Multiparty Computation Schemes: Physical Security and Applications in Secure Implementations

Mohammad Hashemi

A Dissertation
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Doctor of Philosophy
in
Electrical and Computer Engineering

May 2025

APPROVED:

Professor Fatemeh Ganji, Advisor, Worcester Polytechnic Institute

---

Professor Berk Sunar, Committee Member, Worcester Polytechnic Institute

---

Professor Dominic Forte, Committee Member, University of Florida

---

Professor Daniel Holcomb, Committee Member, University of Massachusetts Amherst

I dedicate this dissertation to my beloved mother and father, whose unconditional love and support have been the foundation of my every success. To my brothers and in-laws, thank you for your constant presence and encouragement during the ups and downs of this journey. Most of all, I dedicate this work to my incredible wife, whose love, patience, and unwavering belief in me have been my greatest source of strength. She stood by me through every late night, every setback, and every milestone, celebrating the victories and lifting me through the challenges. Her sacrifices, kindness, and boundless encouragement gave me the courage to keep moving forward. This achievement is as much hers as it is mine.

Abstract

The increasing reliance on cloud-based computation, machine learning services, and collaborative design tools has introduced significant challenges in ensuring data confidentiality and computational integrity. Although Secure Multiparty Computation (MPC) and Fully Homomorphic Encryption (FHE) offer strong theoretical guarantees, their practical adoption remains limited due to performance bottlenecks and exposure to physical attacks such as side-channel leakage and fault injection.

This dissertation addresses these challenges through a set of hardware-assisted secure computation frameworks that improve both efficiency and robustness. GarbledEDA introduces a privacy-preserving electronic design automation solution that secures intellectual property during hardware verification using optimized garbled circuits. GuardianMPC builds on this foundation by accelerating secure neural network inference through parallel garbled circuit evaluation and customized hardware modules for oblivious transfer, while also incorporating backdoor detection mechanisms to ensure model integrity. To highlight practical vulnerabilities, we present Goblin, a timing side-channel attack that targets widely-used MPC frameworks, revealing how variations in execution time can be exploited to recover secret

inputs. FaultyGarble demonstrates that laser fault injection can extract proprietary neural network parameters from garbled circuit-based secure inference systems, exposing the limitations of cryptographic guarantees in the presence of physical adversaries. We also present Bake It Till You Make It!, a novel temperature-based side-channel attack that shows how controlled heating can bypass masking defenses in secure hardware implementations. To address such risks, we propose HWGN^2, a secure inference framework based on secure function evaluation, which is designed to resist power, timing, and electromagnetic side-channel attacks. Finally, Garblet introduces a chiplet-aware architecture for secure MPC deployment across heterogeneous hardware platforms. By distributing garbled computations and integrating hardware-level optimizations, Garblet reduces communication overhead and maintains strong security, even when operating across untrusted components. These contributions demonstrate that secure computation in modern systems requires a multi-layered approach, combining cryptographic techniques with physical defenses and architectural awareness. This dissertation advances the state of the art by bridging the gap between theoretical protocols and practical, secure implementations that can withstand real-world adversaries and physical threats.

Acknowledgments

I am deeply grateful to my advisor, Professor Fatemeh Ganji, for her unwavering support, insightful guidance, and continuous encouragement throughout my Ph.D. studies. Her mentorship has been instrumental not only in shaping the direction of my research but also in refining the way I approach challenges with critical thought.

I would also like to sincerely thank my dissertation committee members, Professor Berk Sunar, Professor Domenic Forte, and Professor Daniel Holcomb, for their valuable feedback, thoughtful insights, and generous contributions of time and expertise. Their input has been vital in enhancing the quality and depth of this work.

This research was supported by the Semiconductor Research Corporation (SRC) under Task IDs 2991.001 and 2992.001, and the National Science Foundation (NSF) under award number 2138420.

I am especially thankful to my colleagues and friends at Vernam Lab and the University of Florida for their collaboration and supportive environment that made this journey both productive and memorable.

Contents

List of Tables

List of Figures

Chapter 1

Publications

The publishers' versions of the following peer-reviewed publications are fully included in this thesis:

1. **M. Hashemi**, S. Tajik, and F. Ganji, "Garblet: Multi-party Computation for Protecting Chiplet-based Systems," in *IEEE VLSI Test Symposium (VTS)*, 2025.

2. **M. Hashemi**, D. J. Forte, and F. Ganji, "GuardianMPC: Backdoor-resilient Neural Network Computation," *IEEE Access*, 2025.

3. **M. Hashemi**, D. Forte, and F. Ganji, "Time is money, friend! Timing side-channel attack against garbled circuit constructions," in *International Conference on Applied Cryptography and Network Security (ACNS)*, Cham: Springer Nature Switzerland, 2024, pp. 325–354.

4. **M. Hashemi**, D. Mehta, K. Mitard, S. Tajik, and F. Ganji, "FaultyGarble: Fault Attack on Secure Multiparty Neural Network Inference," *Workshop on Fault Detection and Tolerance in Cryptography (FDTC), Pages 53-64*, 2024.

5. **M. Hashemi**, S. Roy, D. Forte, and F. Ganji, "HWGN 2: Side-Channel Protected NNs Through Secure and Private Function Evaluation," in *International Conference on Security, Privacy, and Applied Cryptography Engineering (SPACE)*, Cham: Springer Nature Switzerland, 2022, pp. 225–248.

6. **M. Hashemi**, S. Roy, F. Ganji, and D. Forte, "Garbled EDA: Privacy Preserving Electronic Design Automation," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2022, pp. 1–9.

Along with the above items, the following additional peer-reviewed publication was authored by Mohammad Hashemi ("*" denotes that both authors contributed equally to the corresponding work):

1. D. M. Mehta*, **M. Hashemi***, D. S. Koblah, D. Forte, and F. Ganji, "Bake it till you make it: Heat-induced power leakage from masked neural networks," *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, vol. 2024, no. 4, pp. 569–609.

Chapter 2

Contribution

The presented dissertation is based on multiple published co- authored works. In the following, I list my own contributions according to the Doctoral Regulations. I had experimental, editorial, and content responsibilities in these publications. The list was approved by all co-authors.

2.0.1 Publication 1

M. Hashemi, D. J. Forte, and F. Ganji, "GuardianMPC: Backdoor-Resilient Neural Network Computation," *IEEE Access*, 2025.

My contributions are as follows:

- I implemented the entire GuardianMPC framework, including private training and oblivious inference.
- I conducted the full experimental evaluation, including the implementation and performance analysis of private training and oblivious inference.
- I drafted Sections 2-8, which include the methodology, implementation details, results, and discussion sections.

2.0.2 Publication 2

M. Hashemi, D. Forte, and F. Ganji, "Time is Money, Friend! Timing Side-Channel Attack Against Garbled Circuit Constructions," in *International*

Conference on Applied Cryptography and Network Security (ACNS), Cham: Springer Nature Switzerland, 2024, pp. 325–354.

My contributions are as follows:

- I implemented the Goblin attack including the clustering approach, ensuring its effectiveness against various garbled circuit frameworks.
- I identified and analyzed the leaky IF conditions within garbled circuit implementations.
- I evaluated the success rate (SR) of the attack and analyzed its scalability across different benchmark functions.
- I drafted Sections 2-7, which include the methodology, implementation details, and experimental evaluation.

2.0.3 Publication 3

M. Hashemi, D. Mehta, K. Mitard, S. Tajik, and F. Ganji, "FaultyGarble: Fault Attack on Secure Multiparty Neural Network Inference," *Workshop on Fault Detection and Tolerance in Cryptography (FDTC), Pages 53-64*, 2024.

My contributions are as follows:

- I implemented the MIPS architecture on the FPGA to serve as the computation platform for secure neural network inference.
- I conducted the fault injection experiments in simulation, ensuring the effectiveness of the attack under controlled conditions.
- I collaborated with Kyle Mitard and Dev Mehta, where Kyle developed the AlphaNOV laser fault injection setup, and Dev launched the laser fault injection experiments.
- I drafted Sections 2-7, which include the methodology, implementation details, and experimental evaluation. Section V.C was written by Dev Mehta.

2.0.4 Publication 4

M. Hashemi, S. Roy, D. Forte, and F. Ganji, "HWGN2: Side-channel Protected Neural Networks through Secure and Private Function Evaluation," in *International Conference on Security, Privacy, and Applied Cryptography Engineering (SPACE)*, Cham: Springer Nature Switzerland, 2022, pp. 225–248.

My contributions are as follows:

- I implemented the HWGN2 framework, including the secure and private function evaluation approach for side-channel protection.
- I conducted the hardware implementation and optimization of the MIPS-based secure neural network accelerator.
- I performed the side-channel leakage evaluation and validation experiments for HWGN^2.
- I drafted Sections 2-7, which include the methodology, implementation, experimental evaluation, and discussion sections. Steffi Roy assisted with writing the background section of the paper.

2.0.5 Publication 5

M. Hashemi, S. Roy, F. Ganji, and D. Forte, "Garbled EDA: Privacy Preserving Electronic Design Automation," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2022, pp. 1–9.

My contributions are as follows:

- I implemented the Garbled EDA framework, including its secure function evaluation (SFE) and private function evaluation (PFE) approaches.
- I conducted the full experimental evaluation and optimization of Garbled EDA, including the MIPS- and ARM-based secure electronic design automation implementations.
- I performed the security analysis and validation of the framework against potential adversaries.

- I drafted Sections 3-5, which include the methodology, implementation, evaluation, and discussion sections. Steffi Roy assisted with writing the background section of the paper.

2.0.6 Publication 6

M. Hashemi, S. Tajik, and F. Ganji, "Garblet: Multi-party Computation for Protecting Chiplet-based Systems," in *IEEE VLSI Test Symposium (VTS)*, 2025.

My contributions are as follows:

- I implemented the hardware and software components of the Garblet framework, integrating customized Oblivious Transfer (OT) modules and an optimized evaluator engine for chiplet-based secure computation.
- I designed and implemented the circuit decomposition technique, which enables efficient parallel execution across multiple chiplets.
- I conducted the full experimental evaluation, including performance analysis on an AMD/Xilinx UltraScale+ multi-chip module.
- I drafted Sections 2-5, which include methodology, implementation, experimental results, and discussion.

2.0.7 Publication 7

D. M. Mehta*, **M. Hashemi***, D. S. Koblah, D. Forte, and F. Ganji, "Bake It Till You Make It: Heat-Induced Power Leakage from Masked Neural Networks," *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, vol. 2024, no. 4, pp. 569–609.

My contributions are as follows:

- I actively engaged in discussions regarding this project with all co-authors, including Dev M. Mehta, David S. Koblah, Domenic Forte, and Fatemeh Ganji.
- I investigated the timing difference effects caused by temperature variations and their impact on side-channel leakage.

- I implemented the differential power analysis (DPA) algorithm in Python to analyze the power leakage in masked neural networks.
- I launched the DPA attacks based on power traces provided by Dev Mehta, ensuring accurate analysis and validation of the attacks.
- I contributed to writing the manuscript, specifically sections related to DPA results and the timing difference effects of temperature on FPGA-based accelerators (Sections 4, 6.3-6.7).

Chapter 3

Introduction

3.1 Motivation

The growing use of digital data and computing has raised serious concerns about privacy, security, and data integrity. As technologies like cloud computing, machine learning as a service (MLaaS), and shared data processing become more common, it is increasingly important to use secure computation methods that protect data privacy while keeping results accurate [129, 418, 120]. In today's world—where data breaches, cyber spying, and attacks on machine learning systems are more frequent—there is a strong need for computation models that can work on encrypted or hidden data without revealing sensitive information [267, 319].

Traditional cryptographic methods like symmetric and asymmetric encryption offer strong protection for keeping data private during storage and transfer [97]. However, these techniques do not automatically secure data while it is being processed. In many practical situations, computations are carried out on private or sensitive data—such as medical records, financial data, biometric information, or proprietary machine learning (ML) models. If attackers are able to access intermediate results during computation, they may uncover confidential information, which can lead to privacy breaches, financial harm, or theft of intellectual property (IP) [153, 303, 42, 285, 26, 20, 344].

Secure computation approaches like secure multiparty computation (MPC) and homomorphic encryption (HE) help close this gap by allowing computations on encrypted data without revealing the actual values [136, 227]. These

techniques let multiple parties work together to compute a function while keeping their own inputs private, which is important for use cases such as privacy-focused medical research, confidential federated learning, and secure voting systems [82]. Although these methods are strong in theory, they often struggle with real-world challenges like high computation time, communication overhead, and the need to defend against attackers who take advantage of software or hardware weaknesses [319, 267, 217, 290, 68, 404, 44].

One of the main motivations behind secure computation research is the growing understanding that cryptographic security alone is not enough to defend against physical threats. Even if an encryption method is mathematically secure, attackers can still gather information, interfere with computations, or cause errors using techniques like side-channel attacks, fault injection, or hardware backdoors [117, 28, 357, 210, 236, 59]. These risks are especially important in situations where computations are done on hardware that may not be trusted, such as in cloud services, outsourced ML inference, or embedded edge devices [158].

In recent years, secure deep learning (DL) inference has become a major focus in secure computation research, especially as AI models are increasingly used in privacy-sensitive settings. Many DL models used for tasks such as biometric authentication, fraud detection, and national security need to remain confidential from both the users and the service providers [319]. To address this, secure inference frameworks using garbled circuits (GC) [35] and homomorphic encryption (HE) [76] have been developed. These systems let clients query models without revealing their inputs, and also keep the model itself hidden, helping to prevent model inversion attacks or adversarial tampering [267, 154].

However, even advanced secure inference methods can still be vulnerable to physical attacks. For example, the FaultyGarble [155] attack showed that an attacker can use laser fault injection to break the security of GC-based secure inference and recover proprietary neural network (NN) parameters [155]. This case illustrates the practical limits of relying only on cryptographic protections and emphasizes the urgent need for comprehensive security strategies that combine mathematical techniques, physical robustness, and hardware-level defenses [28, 254].

Given the growing range of threats, secure computation needs to go beyond traditional cryptographic methods and include protections against physical tampering, side-channel attacks, and violations of computational integrity [227, 254]. The following sections examine the widening threat

landscape in secure computation, the importance of MPC as a secure computation model, and the difficulties of building truly robust systems that can withstand advanced attackers.

3.2 The Expanding Threat Landscape in Secure Computation

As more computation moves to distributed and untrusted environments, the types of security threats have changed significantly. Secure computation methods, which originally focused on keeping data private during processing, now have to deal with a wider range of attacks that target both software flaws and hardware vulnerabilities [129, 227]. With the growth of cloud computing, edge AI, and collaborative multi-party systems, new attack surfaces have emerged that traditional cryptographic techniques were not built to handle.

While early cryptographic models assumed that attackers could only access data during transmission or storage, modern adversaries go further by directly interfering with the computation process. They take advantage of algorithmic flaws, software bugs, and hardware-level attacks [117, 210]. This shift in attacker capabilities has made traditional security measures insufficient, leading to the need for advanced cryptographic solutions like secure MPC, fully homomorphic encryption (FHE), and trusted execution environments (TEE) [81] to protect both the privacy and correctness of computations, even in hostile settings [120, 267, 68, 81].

The Rise of Cloud and Outsourced Computation

One major reason for the expanding threat landscape in secure computation is the increasing use of cloud based processing. Many organizations outsource their computing tasks to third party cloud providers because it is cost effective and scalable, but this also introduces serious security and privacy risks [196]. Sensitive information, like financial data, medical records, and proprietary ML models, is often processed on hardware that may not be trusted, making it vulnerable to data breaches, insider attacks, and unauthorized monitoring [319].

For instance, MLaaS providers offer inference capabilities where users send their private data to a pre trained model hosted on the cloud [389]. While this approach enhances accessibility, it also introduces risks of model

inversion attacks, where adversaries reconstruct input data from observed outputs, and membership inference attacks, where attackers determine whether a particular data point was used in training [352]. Secure inference protocols based on GC and HE have been proposed to mitigate these risks, but their practicality is still challenged by computational overhead and vulnerability to side channel analysis [267, 319].

Side-Channel and Microarchitectural Attacks

Modern computation does not only face software based vulnerabilities, adversaries increasingly exploit side channel leaks to extract sensitive information from secure systems. These attacks rely on unintended information leakage, such as power consumption, electromagnetic (EM) emissions, timing differences, and cache access patterns, to uncover cryptographic keys or secret data [117, 210].

A well known example of such threats is cache based side channel attacks (SCA), which take advantage of shared memory in modern processors to leak confidential information. Attacks like Flush+Reload [421] and Spectre [207], Meltdown [236] use speculative execution and cache timing differences to extract sensitive data from protected memory [236, 59]. These vulnerabilities have shown that even hardware based security boundaries, such as isolation between processes or virtual machines, can be bypassed using microarchitectural side channels.

Another example is the Goblin attack [153], which revealed how timing side channels could be exploited in secure computation frameworks such as JustGarble [187] and Obliv-C [427]. By analyzing how long certain operations take to run, adversaries were able to recover secret inputs processed using GC, breaking the privacy protection promised by secure computation. These findings show the need for side channel resilient cryptographic methods that do not leak information during execution.

Fault Injection and Physical Tampering Attacks

Beyond passive side channel threats, attackers can actively disrupt computation by injecting faults into hardware and software to uncover secrets or cause incorrect results. Fault injection attacks, including laser based, EM, and voltage glitching methods, have been widely explored as ways to break cryptographic systems [28, 357]. These attacks are especially dangerous because

they do not need direct access to secret keys, instead they cause computation errors that reveal key dependent differences in processed data.

For example, the FaultyGarble [155] attack showed how laser induced faults could be used to extract proprietary DL models from secure inference systems based on GC. By carefully injecting faults during computation, attackers could bypass the privacy protections of MPC protocols and recover NN parameters with high accuracy.

In a similar way, recent work on heat related power leakage has shown that even masked cryptographic systems, which are meant to stop power analysis attacks, can be vulnerable when exposed to controlled temperature changes [254]. These results suggest that traditional defenses like masking and blinding may not be enough in situations where attackers have physical access to the computing device.

Security Challenges in Emerging Hardware Architectures

The development of hardware architectures has made the secure computation landscape more complex. With the growing use of heterogeneous computing, chiplet based systems, and specialized AI accelerators, securing computation now involves addressing new hardware related attack surfaces. Unlike traditional single chip processors, chiplet based designs combine components from different vendors, which brings potential risks such as supply chain attacks, hardware backdoors, and untrusted chiplet behavior [303].

For instance, Garblet [158], a secure MPC framework built for chiplet based systems, showed that even when using cryptographic protocols like GC, communication overhead and trust issues in multi chiplet setups need careful attention. This highlights the importance of hardware assisted secure computation approaches that can reduce the risks from third party components that may not be trusted.

The Need for Multilayered Secure Computation Strategies

Given the rapidly growing range of threats, secure computation must move beyond traditional cryptographic solutions and adopt layered security strategies. This includes cryptographic hardening, which focuses on making MPC and HE based protocols more resistant to side channel and fault injection attacks [227, 267]. It also involves hardware supported security, using TEE, physically unclonable functions (PUF), and hardware root of trust systems

to strengthen cryptographic protections [59, 254]. Another key method is secure system design, which focuses on building fault tolerant, tamper resistant computation frameworks that can detect and respond to attacks on the computation process [154].

The next section discusses MPC as a secure computation model, outlining its theoretical background and practical use in addressing the security challenges introduced by today's advanced attack techniques.

3.3 MPC as a Secure Computation Model

Secure MPC is a cryptographic framework that lets multiple parties compute a shared function over their private inputs without revealing individual data [418, 129, 227, 37, 82]. Unlike standard encryption methods that protect data while stored or sent, MPC keeps data private during the whole computation, making sure participants only learn the final result. This is especially useful when different organizations want to work together on private data without giving up privacy. Examples include financial institutions doing joint risk analysis [47, 282], hospitals working together on research while keeping patient records private [393, 197], and governments performing statistical analysis on sensitive data sets [48].

MPC began with Yao's GC, introduced in the 1980s, which showed how two parties could compute a function together while keeping their inputs hidden [418]. This idea, called Secure Function Evaluation (SFE), allowed private computation of Boolean circuits [35, 429]. Over time, research extended beyond two party settings to more general multi party cases, resulting in protocols based on secret sharing (SS) [37, 87] and HE [120]. While SFE protects fixed functions, it does not naturally support situations where the function itself must stay secret. This led to work on Private Function Evaluation (PFE), where both the inputs and the function remain private [200, 265]. PFE is important for applications like secure IP verification, where revealing the function could expose proprietary algorithms [156].

As computation moved from isolated systems to distributed and cloud based platforms, the need for secure computation became more urgent [196, 259]. Many modern tasks involve data held by multiple parties that do not fully trust each other [319, 267]. For example, cloud based ML services process large amounts of personal data that must stay private [352, 389]. In federated learning, different groups train models together without sharing

their data [48]. Traditional cryptography protects stored and transmitted data, but not the intermediate results during computation. This has made MPC a widely used tool for privacy preserving computation [267, 319].

One of the biggest challenges in MPC is efficiency [202, 87]. Early protocols were mostly theoretical and too slow or complex for practical use. However, improvements in circuit based computation, such as better garbling methods [429, 35] and HE [120], have helped reduce these costs. Secret sharing approaches, which divide data into parts shared among participants, offer another way to enable secure computation with lower overhead [37, 87, 82]. These techniques are used in many modern MPC frameworks that aim to balance security and performance [202, 265].

With its growing adoption, MPC has been applied in many fields [47, 282]. Privacy preserving ML is one of the most promising areas, using GC and HE to train and run models on encrypted data [267, 319, 163, 194, 260]. Financial services use MPC for secure auctions and fraud detection, protecting sensitive financial data [47, 282]. Electronic voting systems rely on MPC to count votes without revealing individual choices [79, 83]. Secure Electronic Design Automation (EDA) uses MPC to protect intellectual property during joint chip design projects [157]. Genomic research also benefits from MPC, enabling private analysis of genetic data without risking personal privacy [393, 197].

Despite this progress, challenges still remain [319, 267, 202]. MPC protocols still require more computation than regular methods. Large scale use needs smart optimizations to reduce communication and support scalability [202]. Real world systems must also protect against side channel threats, where attackers might gather information through timing or power use [117, 254]. While MPC gives strong cryptographic protection, it is often used with trusted hardware like Intel SGX to boost performance [81, 59]. However, this creates new trust issues, since weaknesses in TEE could put security at risk [59, 344].

The rise of MPC marks a big shift in secure computation, making privacy preserving collaboration possible across many applications. As cryptographic tools continue to improve, future work will aim to boost efficiency, support larger systems, and guard against new types of threats. The next section looks at the challenges of building secure computation systems in adversarial settings, with a focus on side channel and fault injection attacks that put the integrity of private computation at risk.

3.3.1 Challenges in Secure Implementations: Side-Channel and Fault Attacks

Secure computation, in its theoretical form, provides strong mathematical guarantees to protect confidentiality and correctness. However, real world implementations of these cryptographic protocols face a very different set of challenges, ones that come not from flaws in cryptography, but from the physical and architectural characteristics of computing systems. Adversaries today do not always break cryptographic schemes in the usual way, instead, they take advantage of how computations are carried out. Two of the most serious threats to secure computation are SCA and fault injection attacks, which can weaken even the strongest cryptographic protocols by using unintended leaks and physical changes [210, 117, 236, 344].

These attacks have serious consequences. They have been used to steal secret cryptographic keys, uncover private inputs, and even break security protections in TEE. For example, attacks on Intel SGX and ARM TrustZone have shown that secure enclaves can be compromised using microarchitectural side channels [59, 344, 81]. Similarly, fault injection attacks have been used to break cryptographic systems, such as the well known Bellcore attack, where attackers recovered RSA private keys by causing errors during computation [49]. As secure computation is used more often in areas like privacy preserving ML, cloud based secure inference, and private financial transactions, dealing with these risks becomes critical.

Side-Channel Attacks: Exploiting Unintended Leaks

Cryptographic protocols assume that adversaries only have access to a system's input and output, but real world implementations leak information in subtle, yet exploitable ways. SCA focus on these unintended leaks, allowing adversaries to recover secret data by observing physical effects such as power use, EM emissions, execution timing, or memory access patterns [210, 111, 117, 422]. One of the earliest and most well known SCA is Differential Power Analysis (DPA), introduced by Kocher et al., which showed that by measuring small changes in power use during encryption, attackers could recover cryptographic keys [210]. This method was later extended to target many cryptographic systems, including smartcards, embedded devices, and hardware security modules (HSMs) [247, 321]. More recently, microarchitectural attacks such as Flush, Reload, Spectre, and Meltdown have shown that even

CPU improvements like speculative execution and branch prediction can leak sensitive data, breaking the separation between processes [208, 236, 422, 59].

In secure MPC and HE, side channel risks present new challenges. Since these methods involve heavy computation, they often depend on hardware accelerators and optimized code. Unfortunately, these optimizations often cause timing differences or memory access patterns that can be exploited. Researchers have shown cache attacks that extract private model parameters from ML models running in cloud systems [319, 389, 239]. In a similar way, side channel analysis has been used to recover cryptographic keys from FHE schemes, raising concerns about using HE based secure inference in real applications [239, 260].

The risk from SCA is not just theoretical, it has been shown in real world cryptographic systems. For example, researchers have recovered AES encryption keys from cloud based virtual machines using cache attacks [435, 422]. EM SCA have also been used to extract RSA keys from cryptographic devices that are physically protected [111, 94]. These attacks show the need for strong countermeasures, including constant time code, randomized memory access, and hardware designed to resist side channel attacks [247, 81].

Fault Injection Attacks: Manipulating Computation

While SCA passively extract information, fault injection attacks actively interfere with computation to cause errors that can be exploited. These attacks use techniques such as laser fault injection, voltage glitching, and EM disturbance to produce incorrect results, which can then be studied to uncover secrets or bypass security protections [356, 28, 94, 321]. One of the most well known examples is the Bellcore attack on RSA cryptography. By adding faults during modular exponentiation, attackers were able to extract private keys by analyzing the wrong outputs [49]. This attack showed that cryptographic systems must not only be mathematically correct, but also resistant to faults introduced on purpose [357, 28].

Modern fault attacks have become more advanced. Laser based fault injection has been used to steal cryptographic keys from smartcards, get around secure boot protections, and even break secure enclave systems like Intel SGX [59, 81, 252]. Similarly, clock glitches and EM interference have been used to bypass PIN checks in embedded devices and payment systems [94, 28]. A particularly concerning example of fault related vulnerabilities is Faulty-Garble [155], an attack on GC based secure inference. Researchers showed

that by adding hardware faults, they could make GC leak information, allowing them to recover the structure and parameters of DL models running on a system that was thought to be secure [153, 254, 170, 386, 420].

Towards Secure and Resilient Implementations

Addressing these vulnerabilities requires a multi layered defense strategy. One effective countermeasure involves masking and blinding techniques, which randomize power use, timing, and memory access patterns to reduce side channel leakage [247, 117]. Oblivious RAM (ORAM) techniques offer another layer of security by making memory access patterns indistinguishable, which helps stop cache based SCA [131, 202]. Also, using secure hardware enclaves like Intel SGX and ARM TrustZone provides isolated environments for execution, though they are still open to microarchitectural attacks and need extra protection to guard against speculative execution issues [59, 81, 16].

Fault tolerant cryptographic systems use repeated computations, error detection, and self repairing circuits to defend against fault injection attacks [28, 357, 254]. For example, modern secure boot systems use integrity checks that can spot and block changes caused by voltage glitching attacks [252]. Likewise, techniques such as dual rail logic and duplicated execution paths help reduce the impact of both side channel and fault injection attacks [247, 28].

As attackers continue to improve their methods, protecting against side channel and fault injection attacks will require ongoing progress in hardware security, cryptographic system design, and secure engineering practices. The next section explores research contributions that build on these challenges, offering new ways to improve secure computation in real systems.

3.4 Research Contributions and Scope

3.4.1 Summarizing the Key Gaps in Prior Work

The field of secure MPC and hardware based security has developed significantly over the past few decades, yet major challenges remain in building practical, scalable, and resilient systems. Traditional cryptographic methods, such as FHE and general GC, though strong in theory, often face computation and communication inefficiencies that limit their use in real world settings. In addition, side channel and fault injection attacks have exposed

serious weaknesses in current secure inference systems, showing the need for stronger protections supported by hardware. Also, as chiplet based architectures become more common in high performance computing, new challenges appear in securing distributed computations across different, and possibly untrusted, hardware components.

Even with many advances, past research has had trouble fully solving these problems. Many current MPC implementations are still not practical because of high overhead, poor scalability, and vulnerability to physical attacks. By carefully reviewing earlier work, we point out several key gaps that have driven our research across different parts of secure computation, including secure EDA, faster MPC systems, and hardware security for NN inference.

Inefficiencies in Secure Computation Protocols

One of the main limitations of traditional secure computation protocols is their high computational and communication overhead. While GC provide a base for two party SFE, the performance slowdowns caused by garbling and evaluating large circuits have limited their broader use [227]. Some improvements, such as free XOR and half gates, have lowered parts of this overhead, but they still do not fully solve the scalability issues in complex secure computations [429, 35]. Fully FHE offers another approach by allowing computations on encrypted data without decryption, but its slow speed and high delay make it a poor fit for real time use [119].

Our work on GarbledEDA tackles these problems by creating a privacy preserving EDA framework that greatly reduces the computation needed for secure circuit verification [157]. By using circuit specific improvements, we show that secure design verification can be done with practical overhead, making it possible to protect IP in shared chip design projects.

Vulnerabilities to Side-Channel and Fault Injection Attacks

Another important limitation in earlier work is the weakness of secure computation systems against side channel and fault injection attacks. Many secure inference protocols are designed with only a cryptographic adversary in mind, and they do not consider real world attack methods such as power analysis, EM leakage, and laser based faults [117, 28]. These attacks give adversaries the ability to extract secret model parameters, change computations, and

bypass cryptographic protections with surprising ease.

Our work on FaultyGarble [155] reveals these weaknesses in secure DL inference by showing how laser fault injection can be used to recover private model weights from GC. This research shows the urgent need for MPC frameworks that can resist faults, and it drives our follow up efforts to build hardware based protections for secure computation.

Communication Overhead in Secure MPC Implementations

Beyond computational inefficiencies, communication overhead is also a major challenge in secure computation. Most existing MPC frameworks depend on frequent message exchanges between the parties involved, which greatly increases both latency and bandwidth usage. This problem is especially noticeable in privacy preserving ML, where large NN models make secure inference even more costly [267, 319].

To reduce these overheads, GuardianMPC [154] presents an improved secure computation framework that uses custom hardware parts to speed up oblivious transfer (OT) and lower communication costs in secure inference. Our framework shows clear improvements in evaluation speed compared to regular MPC setups, making privacy preserving NN inference more practical for real world use.

Security Risks in Emerging Chiplet-Based Architectures

As chiplet-based architectures gain traction in high-performance computing, securing distributed computations across untrusted chiplets becomes a growing concern. Traditional security models assume a monolithic, trusted processor, but in modern multi-chip systems, untrusted chiplets or interposers can act maliciously, intercepting communication or injecting faults into secure computations [303].

Our work on Garblet [158] pioneers the application of MPC techniques to protect computations in chiplet-based systems. By distributing GC across multiple chiplets and leveraging hardware-based OT modules, we demonstrate a significant reduction in communication overhead while maintaining security guarantees even in adversarial hardware environments. This approach sets a precedent for securing next-generation heterogeneous computing platforms.

HWGN^2 and the Limitations of Existing Secure Inference Frameworks

NN hardware accelerators are highly susceptible to SCA, leading to IP theft. Several existing solutions have attempted to mitigate these threats, including masked inference engines such as MaskedNet [103], ModuloNet, and BoMaNet [101]. MaskedNet introduced hardware-level masking techniques to counter differential power analysis (DPA) attacks, incorporating masked adder trees and masked activation functions [103]. Building on this, ModuloNet [102] leveraged modular arithmetic for efficient hardware masking, reducing overhead while maintaining security [101]. BoMaNet further advanced this field by implementing Boolean masking across entire NNs, addressing both linear and non-linear operations to enhance resistance against SCA [102].

HWGN^2 [156] adopts a fundamentally different approach by utilizing GC and SFE to achieve side-channel resistance. Unlike MaskedNet, ModuloNet, and BoMaNet, which focus on masking techniques at the hardware level, HWGN^2 inherently ensures both input privacy and model confidentiality. By implementing GC using a microprocessor without interlocked pipeline stages (MIPS)-based architecture, HWGN^2 significantly reduces logical and memory utilization while maintaining strong security guarantees against side-channel and fault injection attacks. However, this comes at the cost of increased communication overhead. Our work demonstrates that HWGN^2 not only improves resistance to real-world adversarial threats but also expands the scope of privacy-preserving inference to scenarios where traditional masking-based methods may fall short.

Bridging the Gap Between Theory and Practice in Secure Computation

Many cryptographic protocols for secure computation remain confined to theoretical models, lacking real-world implementations that balance security with efficiency. While the field has seen numerous breakthroughs in provable security, translating these techniques into practical, scalable solutions remains an ongoing challenge [202, 37].

Our collective body of work, including Goblin [153] and Bake It Till You Make It [254], bridges this gap by evaluating the real-world performance trade-offs of secure computation techniques. Through rigorous benchmark-

ing and hardware-assisted optimizations, we provide valuable insights into the feasibility of deploying MPC-based solutions in resource-constrained environments.

The limitations of prior work, including inefficiencies in secure computation, vulnerability to side-channel and fault injection attacks, high communication overhead, and security challenges in chiplet-based architectures, underscore the need for more robust and practical solutions. Our research systematically addresses these gaps by introducing novel frameworks, optimizations, and hardware-assisted techniques to enhance the security, efficiency, and scalability of secure computation.

3.5 Dissertation Organization

The organization of the dissertation is as follows. Chapter 4 provides an overview of the fundamental concepts required to understand the dissertation's contributions. It introduces SFE and PFE, explaining their theoretical foundations and practical applications. The chapter then delves into Yao's GC, including mathematical definitions, the garbling process, and various optimizations such as Free-XOR [213], Half-Gates [428], and Row Reduction [428]. Additionally, it covers OT (OT) and adversary models in secure computation, highlighting passive, honest-but-curious, and active adversaries. The chapter concludes with a discussion of side-channel and fault injection attacks, including their impact on cryptographic implementations, as well as a brief introduction to chiplet-based architectures and their security implications.

Chapter 5 presents a survey of secure computation techniques, with a focus on GC, OT, and their role in MPC. It includes an in-depth examination of side-channel and fault injection attacks targeting secure computation models and existing countermeasures. Additionally, this chapter reviews hardware security techniques such as instruction-level obfuscation and power/EM hiding. The discussion extends to the limitations of existing approaches, motivating the need for novel contributions.

Chapter 6 introduces GarbledEDA [157] and GuardianMPC [154], two novel frameworks designed to enhance the security of EDA and DL models. GarbledEDA provides a privacy-preserving approach to secure hardware design, ensuring that IP remains confidential throughout the design and verification process. GuardianMPC extends these principles to DL, introducing

techniques for secure and backdoor-resilient NN computation. The methodologies, implementation details, and experimental results for both frameworks are discussed in detail.

Chapter 7 explores vulnerabilities in secure computation arising from side-channel analysis. The chapter presents the Bake-It attack [254], which leverages heat-induced power leakage in masked NNs, demonstrating a novel class of temperature-based attacks. Additionally, HWGN^2 [156] is introduced as a side-channel-protected NN framework, utilizing secure and PFE to mitigate power and EM leakage. The experimental evaluation highlights the strengths and weaknesses of various countermeasures against such attacks.

Chapter8 focuses on active adversarial techniques, specifically fault injection attacks. The FaultyGarble [155] framework is introduced, demonstrating how laser fault injection can compromise secure multiparty NN inference. The chapter details various fault injection methodologies, their impact on garbled circuit-based computations, and possible countermeasures to mitigate such threats. Experimental results illustrate the effectiveness of these attacks and provide insight into potential improvements in secure hardware design.

Chapter 9 summarizes the key contributions of this dissertation, emphasizing the advancements made in secure computation, hardware security, and side-channel/fault attack resilience. The chapter discusses open challenges and future research directions, including enhancements to secure inference, optimizations for GC, and new methodologies for defending against emerging hardware-based threats.

3.6 Discussion

Advancing the security and efficiency of secure MPC necessitates integrating robust countermeasures into next-generation frameworks. This integration addresses emerging threats and enhances the practical deployment of MPC protocols. Key strategies include incorporating hardware-based TEE, optimizing cryptographic protocols, and implementing comprehensive side-channel attack mitigations.

Leveraging Trusted Execution Environments

TEEs, such as Intel's Software Guard Extensions (SGX) [81], provide isolated environments that protect data during processing, ensuring confidentiality and integrity. By executing sensitive computations within TEEs, MPC frameworks can significantly reduce the attack surface exposed to potential adversaries. This hardware-based isolation complements traditional cryptographic protections, offering a multifaceted defense strategy. However, it's essential to acknowledge that TEEs are not impervious to all attacks; vulnerabilities such as side-channel exploits have been identified, necessitating ongoing research and enhancement [81].

Optimizing Cryptographic Protocols

Enhancing the efficiency of cryptographic operations is crucial for the scalability of MPC systems. Techniques like GC and OT have been foundational but often entail substantial computational and communication overhead. Recent advancements aim to streamline these protocols, reducing latency and resource consumption without compromising security. For instance, implementing parallel processing and leveraging hardware accelerators can expedite cryptographic computations, making MPC more viable for real-time applications [320].

Mitigating Side-Channel Attacks

SCA exploit indirect information leakage, such as timing, power consumption, or EM emissions, to infer sensitive data. To counter these threats, next-generation MPC frameworks must incorporate comprehensive countermeasures, including constant-time algorithms, noise generation, and hardware-level protections. Integrating these defenses ensures that even if an adversary can monitor physical parameters, the extracted information remains insufficient to compromise the system's security [247].

By embedding these countermeasures into the core architecture of MPC frameworks, we can enhance their resilience against sophisticated attacks, paving the way for broader adoption in various applications. Continuous research and development are imperative to adapt to evolving threat landscapes and to maintain the robustness of secure computation methodologies.

Chapter 4

Chapter 4: Background and Preliminaries

4.1 Secure Function Evaluation and Private Function Evaluation

4.1.1 Definition of SFE and PFE

SFE is a cryptographic framework that enables multiple parties to compute a function f over their private inputs while ensuring that no party learns anything beyond the function's output. This property is essential for preserving privacy in applications such as privacy-preserving ML, secure auctions, and medical data analysis.

PFE extends SFE by ensuring that, in addition to hiding inputs, the function f itself remains confidential. PFE is crucial in scenarios where a proprietary function must remain undisclosed, such as secure IP evaluation and hidden rule-based decision-making.

4.2 Yao's GC

GC allow two parties, referred to as a garbler (P_1) and an evaluator (P_2), to securely compute a Boolean function f without revealing their private inputs. The core idea is to transform the original Boolean circuit into a garbled circuit, which the evaluator processes using encrypted values. The only information revealed to the evaluator is the final output.

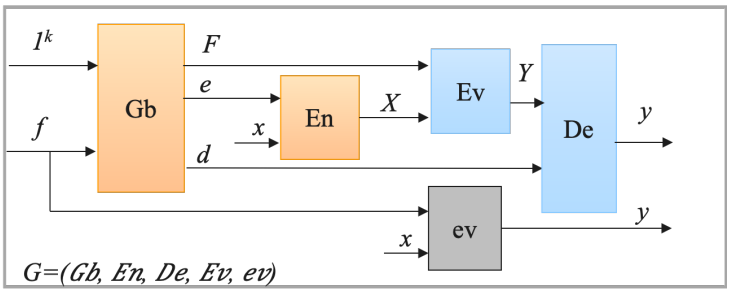


Figure 4.1: A generic garbling scheme $G = (Gb, En, De, Ev, ev)$ cf. [35]. Our proposed secure and private DL accelerator is built upon G. Note that capital letters on the arrows represent garbled (protected) values/functions while lower case represent raw (unprotected) ones. The blocks in orange show the operations performed by the NN vendor, whereas the blues ones indicate the evaluator operations. ev denotes the typical, unprotected evaluation of the function f against the input x, e.g., simulation of an IP using the PDK and the EDA to obtain the output y. F, X, e and d are the counterparts of these in the garbling scheme G that yields y after decoding Y.

The execution of GC consists of two primary processes:

- **Garbling:** The garbler (P_1) transforms the function f into an encrypted circuit representation, ensuring that intermediate values remain hidden.
- **Evaluation:** The evaluator (P_2) processes the garbled circuit using encrypted values to compute the function output without learning any intermediate results.

4.2.1 Mathematical Definition of GC

Figure 4.1, where the strings d, e, f, and F are used by the functions De, En, ev, and Ev. A garbling scheme is a tuple of five probabilistic polynomial-time algorithms:

$$G = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev})$$

where:

- $\mathsf{Gb}(1^\lambda, f) \to (F, e, d)$ is the garbling algorithm that transforms f into a garbled function F, along with an encoding function e and a decoding function d.

- $\mathsf{En}(e, x) \rightarrow X$ is the encoding algorithm, which maps the plaintext input x to garbled input X.
- $\mathsf{Ev}(F, X) \rightarrow Y$ is the evaluation algorithm, which computes the garbled output Y.
- $\mathsf{De}(d, Y) \rightarrow y$ is the decoding algorithm, which maps Y back to the plaintext output y.
- $\mathsf{ev}(f, x) = y$ is the plain evaluation of f on x.

A secure garbling scheme ensures that access to (F, X, d) reveals no additional information beyond $y = f(x)$.

4.2.2 Garbling Process

The garbling process transforms a Boolean circuit into an encrypted version that hides all intermediate values. This process is performed by the garbler (P_1).

Steps in the Garbling Process

Given a circuit C representing f, the garbler executes the following steps:

1. **Wire Label Assignment**: Each wire W_i is assigned two random cryptographic labels:
$$\mathsf{label}_i^0, \quad \mathsf{label}_i^1$$
where label_i^b represents logical value $b \in \{0, 1\}$. These labels are randomly selected from a large space to ensure secrecy.
2. **Gate Garbling**: For each gate G in the circuit, an encrypted truth table is constructed:
$$C_{x,y} = \mathsf{Enc}(\mathsf{label}_a^x, \mathsf{label}_b^y, \mathsf{label}_c^z)$$
where:
   - x, y are the gate input values.
   - $z = G(x, y)$ is the output.
   - $C_{x,y}$ is the encrypted output label.

Algorithm 1: Garbling Algorithm

Input: Boolean circuit C with gates $G_1, G_2, \ldots, G_m$

Output: Garbled circuit F, encoding function e, decoding function d

Generate random labels $\mathsf{label}_i^0, \mathsf{label}_i^1$ for each wire W_i;

foreach *gate* G *in* C **do**

  foreach *input combination* (x, y) **do**

    Compute output $z = G(x, y)$;

    Encrypt $C_{x,y} = \mathsf{Enc}(\mathsf{label}_a^x, \mathsf{label}_b^y, \mathsf{label}_c^z)$;

return F, e, d;

3. **Oblivious Transfer (OT)**: The garbler sends encrypted input labels to the evaluator using OT to ensure that P_2 receives the correct input labels without revealing P_1's private inputs.

4.2.3 Evaluation Process

The evaluation process enables the evaluator (P_2) to compute the function output using encrypted labels.

1. **Input Label Retrieval**: The evaluator receives garbled inputs via OT.

2. **Gate Evaluation**: Using the encrypted truth table, the evaluator decrypts the correct output label:

$$\mathsf{label}_c^z = \mathsf{Dec}(\mathsf{label}_a^x, \mathsf{label}_b^y, C_{x,y})$$

3. **Output Decoding**: The evaluator maps the garbled output label to the plaintext output y.

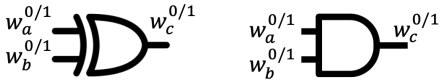
4.3 Optimizations of GC

GC are a cornerstone of SFE, enabling two parties to jointly compute a function over their inputs while preserving privacy [419]. Over time, several

Algorithm 2: Evaluation Algorithm

Input: Garbled circuit F, encoded input X
Output: Decoded output y

foreach *gate* G *in* F **do**
    Retrieve encrypted label $C_{x,y}$;
    Decrypt $\mathsf{label}_c^z = \mathsf{Dec}(\mathsf{label}_a^x, \mathsf{label}_b^y, C_{x,y})$;
Decode $y = \mathsf{De}(d, Y)$;



Input	Garbled Input	XOR Output	Garbled XOR Output	AND Output	Garbled AND Output
0,0	w_a^0, w_b^0	0	$E_{w_a^0,w_b^0}(w_c^0)$	0	$E_{w_a^0,w_b^0}(w_c^0)$
0,1	w_a^0, w_b^1	1	$E_{w_a^0,w_b^1}(w_c^1)$	0	$E_{w_a^0,w_b^1}(w_c^0)$
1,0	w_a^1, w_b^0	1	$E_{w_a^1,w_b^0}(w_c^1)$	0	$E_{w_a^1,w_b^0}(w_c^0)$
1,1	w_a^1, w_b^1	0	$E_{w_a^1,w_b^1}(w_c^0)$	1	$E_{w_a^1,w_b^1}(w_c^1)$

(a)

Figure 4.2: Garbled gates look-up table with no optimization.

optimizations have been introduced to enhance the efficiency of GC protocols [35, 225]. This section delves into three pivotal optimizations: Free-XOR [214], Half-Gates [428], and Row Reduction [299]. Figure 4.2 illustrates the garbled gates look-up table without any optimization.

4.3.1 Free-XOR Optimization

The Free-XOR technique, introduced by Kolesnikov and Schneider [214], revolutionizes the efficiency of GC by allowing XOR gates to be evaluated without any cryptographic operations or communication overhead. This optimization leverages the linearity of the XOR operation to simplify the garbling process, making it significantly more efficient than traditional garbling techniques [35].

In traditional GC, each gate—whether an AND, OR, or XOR gate—requires the construction of a garbled table containing encrypted output values for

Input	Garbled Input	XOR Output	Garbled XOR Output	AND Output	Garbled AND Output
0,0	A_i^0, B_i^0	0	$w_c^0 = A_i^0 \oplus B_i^0$	0	$E_{A_i^0, B_i^0}(w_c^0)$
0,1	$A_i^0, B_i^0 \oplus R$	1	$w_c^0 \oplus R$	0	$E_{A_i^0, B_i^0 \oplus R}(w_c^0)$
1,0	$A_i^0 \oplus R, B_i^0$	1	$w_c^0 \oplus R$	0	$E_{A_i^0 \oplus R, B_i^0}(w_c^0)$
1,1	$A_i^0 \oplus R, B_i^0 \oplus R$	0	w_c^0	1	$E_{A_i^0 \oplus R, B_i^0 \oplus R}(w_c^0 \oplus R)$

(a)

Figure 4.3: Garbled gates look-up table with free-XOR optimization.

every possible combination of inputs. This process is computationally intensive and increases the size of the garbled circuit [362]. However, XOR gates possess a unique property: the XOR of two encrypted values can be directly obtained by XORing the two ciphertexts. Exploiting this property, the Free-XOR technique eliminates the need to garble XOR gates altogether [299].

Figure 4.3 illustrates the garbled gate look-up table with free-XOR optimization. The core idea behind Free-XOR is to assign a global difference, denoted as Δ, between the wire labels corresponding to the binary values 0 and 1. Specifically, for any wire w, the labels are defined as:

$$\begin{aligned} \text{Label for } 0 &: X_w^0 = k_w, \\ \text{Label for } 1 &: X_w^1 = k_w \oplus \Delta, \end{aligned}$$

where k_w is a randomly chosen value, and Δ is a fixed global offset shared across all wires in the circuit [146].

Consider an XOR gate with input wires w_1 and w_2, and output wire w_3. The output of the XOR gate is defined as:

$$w_3 = w_1 \oplus w_2.$$

Using the Free-XOR optimization, the label for the output wire can be computed directly:

$$X_{w_3} = X_{w_1} \oplus X_{w_2}.$$

This computation is possible because:

$$\begin{aligned} X_{w_1} &= k_{w_1} \oplus b_1 \cdot \Delta, \\ X_{w_2} &= k_{w_2} \oplus b_2 \cdot \Delta, \end{aligned}$$

Algorithm 3: Free-XOR Garbling

Input: Circuit C with gates G and wires W
Output: Garbled circuit $\widetilde{C}$ and decoding information

Choose a random global difference Δ;
foreach *wire* $w \in W$ **do**
    Choose a random label k_w for w;
    Set $X_w^0 = k_w$;
    Set $X_w^1 = k_w \oplus \Delta$;
foreach *gate* $g \in G$ **do**
    if *g is an XOR gate with input wires w_1, w_2 and output wire w_3* **then**
        Set $k_{w_3} = k_{w_1} \oplus k_{w_2}$;
    else
        Garble the gate g as in the standard method [362];
Output the garbled circuit $\widetilde{C}$ and decoding information;

where b_1 and b_2 are the binary values (0 or 1) on wires w_1 and w_2, respectively. Therefore:

$$\begin{aligned} X_{w_3} &= (k_{w_1} \oplus b_1 \cdot \Delta) \oplus (k_{w_2} \oplus b_2 \cdot \Delta) \\ &= (k_{w_1} \oplus k_{w_2}) \oplus (b_1 \oplus b_2) \cdot \Delta \\ &= k_{w_3} \oplus (b_1 \oplus b_2) \cdot \Delta, \end{aligned}$$

where $k_{w_3} = k_{w_1} \oplus k_{w_2}$. This shows that the output label X_{w_3} corresponds to the correct value without additional encryption [214, 78].

The security of the Free-XOR technique relies on the secrecy of the global difference Δ. If an adversary learns Δ, they can distinguish between the labels for 0 and 1, compromising the security of the protocol [78, 146]. Therefore, Δ must be kept confidential and chosen randomly. The Free-XOR optimization is proven secure under the random oracle model, assuming that the underlying cryptographic primitives are secure [214].

The Free-XOR technique reduces both the computational and communication overhead of the garbling process. The evaluation of XOR gates becomes a simple XOR operation on the wire labels, streamlining the evaluation phase of the protocol. However, this technique requires the random oracle model for its security proof, which is a stronger assumption than standard

Known input	Other input	Garbled Input	XOR output	Garbled XOR output	AND Output	Garbled AND output
0	0	A_i^0, B_i^0	0	$w_c^0 = A_i^0 \oplus B_i^0$	0	$E_{B_i^0}(w_c^0)$
	1	$A_i^0, B_i^0 \oplus R$	1	$w_c^0 \oplus R$		
1	0	$A_i^0 \oplus R, B_i^0$	1	$w_c^0 \oplus R$	0	
	1	$A_i^0 \oplus R, B_i^0 \oplus R$	0	w_c^0	1	$E_{B_i^0 \oplus R}(w_c^0 \oplus R)$

(a)

Figure 4.4: Garbled gates look-up table with half-gate optimization.

models [229]. Additionally, all wire labels must share the same global difference Δ, which may impose constraints on certain circuit constructions [35].

The Free-XOR optimization significantly enhances the efficiency of GC by leveraging the properties of the XOR operation, making SFE more practical for complex computations [428]. It has been widely adopted in various GC-based privacy-preserving applications, including secure MPC and privacy-preserving ML [318].

4.3.2 Half-Gates Optimization

The Half-Gates optimization, introduced by Zahur et al. [428], is a significant improvement over traditional garbled circuit constructions, particularly in reducing the number of ciphertexts required for AND gates. While the Free-XOR technique eliminates the cost of XOR gates, AND gates still require encryption operations, making them the primary bottleneck in GC-based protocols. The Half-Gates technique addresses this challenge by reducing the number of ciphertexts per AND gate from four to two, thereby improving both the communication and computation efficiency of the protocol [35].

In a standard garbled circuit, each AND gate requires four ciphertexts, as each row in the truth table must be encrypted separately [299]. The Half-Gates optimization leverages a novel encoding technique that allows the evaluator to compute the output using only two ciphertexts, without compromising security. This is achieved by decomposing the AND gate computation into two separate components, each of which is garbled independently and efficiently [214].

Figure 4.4 illustrates the garbled gates look-up table with half-gate optimization. The fundamental idea behind Half-Gates is to split the evaluation

of an AND gate G into two intermediate computations, ensuring that the evaluator can derive the correct output using only two ciphertexts. Let A and B be the two input wires to the AND gate, and let C be the output wire. The wire labels are denoted as:

$$\begin{aligned} X_A^0 &= k_A, \quad X_A^1 = k_A \oplus \Delta, \\ X_B^0 &= k_B, \quad X_B^1 = k_B \oplus \Delta, \\ X_C^0 &= k_C, \quad X_C^1 = k_C \oplus \Delta. \end{aligned}$$

In the traditional approach, the evaluator must decrypt four ciphertexts to determine the output label. With Half-Gates, the AND gate is decomposed into two sub-gates, each requiring only one ciphertext. The output label for the AND operation is computed as:

$$X_C = \mathcal{E}(X_A, X_B),$$

where $\mathcal{E}$ represents the encryption operation [428].

To construct the garbled table efficiently, the garbler generates two masking values:

$$\lambda_A = H(k_A) \oplus (X_C \oplus g_A \cdot \Delta),$$
$$\lambda_B = H(k_B) \oplus (X_C \oplus g_B \cdot \Delta),$$

where $H(\cdot)$ is a cryptographic hash function, and g_A, g_B are deterministic functions ensuring that the evaluator selects the correct masked value during decryption.

During evaluation, the evaluator receives the input labels X_A and X_B and computes the output label using:

$$X_C = \lambda_A \oplus H(X_B) \text{ if } g_A = 1,$$

$$X_C = \lambda_B \oplus H(X_A) \text{ if } g_B = 1.$$

This ensures that only two ciphertexts are needed, significantly reducing the garbled circuit size and improving efficiency [428].

The Half-Gates optimization improves upon prior methods by minimizing the number of encryptions needed per gate. It reduces the communication complexity of GC by approximately 25% compared to traditional garbling techniques [362]. Additionally, it ensures that the evaluator only needs to process two ciphertexts per AND gate, leading to faster execution times.

Algorithm 4: Half-Gates Garbling

Input: Circuit C with gates G and wires W
Output: Garbled circuit $\widetilde{C}$ and decoding information

foreach *wire* $w \in W$ **do**
    Choose a random label k_w;
    Set $X_w^0 = k_w$;
    Set $X_w^1 = k_w \oplus \Delta$;

foreach *AND gate* G *with inputs* A, B *and output* C **do**
    Compute $\lambda_A = H(k_A) \oplus (X_C \oplus g_A \cdot \Delta)$;
    Compute $\lambda_B = H(k_B) \oplus (X_C \oplus g_B \cdot \Delta)$;
    Store λ_A, λ_B as the garbled table;

Output the garbled circuit $\widetilde{C}$ and decoding information;

Security is preserved because the garbler never reveals more than necessary to the evaluator. The use of a cryptographic hash function $H(\cdot)$ ensures that no information about the input values is leaked beyond what is required to compute the output [428]. This makes the Half-Gates technique particularly effective for large-scale secure computation applications, such as privacy-preserving ML and secure MPC [318].

Overall, the Half-Gates optimization represents a significant advancement in GC, reducing computational overhead while maintaining strong security guarantees. When combined with Free-XOR, it provides a highly efficient framework for SFE, making GC more practical for real-world applications [428, 214].

4.3.3 Row Reduction Optimization

The Row Reduction optimization, introduced by Naor et al. [276], is a fundamental improvement in GC aimed at reducing the number of ciphertexts that need to be transmitted and stored. In a conventional garbled circuit, each gate is associated with four ciphertexts, corresponding to the four possible input combinations. The Row Reduction technique exploits the structure of garbled truth tables to eliminate one of these ciphertexts, reducing communication complexity and improving efficiency [35].

The primary motivation behind Row Reduction is to reduce the size of

garbled tables without compromising security. The technique works by fixing one of the four ciphertexts in each gate's truth table to a known constant, eliminating the need to transmit it. This effectively reduces the garbled table size by 25%, leading to lower memory requirements and faster evaluation [299].

Garbling Process with Row Reduction

Let G be a Boolean gate with two input wires A and B, and an output wire C. Each wire is associated with garbled labels:

$$X_A^0 = k_A, \quad X_A^1 = k_A \oplus \Delta,$$

$$X_B^0 = k_B, \quad X_B^1 = k_B \oplus \Delta,$$

$$X_C^0 = k_C, \quad X_C^1 = k_C \oplus \Delta.$$

where Δ is the global fixed offset used for encoding wire labels [35].

For a standard AND gate, the garbler constructs a truth table by encrypting the output labels for each possible input pair:

$$T_{ab} = \mathcal{E}(X_A^a, X_B^b, X_C^{G(a,b)}),$$

for $a, b \in \{0, 1\}$. This results in four ciphertexts, each encrypting the corresponding output label X_C.

In Row Reduction, the garbler sets one ciphertext, typically the first one (corresponding to input $(0, 0)$), to an all-zero string:

$$T_{00} = 0.$$

This eliminates the need to transmit this value, as the evaluator can derive it implicitly during evaluation. The remaining three ciphertexts are computed as:

$$T_{01} = \mathcal{E}(X_A^0, X_B^1, X_C^{G(0,1)}),$$

$$T_{10} = \mathcal{E}(X_A^1, X_B^0, X_C^{G(1,0)}),$$

$$T_{11} = \mathcal{E}(X_A^1, X_B^1, X_C^{G(1,1)}).$$

Algorithm 5: Row Reduction Garbling

Input: Circuit C with gates G and wires W
Output: Garbled circuit $\widetilde{C}$ and decoding information

foreach *wire* $w \in W$ **do**
  Choose a random label k_w;
  Set $X_w^0 = k_w$;
  Set $X_w^1 = k_w \oplus \Delta$;

foreach *AND gate* G *with inputs* A, B *and output* C **do**
  Set $T_{00} = 0$;
  Compute $T_{01} = \mathcal{E}(X_A^0, X_B^1, X_C^{G(0,1)})$;
  Compute $T_{10} = \mathcal{E}(X_A^1, X_B^0, X_C^{G(1,0)})$;
  Compute $T_{11} = \mathcal{E}(X_A^1, X_B^1, X_C^{G(1,1)})$;
  Store T_{01}, T_{10}, T_{11} as the garbled table;

Output the garbled circuit $\widetilde{C}$ and decoding information;

Evaluation Process with Row Reduction

During evaluation, the evaluator receives garbled input labels X_A and X_B. To determine the correct ciphertext to decrypt, they use a selection function $\sigma(A, B)$, which maps input labels to their corresponding ciphertext index:

$$\sigma(A, B) = 2A + B.$$

The evaluator decrypts the corresponding entry in the garbled table:

$$X_C = \mathcal{D}(X_A, X_B, T_{\sigma(A,B)}).$$

For the case where $(A, B) = (0, 0)$, the evaluator directly assigns:

$$X_C = X_C^0.$$

since T_{00} is set to zero and does not require decryption [276].

Benefits and Applications

The Row Reduction optimization reduces the total garbled circuit size by approximately 25%, improving the efficiency of SFE [276]. This technique is

particularly beneficial in large-scale secure computation applications, where minimizing communication overhead is crucial [299].

Security is preserved because the missing ciphertext is deterministic and does not reveal any additional information about the underlying data. The cryptographic properties of the encryption function $\mathcal{E}$ ensure that no partial information about the output wire labels can be inferred from the reduced garbled table [35].

Row Reduction is commonly used in conjunction with other optimizations, such as Free-XOR and Half-Gates, to achieve further efficiency gains. When combined, these techniques make GC more practical for applications such as privacy-preserving ML, secure cloud computing, and MPC [428, 318].

4.4 Oblivious Transfer

Oblivious Transfer (OT) is a fundamental cryptographic primitive that enables a sender to transmit multiple messages to a receiver while ensuring that the receiver learns only one of the messages without revealing which one was chosen. Simultaneously, the sender remains oblivious to the receiver's choice. This property makes OT a critical building block for secure computation protocols, particularly in applications such as secure MPC and GC [203]. By ensuring the confidentiality of inputs, OT minimizes information leakage during SFE.

The most basic form of OT, known as 1-out-of-2 OT, involves two parties: a sender P_1, who holds two secret messages m_0 and m_1, and a receiver P_2, who has a selection bit $i \in \{0, 1\}$. After executing the OT protocol, the receiver learns only m_i, while the sender does not gain any information about i [301]. More generally, an 1-out-of-N OT allows the receiver to choose one message from N possible options [275]. The security of OT is defined by two fundamental properties: receiver privacy, which ensures that the sender cannot determine which message the receiver has chosen, and sender privacy, which guarantees that the receiver learns only the selected message and gains no information about the unselected message [51].

A standard OT scheme consists of three probabilistic polynomial-time algorithms: $\mathsf{OT.Setup}(1^\lambda)$, $\mathsf{OT.Send}(\mathrm{pk}, m_0, m_1)$, and $\mathsf{OT.Receive}(\mathrm{sk}, T, i)$. The setup algorithm generates the necessary cryptographic keys, where λ denotes the security parameter. The sender encrypts the messages m_0 and m_1 using the public key pk and sends the ciphertext T to the receiver. Finally, the

receiver, using their private key sk, selects the desired message m_i without revealing i. These operations ensure that the receiver obtains exactly one message while maintaining privacy.

A commonly used implementation of 1-out-of-2 OT is based on public-key cryptography, often employing HE or Diffie-Hellman key exchange. The protocol begins with the sender generating a public-private key pair (pk, sk). The receiver selects a bit $i \in \{0, 1\}$ and constructs an encryption request using pk, ensuring that the sender cannot determine the chosen bit. The sender then encrypts m_0 and m_1 under two different encryptions and sends them to the receiver. The receiver decrypts only the ciphertext corresponding to their selection bit i and retrieves m_i, while learning nothing about m_{1-i} [275].

Algorithm 6: 1-out-of-2 OT Protocol

1: **Input:** Sender P_1 has messages m_0, m_1; receiver P_2 has selection bit i
2: **Output:** Receiver learns m_i; sender learns nothing about i
3: Sender generates a key pair (pk, sk)
4: Receiver selects a random value r and computes $v = \mathsf{Enc}(\text{pk}, r)$
5: Receiver sends v to sender
6: Sender encrypts:

$$c_0 = \mathsf{Enc}(v, m_0), \quad c_1 = \mathsf{Enc}(v \oplus \text{pk}, m_1)$$

7: Sender sends c_0, c_1 to receiver
8: Receiver decrypts:

$$m_i = \mathsf{Dec}(c_i)$$

9: **return** m_i

While OT is a powerful tool for secure computation, its reliance on public-key cryptography introduces computational overhead, making it inefficient for large-scale applications. To address this, OT extension techniques have been developed to enable multiple OT instances to be executed with minimal cryptographic cost [188]. Instead of performing full-fledged OT for each message transfer, a small number of base OT instances are used to generate correlated keys for subsequent OT executions. This approach significantly reduces computational costs, making large-scale secure computation feasible [211].

A further optimization, known as Silent OT, enhances efficiency by reducing the communication cost while maintaining security. Silent OT replaces explicit OT queries with precomputed pseudo-random correlations, allowing for fast key derivation [437]. This technique results in near-constant overhead per OT instance, making it particularly advantageous for large-scale cryptographic applications. Additionally, OT has been integrated into hardware-based cryptographic accelerators to improve execution speed and reduce latency in secure MPC frameworks [308].

OT finds applications in numerous privacy-preserving protocols. In GC, OT is used to transfer garbled input labels securely, ensuring that each party receives only the necessary information to evaluate the circuit without learning additional secrets [130]. In MPC, OT ensures that private inputs remain confidential while enabling joint computations. Beyond secure computation, OT is employed in password-based authentication, secure voting systems, private information retrieval, and digital rights management.

The security of OT protocols is based on standard cryptographic hardness assumptions such as the Decisional Diffie-Hellman assumption and the Learning with Errors assumption [275]. These assumptions ensure that a computationally bounded adversary cannot break receiver or sender privacy. Several formal security proofs have demonstrated that under these assumptions, OT remains secure against various adversarial models, including semi-honest and malicious attackers [51].

Overall, OT serves as an essential building block for SFE, ensuring that private data can be transferred securely between parties without revealing unnecessary information. The continual development of optimizations such as OT extensions and Silent OT contributes to making OT-based secure computation protocols more efficient and scalable for practical applications.

4.5 Adversary Models in Secure Computation

The security of cryptographic protocols, including secure MPC and hardware security techniques, is evaluated against different adversarial models. The choice of an adversary model significantly influences protocol design, proof of security, and countermeasures against potential attacks. Throughout the literature, adversary models are often categorized into two major

pairs: Passive (also known as Honest-but-Curious (HbC)) and Active (often equated with Malicious). These models define the capabilities of an attacker, dictating the security assumptions of cryptographic protocols and hardware countermeasures.

4.5.1 Passive and Honest-but-Curious Adversary Model

A passive adversary, also commonly referred to as an HbC adversary, follows the protocol honestly but attempts to infer additional information from observed communications and computations. The terms passive and HbC are often used interchangeably in the literature, particularly in the context of semi-honest secure computation protocols, where privacy is preserved as long as participants do not deviate from their prescribed execution [132, 36].

The semi-honest adversarial model is widely studied in the MPC community, particularly in GC [419, 34] and hybrid cryptographic protocols [267, 362]. Protocols such as SecureML [267] and CryptFlow2 [310] ensure that intermediate computation remains hidden, providing privacy against HbC adversaries.

From a hardware security perspective, passive adversaries are often considered in SCA, where an attacker observes unintended leakage such as power consumption, EM emissions, or timing variations [369, 295, 370]. These adversaries do not interfere with execution but analyze information leakage to recover cryptographic keys [56, 336]. Techniques such as DL-based side-channel analysis have further strengthened passive attack capabilities [169, 164].

In certain settings, passive adversaries may be assumed to analyze only publicly available data but not attempt cryptographic breaks. For example, privacy-preserving DL frameworks such as DELPHI [365] and CHET [90] operate under this assumption, ensuring no unintended data leakage in collaborative ML.

4.5.2 Active and Malicious Adversary Model

An active adversary, often equated with a malicious adversary, is significantly more powerful than a passive adversary. The terms active and malicious are sometimes used interchangeably in cryptographic literature, although malicious adversaries are generally considered a strict superset of active adversaries, meaning they can perform all actions of an active adversary with even

fewer constraints [299, 401]. Unlike semi-honest adversaries, active adversaries can arbitrarily deviate from the protocol, inject faults, forge messages, and manipulate intermediate computations.

In MPC, protocols such as Secure Protocols with Dishonest Majority (SPDZ) [231] and cut-and-choose-based GC constructions [226] provide protection against active adversaries. These protocols employ message authentication codes (MACs), zero-knowledge proofs (ZKPs), and consistency checks to prevent malicious tampering. Malicious adversaries are also a major threat in FHE-based secure computations, where attackers might attempt to manipulate encrypted data [119, 194].

In hardware security, active adversaries are responsible for sophisticated attacks such as fault injection techniques [54], including laser fault injection (LFI) [431], electromagnetic fault injection (EMFI) [54], and clock glitching [370]. These attacks attempt to alter execution flow and extract sensitive data through induced faults.

In practical secure computation settings, adversaries classified as malicious require more computationally expensive countermeasures. Protocols such as JustGarble [187], EMP-toolkit [246], and Chameleon [318] address malicious adversaries by incorporating cryptographic commitments, ZKPs, and redundant computations to detect and prevent tampering. Hybrid secure computation frameworks such as F1 [332] and Delphi [365] also employ additional security layers to mitigate risks from active attackers.

From a hardware perspective, malicious adversaries can perform a combination of passive and active attacks, such as power analysis combined with fault injection [368]. Advanced ML-assisted attacks can further amplify these threats, enabling attackers to recover cryptographic keys by leveraging correlations between power traces and computational operations [166, 204].

4.6 Side-Channel Attacks: Leakage Sources and Analysis

Secure computation protocols, including GC, OT, and Secure MPC, rely on strong cryptographic assumptions to ensure data privacy and computation security. However, real-world implementations of these protocols are often susceptible to *SCA*, where an adversary exploits unintended physical leakage to infer sensitive information [210, 29, 342]. Unlike conventional crypt-

analysis, which targets algorithmic weaknesses, SCAs leverage the physical characteristics of hardware implementations, making them a potent threat against secure computing devices.

4.6.1 Side-Channel Leakage: Sources and Classification

Side-channel leakage arises from variations in the physical behavior of cryptographic implementations, which are influenced by internal data processing. The most commonly studied side-channel sources include *power consumption*, *EM emissions*, *execution timing variations*, and *photonic emissions* [112, 209, 338]. Each of these sources provides adversaries with observable patterns that can be correlated with secret information.

Power Consumption Leakage

The power consumption of a cryptographic device is not uniform across operations. When performing computations, modern processors and cryptographic accelerators exhibit variations in power usage, which correlate with the processed data and executed instructions [210, 247]. This leakage is modeled using *Hamming weight (HW)* or *Hamming distance (HD)* power models.

For a processor executing an operation involving bit-wise data manipulation, the power consumption at time t is expressed as:

$$P(t) = P_{\text{static}} + \alpha V_{\text{dd}}^2 f_{\text{clk}} + \sum_{i=1}^{n} C_i V_{\text{dd}}^2 D_i(t) \tag{4.1}$$

where P_{static} represents the static power consumption, α is the switching factor, V_{dd} is the supply voltage, f_{clk} is the clock frequency, C_i is the load capacitance of the i-th switching node, and $D_i(t)$ represents the HW or HD of the transitioning bits [256].

The *Hamming weight model* assumes that power consumption depends on the number of '1' bits in a processed value:

$$P_{\text{HW}}(X) \propto \sum_{i=0}^{n} X_i \tag{4.2}$$

where X_i represents the bit values of the data word [369].

The *Hamming distance model*, in contrast, considers the number of bits flipped between consecutive states of a register:

$$P_{\text{HD}}(X, Y) \propto \sum_{i=0}^{n} (X_i \oplus Y_i) \tag{4.3}$$

where X and Y represent two successive states of the register [247]. These models form the foundation of *power analysis attacks*, where adversaries collect power traces and apply statistical techniques to recover cryptographic keys or other sensitive data.

Electromagnetic Emission Leakage

EM emissions arise from transient current variations within a microprocessor's power rails and data buses. These emissions can be captured using specialized probes and correlated with internal computations [112]. The EM field strength at a given observation point is modeled using *Maxwell's equations*, which relate the electric and magnetic fields to the current density in the chip:

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}, \quad \nabla \times \mathbf{H} = \mathbf{J} + \frac{\partial \mathbf{D}}{\partial t} \tag{4.4}$$

where $\mathbf{E}$ and $\mathbf{H}$ are the electric and magnetic field vectors, respectively, $\mathbf{B}$ is the magnetic flux density, $\mathbf{D}$ is the electric displacement field, and $\mathbf{J}$ is the current density [284].

Since cryptographic operations involve data-dependent current flow through the processor's power grid, these fields exhibit measurable variations correlated with the processed values. Attacks exploiting this phenomenon, such as *Template EM Attacks*, rely on recording multiple traces and statistically linking them to known cryptographic operations [124].

The strength of the EM emissions depends on the distance from the device and the loop area of the current paths inside the chip, modeled as:

$$E_{\text{EM}}(t) \propto \frac{I(t)}{r^2} \tag{4.5}$$

where $I(t)$ is the instantaneous current drawn by the circuit, and r is the observation distance [93].

Execution Timing Leakage

Timing attacks exploit variations in execution latency caused by conditional branching, memory access, or cryptographic table lookups. Many cryptographic implementations contain operations with data-dependent execution paths, making them susceptible to timing-based leakage [209].

The execution time of a function $f(x)$ can be approximated as:

$$T(x) = T_{\text{base}} + \sum_{i=1}^{m} \delta_i \cdot f_i(x) \tag{4.6}$$

where T_{base} is the base execution time, δ_i represents the additional latency introduced by a data-dependent operation $f_i(x)$, and m denotes the number of such operations [57].

To mitigate timing attacks, cryptographic implementations enforce *constant-time execution*, ensuring that all operations execute in a uniform number of cycles regardless of input:

$$T(x) = c, \quad \forall x \tag{4.7}$$

where c is a constant execution time for all inputs [42].

Besides the commonly studied leakage vectors such as timing, power, and EM emissions, there exist several other physical side-channel leakage sources. These include photonic emissions, where CMOS transistors emit light during switching [337]; acoustic leakage, where computational operations induce audible or ultrasonic vibrations [118]; thermal leakage through observable heat patterns [152]; current variations captured through USB or power lines [380]; capacitive coupling and crosstalk effects between neighboring wires [442]; close-range magnetic flux leakage using fluxgate magnetometers [443]; visible state leakage through peripheral indicators such as LED blinking [242]; and radio frequency (RF) emissions exploitable via long-range eavesdropping [392]. While these channels have demonstrated their potential in prior work, they are beyond the scope of this dissertation and are mentioned here for completeness.

4.7 Side-Channel Attacks and Evaluation

Side-channel leakages such as power consumption, EM emissions, timing variations, and photonic emissions can be exploited to extract sensitive cryptographic information. These physical emanations provide an attacker with indirect access to secret values processed by a secure implementation. The attacks leveraging these leakages are referred to as SCA, and they have been widely studied due to their practical effectiveness in breaking cryptographic protocols.

SCA generally follow a structured process where an attacker collects physical leakage signals while the target device executes cryptographic operations. The collected data is then analyzed using statistical or machine-learning-based methods to infer secret information. SCAs range from simple power analysis (SPA) attacks, which rely on visually inspecting power traces, to more advanced techniques such as Differential Power Analysis (DPA) and Template Attacks, which leverage statistical models and ML.

4.7.1 Differential Power Analysis

DPA is a statistical technique for extracting cryptographic keys from power consumption traces [210]. Unlike SPA, which relies on visual inspection of power waveforms, DPA uses mathematical techniques to analyze multiple traces and identify small correlations between power consumption and cryptographic computations.

The fundamental principle behind DPA is that some intermediate values in cryptographic algorithms depend on the secret key. By modeling how these intermediate values influence power consumption and comparing predictions with real measurements, an attacker can recover secret keys.

The DPA attack follows a structured approach:

Algorithm 7: Differential Power Analysis Attack

Input: Set of power traces $\{P_j(t)\}_{j=1}^{N}$, corresponding plaintexts $\{X_j\}_{j=1}^{N}$

Output: Recovery of key candidate k^*

foreach *possible key guess* k **do**

Compute hypothetical intermediate values $V_j = f(X_j, k)$;

Apply power model to estimate power consumption $M_j(k)$ (e.g., HW model);

Partition traces into two sets based on the predicted power value;

Compute the mean traces for each set:;

$$P^{(0)}(t) = \frac{1}{N} \sum_{j \in S_0} P_j(t), \quad P^{(1)}(t) = \frac{1}{N} \sum_{j \in S_1} P_j(t)$$

Compute differential trace:;

$$\Delta P(t) = P^{(0)}(t) - P^{(1)}(t)$$

The correct key guess k^* corresponds to the maximum peak in $\Delta P(t)$;

return k^*;

DPA is effective against cryptographic implementations that do not exhibit clear patterns in power traces. By averaging power traces across multiple executions, DPA mitigates noise and isolates small variations linked to secret key computations.

Mathematically, the success of a DPA attack depends on the correlation between power consumption and the computed values:

$$\Delta P(t) = \frac{1}{N} \sum_{j=1}^{N} P_j^{(0)}(t) - \frac{1}{N} \sum_{j=1}^{N} P_j^{(1)}(t) \tag{4.8}$$

A strong peak in $\Delta P(t)$ indicates the correct key guess. Countermeasures such as masking and randomized execution can be used to thwart DPA attacks by reducing the correlation between power consumption and intermediate values [284].

4.8 Side-Channel Evaluation Techniques

Secure computation implementations must undergo rigorous side-channel evaluations to ensure resilience against potential attacks. Side-channel leakages, such as power consumption, EM emissions, timing variations, and photonic emissions, serve as valuable signals for adversaries attempting to recover secret data. The effectiveness of countermeasures such as masking, hiding, and obfuscation is contingent upon their ability to mitigate these leakages. To validate the security of cryptographic implementations, various statistical and machine-learning-based evaluation techniques have been developed. These methodologies analyze the correlation between observed leakage and secret data to determine the presence and severity of side-channel vulnerabilities.

The evaluation process typically begins by collecting power traces, EM signals, or timing measurements from a cryptographic device executing sensitive operations. The acquired traces are then subjected to statistical analysis to determine whether information about the secret key or intermediate states can be inferred. A variety of statistical tests, including Welch's t-test, Chi-Square (χ^2) analysis, and Mutual Information Analysis (MIA), provide different perspectives on leakage detectability. Additionally, more advanced techniques such as DCA employ ML to uncover subtle non-linear dependencies in leakage patterns.

4.8.1 Welch's t-Test for Leakage Detection

One of the most widely used evaluation methods for detecting leakage in cryptographic implementations is Welch's t-test [33]. This test quantifies differences in mean power consumption between two sets of traces: one with a fixed input and another with random inputs. A significant difference between these sets suggests the presence of data-dependent leakage.

Mathematically, Welch's t-test is defined as follows. Given two sets of leakage traces, one corresponding to a fixed input (X_f) and the other corresponding to a randomly varying input (X_r), the test statistic is computed as:

$$t = \frac{\mu_f - \mu_r}{\sqrt{\frac{\sigma_f^2}{n_f} + \frac{\sigma_r^2}{n_r}}} \tag{4.9}$$

where μ_f and μ_r represent the mean values of the two sets, σ_f^2 and σ_r^2 denote the variances, and n_f, n_r are the respective sample sizes. If the computed $|t|$ value exceeds a predefined threshold (typically 4.5 for a significance level of 99.99%), the implementation is deemed to exhibit leakage. This method is effective in detecting first-order leakage but may be inadequate for higher-order leakage, where multiple variables need to be combined before a distinguishable pattern emerges.

The following algorithm provides a systematic approach to applying the Welch's t-test for leakage detection.

Algorithm 8: Welch's t-Test for Side-Channel Leakage Detection

Input: Set of power traces X_f (fixed input), X_r (random input)
Output: Leakage detection result
Compute mean and variance for both distributions:

$$\mu_f = \frac{1}{n_f}\sum_{i=1}^{n_f} X_f^i, \quad \mu_r = \frac{1}{n_r}\sum_{i=1}^{n_r} X_r^i \tag{4.10}$$

$$\sigma_f^2 = \frac{1}{n_f - 1}\sum_{i=1}^{n_f}(X_f^i - \mu_f)^2, \quad \sigma_r^2 = \frac{1}{n_r - 1}\sum_{i=1}^{n_r}(X_r^i - \mu_r)^2 \tag{4.11}$$

Compute the t-statistic:

$$t = \frac{\mu_f - \mu_r}{\sqrt{\frac{\sigma_f^2}{n_f} + \frac{\sigma_r^2}{n_r}}} \tag{4.12}$$

Compare $|t|$ against the critical threshold (typically 4.5). If $|t| > 4.5$, report leakage detected.

This algorithm provides an efficient and systematic method for analyzing whether an implementation leaks information through power consumption traces.

Several additional side-channel evaluation techniques exist, including:

- **Correlation Power Analysis (CPA) Evaluation** [56] uses Pearson's correlation coefficient to determine the statistical relationship between predicted power consumption and observed leakage traces.
- **Kolmogorov-Smirnov (K-S) Test** [124] is a non-parametric test

that compares the distributions of two sets of leakage traces to detect statistical differences.

- **Principal Component Analysis (PCA)** [249] is a dimensionality reduction technique used to identify dominant leakage components in high-dimensional side-channel traces.
- **Differential Cluster Analysis (DCA)** [165] applies ML-based clustering techniques to side-channel traces to differentiate key-dependent operations.
- **Linear Regression Analysis** [341] models side-channel leakage as a function of secret-dependent variables to determine statistical dependencies.

While these methods provide valuable insights into the resilience of cryptographic implementations, they were not utilized in the evaluation presented in this dissertation.

4.9 Fault Injection Attacks

Fault injection attacks (FIA) are a class of active physical attacks that exploit vulnerabilities in hardware and software implementations by deliberately inducing errors during execution. Unlike SCA, which passively observe leakages from cryptographic operations, FIA actively modify the computation process to force incorrect outputs, extract sensitive data, or bypass security checks [28, 338]. These attacks have become a significant threat to secure computation protocols such as MPC, GC, and OT, particularly in hardware implementations.

The fundamental principle of fault injection is to introduce controlled disturbances in a computation such that specific exploitable conditions arise. Given an input x and a function $f(x)$, an attacker aims to introduce a fault δ such that the output is altered from the expected value $y = f(x)$ to a faulty result $y' = f(x + \delta)$. The attacker then exploits the differences between y and y' to infer confidential information [354].

4.9.1 Mathematical Model of Fault Injection

To formally describe a fault injection attack, consider a cryptographic function $f(x)$ operating on an input x. The execution of this function is affected by an injected fault δ, which modifies the input, intermediate computation, or output as follows:

$$y' = f(x + \delta), \tag{4.13}$$

where δ represents the induced error. Depending on the attack strategy, δ may be introduced at various levels:

1. **Input perturbation:** The attacker modifies the initial state of the system by altering input values or cryptographic keys.
2. **Intermediate state corruption:** Errors are injected during execution, affecting registers, logic gates, or memory contents.
3. **Output manipulation:** The attacker attempts to manipulate the final output directly by introducing transient or permanent faults.

In differential fault analysis (DFA), an attacker compares the faulty output y' with the correct output y to derive relationships between internal states [46]. Given multiple faulted computations $f(x + \delta_i)$, statistical analysis can recover secret keys by solving a system of equations:

$$f(x) \oplus f(x + \delta_i) = \Delta y_i. \tag{4.14}$$

where $\oplus$ denotes the bitwise XOR operation, and Δy_i is the observed output difference caused by fault δ_i.

4.9.2 Types of Fault Injection Attacks

Fault injection techniques vary based on the physical method used to introduce errors.

Laser Fault Injection

LFI employs focused laser beams to induce errors in semiconductor devices [338]. By directing a laser pulse at specific regions of a chip, attackers can selectively introduce bit flips or induce timing violations.

The energy delivered by the laser is given by:

$$E = P \cdot t, \tag{4.15}$$

where P is the laser power and t is the exposure time. If E exceeds a critical threshold, charge carriers are generated in semiconductor junctions, causing transient or permanent disruptions.

In addition to laser fault injection, which is the primary focus of this dissertation, there exist several other fault injection techniques that can compromise hardware integrity. These include clock glitching, where precise timing violations are introduced into the clock signal to disrupt normal instruction execution [28]; voltage fault injection, where deliberate fluctuations in the power supply induce computation errors or system instability [358]; and electromagnetic fault injection (EMFI), which uses high-intensity electromagnetic pulses to corrupt logic operations or memory states [347]. While these approaches are effective and relevant to hardware-level attack strategies, a detailed analysis of them is beyond the scope of this dissertation and is mentioned here for completeness.

4.9.3 Fault Injection Methods

FIA pose a severe threat to cryptographic and secure computation systems by deliberately introducing errors into hardware or software implementations. These faults, when successfully injected, can leak sensitive information or cause unexpected behavior, compromising the integrity and confidentiality of the computation. Faults can be broadly classified into transient faults, permanent faults, and bit-flipping faults, each with different implications for secure computation models. These faults occur due to environmental influences, intentional adversarial interventions, or defects in hardware fabrication.

Bit-Flip Faults

Bit-flip faults are one of the most common fault injection methods, where an adversary induces single or multiple bit inversions within a computation. A bit-flip occurs when a stored or processed value of $b \in \{0, 1\}$ is flipped from b to $\bar{b} = 1 - b$. Mathematically, a bit-flip attack can be described as:

$$b^* = b \oplus 1 \tag{4.16}$$

Algorithm 9: Bit-Flip Fault Injection on Secure Computation

Require: Secure function $f(x)$, input x, fault model $\mathcal{F}$
Ensure: Faulted output y^*
1: Evaluate $y = f(x)$ under normal conditions
2: Apply bit-flip fault: $b^* = b \oplus 1$
3: Inject modified value into computation: $x^* = x \oplus \mathcal{F}$
4: Recompute output: $y^* = f(x^*)$
5: Return y^* and compare with y

where b^* represents the corrupted bit value after a fault injection event. In the context of secure computation, bit-flip faults can affect *cipher execution, key scheduling, and intermediate values* in cryptographic algorithms such as AES and RSA [210, 28]. Bit-flip faults are particularly dangerous in *GC* and *MPC* because a single bit modification can propagate through the circuit, altering the final result while remaining undetected.

An attacker can induce bit-flips through various methods, including voltage glitching, clock manipulation, and EMFI [358, 347]. These attacks exploit physical vulnerabilities in integrated circuits (IC), where a sudden change in environmental conditions leads to the corruption of specific data registers.

The algorithm above outlines the process of injecting a *bit-flip fault* into a secure computation model. By carefully choosing when and where to inject the fault, an attacker can manipulate specific outputs, allowing for cryptographic key recovery or function leakage.

Transient Faults

Transient faults, also known as *soft errors*, occur when an external disturbance temporarily alters the state of a digital circuit without causing permanent damage. These faults are often caused by *cosmic radiation, EM interference, or power supply fluctuations* [28, 338]. Unlike bit-flip faults, which are typically controlled by an adversary, transient faults can occur naturally or be induced by external attackers using *laser pulses* or *EMFI*.

The mathematical representation of a transient fault in secure computation is given by:

$$y^* = f(x) + \epsilon \tag{4.17}$$

Algorithm 10: Stuck-At Fault Simulation

Require: Circuit inputs x, stuck-at fault location i
Ensure: Faulted output y^*
1: Compute normal execution: $y = f(x)$
2: Inject stuck-at fault: $x_i^* = s_i$
3: Evaluate new output: $y^* = f(x^*)$
4: Compare y^* with y

where ϵ represents an *error function* that models transient disturbances. If ϵ is predictable, attackers can exploit it to reveal secret information by *DFA* [46].

Transient faults are particularly problematic in cryptographic protocols because they can cause predictable *biases in key schedules or S-box computations*. Attackers can induce transient faults at precise moments during execution to extract *intermediate encryption values* [29].

Stuck-At Faults

Stuck-at faults (SAF) occur when a circuit node remains *permanently stuck* at a logical 0 or 1, preventing normal operation. These faults are common in hardware security evaluations, as they simulate *defects in physical circuits* or *deliberate hardware trojans* implanted during fabrication.

In mathematical terms, a stuck-at fault at position i in an n-bit vector can be represented as:

$$x_i^* = s_i, \quad \forall i \in [1, n] \tag{4.18}$$

where $s_i \in \{0, 1\}$ is a fixed stuck-at value, independent of normal computation.

SAF are *highly effective against cryptographic hardware* since they can alter critical registers, leading to leakage of *secret keys* or faulty computations in secure execution environments [28].

Permanent Faults

Permanent faults are persistent errors in a circuit caused by *hardware aging, fabrication defects, or laser-induced damage*. Unlike transient faults, these

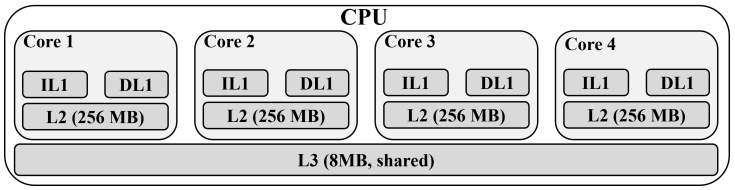


Figure 4.5: Intel core-i7 cache architecture [274].

faults do not self-correct and can be *exploited repeatedly* by attackers [28, 358].

$$\forall t > t_0, \quad x^*(t) = x^*(t_0) \tag{4.19}$$

where t_0 is the moment at which the permanent fault is introduced.

4.10 Cache Architecture

Modern computer architectures rely heavily on cache memory to bridge the speed gap between the Central processing unit (CPU) and main memory. Cache memory is a smaller, high-speed storage unit placed between the CPU and dynamic random-access memory (DRAM) to store frequently accessed instructions and data. The presence of caches significantly improves the performance of processors by reducing the number of direct accesses to main memory, which are substantially more time-consuming [162]. The design and organization of the cache hierarchy, including its size, associativity, replacement policies, and coherency mechanisms, are crucial in determining system efficiency and responsiveness [184].

4.10.1 Cache Hierarchy and Levels

Figure 4.5 presents the Intel core-i7 cache architecture. Most modern CPUs implement a multi-level cache hierarchy, typically consisting of three levels.

The first level of cache, also known as primary cache, is the fastest but smallest. It is typically divided into separate instruction and data caches. The instruction cache holds executable code, while the data cache stores frequently accessed operands [116].

The second level cache serves as an intermediate buffer between the first and last cache levels. While slightly slower than the first level, it has a

larger capacity, often dedicated per CPU core, and provides rapid access to frequently used data that has been evicted from the first level [270].

The third level cache is the last-level cache, which is the largest and slowest among the three but serves as a shared resource for all CPU cores, reducing redundant DRAM accesses and enhancing inter-core communication efficiency [237].

4.10.2 Cache Inclusion Policies

The organization of cache levels is determined by the inclusion policy, which dictates whether data present in a lower cache level must also exist in the higher levels. The three major inclusion policies are described as follows.

A cache hierarchy is inclusive when all data present in a lower-level cache is guaranteed to be in higher-level caches. This policy simplifies coherence enforcement because invalidating an entry at the last-level cache ensures that the corresponding data is also removed from lower caches [184]. However, inclusivity reduces the effective cache capacity due to redundancy, as multiple levels store the same data [116].

In an exclusive cache design, each data block is stored in only one cache level at a time. If a block is promoted from the second level to the first level, it is evicted from the second level, ensuring efficient utilization of cache space and reducing redundancy. This approach maximizes cache storage but adds complexity to the cache management policies [400].

A hybrid approach known as non-inclusive non-exclusive caches allows adaptive cache management strategies where the presence of data at different levels is not strictly enforced, optimizing performance based on workload characteristics [219].

4.10.3 Cache Coherence in Multi-Core Processors

With the widespread adoption of multi-core processors, cache coherence is a fundamental challenge that must be addressed to maintain consistency among multiple cache copies of shared data. Several cache coherence protocols are widely used.

The modified, shared, and invalid protocol is a simple state-based approach where a cache line can be in one of three states. Modified means exclusively owned and changed, shared means read-only access, and invalid

means not present. Write operations require broadcasting updates to other cores [184].

The modified, exclusive, shared, and invalid protocol extends the previous protocol by introducing an exclusive state that allows a core to modify a cache block without broadcasting until another core requests it. This optimization reduces unnecessary communication overhead [162].

The modified, owned, exclusive, shared, and invalid protocol further improves performance by allowing a cache block to be owned by a core while still permitting shared access. This reduces memory traffic when multiple cores require access to frequently used data [219].

Maintaining cache coherence is essential for preventing data inconsistencies that can arise due to simultaneous accesses from different cores, ensuring correctness in multi-threaded applications [270].

4.10.4 Cache Replacement and Eviction Policies

When a cache is full and a new data block needs to be loaded, an existing block must be evicted. Several cache replacement policies exist to determine which block is removed.

The least recently used policy evicts the block that has not been accessed for the longest time. This policy is widely used in CPUs and achieves good performance in workloads with temporal locality [141].

The first-in-first-out policy removes the oldest cache block regardless of how frequently it has been accessed. While simple to implement, this method can lead to suboptimal eviction decisions when old but frequently used data is removed [237].

The random replacement policy selects a block randomly for eviction. While unpredictable, this strategy prevents adversaries from exploiting deterministic eviction behaviors in cache-based attacks [423].

The least frequently used policy tracks access counts and removes the least accessed block over time. While theoretically effective, it can suffer from high overhead due to tracking access frequencies [438].

Different cache architectures and workloads benefit from different eviction strategies, and some modern processors employ adaptive policies that dynamically adjust based on application behavior [116].

4.10.5 Memory Access and Prefetching Mechanisms

The efficiency of memory access is a crucial factor in overall system performance. If requested data is not found in the cache, the CPU must fetch it from DRAM, incurring high latency.

To reduce memory access delays, modern processors employ prefetching mechanisms, which attempt to anticipate future memory accesses and load the relevant data into the cache in advance. Prefetching techniques include hardware and software approaches.

Hardware prefetching is implemented at the processor level, where techniques analyze access patterns and fetch data proactively. Examples include stride-based prefetching and adjacent-line prefetching [184].

Software prefetching relies on compilers inserting explicit prefetch instructions based on program behavior, improving cache utilization for specific workloads [438].

Efficient prefetching significantly enhances performance by reducing the number of cache misses and minimizing DRAM accesses [162].

4.11 Neural Networks: Foundations and Architectures

NN are a class of ML models inspired by the structure and function of biological neural systems [251, 325, 220]. They are composed of interconnected layers of artificial neurons, where each neuron processes input data and transmits an activation signal to subsequent layers. The foundation of NN lies in the weighted summation of inputs followed by an activation function, mathematically expressed as:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right), \tag{4.20}$$

where x_i represents the input features, w_i are the weights associated with each input, b is the bias term, and $f(\cdot)$ is the activation function that introduces non-linearity into the model [138].

4.11.1 Feedforward and Deep Neural Networks

A feedforward neural network (FNN) is the simplest form of NN where information flows strictly in one direction: from the input layer, through hidden layers, to the output layer [330, 159]. Unlike recurrent NN, FNNs do not have cycles or feedback connections, making them suitable for static input-output mappings.

The depth of an NN is defined by the number of hidden layers, distinguishing between shallow and deep networks. A deep neural network (DNN) consists of multiple hidden layers, enabling hierarchical feature extraction and representation learning [40, 220]. Each layer in a DNN transforms its input using a weight matrix W, a bias vector b, and a non-linear activation function $f(\cdot)$:

$$\mathbf{h}^{(l)} = f\left(W^{(l)}\mathbf{h}^{(l-1)} + b^{(l)}\right), \tag{4.21}$$

where $\mathbf{h}^{(l)}$ denotes the activations at layer l, and $\mathbf{h}^{(0)}$ corresponds to the input features.

4.11.2 Training Neural Networks: Backpropagation and Optimization

Training a NN involves adjusting the weights and biases to minimize the error between predicted and actual outputs. This optimization is achieved through backpropagation, an algorithm that computes the gradient of a loss function $\mathcal{L}$ with respect to model parameters using the chain rule of differentiation [330]. The weight update at each layer follows the gradient descent method:

$$W^{(l)} \leftarrow W^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial W^{(l)}}, \tag{4.22}$$

where η is the learning rate that controls the step size of updates. The loss function is typically defined as the mean squared error (MSE) for regression problems [159]:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2, \tag{4.23}$$

or the cross-entropy loss for classification tasks [138]:

$$\mathcal{L} = -\sum_{i=1}^{N} y_i \log \hat{y}_i. \tag{4.24}$$

Various optimizers, including stochastic gradient descent (SGD), Adam, and RMSprop, improve convergence by adapting the learning rate dynamically [206].

4.11.3 Activation Functions and Their Role

Activation functions introduce non-linearity into NN, enabling them to learn complex patterns [273]. Common activation functions include the sigmoid, hyperbolic tangent (tanh), and rectified linear unit (ReLU). The ReLU function, defined as:

$$f(x) = \max(0, x), \tag{4.25}$$

has become the standard choice due to its efficient gradient propagation and ability to mitigate the vanishing gradient problem [127].

4.11.4 Convolutional Neural Networks

A convolutional neural network (CNN) is a specialized architecture designed for spatial data such as images [221, 216]. Instead of using fully connected layers, CNNs leverage convolutional layers that apply learnable filters to local receptive fields:

$$h_{ij}^{(l)} = f\left(\sum_m \sum_n W_{mn}^{(l)} x_{(i+m)(j+n)} + b^{(l)}\right). \tag{4.26}$$

Pooling layers, such as max pooling and average pooling, reduce spatial dimensions and enhance translational invariance [220].

4.11.5 Neural Network Architectures and Applications

Beyond conventional architectures, various NN models are tailored for specific applications. Generative adversarial networks (GANs) generate realistic data distributions through adversarial training between a generator and a

discriminator [139]. Transformer-based architectures, such as BERT and GPT, achieve state-of-the-art performance in natural language processing by leveraging self-attention mechanisms [394].

NN have revolutionized fields such as computer vision, speech recognition, and autonomous systems, demonstrating superior performance in complex pattern recognition tasks [221, 220].

4.12 Clustering

Clustering is a fundamental task in unsupervised ML [190, 201, 159], where the objective is to group a set of data points into clusters based on their similarities. The similarity of data points is typically determined using a distance metric, which measures how close or far apart two points are in the feature space. Various clustering techniques exist, each leveraging different distance measures and optimization strategies to form coherent groupings. One of the most widely used clustering algorithms is the k-means algorithm, which aims to partition data points into k clusters by minimizing intra-cluster variance, often using the squared Euclidean distance as the similarity metric [159, 405, 351].

Different distance metrics have been proposed in the literature to measure the similarity between data points, including Euclidean, Manhattan, Mahalanobis, and cosine distances [190, 100, 105]. The Euclidean distance, given by

$$d(c_i, c_j) = \sqrt{\sum_{m=1}^{M} (c_{im} - c_{jm})^2}, \tag{4.27}$$

is the most commonly used metric in clustering, particularly in k-means, because of its straightforward interpretation and computational efficiency [159]. Another alternative, the Manhattan distance, defined as

$$d(c_i, c_j) = \sum_{m=1}^{M} |c_{im} - c_{jm}|, \tag{4.28}$$

measures the sum of absolute differences between coordinates, making it more robust to outliers in some cases [100]. The Mahalanobis distance accounts for correlations between variables and scales features accordingly,

providing a more adaptive similarity measure [92]. Finally, the cosine similarity metric measures the angle between two feature vectors, making it particularly useful for text and high-dimensional clustering problems [371].

Despite the availability of multiple distance metrics, our approach employs the squared Euclidean distance due to its effectiveness in minimizing intra-cluster variance, as demonstrated in numerous studies [351, 405]. The k-means algorithm specifically benefits from this distance measure, as it simplifies the optimization process and enables efficient centroid updates. While other distance metrics provide alternative perspectives on similarity, they have not been utilized in our method. Nonetheless, we acknowledge their significance in various clustering paradigms and highlight their applicability in specific domains such as high-dimensional data clustering, density-based clustering, and hierarchical methods [190, 105].

4.13 Chiplet-based Processing

4.13.1 Introduction to Chiplet Architectures

Chiplet-based architectures represent a transformative shift in semiconductor design, providing a modular alternative to monolithic IC. As transistor scaling nears its physical limitations, conventional monolithic designs struggle with escalating fabrication costs, yield challenges, and thermal constraints [50, 349]. Chiplets address these challenges by partitioning complex systems into smaller, functionally distinct dies that can be fabricated independently and then integrated within a Multi-Chip Module (MCM) or a 2.5D/3D packaging technology [309, 258, 89].

Chiplet-based architectures offer significant advantages in design flexibility, scalability, and performance optimization. Unlike traditional System-on-Chip (SoC) designs that require all components to be fabricated using the same technology node, chiplet-based processors enable heterogeneous integration, allowing different chiplets to be manufactured using process nodes best suited to their functionality [376, 324]. This approach reduces development costs and improves overall yield, as smaller dies are less prone to defects than large monolithic chips [107, 306].

Inter-chiplet communication is facilitated by high-speed interconnects such as silicon interposers, Through-Silicon Vias (TSVs), and embedded bridges, reducing data transfer latency while maintaining power efficiency [374,

241]. Standardized interconnect frameworks, including Universal Chiplet Interconnect Express (UCIe), AMD's Infinity Fabric, and Intel's EMIB (Embedded Multi-Die Interconnect Bridge), enable interoperability among chiplets from different vendors, fostering a new ecosystem for reusable and modular chiplet designs [399, 12, 122].

The widespread adoption of chiplet-based architectures extends beyond high-performance computing (HPC) and artificial intelligence (AI) accelerators to cloud computing, embedded systems, and mobile applications, where power efficiency and scalability are critical [123, 88]. Major semiconductor manufacturers, including AMD, Intel, and TSMC, have embraced chiplet-based architectures to develop next-generation processors with enhanced compute density and improved power-performance trade-offs [289, 293].

4.13.2 Security Threats in Chiplet-based Systems

While chiplet architectures enhance design modularity and scalability, they introduce new security concerns that must be addressed for secure deployment in critical applications. The reliance on third-party vendors for fabricating individual chiplets increases the risk of supply chain attacks, including hardware Trojans, malicious circuit modifications, and backdoors [326, 383, 168]. Since chiplets originate from multiple sources, ensuring trust in outsourced manufacturing is a major challenge [41, 425].

One of the primary concerns in chiplet-based architectures is inter-chiplet communication security. Unlike monolithic processors, where all components are integrated onto a single die, chiplet-based systems rely on high-speed interconnects for data exchange. This opens new attack vectors, including SCA, where adversaries can exploit EM emissions, power consumption variations, or timing anomalies to extract sensitive data [41, 346]. To mitigate these threats, secure serialization and encryption mechanisms such as authenticated encryption, bus obfuscation, and random delay injection are employed to protect data integrity and confidentiality across chiplets [145, 65].

FIA pose another severe threat, particularly for chiplet-based cryptographic accelerators and AI processors. Techniques such as LFI, EMFI, and power glitching can induce transient or permanent faults in chiplets, altering their behavior or leaking cryptographic keys [312, 433, 244]. Countermeasures such as error correction codes (ECC), redundant computation, and anomaly detection can enhance resilience against such attacks [304, 294].

Another critical security challenge is chiplet authentication and attesta-

tion. Given the heterogeneous nature of chiplets, robust mechanisms are needed to ensure that only trusted and verified components are integrated into a system. Hardware-based solutions, such as PUFs, cryptographic signatures, and blockchain-backed supply chain tracking, are being explored to authenticate chiplets before deployment [288, 5]. Runtime security monitoring frameworks utilizing hardware root-of-trust (RoT) can further enhance trust by continuously verifying chiplet integrity and detecting anomalies [272, 413].

4.13.3 Trusted Execution in Multi-Chip Modules

TEE are critical for ensuring the security and confidentiality of workloads in chiplet-based architectures. Secure computing frameworks, such as Intel SGX and ARM TrustZone, provide isolated execution environments that can be extended to multi-chip systems to prevent unauthorized data access and tampering [22, 314]. These secure enclaves enable privacy-preserving computation for cryptographic applications, AI inference, and financial transactions [244, 391].

Secure boot mechanisms play a crucial role in ensuring that only authenticated chiplets and firmware are executed within a system. Secure boot leverages cryptographic hash functions, digital signatures, and key management schemes to validate chiplet integrity at startup, preventing unauthorized firmware modifications or trojan-infected chiplets from being deployed [302, 348]. Additionally, dynamic reconfiguration of trusted chiplets allows real-time isolation and replacement of untrusted components, maintaining system security even in the presence of active threats [413, 311].

Confidential computing techniques such as FHE and secure MPC offer robust solutions for protecting data privacy in multi-chip environments. These cryptographic methods allow computations to be performed on encrypted data without revealing the underlying information, ensuring secure data processing across distributed chiplets [317, 149]. Such methods are particularly relevant for cloud-based AI and federated learning applications, where data confidentiality is paramount [327, 318].

Emerging trends in zero-trust hardware architectures focus on continuous authentication and verification of chiplets before and during execution. Unlike traditional security models that assume trust in pre-verified hardware, zero-trust architectures require continuous attestation, secure enclaves, and anomaly detection to mitigate risks in heterogeneous multi-chip systems [43, 294]. This approach is expected to be a cornerstone of future

HPC, AI accelerators, and cloud security solutions [7, 45].

Chapter 5

Chapter 5: Literature Review

5.1 Overview of Secure Computation Approaches

Secure computation allows multiple parties to compute a function over their private inputs while ensuring that no party learns anything beyond the intended output. This cryptographic concept is fundamental for preserving data privacy in various applications, including privacy-preserving ML, secure cloud computing, financial transactions, and biomedical data analysis. The growing need for privacy in modern computing has fueled extensive research into secure computation techniques that balance efficiency, security, and practicality.

Over the years, several secure computation protocols have been developed, each designed to offer different trade-offs in terms of security guarantees, computational efficiency, and communication overhead. Among the most prominent approaches are secure MPC, GC, and OT. These techniques serve as building blocks for privacy-preserving computation, and continuous optimizations have been proposed to make them more scalable and efficient.

Several cryptographic paradigms have been developed to enable secure computation. Among them, secure MPC has emerged as a powerful framework that allows multiple parties to compute joint functions while ensuring input privacy. MPC techniques can be broadly categorized into two main approaches: secret-sharing-based MPC and GC-based MPC, each optimized for different use cases and security settings. The following sections explore these two paradigms in detail, highlighting their advantages, challenges, and optimizations.

5.1.1 Secure MPC

Secure MPC allows a group of parties to jointly compute a function on their private inputs without revealing those inputs to one another. This technology eliminates the need for a trusted third party, making it an essential tool for privacy-preserving applications in settings where mutual trust cannot be assumed.

There are two primary types of MPC:

- Secret-sharing-based MPC: This approach splits the input into multiple random shares distributed among the parties. Each party holds a share and participates in computations over their respective shares, ensuring that no individual party can infer the original input. Only after the computation is complete can the final result be reconstructed from the shares.

- GC-based MPC: Instead of SS, this method transforms a function into an encrypted Boolean circuit. The circuit is then evaluated securely, ensuring that intermediate values remain hidden while only revealing the final output.

Both approaches have their strengths and weaknesses. Secret-sharing-based MPC excels in distributed settings where parties collaborate with minimal trust, offering strong fault tolerance and resilience against failures. However, it often requires high communication complexity due to the need for share exchanges at each computation step. On the other hand, GC-based MPC is well-suited for two-party scenarios, reducing the need for extensive communication but introducing higher computational costs due to encryption overhead.

Several notable MPC protocols have been developed, each tailored for different security models and efficiency requirements:

Table 5.1: Comparison of Notable MPC Protocols

Protocol	Year	Security Model	Communication Complexity	Computation Complexity
GMW [130]	1987	Honest Majority	$O(n^2\|C\|)$	$O(\|C\|)$
BGW [37]	1988	Honest Majority	$O(n^2\|C\|)$	$O(\|C\|)$
SPDZ [86]	2012	Dishonest Majority	$O(n\|C\|)$	$O(\|C\|)$

The Goldreich-Micali-Wigderson protocol [130] was one of the earliest MPC constructions, relying on SS and Boolean circuit evaluation. While it

introduced a foundation for secure computation, its quadratic communication complexity limited its scalability. The Ben-Or, Goldwasser, and Wigderson protocol [37] improved upon this by enabling fault tolerance and supporting computations under an honest-majority assumption. However, practical limitations arose in large networks due to their reliance on multiparty consistency checks.

A significant breakthrough came with the SPDZ protocol [86], which optimized secret-sharing-based MPC for adversarial settings. By introducing a preprocessing phase, SPDZ significantly reduced online computation costs, making it more practical for real-world privacy-preserving applications such as financial analytics, medical data processing, and secure auctions.

While secret-sharing-based MPC techniques focus on distributing computations across multiple parties, GC provides an alternative method for secure two-party computation. Originally introduced by Yao, the garbled circuit framework allows SFE using encrypted Boolean circuits. This technique has been widely adopted not only in two-party computations but also in hybrid MPC frameworks. The next section details the fundamentals of GC, optimizations, and their integration with other secure computation methods.

5.1.2 Garbled Circuits

GC, first introduced by Yao [419], provides a secure way to evaluate a Boolean function between two parties without revealing intermediate values. This approach encrypts a function into a set of garbled logic gates, which the evaluator can process without learning the underlying computations.

A garbled circuit execution involves three main phases:

- GC: The garbler encrypts the function by assigning random labels to each wire and encrypting the truth tables of all logic gates.
- OT: The evaluator receives encrypted labels corresponding to their private input, ensuring that the garbler does not learn these values.
- Circuit Evaluation: The evaluator processes the garbled circuit using the received labels, decrypting gates sequentially until reaching the final output.

While GC provide strong security guarantees, they introduce significant communication overhead due to the need for transmitting garbled truth ta-

bles. Over the years, several optimizations have been proposed to improve their efficiency:

Table 5.2: Key Optimizations in GC

Optimization	Year	Primary Benefit
Free-XOR [214]	2008	Eliminates encryption for XOR gates
Half-Gates [429]	2014	Reduces ciphertexts per AND gate
Row Reduction [299]	2009	Minimizes encryption operations

The Free-XOR optimization [214] eliminated encryption for XOR gates by leveraging correlated randomness, significantly reducing circuit size. The Half-Gates technique [429] further reduced the cost of AND gates by halving the number of ciphertexts needed for evaluation. Additionally, row reduction [299] optimized garbled table storage, minimizing memory and computation costs.

These improvements have made GC more practical for applications such as secure cloud computing and privacy-preserving ML.

A fundamental component of GC is OT, a cryptographic primitive that enables private input selection. OT is essential in ensuring that parties obtain only the necessary encryption keys without revealing their actual inputs. Since OT plays a vital role in many secure computation protocols, including MPC and GC, it has been the subject of extensive research to improve its efficiency. The next section explores OT techniques, optimizations, and their impact on secure computation.

5.1.3 Oblivious Transfer

OT is a cryptographic protocol that allows a sender to transfer multiple messages while ensuring that the receiver learns only one, without revealing which message was selected. OT is a crucial building block for secure computation, particularly in GC, where it facilitates private input selection.

Over time, several optimizations have improved the efficiency of OT:

Table 5.3: Optimizations in OT

Optimization	Improvement
OT Extension [188]	Reduces expensive base instances
Silent OT [437]	Reduces computational overhead
Hardware OT [308]	Dedicated OT accelerators

The OT Extension technique [188] significantly reduced computational costs by requiring only a small number of base OTs. More recently, Silent

OT [437] replaced expensive public-key operations with fast symmetric-key cryptography, making OT more practical for large-scale secure computation. Additionally, hardware-assisted OT [308] introduced specialized OT accelerators, further reducing latency and improving scalability.

While cryptographic primitives such as MPC, GC, and OT provide strong theoretical security guarantees, real-world implementations are susceptible to physical and SCAs. These attacks exploit unintended leakage from hardware components, such as power consumption, EM emissions, and memory access patterns. Furthermore, adversaries can introduce faults to manipulate computations and extract secret data. The next section surveys various SCA and FIA that threaten secure computation implementations.

5.2 Survey of Side-Channel and Fault Injection Attacks on Secure Computation

Secure computation protocols, such as MPC, GC, and OT, are designed to protect sensitive data during collaborative computations. While these protocols offer robust cryptographic guarantees, their implementations can be susceptible to physical and SCAs that exploit unintended information leakages or induce faults in the system.

SCAs and FIAs represent significant threats to the integrity of secure computation, allowing adversaries to recover sensitive data or bypass cryptographic protections. This section explores the key SCAs and FIAs targeting secure computation, their methodologies, and the impact they have on cryptographic implementations.

5.2.1 Side-Channel Attacks on Secure Computation

SCAs extract sensitive information by analyzing physical emanations or resource usage patterns of cryptographic devices. Unlike cryptographic attacks, which target mathematical weaknesses, SCAs leverage physical leakage that occurs due to variations in power consumption, EM emissions, execution timing, or cache access patterns.

Power Analysis Attacks: Power analysis attacks measure the power consumption of a device during cryptographic operations to infer secret keys or data. SPA directly interprets power traces, while DPA statistically analyzes power variations across multiple executions to extract cryptographic

keys [210]. DPA is particularly effective against secure computation protocols, including MPC-based secure inference and GC-based privacy-preserving DL [29].

Electromagnetic Analysis Attacks: EM SCAs capture EM emissions from hardware components during computation. By analyzing these emissions, attackers can reconstruct processed data or extract secret keys from cryptographic protocols [112]. Secure computation frameworks implemented on hardware accelerators, such as field-programmable gate arrays (FPGA)-based GC evaluators, are particularly vulnerable to such attacks due to their predictable power and EM leakage patterns [342].

Timing Attacks: Timing attacks exploit variations in execution time to deduce secret information. Cryptographic algorithms that involve conditional branches or varying memory access times can leak information through execution delays [57]. In secure computation, OT protocols and SFE implementations can be vulnerable to timing-based attacks, particularly when they rely on non-constant-time cryptographic primitives [440].

Cache Attacks: Cache attacks exploit shared memory resources in modern processors to infer data being processed. Some techniques such as Flush+Reload [421] and Prime+Probe [353] allow an attacker to observe cache access patterns and deduce secret information. These attacks are especially relevant in cloud-based secure computation frameworks, where multiple parties may share computation resources [416].

Table 5.4: Summary of SCAs Against Secure Computation

Attack Type	Target	Methodology
Power Analysis [210]	MPC	Measures power variations to extract keys
EM Analysis [112]	GC	Uses EM emissions to recover garbled values
Timing Attack [57]	OT	Exploits execution timing variations
Cache Attack [421]	Secure Computation Frameworks	Monitors cache access patterns

5.2.2 FIAs on Secure Computation

FIAs deliberately introduce errors into a system to disrupt its normal operation, potentially revealing sensitive information or compromising security mechanisms. By inducing faults during execution, adversaries can force cryptographic protocols to leak secret data or behave incorrectly.

Voltage Glitching: Voltage glitching involves momentarily altering the supply voltage to introduce transient faults in computations [358]. Secure computation protocols that rely on cryptographic primitives such as advanced encryption standards (AES) [84] or Rivest–Shamir–Adleman (RSA) [322] can be compromised by voltage-induced faults, leading to partial key exposure [347].

Clock Glitching: Clock glitching manipulates the clock signal to induce timing errors in processing [28]. By accelerating or slowing down computation cycles, attackers can cause unintended behaviors in secure computation frameworks, leading to data leakage or cryptographic failures [23].

Laser Fault Injection: LFI uses focused laser beams to disrupt specific areas of a chip, causing faults in the execution of cryptographic algorithms [338]. This attack is particularly effective against hardware-based secure computation, including FPGA implementations of GC and OT modules [29].

Electromagnetic Fault Injection: EMFI applies strong EM pulses to induce transient faults in electronic circuits [347]. Secure processors and MPC hardware accelerators are susceptible to EMFI, which can bypass security checks or introduce computational errors that reveal sensitive data [342].

Table 5.5: Summary of FIAs on Secure Computation

Fault Type	Target	Methodology
Voltage Glitching [358]	Secure Hardware	Induces power fluctuations to alter computations
Clock Glitching [28]	MPC Circuits	Modifies clock signals to disrupt execution
Laser Fault Injection [338]	Cryptographic Engines	Uses laser pulses to induce controlled faults
EM Fault Injection [347]	Secure Processors	Uses EM pulses to introduce transient faults

5.2.3 Impact of Side-Channel and FIAs on Secure Computation

The practical implications of SCAs and FIAs on secure computation protocols are significant:

MPC: Power analysis and timing attacks can compromise the confidentiality of participants' inputs by revealing secret shares or intermediate computations. Cache attacks further threaten the security of MPC imple-

mentations in cloud environments, where shared hardware resources create additional attack surfaces [441].

GC: EM analysis can leak information about the garbled values, undermining the security of the evaluated function. Furthermore, LFI can manipulate the garbled circuit evaluation, leading to incorrect outputs or information leakage [29].

OT: Timing discrepancies can allow adversaries to determine which inputs were selected, breaching protocol privacy. FIAs targeting OT modules can compromise key agreement mechanisms, leading to information disclosure [440].

Secure Hardware Implementations: FIAs can bypass security features, extract cryptographic keys, or alter control flows, leading to unauthorized data access. Hardware obfuscation techniques and circuit masking can mitigate such attacks, but they often come with increased implementation complexity [342].

Given the wide range of attacks that can compromise secure computation protocols, researchers have developed various countermeasures to mitigate these threats. Masking techniques, which aim to randomize power and data dependencies, and hiding techniques, which obfuscate execution behavior, are two fundamental strategies used to protect cryptographic implementations. The next section details these techniques, their effectiveness, and the trade-offs they introduce.

5.3 Masking and Hiding Techniques

To mitigate SCAs, researchers have proposed various masking and hiding techniques that aim to obfuscate power consumption, EM emissions, and timing behavior during computation. These approaches prevent adversaries from extracting sensitive information through unintended leakage channels. However, while effective, these countermeasures introduce trade-offs in terms of performance overhead and implementation complexity.

5.3.1 Power Analysis and EM Hiding Countermeasures

One of the primary attack vectors in SCAs is power analysis, where attackers measure power consumption fluctuations to deduce cryptographic keys or NN

parameters [210]. Masking techniques have been widely studied to counteract such threats.

Boolean Masking: In Boolean masking, sensitive variables are split into multiple shares such that each individual share leaks no information about the original value. Computations are then performed on masked values, ensuring that intermediate power consumption does not correlate with the actual secret values [80]. Boolean masking has been implemented in various secure computation frameworks, including MPC protocols and hardware-accelerated cryptographic systems [267].

Arithmetic Masking: Unlike Boolean masking, arithmetic masking represents sensitive values as random shares under modular arithmetic operations. This approach is particularly useful for securing DL accelerators that rely on integer or floating-point operations [103]. Despite its effectiveness, arithmetic masking requires additional computation to maintain masked values throughout complex mathematical transformations, leading to increased latency [101].

Higher-Order Masking: Traditional masking schemes operate at the first order, meaning they protect against first-order SCAs but remain vulnerable to higher-order power analysis. To address this, higher-order masking schemes expand the number of shares used, making statistical analysis of power traces significantly more difficult for attackers [61]. However, the computational cost grows exponentially with the order of masking, making higher-order techniques impractical for resource-constrained environments [342].

Beyond power analysis, EM SCAs exploit EM emissions from hardware components to reconstruct sensitive computations [112]. Countermeasures against EM-based SCAs focus on reducing signal correlation with secret data or introducing noise to mask relevant information.

Randomized Execution: Randomizing the order of execution for instructions and data processing steps disrupts consistent EM leakage patterns, making attacks less effective [269]. However, achieving full randomization without impacting performance remains an ongoing challenge.

Shielding and Filtering: Physical shielding using conductive enclosures can prevent EM emissions from being captured externally [342]. Similarly, low-pass filters can be used to suppress high-frequency leakage signals that carry the most information [436].

Signal Blinding and Noise Injection: By injecting controlled noise into power and EM channels, adversaries are prevented from reliably distin-

guishing actual signal variations due to computation [255]. While effective, noise injection increases power consumption and may degrade overall system efficiency.

5.3.2 Instruction-Level Obfuscation

Instruction-level obfuscation techniques aim to modify execution patterns such that an adversary observing power or timing behavior cannot reliably infer secret data. This method is commonly employed in hardware-accelerated secure computation [385].

Dummy Operations: Introducing non-functional dummy instructions ensures that timing and power consumption remain uniform across different execution paths [28]. This technique is widely used in cryptographic implementations such as AES and RSA [338].

Randomized Branch Execution: Conditional branches can be randomized to obscure execution flow, preventing timing-based SCAs [430]. However, this method may introduce significant control overhead and reduce performance efficiency.

Hardware-Level Obfuscation: Some secure hardware architectures integrate obfuscation at the microarchitectural level, ensuring that identical operations exhibit different power signatures across executions [385]. This is particularly useful in secure DL accelerators and cryptographic processors [240].

Table 5.6: Comparison of Masking and Hiding Techniques

Technique	Operation Domain	Targeted Side-Channel
Boolean Masking [80]	Boolean Circuits	Power (First-order)
Arithmetic Masking [103]	Modular Arithmetic	Power (First-order)
Higher-Order Masking [61]	Boolean/Arithmetic	Power (Higher-order)
Randomized Execution [269]	Instruction Execution	Electromagnetic
Shielding and Filtering [342]	Hardware-Level	Electromagnetic
Signal Blinding [255]	Power/EM Manipulation	Power/EM
Dummy Operations [28]	Instruction Execution	Power
Randomized Branch Execution [430]	Control Flow	Timing/Power
Hardware-Level Obfuscation [385]	Microarchitecture	Power/Timing

5.3.3 Limitations and Practical Challenges

While masking and hiding techniques significantly enhance security, they come with notable drawbacks:

Performance Overhead: Many masking techniques require additional computation for share management, while hiding techniques often introduce redundant operations or require specialized hardware, leading to higher power consumption [430]. Higher-order masking, for instance, dramatically increases the computational cost due to the exponential growth of required shares [342].

Implementation Complexity: Designing hardware and software implementations that maintain security while minimizing overhead is highly challenging [101]. Incorrect implementations can lead to security loopholes that compromise the entire system [284]. Additionally, countermeasures such as randomized execution demand precise synchronization to avoid introducing unintended vulnerabilities [269].

Scalability Issues: While effective for small-scale applications, higher-order masking and noise injection techniques struggle to scale efficiently for complex DL workloads and large secure computation frameworks [267]. For example, hardware-level obfuscation may be effective for cryptographic functions but impractical for DL inference due to resource constraints [240].

Among various secure computation techniques, GC provide strong security guarantees through encrypted Boolean function evaluation. However, they remain susceptible to certain side-channel and implementation-level attacks. Recent research has explored integrating countermeasures such as circuit masking and leakage-resistant encoding schemes to enhance the security of GC-based computations. The following section examines state-of-the-art GC implementations, their optimizations, and their role in S/PFE.

5.4 Garbled Circuit and Secure/Private Function Evaluation

GC is fundamental to S/PFE, enabling privacy-preserving computation by transforming Boolean circuits into encrypted representations that can be evaluated without revealing intermediate values. Various implementations of GC have been proposed to enhance efficiency and security, including hardware-accelerated GC and hybrid cryptographic approaches. This section presents an overview of recent advances in GC-based computation, highlighting key methodologies and their applications.

5.4.1 Garbled Accelerators

Table 5.7 provides a summary of recent frameworks developed based on garbled circuit principles. Among these, GarbledCPU [364] and RedCrypt [327] stand out for their hardware-based implementations, while other studies focus on software-based garbling engines and evaluators [362, 25, 328, 318, 317, 173]. Additional approaches utilizing hybrid secret-sharing or ZKPs have also been explored to counteract malicious adversaries, such as BLAZE [291], SIMC [66], and MUSE [222], which integrate GC with cryptographic protocols to enhance security.

General-Purpose Hardware Accelerators

GarbledCPU [364] introduces a hardware-based garbled circuit evaluator implemented on general-purpose sequential processors. This approach ensures privacy preservation for NN architectures. However, its design is specific to the Microprocessor without Interlocked Pipelined Stages (MIPS) architecture, limiting broader applicability. To address this constraint, the ARM2GC framework [363] was developed, synthesizing an ARM processor circuit to support pervasiveness and conditional execution. The efficiency of these solutions has been reported in terms of hardware resource utilization and communication costs.

Another contribution is the FPGA-based garbling engine FASE [174], which extends the hardware evaluation of GC. FASE enables cloud servers to provide secure services to multiple clients without violating data privacy. While these implementations focus on secure computation, they do not explicitly address side-channel protection, which remains a challenge in hardware accelerators.

Hardware Deep Learning Accelerators

RedCrypt [327] was designed to enhance cloud-based secure computation by achieving high-throughput and energy-efficient GC evaluation in real-time. Using an FPGA-based garbling core (Virtex UltraSCALE VCU108), RedCrypt optimizes gate-level control per clock cycle, reducing idle states and significantly increasing processing throughput. Compared to previous implementations, such as GarbledCPU and TinyGarble [364, 362], RedCrypt provides notable computational improvements.

However, RedCrypt assumes that the network architecture is publicly known, making it more susceptible to SCAs (SCAs) [29]. This limitation is addressed by HWGN^2, which ensures model privacy by obfuscating the architecture. HWGN^2 offers an NN-agnostic approach, making it a more versatile and secure alternative to previous GC accelerators. Additionally, approaches such as BLAZE [291] and SIMC 2.0 [412] have extended garbled circuit techniques by integrating them with ZKPs and MPC methods to counteract malicious adversaries.

Zero-Knowledge Proof-Based Approaches

MUSE [222] combines ZKPs with HE to protect against malicious clients in secure DL inference. By using ZKPs, MUSE ensures that computation results are verifiable without revealing sensitive model parameters. However, ZKP approaches suffer from computational complexity and require extensive setup, limiting their practical deployment.

Building on MUSE, SIMC [66] introduced optimized verification steps, reducing the proof size and computational overhead. The latest iteration, SIMC 2.0 [412], further enhances security against malicious adversaries by refining the ZK proof framework and improving scalability for large-scale DL inference.

Secret-Sharing and Hybrid Garbled Circuit Approaches

BLAZE [291] integrates SS and GC to create a more efficient MPC framework for secure inference. By leveraging a combination of MPC and GC, BLAZE achieves lower communication overhead while ensuring robustness against malicious adversaries.

Table 5.7 presents a high-level comparison of state-of-the-art approaches, emphasizing security features such as parameter secrecy, protection against malicious adversaries, and architecture obfuscation.

While GC enable efficient SFE, certain security challenges remain, particularly in the presence of malicious adversaries. ZKPs provide an additional layer of security by allowing verifiable computation without revealing private data. Furthermore, hybrid secure computation approaches combine multiple cryptographic techniques—such as authenticated garbling (AG), SS, and HE—to balance performance and security. The next section explores the

Table 5.7: Summary of garbled DL accelerators and their features.

Paper	Adversary Model	Approach	Contribution
DeepSecure [328]	HbC	Garbling	• Pre-processing to improve efficiency
Chameleon [318]	HbC	Hybrid	• Uses additive SS for linear operations • Uses Yao's GC for nonlinear operations
Ball et al. [24]	HbC	Hybrid	• Extends the BMR scheme [25] • Supports efficient non-linear operations
TinyGarble2 [173]	HbC + Malicious	Garbling	• Protection against malicious adversaries • Reduces memory cost of garbling
BLAZE [291]	Malicious	Secret Sharing + GC	• Efficient MPC protocol for secure NN • Reduces communication overhead
SIMC [66]	Malicious	GC + ZK Proofs	• Enhances MUSE with faster verification • Reduces proof size in ZK framework
SIMC 2.0 [412]	Malicious	GC + ZK Proofs	• Improves security model for malicious adversaries • Optimized for large-scale NN inference

role of ZKPs and hybrid approaches in enhancing secure computation frameworks.

5.5 Zero-Knowledge Proofs and Hybrid Secure Computation Approaches

Secure computation frameworks often rely on a combination of cryptographic primitives to enhance efficiency, security, and scalability. ZKPs play a crucial role in ensuring the integrity of SFE, while hybrid approaches integrate multiple cryptographic techniques such as AG, SS, and HE to balance performance and security. This section explores the role of ZKPs in secure computation, discusses hybrid cryptographic approaches, and compares their practical implementations in privacy-preserving DL.

5.5.1 Zero-Knowledge Proofs for Secure Computation

ZKPs allow one party (the prover) to convince another party (the verifier) that a statement is true without revealing any additional information beyond its validity [137, 39, 375]. This property makes ZKPs particularly useful in secure MPC and GC by ensuring correctness without requiring full trust

between parties. In the context of secure computation, ZKPs are commonly used for:

Input Consistency: Ensuring that participants use the same input across multiple executions prevents malicious deviations [233, 66]. Without such verification, an adversary could manipulate its inputs across different computation rounds, leading to inconsistencies in the results.

Circuit Correctness: A fundamental challenge in GC-based computation is ensuring that the evaluated circuit correctly represents the agreed function. ZKPs provide a mechanism for the evaluator to verify circuit correctness without learning anything about the underlying function [232].

Malicious Adversary Mitigation: In malicious settings, adversaries may attempt to manipulate computation by deviating from the protocol. ZKPs mitigate such risks by enforcing compliance and allowing verifiers to detect unauthorized modifications [222, 412].

Several frameworks have integrated ZKPs to secure DL computations. MUSE [222] employs a combination of HE, SS, and ZKPs to prevent model inference leakage. This ensures secure NN inference even in the presence of malicious clients. Similarly, SIMC [66] and its improved variant, SIMC 2.0 [412], optimize proof generation and execution times for secure inference while leveraging ZKPs to enhance robustness. However, the adoption of ZKPs introduces computational complexity, requiring additional proof verification steps that increase runtime overhead, especially in large-scale DL applications.

5.5.2 Hybrid Cryptographic Approaches

To balance security and efficiency, several hybrid approaches have been proposed, integrating GC, SS, and HE.

Authenticated Garbling

AG extends traditional GC protocols by embedding authentication mechanisms to detect circuit tampering. This method enhances security by enabling more efficient cut-and-choose GC protocols, ensuring correctness and input consistency [402, 233]. A key advantage of AG is its ability to strengthen GC protocols against selective OT attacks and incorrect circuit construction [233]. However, integrating authentication increases computation overhead due to the additional checks required for circuit validation.

SS with GC

Combining SS with GC offers a trade-off between communication and computation efficiency. Several protocols have leveraged this hybrid approach:

Chameleon [318] employs additive SS for linear operations and GC for nonlinear computations, ensuring efficiency while maintaining security. However, its reliance on SS introduces additional rounds of communication, which can become a bottleneck in high-latency networks.

FLASH [60] enhances secure inference by optimizing GC and SS for three-party computation (3PC), reducing overhead compared to purely GC-based methods. Although it improves efficiency, FLASH depends on a three-party setup, which may not be feasible in two-party settings.

Trident [70] focuses on ensuring fairness and robustness in secure computation by integrating SFE primitives. It prevents adversaries from prematurely aborting computations but requires additional cryptographic checks, which contribute to higher communication complexity.

Despite these benefits, hybrid approaches that incorporate SS necessitate precise synchronization between parties. This dependency increases communication rounds and may impact overall latency in large-scale deployments [396, 85].

HE with Secure Computation

Hybrid HE-based approaches leverage HE for performing secure computations on encrypted data [76, 430]. However, standalone HE techniques introduce significant computational overhead [283], making them inefficient for large-scale DL. To address this, several frameworks integrate HE with other cryptographic primitives:

MUSE [222] incorporates HE alongside SS and ZKPs to protect NN inference. This approach reduces inference leakage but still suffers from high verification costs due to the complexity of HE-based computations.

CRYPTGPU [379] accelerates HE using GPUs, making privacy-preserving NN inference more practical. While this improves efficiency, its scalability remains limited to HPC environments.

SWIFT [215] combines HE with differential privacy to prevent data leakage in ML applications. However, the added security comes at the cost of increased computational complexity, limiting its real-time applicability.

While HE enables secure computation over encrypted data, it remains

Table 5.8: Comparison of Zero-Knowledge and Hybrid Secure Computation Approaches.

Approach	Security Model	Techniques
MUSE [222]	Malicious	HE + ZKPs + SS
SIMC [66]	Malicious	GC + ZKPs
Trident [70]	Malicious	GC + SS
FLASH [60]	Malicious	GC + SS
Chameleon [318]	Semi-honest	SS + GC
CRYPTGPU [379]	Semi-honest	HE (GPU-accelerated)
SWIFT [215]	Semi-honest	HE + DP

computationally expensive compared to GC-based methods. Hybrid frameworks attempt to reduce reliance on HE by leveraging SS and ZKPs, optimizing efficiency in secure computation scenarios [328].

Table 5.8 provides a comparative overview of different ZKP and hybrid secure computation approaches, outlining their key techniques and security guarantees.

Chapter 6

MPC for IP Protection

6.1 Motivation

The increasing complexity of semiconductor design and fabrication has raised significant concerns regarding the security and privacy of IP. As chip manufacturing processes evolve, they involve multiple stakeholders, including design houses, third-party intellectual property (3PIP) vendors, and foundries, each introducing potential risks to confidentiality and integrity. The previous chapters have explored secure computation techniques and their applicability in protecting sensitive computations. This chapter builds on that foundation and focuses on leveraging MPC to enhance IP security in EDA workflows and secure computation frameworks.

Several hardware security mechanisms have been proposed to protect IP, including logic locking [329], encryption-based protections [14], and obfuscation techniques [305]. These methods aim to prevent unauthorized access to proprietary designs and thwart reverse engineering attacks. However, conventional approaches often face challenges such as susceptibility to oracle-guided attacks [415], vulnerability to side-channel leakages [388], and adversarial model extraction [424]. As a result, secure computation techniques, including SFE and PFE, have emerged as promising solutions to ensure the confidentiality of hardware designs while enabling collaborative workflows among distrusting parties.

This chapter introduces two key approaches that leverage MPC for securing IP: Garbled EDA [157] and GuardianMPC [154]. Garbled EDA integrates cryptographic techniques into the EDA toolchain, enabling secure and

privacy-preserving circuit design and verification without exposing confidential design details to untrusted tools or external entities. This method addresses the risk of IP leakage in outsourced design and verification workflows by ensuring that only authorized computations are performed on encrypted representations of circuits, preventing unauthorized inference of design functionalities [157].

GuardianMPC extends MPC principles to broader secure computation settings, safeguarding critical hardware security applications by enabling distributed and privacy-preserving execution of computations. Unlike conventional hardware protection mechanisms, GuardianMPC leverages secure multi-party protocols to execute IP-sensitive operations in a way that minimizes leakage risks and ensures resilience against adversarial tampering [154]. Through efficient cryptographic protocols and hardware-optimized implementations, GuardianMPC reduces computation overhead while maintaining strong security guarantees [154].

The following sections provide a detailed methodology and evaluation of these frameworks. By incorporating secure computation principles into modern IP protection schemes, these approaches significantly enhance the security and privacy of semiconductor designs, ensuring that sensitive hardware components remain protected throughout the design, verification, and deployment phases.

6.2 GarbledEDA: Privacy-Preserving Electronic Design Automation

6.2.1 Methodology

The GarbledEDA [157] framework introduces a novel approach to secure and private EDA by leveraging SFE and PFE techniques. This methodology ensures that critical IP remains confidential throughout the compilation and simulation processes while protecting against adversaries with varying levels of sophistication. Unlike conventional security mechanisms such as IEEE P1735 [175], which rely on encryption-based access control, GarbledEDA employs cryptographic protocols that prevent IP disclosure even in untrusted environments [157]. Prior research has demonstrated the vulnerabilities of IEEE P1735, particularly concerning key extraction and reverse engineering attacks [444, 64], making it an insufficient solution against well-equipped

adversaries.

At its core, GarbledEDA builds upon Yao's GC principle [417], a cryptographic method originally designed for secure two-party computation. In this context, the EDA tool vendor assumes the role of a garbler, transforming the raw IP and EDA tool into an encrypted (garbled) representation. The IC designer, who acts as an evaluator, processes the garbled version without gaining access to the underlying sensitive design details. This ensures a secure EDA workflow where neither the IC designer nor the tool vendor can extract unauthorized information [130, 364]. The security guarantees of GarbledEDA hold even in the presence of an adversary attempting to manipulate protocol execution, as the garbled representation remains computationally indistinguishable from random data.

To support both compilation and simulation securely, GarbledEDA employs distinct cryptographic procedures tailored to each stage. During secure compilation, the IP owner provides a design description that is processed by the CAD tool vendor, incorporating process design kit (PDK) information. The output is a garbled netlist, which is forwarded to the IC designer. Since this garbled netlist is encrypted and evaluated in a privacy-preserving manner, unauthorized access to proprietary logic is prevented, mitigating risks posed by reverse engineering attempts. Similarly, during secure simulation, proprietary simulation models and input stimuli are protected to ensure that sensitive design information is not leaked to untrusted CAD vendors [157]. By enforcing SFE on both inputs and outputs, GarbledEDA ensures end-to-end confidentiality during the entire design flow.

The GarbledEDA framework also addresses security threats arising from different adversarial models. The compilation and simulation environments are susceptible to both HbC and malicious adversaries, as illustrated in Figure 6.1. An HbC adversary, such as an IC designer or an untrusted EDA vendor, follows the protocol correctly but attempts to infer additional information from intermediate data. In contrast, a malicious adversary actively deviates from the protocol by tampering with inputs, modifying garbled tables, or executing unauthorized queries to extract information [130]. GarbledEDA mitigates these threats by ensuring that GC are resistant to reverse engineering and implementing cryptographic techniques such as OT to securely exchange encrypted values between parties [226].

Furthermore, to ensure robustness in a real-world EDA workflow, GarbledEDA integrates PFE techniques to prevent adversarial leakage of EDA tool logic. This is particularly critical for securing proprietary CAD tool im-

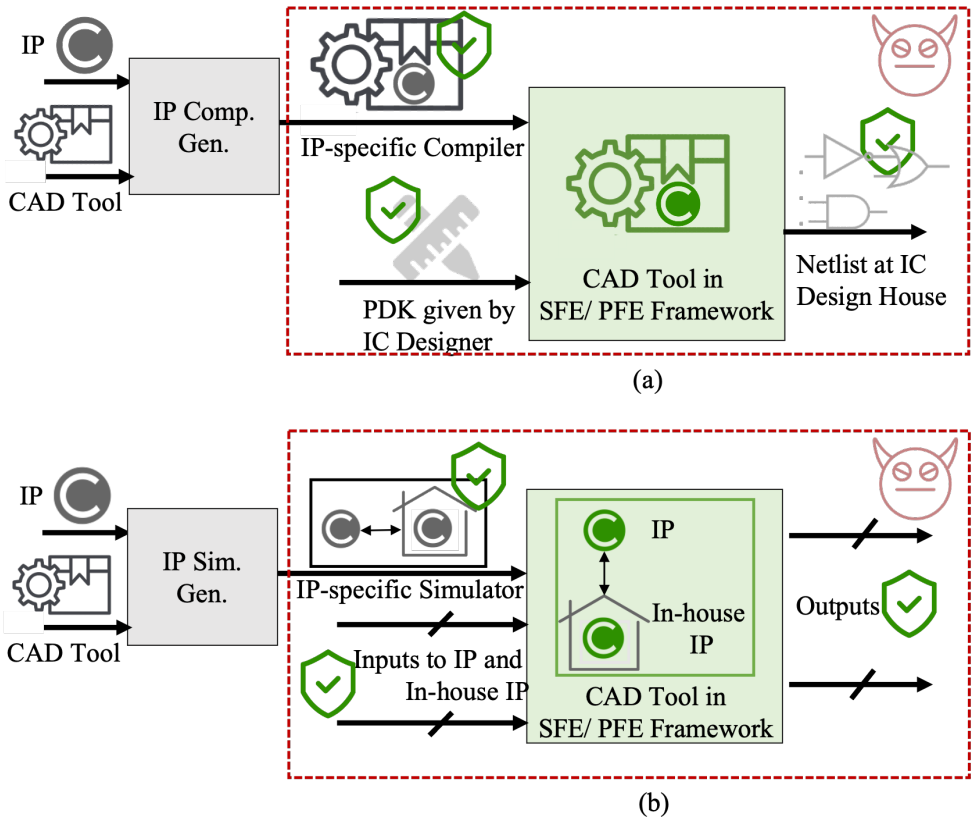


Figure 6.1: Proposed CAD/EDA compilation and simulation of IP under various secure scenarios. The adversary at the design house could be either HbC or malicious, attempting to tamper with the IP-specific compiler or simulator to extract the IP. In a secure compilation scenario, both the IP and PDK inputs remain protected, preventing unauthorized access to proprietary technology. Similarly, during simulation, secure execution ensures that inputs remain private while restricting an untrusted CAD vendor from gaining access to the simulation output.

plementations, which could otherwise be reverse-engineered by an untrusted user. Unlike standard SFE, which only hides inputs and outputs, PFE ensures that the functionality of the EDA tool itself remains confidential [264]. As a result, GarbledEDA provides comprehensive security coverage that extends beyond data protection, ensuring that both the design IP and EDA tool logic remain hidden from adversaries.

By leveraging GC, OT, and PFE, GarbledEDA presents a scalable and

cryptographically secure alternative to existing IP protection mechanisms. It eliminates reliance on hardware obfuscation techniques, which have been shown to be susceptible to SAT-based deobfuscation attacks [373], and mitigates key leakage risks associated with encryption-based schemes such as IEEE P1735 [444]. The approach is designed to support various EDA operations, including compilation, simulation, and verification, making it adaptable for use in next-generation secure design workflows.

Figure 6.1 illustrates the secure execution of compilation and simulation under different adversarial scenarios. In a secure compilation scenario, both the IP and PDK inputs remain protected, preventing unauthorized access to proprietary technology. Similarly, during simulation, secure execution ensures that inputs remain private while restricting an untrusted CAD vendor from gaining access to the simulation output. The use of garbled representations and secure evaluation techniques guarantees that neither the IC designer nor the CAD vendor can compromise the integrity or confidentiality of the design IP.

6.2.2 Secure Computation for IP Protection

The GarbledEDA [157] framework employs secure computation techniques to ensure the confidentiality and integrity of IP throughout the EDA workflow. The methodology is built upon cryptographic protocols, including OT, GC, and PFE, to prevent adversarial entities from gaining unauthorized access to proprietary design information. Unlike traditional encryption-based protections, which are vulnerable to key extraction attacks [444, 64], secure computation guarantees that IP remains concealed even in hostile execution environments [130].

At the core of GarbledEDA lies the interactive cryptographic protocol that enables privacy-preserving compilation and simulation. The execution follows a two-party model where the CAD tool vendor, acting as the garbler, transforms the IP-specific compiler or simulator into an encrypted (garbled) representation [35]. This garbled version is then transmitted to the IC designer, who functions as the evaluator, executing the computation on encrypted data without learning the underlying IP content. By applying Yao's garbled circuit methodology [417], this approach extends prior work on secure processors that employ encrypted instruction sets to obfuscate execution details [364, 401].

The GarbledEDA framework integrates multiple layers of security to de-

fend against potential adversarial threats. One of the primary attack vectors in SFE is tampering with GC to extract sensitive information. To mitigate this risk, GarbledEDA incorporates cut-and-choose protocols [228], a widely used cryptographic technique for ensuring malicious security in garbled circuit execution. This method involves generating multiple garbled versions of the IP-specific compiler or simulator and requiring the evaluator to verify a subset before executing the computation. Since a malicious adversary cannot predict which versions will be inspected, the probability of successfully injecting manipulated circuits remains negligibly low, thereby significantly reducing the risk of undetected tampering.

Furthermore, message authentication codes (MACs) are embedded within garbled outputs to prevent unauthorized modifications and replay attacks [133]. By applying one-time MACs, the framework ensures that any unauthorized attempt to alter output data will be detected, thereby preserving computation integrity. This is particularly critical in scenarios where an untrusted IC designer may seek to manipulate simulation results to extract insights about the underlying IP. The inclusion of authenticated encryption mechanisms provides additional safeguards, ensuring that even in the presence of an adversarial execution environment, only the intended function output is revealed to the evaluator [214].

Beyond execution integrity, GarbledEDA also addresses the challenge of securely handling private inputs from multiple parties. In secure compilation, the IP owner's design files, the CAD tool vendor's proprietary compiler, and the IC designer's process constraints must be protected from unauthorized inference. Similarly, in simulation, the confidentiality of proprietary test vectors and design behavior must be maintained. To achieve this, GarbledEDA extends PFE techniques, enabling secure execution without revealing the EDA tool's internal logic [264]. This feature is crucial for preventing reverse engineering attacks on proprietary simulation models, which has been a major challenge in conventional security frameworks such as IEEE P1735 [175].

To further enhance security in multiparty settings, GarbledEDA incorporates OT as a foundational building block for input privacy. OT enables one party (evaluator) to retrieve an encrypted function input from another party (garbler) without revealing which specific input was chosen [218]. This guarantees that the IC designer can securely execute simulation or compilation tasks without directly accessing sensitive IP components. Additionally, the integration of randomized circuit evaluation techniques further obfuscates computational patterns, preventing adversaries from using power analysis or

timing side channels to infer IP characteristics [288, 244].

By combining GC, OT, and PFE, GarbledEDA provides a robust and cryptographically secure alternative to conventional IP protection mechanisms. It mitigates vulnerabilities inherent in hardware-based obfuscation techniques, which are often susceptible to deobfuscation via Boolean satisfiability (SAT) solvers [373], and eliminates the reliance on encryption schemes that can be compromised through key leakage [444]. The application of these techniques ensures that the security of the IP remains intact even in adversarial environments, making GarbledEDA a viable solution for next-generation secure EDA workflows.

6.2.3 GarbledEDA Implementation Flow

The implementation of GarbledEDA follows a structured pipeline that enables privacy-preserving compilation and simulation of IP cores. This methodology ensures that the IP remains confidential while passing through different stages of an EDA flow. The approach is based on SFE and PFE, utilizing state-of-the-art cryptographic primitives such as Yao's GC and OT [130].

The workflow begins with hardware description preprocessing, in which the given design, typically written in Verilog or C, is transformed into an intermediate representation for secure evaluation. This transformation is critical for compatibility with garbling-based computation frameworks that require functionally equivalent software representations of circuit descriptions. For Verilog-based designs, the V2C tool is employed to translate the Hardware Description Language (HDL) specification into a functionally equivalent C representation [71]. V2C is specifically designed to facilitate the conversion of Verilog constructs, including combinational and sequential logic, into a format that can be processed by software-based synthesis tools. The generated C code accurately represents the logic gates and signal propagation found in the original Verilog description, ensuring that subsequent processing steps maintain the correctness of the original hardware design.

Once the intermediate representation is generated, the GarbledEDA framework processes it further based on whether the execution is intended for a MIPS-based or ARM-based simulation. The choice of architecture influences how the IP is compiled, garbled, and evaluated securely. For MIPS-based execution, the translated C representation is first compiled using a MIPS cross-compiler, which converts the high-level design into machine-level instructions tailored for a MIPS processor. The compiled binary is then processed by the

GarbledCPU framework, which is specifically designed for secure computation on MIPS architectures [364]. GarbledCPU applies the principles of secure MPC by generating garbled MIPS instructions, which ensure that every operation in the circuit remains encrypted throughout execution. This transformation prevents unauthorized parties from learning the functionality of the IP. Once the garbled instructions are generated, they are executed using TinyGarble, an optimized garbled circuit framework that enhances the efficiency of secure circuit evaluation [362]. TinyGarble reduces the computational overhead of SFE by applying logic optimization techniques, such as gate reordering and precomputed garbled tables, to minimize resource consumption. By executing the MIPS instructions in a garbled format, TinyGarble ensures that no meaningful information about the IP structure is revealed, even when the computation is performed in an untrusted environment.

For ARM-based execution, a different transformation pipeline is utilized. Instead of using GarbledCPU, the ARM2GC framework is employed to process the IP design into garbled ARM instructions [363]. ARM2GC extends the GC approach to embedded and low-power processing environments, ensuring secure execution on ARM architectures commonly found in Internet of Things (IoT) devices and mobile processors. The ARM2GC framework takes the translated C representation of the IP and applies garbled logic transformations to produce an encrypted instruction set. These garbled instructions are then evaluated in a trusted execution environment, where security guarantees prevent side-channel leakage and unauthorized access. This methodology ensures that IP functionality is preserved while maintaining cryptographic confidentiality.

As depicted in Figure 6.2, GarbledEDA integrates a multi-step transformation process to secure both compilation and simulation of IPs. This process ensures that no sensitive information about the IP is exposed to untrusted parties, such as IC designers or CAD tool vendors. The parsing and conversion of an IP description in Verilog or C format ensures compatibility with cryptographic frameworks while maintaining logical correctness. The secure compilation phase follows, where the intermediate representation is processed through MIPS or ARM-based compilers and transformed into garbled instructions. This guarantees that IP security is preserved at the compilation stage, preventing unauthorized entities from reverse-engineering the design. Finally, the encrypted execution stage ensures that the garbled instructions are evaluated in a secure manner, where only authorized outputs are revealed, and all intermediate computations remain concealed.

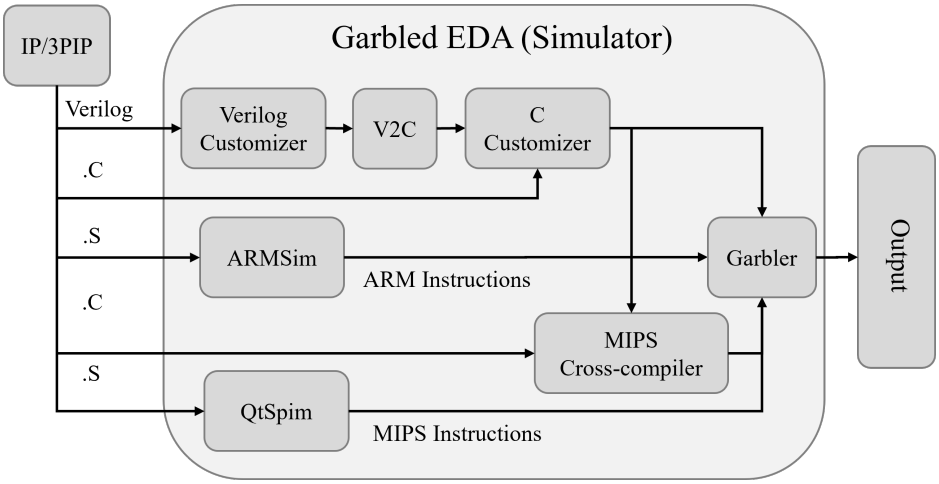


Figure 6.2: General flow of generating GarbledEDA. The process starts with parsing an IP description in Verilog or C format, which is then converted to an appropriate instruction set for secure evaluation. The garbler consists of two main components: ARM2GC for ARM-based execution and GarbledCPU for MIPS-based execution. The converted instructions are processed through these frameworks to generate garbled MIPS or ARM instructions that can be executed without exposing the original IP.

To further enhance security and performance, GarbledEDA incorporates several optimization techniques. One key optimization is circuit minimization, in which redundant logic operations are eliminated before garbling to reduce computation time [173]. Additionally, batch OT protocols are utilized to accelerate secure input selection, allowing multiple instances of an IP design to be processed efficiently [19]. These optimizations significantly reduce the performance overhead associated with secure computation, making GarbledEDA a practical solution for industrial-scale EDA applications.

Another critical aspect of GarbledEDA's security architecture is its resistance to tampering and replay attacks. To prevent adversarial modifications, the framework employs one-time message authentication codes (MACs), which verify the integrity of garbled instruction sequences before execution [133]. This mechanism ensures that even if an attacker attempts to modify the garbled circuit, the execution remains secure.

6.2.4 Optimizing Performance and Hardware Utilization

The GarbledEDA framework supports two primary execution models: maximum performance mode and resource-efficient mode. These configurations cater to different design constraints, ensuring flexibility in secure EDA. The first mode prioritizes execution speed by reducing communication overhead, while the second aims to optimize hardware utilization by efficiently managing computation and memory resources. The ability to switch between these execution strategies enables GarbledEDA to be applied across diverse hardware environments, from HPC clusters to resource-constrained embedded systems.

The maximum performance mode is designed for scenarios where computation speed is the primary concern. In this setup, the garbler transmits the entire set of garbled instructions upfront, allowing the evaluator to perform secure computations with minimal interaction. By reducing the number of required OTs (OT), this approach mitigates communication latency and improves execution efficiency [172]. This mode is particularly beneficial for large-scale simulations and high-speed cryptographic applications where frequent interaction between the parties would otherwise introduce significant bottlenecks. The use of pre-generated GC ensures that the execution pipeline remains uninterrupted, allowing for faster evaluation of complex IP designs.

Conversely, the resource-efficient mode is tailored for hardware-limited environments where minimizing memory usage and computational complexity is paramount. Instead of evaluating the entire circuit at once, GarbledEDA decomposes the design into smaller sub-netlists, which are processed sequentially [173]. This technique follows the principles of incremental garbled circuit execution, ensuring that only the necessary parts of the circuit are active at any given time [401]. By restricting the number of simultaneously loaded gates, the resource-efficient mode significantly reduces the memory footprint and enables the execution of secure computations on devices with constrained hardware resources, such as FPGA-based accelerators and embedded processors [173, 401, 363, 361].

The partitioning strategy used in the resource-efficient mode is inspired by prior optimizations in secure computation, where sub-circuits are evaluated iteratively to manage hardware constraints efficiently [434]. The decomposition process groups neighboring gates into clusters, ensuring that dependencies are preserved while minimizing redundant computations. Addition-

ally, this method allows for more effective parallelization in multiprocessor systems, where sub-netlists can be evaluated independently before merging results. The trade-off, however, lies in the increased number of OT interactions required to process each sub-netlist, potentially introducing additional communication overhead.

A key feature of GarbledEDA's optimization framework is its ability to dynamically adapt between these two execution modes based on available resources and application requirements. For instance, a designer working in a high-speed simulation environment might opt for the maximum performance mode to achieve rapid verification, while a developer targeting low-power hardware might favor the resource-efficient mode to reduce computational overhead. This adaptability ensures that GarbledEDA can be integrated into a wide range of secure EDA workflows, spanning from ASIC and FPGA design to cloud-based CAD services.

6.2.5 GarbledEDA Simulator Implementation Flow

The GarbledEDA [157] framework implements a structured, privacy preserving execution model that enables secure EDA workflows, particularly in scenarios where IP protection is paramount. The simulator execution flow ensures that sensitive design data remains confidential throughout the compilation and simulation processes while offering optimizations tailored for either high-performance execution or memory-efficient secure evaluation.

At the core of GarbledEDA is the garbled circuit paradigm, which transforms IP designs into encrypted Boolean circuits that can be evaluated securely without revealing their structure. The framework supports both MIPS- and ARM-based execution environments, leveraging cryptographic techniques such as OT, cut-and-choose protocols, and integrity verification mechanisms to prevent adversarial interference [130, 364]. Figure 6.3 presents a detailed overview of the entire execution pipeline, highlighting two distinct execution strategies: maximum performance execution mode and resource-efficient execution mode.

Maximum Performance Execution Flow

The maximum performance execution flow is designed to optimize execution speed by minimizing communication overhead. In this approach, all garbled instructions are precomputed and transmitted in a single batch before exe-

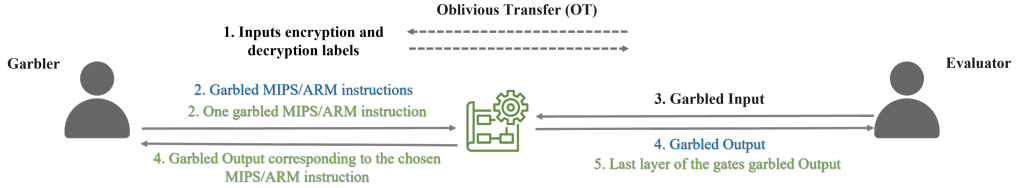


Figure 6.3: Flow of GarbledEDA simulator implementation. The figure illustrates (a) the maximum performance implementation (blue), which minimizes communication overhead and maximizes speed, and (b) the improved hardware resource efficiency implementation (green), which prioritizes memory efficiency by evaluating smaller sub-netlists sequentially. The first approach is optimized for high-performance applications, while the second is suited for hardware-constrained environments.

cution begins. This strategy reduces the number of OT interactions, which typically introduce significant computational and bandwidth overhead. By performing a one-time transmission of encrypted logic gates, the maximum performance mode is particularly well-suited for high-speed circuit simulations where low-latency processing is crucial.

The execution pipeline in this mode begins with the garbler, typically the CAD tool vendor or a secure IP processing unit, generating garbled MIPS/ARM instructions corresponding to the given logic function. Each logic gate in the IP design is converted into an encrypted truth table using Yao's GC method [417]. Alongside the garbled circuit, encryption and decryption labels are generated for each possible input/output combination. To ensure that the evaluator can perform computations without learning sensitive design information, an OT protocol is executed between the garbler and evaluator [19]. During this phase, the evaluator securely obtains the cryptographic labels corresponding to its inputs without revealing any information to the garbler.

Once the necessary cryptographic material is exchanged, the evaluator processes the encrypted circuit using a garbled MIPS/ARM core [364, 362]. Execution takes place without exposing intermediate results, ensuring that the IP remains confidential throughout the process. Upon completion, the evaluator decrypts the garbled output labels using the decryption keys provided by the garbler, thereby obtaining the final computation results. This approach significantly reduces OT interactions, a major performance bottleneck in secure computation. However, it requires a substantial amount

of memory to store precomputed garbled instructions, making it ideal for scenarios with ample computational resources but stringent execution time constraints [172].

Resource-Efficient Execution Flow

The resource-efficient execution mode is designed for environments with constrained memory resources, such as FPGA-based simulation frameworks and embedded systems [173]. Unlike the maximum performance mode, which transmits all garbled instructions in a single batch, this approach processes the circuit in smaller, sequential sub-netlists to optimize memory usage.

The execution pipeline begins by partitioning the full IP circuit into smaller sub-netlists, each representing a subset of logic gates. The number of gates per sub-netlist is adjustable based on available memory resources, ensuring that only a limited portion of the design is active at any given time [401]. Instead of sending all encrypted circuit data upfront, the garbler transmits one garbled sub-netlist at a time. This significantly reduces memory consumption on the evaluator's side and allows for incremental processing.

Each sub-netlist is processed sequentially within the garbled MIPS/ARM core, ensuring that only a small portion of the circuit is evaluated at any given time. After processing a sub-netlist, the partial results are decrypted and passed as inputs to the next sub-netlist in the sequence, maintaining data privacy while preserving execution integrity. The total number of OT interactions in this mode is calculated as:

$$M = \frac{N_{gates}}{4} + (I_{size} + O_{size}) \tag{6.1}$$

where N_{gates} is the total number of gates in the circuit, I_{size} represents the input size of the netlist, and O_{size} denotes the output size of the computation. While this approach requires additional interactive steps, it remains a practical solution for executing secure computations in memory-constrained settings.

Secure Execution in the Presence of Malicious Adversaries

Security in garbled circuit execution is crucial, particularly in scenarios where an adversarial evaluator may attempt to manipulate the evaluation process to

extract sensitive information. Unlike many existing garbled circuit protocols that assume an HbC adversary, GarbledEDA employs techniques for securing execution against active (malicious) adversaries.

To mitigate tampering and unauthorized inference, GarbledEDA incorporates cut-and-choose protocols [228] and randomized execution orders [173]. The randomized execution order ensures that the evaluator cannot predict the sequence of execution, making it extremely difficult to perform differential analysis on encrypted computations. Additionally, the final stages of execution integrate cryptographic authentication mechanisms, such as one-time Message Authentication Codes (MACs), ensuring that the output has not been tampered with or replayed [133].

The implementation also introduces countermeasures to prevent fault-injection attacks, which could alter computation results by modifying garbled tables or input labels [244]. By integrating error correction mechanisms and redundant computation techniques, GarbledEDA ensures that any adversarial manipulation is detected and mitigated. These security enhancements make GarbledEDA robust against both passive and active adversaries, ensuring the confidentiality and integrity of IP during secure compilation and simulation processes.

6.2.6 Evaluation Setup

The evaluation of GarbledEDA is conducted by implementing garbled IP-specific simulators that securely compile and simulate a variety of benchmark circuits. The evaluation setup involves synthesizing garbled MIPS and ARM instruction sets and executing them using modified processor architectures that support garbled circuit execution. Specifically, we utilize customized cores of Amber (ARM-based) and Plasma (MIPS-based) processors to execute garbled instructions corresponding to the given IP descriptions [363, 364].

To ensure an accurate evaluation of real-world circuits, we use the ISCAS-85 benchmark suite, a widely used set of industrial logic circuits. The suite includes C432, C499, C1355, C1908, C3540, and C6288, which are representative of random logic circuits. Since the high-level design details of these benchmarks are not publicly available, they serve as ideal candidates for evaluating secure execution frameworks by simulating circuits that resemble practical industrial applications [55].

Additionally, we compare the implementation cost of GarbledEDA us-

ing ISCAS-85 benchmarks with other widely used SFE benchmarks, such as AES encryption, 8-bit SUM, HD computation, and 8-bit/256-bit multiplication (MULT). These benchmarks cover a broad range of computational complexities and circuit sizes, allowing us to assess how GarbledEDA scales across different workloads.

For implementation, we generate garbled ARM/MIPS instructions corresponding to different IP descriptions and store them in the instruction memory of garbled IP-specific simulators. The synthesis process is carried out using Vivado 2021, a well-established FPGA development tool. The generated circuits are implemented and evaluated on an ARTIX-7 FPGA, which provides hardware-accelerated execution of GC. This setup enables a practical assessment of performance, resource utilization, and feasibility of deploying GarbledEDA in secure hardware design environments.

By leveraging FPGA-based execution, we gain insight into the practical overhead introduced by secure computation techniques. Unlike software-based implementations of SFE and PFE that rely on CPU or GPU processing, FPGA-based execution allows us to observe real-world constraints in terms of logic utilization, memory footprint, and execution time. The FPGA implementation also allows us to compare ARM-based and MIPS-based architectures, providing insights into their relative efficiency for garbled execution.

Using an FPGA-based evaluation provides several advantages. FPGA synthesis exposes the actual resource overhead of GarbledEDA, including LUTs, FFs, MUX usage, and digital signal processing blocks (DSP) consumption. Unlike traditional CPU-based evaluation, FPGA accelerates garbled circuit computation through its ability to process multiple logic gates in parallel. Additionally, the FPGA execution platform helps assess hardware-specific vulnerabilities such as side-channel leakages in the presence of FIA [312, 413].

Thus, GarbledEDA's evaluation methodology combines practical hardware execution with strong cryptographic guarantees to ensure secure compilation and simulation of IPs.

6.2.7 Resource Utilization Evaluation

The evaluation of GarbledEDA's resource utilization provides a comprehensive understanding of the computational overhead introduced by SFE and PFE techniques. By analyzing the logical and memory resources required

to implement GarbledEDA on an FPGA, we can assess the feasibility of deploying privacy-preserving IP simulation and compilation in real-world EDA workflows. This evaluation includes analyzing the synthesis and implementation costs of GarbledEDA when applied to various benchmarks across different computational complexities, including cryptographic functions, arithmetic operations, and industrial logic circuits.

To quantify the resource requirements of GarbledEDA, we implement a series of benchmark circuits using both ARM- and MIPS-based architectures. The evaluation covers a broad range of benchmarks, from simple arithmetic functions such as 8-bit SUM and HD to complex cryptographic and industrial benchmarks like AES encryption and ISCAS-85 circuits. These benchmarks provide a representative set of workloads to analyze the scalability of GarbledEDA across different domains. The synthesis process is carried out using Vivado 2021, with the resulting designs implemented on an ARTIX-7 FPGA to measure logic utilization, FF consumption, and memory requirements.

Table 6.1 presents the resource utilization of GarbledEDA in its maximum performance mode. The results include the number of lookup tables (LUTs), flip-flops (FF), and multiplexers (MUXes) required to implement each benchmark, along with the total number of garbled instructions executed. The table also provides a breakdown of XOR and non-XOR gates in each benchmark, highlighting the impact of Free-XOR optimization. Values outside the parentheses correspond to ARM-based implementations, while those inside the parentheses represent MIPS-based implementations.

A key insight from Table 6.1 is the impact of Free-XOR optimization, which significantly reduces the overhead of GarbledEDA when applied to benchmarks with a high number of XOR gates. For example, the AES benchmark, which primarily consists of XOR operations, demonstrates substantially lower hardware costs compared to non-XOR-intensive benchmarks of similar gate complexity. Despite having only 272 more gates than C6288, GarbledEDA for C6288 consumes four times more logical and memory resources due to the absence of XOR operations, which limits the effectiveness of Free-XOR optimization.

Benchmarks with more XOR gates exhibit lower resource utilization even when their total gate count is comparable to benchmarks dominated by non-XOR gates. The 256-bit multiplication benchmark, for instance, contains 163 more gates than C6288 but requires only half the logic and memory resources for its GarbledEDA implementation. This trend underscores the efficiency of Free-XOR optimization in reducing the cost of secure computation by

Table 6.1: GarbledEDA with maximum performance implementation cost of different benchmarks in ARM(MIPS).

Benchmark	XOR	Other	Inst.	LUT	FF	MUX
Amber(Plasma)	N/A	N/A	64	3526 (1817)	1830 (1255)	229 (292)
8-bit SUM	48	96	30	22796 (9319)	20169 (7166)	1672 (523)
16-bit Hamming	3	39	47	25116 (17591)	21835 (13742)	1646 (1179)
8-bit MULT	43	139	93	32075 (32973)	26262 (25598)	1840 (1864)
C499	104	198	236	86445 (110012)	58582 (83278)	4016 (9105)
C432	18	222	276	91829 (130163)	69104 (97263)	5487 (11390)
AES	2112	576	426	158870 (287334)	102112 (200179)	6896 (16085)
C1355	0	746	1335	237967 (506471)	145800 (394247)	9816 (26025)
C1908	0	880	1560	271495 (528364)	167340 (417282)	10911 (27752)
256-bit MULT	1303	1276	2012	361637 (712398)	220591 (512895)	14856 (39063)
C3540	0	1669	3008	513105 (1009021)	306512 (787454)	19792 (52407)
C6288	0	2416	4669	788181 (1523564)	465723 (944478)	30080 (60445)

leveraging fast XOR gate evaluations in GC.

To assess the worst-case overhead of GarbledEDA, we analyze the resource consumption of benchmarks with no XOR gates, including C1355, C1908, C3540, and C6288. These benchmarks serve as an upper bound for hardware cost since they do not benefit from XOR-based optimizations. On average, the implementation of GarbledEDA for these benchmarks requires 8.5 times more logic resources and 10.5 times more memory resources compared to their ARM-based architecture designs. For MIPS-based implementations, the overhead is slightly higher, with an increase of approximately 9.5 times in logic utilization and 8 times in memory consumption.

Comparing ARM- and MIPS-based implementations provides additional insights into architectural trade-offs. While both architectures impose significant overhead compared to unprotected designs, the specific resource demands vary based on benchmark characteristics. A notable distinction is the use of digital signal processing (DSP) blocks. ARM-based implementations require DSPs, while MIPS-based implementations do not, allowing greater flexibility in selecting architectures based on available hardware resources. If DSP availability is a constraint, MIPS-based implementations provide an alternative that avoids DSP dependency.

Another key observation is that MIPS-based implementations tend to be more resource-efficient for benchmarks with fewer than approximately

90 garbled instructions. For benchmarks exceeding this threshold, ARM-based implementations become more efficient. This suggests that the choice of architecture should be guided by the computational complexity of the target design. For lightweight designs, MIPS offers a better balance between performance and resource utilization, whereas for more complex benchmarks, ARM provides superior scalability.

Despite the non-trivial overhead associated with GarbledEDA, the results confirm its viability for secure IP protection in practical scenarios. The scalability of the framework across different computational workloads and architectures indicates that privacy-preserving EDA can be integrated into secure design flows without prohibitive resource costs. The next section explores further optimizations to reduce hardware overhead while maintaining the security guarantees of the GarbledEDA framework.

6.2.8 GarbledEDA with a Selector

One of the main challenges associated with GarbledEDA is the high hardware resource overhead introduced by the secure evaluation process. As demonstrated in previous sections, SFE and PFE incur significant logic and memory resource utilization due to the transformation of circuit elements into cryptographic representations. To address this concern, we propose an optimization technique that allows multiple IPs to be merged into a single GarbledEDA instance using a selector, thereby reducing overall implementation costs.

The fundamental idea behind this optimization is to leverage a unified evaluation core while allowing multiple IPs to be processed within the same hardware instance. Instead of implementing separate GarbledEDA instances for each IP, a single implementation is designed to accommodate multiple IPs by incorporating a selector mechanism. This selector is responsible for dynamically switching between different instruction sets and garbled tables corresponding to the various IPs included in the system. By doing so, the overhead associated with maintaining multiple independent evaluator cores is eliminated, leading to substantial reductions in resource consumption.

To demonstrate the efficacy of this approach, we implemented a GarbledEDA instance with a selector that can process three different benchmarks: C499, C432, and AES. Table 6.2 presents a comparison between the resource utilization of GarbledEDA implementations for individual benchmarks and the combined GarbledEDA with a selector. The results indicate that the combined implementation significantly reduces the number of LUT

Table 6.2: Comparison between implementation costs of GarbledEDA (maximum performance) with a selector vs. GarbledEDA of individual benchmarks.

Benchmark	XOR	Other	Inst.	LUT	FF	MUX
C499	104	198	236	86445 (110012)	58582 (83278)	4016 (9105)
C432	18	222	276	91829 (130163)	69104 (97263)	5487 (11390)
AES	2112	576	426	158870 (287334)	102112 (200179)	6896 (16085)
Combination	**2366**	**1053**	**491**	**176134 (290873)**	**264937 (417391)**	**4608 (14936)**

and MUX while slightly increasing FF utilization. This trade-off is expected, as the selector-based approach requires additional storage to maintain the garbled tables and input labels for all included IPs.

As seen in Table 6.2, the use of a selector reduces LUT utilization by 47.3% and 44.8% in ARM and MIPS architectures, respectively. Similarly, the number of MUXes is reduced by 78.9% and 59.2%, demonstrating the efficiency of combining multiple IPs under a unified GarbledEDA instance. However, the FF utilization is slightly increased by 13.3% in ARM and 8.79% in MIPS architectures. This increase is due to the need for additional storage to accommodate the instruction sets and intermediate data corresponding to each IP.

A key advantage of this approach is that it significantly reduces the number of evaluator cores required for execution. Without a selector, implementing GarbledEDA for three separate IPs would necessitate three independent instances, each with its own set of evaluator cores and instruction storage. In contrast, the selector-based design consolidates these resources, allowing a single evaluator core to process multiple IPs dynamically. This leads to improved hardware efficiency and lower FPGA area consumption.

Another benefit of using a selector-based approach is its applicability to real-world use cases where multiple secure IPs need to be simulated or compiled within a single environment. Many modern IC incorporate multiple functional blocks, each designed by different third-party vendors. Ensuring the confidentiality of these blocks while maintaining efficient hardware utilization is a critical challenge. The selector mechanism allows an IC designer to compile and simulate multiple IPs securely without incurring the excessive overhead associated with independent implementations.

From an implementation perspective, incorporating a selector requires careful management of instruction memory and label assignments. Each

IP has a predefined instruction set that must be stored separately, and the selector dynamically determines which instruction set is executed at any given time. The execution flow is adjusted accordingly, ensuring that only the relevant portions of the garbled circuit are evaluated based on the selected IP. This prevents unnecessary computation and minimizes power consumption, making the approach suitable for resource-constrained environments.

It is important to note that while this optimization provides substantial resource savings, it does introduce some trade-offs. One of the main challenges is ensuring that the selector mechanism does not introduce additional latency or processing overhead. However, our results indicate that this overhead is minimal compared to the overall savings in logic and memory utilization. Additionally, the increased FF utilization is an expected consequence of consolidating multiple IPs into a single instance, but it remains within an acceptable range.

Overall, the selector-based GarbledEDA implementation offers a practical and scalable solution for reducing hardware resource overhead while maintaining the strong security guarantees provided by SFE. By consolidating multiple IPs under a single execution framework, this approach enables more efficient utilization of FPGA resources, making it well-suited for secure EDA in large-scale applications.

6.2.9 GarbledEDA with an Improved Hardware Resource Efficiency Evaluation

The implementation of GarbledEDA in its maximum performance mode prioritizes execution speed at the cost of significant resource utilization. However, in practical applications where FPGA resources are constrained, a more hardware-efficient execution strategy is required. To address this, the GarbledEDA framework incorporates an improved hardware resource efficiency mode, which focuses on reducing resource utilization while maintaining the security guarantees of the garbled circuit approach. This optimization is particularly important for large-scale benchmarks where maximum performance execution would result in prohibitive FPGA overhead.

In the improved hardware resource efficiency mode, GarbledEDA takes advantage of a gate-level decomposition approach, where the circuit is broken down into smaller sub-netlists. Rather than evaluating the entire circuit in a single execution cycle, each gate or small group of gates is processed

Table 6.3: Garbled EDA with an improved hardware resource efficiency implementation cost of different benchmarks in ARM(MIPS).

Benchmark	XOR	Other	Inst.	LUT	FF	MUX	OT
8-bit SUM	48	96	30	1941(1775)	823(1406)	152(289)	64
AES	2112	576	426				682
256-bit MULT	1303	1276	2012				3036
C6288	0	2416	4669				4733

separately, reducing the peak memory and computational load at any given time. This approach ensures that only the necessary portions of the design are active during execution, thereby minimizing the logic and memory resources required. Unlike the maximum performance mode, where the garbled instructions corresponding to the entire IP description are loaded and executed at once, this approach sequentially evaluates each garbled instruction, preventing unnecessary hardware overhead.

Since each sub-netlist consists of a small number of neighboring gates from the original design, the evaluation process follows a structured order where each set of encrypted operations is sequentially fed into the evaluation core. The mapping between the sub-netlists and their original design positions is securely handled by the garbler, ensuring that an adversary cannot gain insights into the circuit structure even if they attempt to analyze the order of execution. This methodology aligns with previously proposed hardware-optimized garbled circuit execution frameworks, such as those explored in [173], where netlist partitioning was shown to be effective in reducing execution overhead.

The key advantage of this approach is the significant reduction in FPGA resource utilization. Table 6.3 provides a comparative evaluation of GarbledEDA implementations under the maximum performance mode and the improved resource efficiency mode. The hardware resource costs are shown for benchmarks of varying complexities, including a small-scale computation such as 8-bit SUM, a cryptographic benchmark like AES, a moderate-scale computation such as 256-bit multiplication (MULT), and a large-scale benchmark like C6288. Despite the differences in circuit size, the improved hardware efficiency mode maintains a consistent and predictable resource consumption across all benchmarks.

As seen in Table 6.3, the implementation costs in terms of logic utilization (LUT), FFs, and multiplexers (MUX) remain nearly constant across differ-

ent benchmarks when the improved hardware resource efficiency approach is used. This is because the underlying architecture only evaluates a single instruction per cycle, meaning that the required FPGA logic remains fixed regardless of the circuit size. Unlike the maximum performance mode, which scales resource utilization linearly with the circuit complexity, the improved efficiency mode ensures that resource usage is predictable and manageable.

A key trade-off in this optimization is the increase in the number of OT interactions. Since each gate evaluation requires a separate garbled instruction to be loaded and executed, the number of OTs increases proportionally with the circuit size. For instance, while the 8-bit SUM benchmark requires only 64 OT operations, the AES benchmark requires 682 OTs, and the C6288 benchmark, one of the largest circuits in the ISCAS-85 suite, requires 4733 OTs. This increase in OT operations introduces a slight performance overhead, as each OT requires additional computation and communication between the garbler and the evaluator. However, in applications where FPGA resources are the primary constraint, the trade-off between execution time and resource savings is often acceptable.

Another advantage of the improved hardware efficiency mode is its applicability to scenarios where circuit inputs and outputs are minimal compared to the total number of logic gates. As observed in [173], when the I/O size of a circuit is negligible in comparison to the number of logic gates, this mode achieves optimal security and efficiency trade-offs. Moreover, even for small benchmarks such as 8-bit SUM, the resource-efficient mode results in considerable savings in hardware usage, making it a viable approach for a wide range of secure hardware designs.

The results in Table 6.3 also highlight an important observation regarding architecture selection. The ARM-based implementation of GarbledEDA exhibits slightly higher hardware resource utilization compared to the MIPS-based implementation. This discrepancy is attributed to the differences in how ARM and MIPS architectures handle FF storage and instruction decoding. Specifically, the MIPS-based implementation avoids using DSP blocks, which can be a crucial factor when deploying GarbledEDA in resource-constrained environments. As a result, selecting the appropriate architecture depends on the specific hardware constraints of the target system.

In conclusion, the improved hardware resource efficiency implementation of GarbledEDA provides a significant reduction in FPGA resource consumption while maintaining secure computation guarantees. By adopting a sequential execution model that processes smaller circuit fragments, the

Table 6.4: Comparison between implementation costs of GarbledEDA (maximum performance) vs. GarbledEDA (resource-efficient) for small, moderate, and large benchmarks.

Benchmark	**Inst.**	**OT (Resource Efficient)**	**Time (Sec)**		**Peak Memory (MB)**	
			Maximum Performance	Resource Efficient	Maximum Performance	Resource Efficient
8-bit SUM	30	64	4.9E-5	3E-3	6.8	0.33
AES	426	682	6.2E-5	1.6E-2	51.2	3.54
256-bit MULT	2012	3036	1E-4	7.3E-2	102.4	15.12
C6288	4669	4733	2.3E-4	1.1E-1	25.6	23.66

framework enables practical deployment of secure EDA tools on constrained hardware platforms. The trade-off between resource savings and OT overhead must be carefully balanced depending on the application requirements. For use cases where execution speed is a priority, the maximum performance mode remains preferable, whereas applications with strict resource limitations benefit from the optimized resource-efficient execution model.

6.2.10 GarbledEDA Execution Time and Peak Memory Cost Evaluation

The execution time and memory overhead of GarbledEDA are critical factors in assessing its viability for secure hardware design automation. The evaluation of execution performance is conducted by comparing the maximum performance and resource-efficient implementations across multiple benchmarks. The experiments were performed on a computing setup consisting of an Intel Core i7-7700 CPU running at 3.60 GHz with 16 Gigabyte (GB) of RAM, executing the garbling process, and an ARTIX-7 FPGA board running the Garbled MIPS/ARM evaluator core. The FPGA operates at a 20 MHz clock frequency, ensuring a practical representation of real-world execution constraints.

Table 6.4 presents a detailed comparison of execution time and peak memory usage for both implementations. The execution time results indicate the performance trade-offs introduced by SFE techniques, while the memory footprint analysis provides insights into the hardware resource constraints imposed by different implementation flows.

The maximum performance implementation follows an unrolled execu-

tion flow, where all encrypted instructions and labels are precomputed and transferred to the evaluator upfront. This minimizes interactive overhead by requiring only one OT interaction per input-output transaction. As a result, the execution time remains close to the baseline MIPS/ARM execution time, with a marginal increase due to the additional cryptographic computations involved in garbled circuit evaluation. Each instruction in the maximum performance implementation is executed in 50 nanoseconds (ns), corresponding to a single clock cycle on the FPGA.

In contrast, the resource-efficient implementation follows a sequential execution model, where encrypted gates are evaluated one at a time, significantly reducing the memory footprint. This approach, while beneficial in minimizing logic utilization, introduces additional OT interactions, leading to increased execution latency. Each instruction in the resource-efficient model takes 150 ns, equivalent to three clock cycles per operation. Additionally, since OT is performed per gate rather than per batch of instructions, the overall execution time scales with the number of evaluated gates, making it suitable only for applications where hardware resource constraints outweigh performance requirements.

The impact of OT interactions on execution latency is evident in Table 6.4. The 8-bit SUM benchmark, which consists of only 30 instructions, executes in 4.9E-5 seconds in the maximum performance mode but requires 3E-3 seconds in the resource-efficient mode due to the additional OT interactions. The effect is even more pronounced in complex benchmarks such as AES, where execution time increases from 6.2E-5 seconds in maximum performance mode to 1.6E-2 seconds in the resource-efficient mode. This 260x slowdown highlights the trade-offs between execution speed and hardware resource conservation.

Peak memory usage is another crucial consideration in GarbledEDA evaluation. The maximum performance implementation requires all garbled instructions, encryption labels, and middle wire labels to be preloaded into the FPGA's memory, leading to a significantly larger memory footprint. On average, the maximum performance implementation consumes 5-25x more memory than its resource-efficient counterpart. The AES benchmark, for instance, requires 51.2 MB in maximum performance mode, while the resource-efficient mode reduces this footprint to 3.54 MB. Similarly, the 256-bit MULT benchmark, one of the most computationally demanding workloads, requires 102.4 MB in the maximum performance setup, compared to 15.12 MB in the resource-efficient implementation.

The difference in memory consumption stems from how garbled circuit evaluations are structured in each mode. In the maximum performance implementation, all garbled tables and cryptographic labels are prepared in advance, requiring a large buffer to store intermediate values. Meanwhile, in the resource-efficient model, only a small subset of encrypted instructions are processed at a given time, significantly reducing peak memory usage. However, the increased number of OT interactions in this model results in higher communication overhead and longer execution times.

Furthermore, the number of OT interactions directly influences execution time and memory utilization. In the maximum performance implementation, the total OT overhead is limited to the summation of input and output sizes, while in the resource-efficient implementation, OT overhead scales with the number of evaluated gates. The execution of each OT interaction requires 24 microseconds (µs), adding to the cumulative execution time. Additionally, for each input-output bit, 3.2 kilobytes (kB) of memory are required for OT, contributing to the overall resource demand.

A significant observation from Table 6.4 is that benchmark complexity does not always correlate linearly with execution time and memory cost. For instance, although AES contains significantly more logic operations than the 8-bit SUM benchmark, its execution time and memory footprint do not increase proportionally due to the presence of XOR gates, which benefit from Free-XOR optimizations [214]. In contrast, C6288, a large arithmetic circuit, incurs the highest computational overhead due to its complex gate structure, requiring 4733 OT interactions in the resource-efficient implementation.

The results confirm that the choice between maximum performance and resource-efficient implementations depends on specific application constraints. Applications requiring high-speed execution should opt for the maximum performance mode, while scenarios with tight hardware resource limitations can leverage the resource-efficient model to reduce memory footprint at the cost of longer execution times.

By thoroughly analyzing execution time and memory consumption, this study provides valuable insights into the trade-offs involved in SFE for EDA workflows. The findings highlight the need for a balanced approach that optimizes both computational efficiency and hardware feasibility, making GarbledEDA a practical solution for secure IP compilation and simulation.

The previous sections have presented GarbledEDA, an efficient framework that enables privacy-preserving EDA by leveraging GC and secure computation techniques [157]. By transforming hardware description lan-

guages (HDLs) into secure, functionally equivalent garbled representations, GarbledEDA ensures that sensitive IP remains protected from unauthorized access and adversarial tampering. Through optimizations such as gate-level decomposition and selective circuit evaluation, GarbledEDA effectively balances computational overhead and hardware resource constraints while securely executing EDA operations.

However, as the domain of secure computation expands beyond EDA applications, new challenges arise in securing DL models against malicious adversaries. In modern AI deployments, ML models are often trained or executed in untrusted environments, where adversaries may attempt to manipulate model parameters, insert backdoors, or extract sensitive information through inference attacks [240, 424]. Given the increasing reliance on pre-trained models and outsourced training [267, 316], the risk of backdoor insertion and adversarial weight manipulation has become a critical concern in secure DL inference.

To address these threats, we introduce GuardianMPC, a novel framework that extends the principles of secure computation to protect DL models from malicious adversaries [154]. Similar to GarbledEDA, GuardianMPC is built on GC and employs secure MPC to ensure privacy-preserving execution. However, while GarbledEDA focuses on securing IP blocks in hardware design, GuardianMPC specifically targets DL models by introducing mechanisms for oblivious inference, private training, and backdoor-resilient model execution.

6.3 GuardianMPC: Backdoor-resilient Neural Network Computation

In modern AI applications, deep NN are deployed in untrusted environments where both the training process and the model parameters are vulnerable to backdoor insertion, direct weight manipulation, and architectural tampering [113, 167, 142].

To address these emerging threats, we introduce GuardianMPC, a novel secure MPC framework tailored for NN computation. GuardianMPC extends the principles of secure computation developed in GarbledEDA and adapts them to the domain of DL. By integrating techniques from the LEGO protocol family [280, 108] and applying them to NN inference and private training,

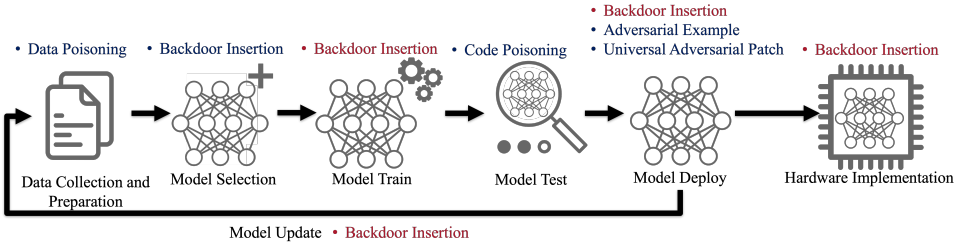


Figure 6.4: Well-known attack types against each stage of the DL pipeline (Inspired by [113]). The red font means that the attacks fall within the scope of this paper. The backdoor insertion during different phases involves architectural backdoor insertion in *Model Selection*, direct weight manipulation in *Model Train*, architectural backdoor insertion and direct weight manipulation in *Model Deploy*, and direct weight manipulation in *Model Update*.

GuardianMPC provides robust protection against both passive and active adversaries. This framework not only ensures the confidentiality of model architectures and weights during inference but also safeguards the training process against malicious modifications and backdoor attacks. In this way, GuardianMPC bridges the gap between secure hardware design and secure DL, offering a comprehensive solution for protecting sensitive NN models in outsourced and untrusted environments. The following sections elaborate on the design, methodology, and evaluation of GuardianMPC, highlighting its advantages and demonstrating its resilience against adversarial attacks.

6.3.1 Backdoor Attacks in DL Pipeline

Figure 6.4 illustrates well-known attack types at various stages of the DL pipeline, including adversarial example attacks [287], universal adversarial patches [268, 359, 278], data poisoning [189, 74], backdoor insertion [142, 74, 167, 135], and outsourcing attacks [142]. Backdoor attacks are especially insidious because a backdoor refers to the intentional insertion of a hidden vulnerability within a model that enables it to perform correctly on most inputs while producing malicious behavior when a specific trigger is present [113, 331]. These backdoors can be injected either during the outsourcing of model training or within pre-trained models. In the pre-trained model scenario, an attacker may modify the model's weights directly [167, 135] so that the trigger activates a malicious sub-task, whereas in an outsourcing scenario the training process itself can be manipulated to

insert such vulnerabilities [113].

Direct Weight Manipulation

Direct weight manipulation involves altering a pre-trained model's weights, either partially or entirely, to embed a backdoor without modifying the training data. This method gives the attacker precise control over the model's behavior and allows the backdoor to remain undetected by standard evaluation metrics [167, 135]. Such modifications can preserve overall performance on legitimate tasks while triggering malicious outputs when specific inputs are presented [113]. Researchers have demonstrated that even minor modifications in weight values can lead to effective backdoor insertion, thereby evading many current detection and removal defenses.

Outsourcing Backdoor Insertion and Architectural Attacks

In outsourcing scenarios, a user who lacks the necessary resources or expertise delegates the training process to an external provider. This creates an opportunity for the provider to perform direct weight manipulation during training while still maintaining high overall accuracy [113]. Additionally, architectural backdoor insertion is possible, where the attacker modifies the NN's structure itself. These modifications are subtle and may involve changes to the network's connectivity or the introduction of malicious sub-circuits that remain dormant until triggered [142, 74]. Hardware backdoor insertion is another form, where alterations are made at the register-transfer level of the underlying accelerator hardware, potentially compromising the security of the DL model [403]. Such architectural modifications can be particularly difficult to detect, as they do not significantly alter the model's performance on standard inputs.

6.3.2 Targets of Malicious Adversaries in Garbled Circuits

In secure MPC protocols that employ GC, a malicious adversary may attempt to compromise the security by constructing incorrect circuits. If an adversary introduces faulty GC without detection, the resulting computation might yield incorrect outputs or leak sensitive information. To mitigate this risk, cut-and-choose protocols are employed; by generating multiple GC

and randomly verifying a subset, the evaluator gains statistical assurance of correctness [228, 226]. Similarly, an adversary may execute an input inconsistency attack by providing different inputs in different instances of the protocol, forcing the protocol to abort and inadvertently leaking information about the evaluator's input [350]. Recent works have addressed these vulnerabilities by incorporating commitment schemes that bind the garbler to a fixed input across multiple circuit instances [108, 281].

6.3.3 Our Adversary Model

Our adversary model considers two primary parties: a user and a NN provider. In an outsourcing scenario, the user sends a detailed NN description to the provider, who returns trained model parameters. The user, despite verifying the model on a held-out validation set, may still face the risk that the provider modifies the NN's weights or architecture after validation. Alternatively, when using a pre-trained model, the user downloads a model that may have been manipulated by the provider to include backdoors. In both cases, the provider can alter the model without significantly affecting its performance on standard inputs, yet causing it to behave maliciously when a specific trigger is present [113]. This situation in secure computation is analogous to the concept of input inconsistency and incorrect circuit construction in MPC protocols. Our framework leverages secure computation techniques to protect against these attacks, ensuring that the evaluator obtains only the intended output while the NN provider cannot extract additional information or insert backdoors.

6.3.4 Similarities between Adversarial Models

The adversarial models used in MPC distinguish between passive adversaries, often called HbC, and active adversaries, also known as malicious. Passive adversaries follow the protocol as specified while trying to learn additional information from the exchanged data [35, 130]. Active adversaries, on the other hand, may deviate arbitrarily from the protocol in an attempt to influence the output or extract confidential information. In DL pipelines, similar adversarial behaviors are observed. For instance, a malicious NN provider may modify a model's weights or structure—actions that correspond conceptually to incorrect garbled circuit constructions or input inconsistencies in MPC protocols [228, 226]. These similarities allow for the application of

secure computation techniques to defend against both classes of adversaries, thereby protecting both the integrity of the computation and the confidentiality of the underlying model [133, 413].

6.3.5 GuardianMPC Flow

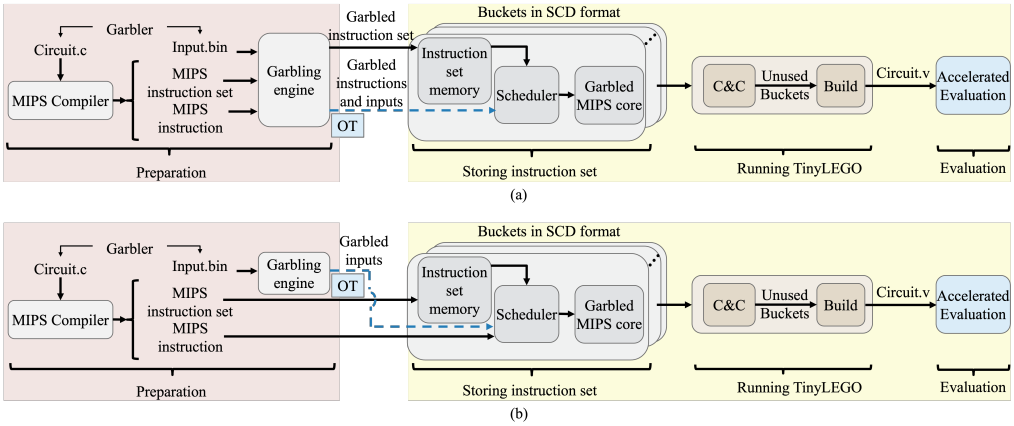


Figure 6.5: A high-level flow of GuardianMPC. The processes highlighted in red and yellow run on the garbler's (NN provider's) and user's machines, respectively. (a) In oblivious inference, garbling the instructions and instruction sets is included in the computation flow to ensure function privacy. (b) In private training, the instructions and instruction sets are not garbled. Instead, the garbler's input (weights) are garbled and obliviously sent to the user via OT.

GuardianMPC extends secure MPC to protect NN models against backdoor attacks and weight manipulation during both private training and oblivious inference. The framework is built on the principles of GC and the LEGO protocol family [280, 108], while also incorporating hardware-based acceleration techniques [156]. In GuardianMPC, the NN provider (garbler) transforms the NN model into a garbled representation that conceals both the model architecture and its parameters. The user, acting as the evaluator, processes this encrypted representation and computes outputs without learning any sensitive details about the model. This secure computation flow is designed to protect both the training and inference phases in untrusted environments.

Figure 6.5 provides an overview of GuardianMPC's architecture. The processes executed on the garbler's machine (NN provider) and those on the user's machine are distinctly separated. In the oblivious inference scenario, the NN provider converts a pre-trained model into a set of low-level instructions, typically using MIPS instructions, and garbles them using Yao's garbled circuit protocol [419]. This transformation ensures that the NN architecture remains confidential while still allowing secure evaluation. The resulting garbled instruction set and cryptographic labels are stored in a dedicated memory unit, reducing interaction overhead between the parties during evaluation. The user, acting as the evaluator, securely obtains garbled input labels via OT [133], ensuring that only the necessary labels corresponding to private inputs are revealed. The garbled instructions are then executed on hardware platforms such as FPGA, which leverage specialized arithmetic logic units and LUT for optimized performance.

Conversely, in private training, the model is trained iteratively on encrypted weights to prevent unauthorized access or modifications. Unlike the oblivious inference scenario, where the function privacy mechanism garbles the entire instruction set, the private training setup keeps the instruction set in plaintext while encrypting the model weights. This allows for efficient backpropagation updates while still preserving security. Secure exchange of garbled weights between the garbler and evaluator occurs via OT, preventing an adversary from gaining insights into the evolving model parameters.

6.3.6 Protection Against Malicious Adversaries

GuardianMPC employs multiple layers of protection against adversarial threats that target either private training or oblivious inference. These threats include backdoor insertion attacks, weight manipulation, and unauthorized architecture modifications. Figures 6.6 and 6.7 demonstrate GuardianMPC's robustness against such threats.

Figure 6.6 illustrates the GuardianMPC defense mechanism in private training. A malicious NN provider might attempt to inject a backdoor by altering weight values or modifying the circuit representation. To counteract this, GuardianMPC integrates cut-and-choose verification, where a subset of GC is randomly opened and checked for correctness. The evaluator verifies these circuits against expected values, ensuring that any tampering is detected with high probability. If inconsistencies are found, the entire computation is rejected. This process guarantees that the training data remains

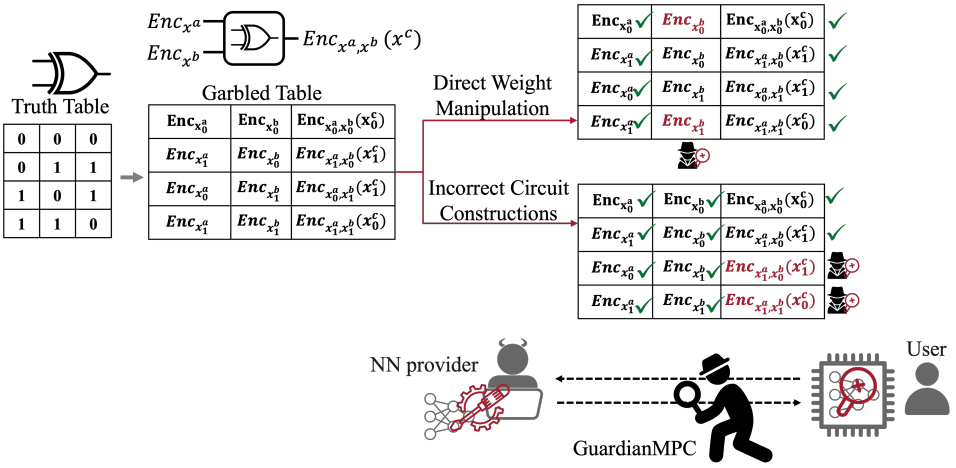


Figure 6.6: GuardianMPC protects NN against malicious modifications during private training. The framework employs a cut-and-choose mechanism to verify the consistency of GC, preventing an attacker from inserting backdoors through weight manipulation or incorrect circuit construction. The verification process, based on random selection and cryptographic commitments, ensures that any tampering is detected with high probability.

private and the model parameters remain untampered throughout the training phase.

Figure 6.7 highlights how GuardianMPC ensures the confidentiality of pre-trained models during oblivious inference. A malicious NN provider could attempt to embed hidden functionalities or modify the architecture to extract information from the user's input. To prevent this, GuardianMPC garbles the entire instruction set and encrypts both model parameters and control flow. As a result, the evaluator cannot modify the computation process nor access any intermediate values. Even if an adversary gains partial access to the computation flow, the use of cryptographic commitments and secure evaluation ensures that the model remains secure against architectural backdoor attacks.

6.3.7 Efficient Execution with Hardware Acceleration

One of the key optimizations in GuardianMPC is its use of hardware acceleration to improve the efficiency of garbled circuit evaluation. The garbled MIPS evaluator [156] is designed to process encrypted instructions in parallel,

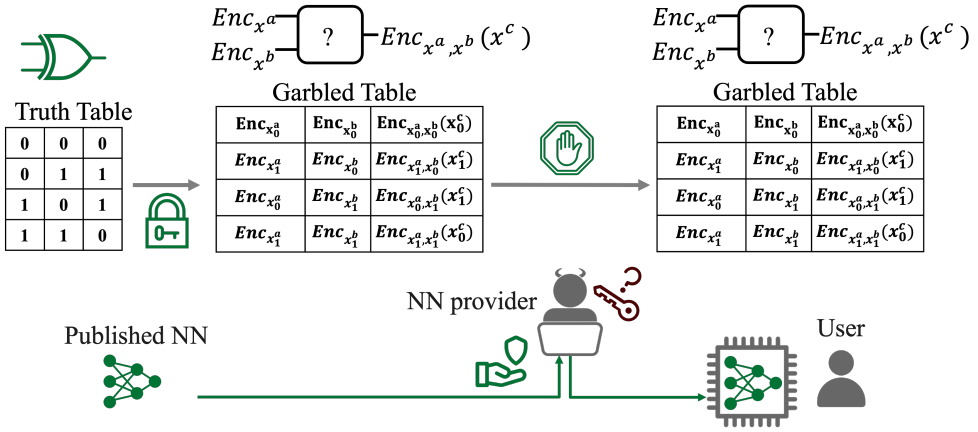


Figure 6.7: In the oblivious inference scenario, GuardianMPC ensures the privacy of pre-trained NN by encrypting the model architecture and weights. The garbling of the NN prevents a malicious provider from modifying the model, as the evaluator is unable to decrypt the garbled inputs and tables, thereby preserving the integrity of the model even in the presence of adversarial behavior.

significantly reducing execution latency.

During evaluation, the evaluator retrieves the encrypted input labels and processes the garbled instructions using a parallel execution framework. FPGA-based evaluation enables fast arithmetic computations, leveraging LUT and on-chip memory to minimize latency. The evaluation phase is further optimized using garbled arithmetic logic units (ALUs) [156], allowing for efficient encrypted operations.

Unlike traditional garbled circuit implementations, which rely on software-based computation, GuardianMPC takes full advantage of hardware primitives to accelerate secure inference and training. This design minimizes overhead while ensuring that the security guarantees of GC remain intact. The result is a system that not only enhances privacy but also maintains computational feasibility for large-scale deep-learning models.

The preceding section detailed the design and operational flow of GuardianMPC, highlighting its secure computation framework, garbling mechanisms, and FPGA-based acceleration. With a structured methodology that ensures both function privacy and efficient execution, GuardianMPC presents a compelling approach for secure NN inference and training. To validate its practical feasibility, we now present a comprehensive experimental evaluation.

This evaluation examines the performance of GuardianMPC across different NN architectures and compares it against existing state-of-the-art secure computation frameworks. The following section outlines the experimental setup, benchmark models, and execution-time comparisons, demonstrating the efficiency and security trade-offs of GuardianMPC in real-world applications.

6.3.8 Experimental Setup

To evaluate the performance of GuardianMPC, we conduct experiments in two distinct secure computation scenarios: oblivious inference and private training. The experimental setup comprises two computational entities—the garbler, responsible for preparing and garbling the NN model, and the evaluator, which executes the secure computation on encrypted inputs. Each entity is equipped with hardware tailored for its role in the protocol.

The garbler machine operates on a HPC platform featuring an Intel Xeon Silver 16-core CPU running at 2.5 GHz, an NVIDIA RTX-A4000 GPU, and 128 GB of RAM. The system runs on Linux Ubuntu 20, providing a stable and compatible environment for executing GuardianMPC. Given the extensive cryptographic computations involved in garbling circuits, a large memory capacity and multi-core processing are essential to optimize the pre-processing phase. Additionally, the GPU is leveraged for matrix multiplications in NN computations, reducing the burden on the CPU. The compiled version of the TinyLEGO framework [52] is used for baseline comparisons, as it provides a standard implementation of SFE.

For private training, the evaluator operates on a system with an Intel Core i7-7700 CPU running at 3.60 GHz and 16 GB of RAM. This hardware configuration is sufficient for executing GC, as the evaluator does not engage in complex cryptographic operations but instead processes encrypted labels using the garbled instruction set. The execution of GuardianMPC's garbled computations is further enhanced through FPGA-based acceleration. The evaluator runs its inference tasks on an Artix-7 FPGA device (XC7AT100T) using the Xilinx Vivado Design Suite 2021 [409]. The FPGA is programmed to accelerate the execution of GC, significantly reducing the computational overhead typically associated with software-based secure MPC implementations.

To facilitate low-latency communication between the garbler and evaluator, the two machines are connected via a high-speed local area network

(LAN). The network configuration ensures an average delay of 0.2 milliseconds and a bandwidth of 1 GB/s, closely resembling the experimental setup used in SecureML [267]. The use of a high-bandwidth, low-latency network infrastructure is crucial for reducing communication bottlenecks, a common challenge in secure computation protocols that require frequent data exchanges between parties. The efficiency of GuardianMPC relies on its ability to minimize network overhead while securely transmitting garbled instructions and encrypted labels.

The experimental evaluation follows a structured methodology for comparing GuardianMPC against state-of-the-art secure computation frameworks, including TinyLEGO [108] and SecureML [267]. These comparisons focus on execution time, computational efficiency, and security guarantees in both the preparation and online phases. The primary goal is to assess the feasibility of GuardianMPC in real-world applications where NN are either evaluated on encrypted data (oblivious inference) or trained on private datasets (private training) without exposing model parameters.

In the private training scenario, GuardianMPC ensures that NN training remains confidential by garbling the model weights and securely exchanging encrypted weight updates through OT [133]. The model undergoes forward and backward propagation in an encrypted form, preventing any leakage of sensitive information. The evaluator processes encrypted gradients, while the garbler ensures that weight updates remain concealed throughout the training process.

During oblivious inference, the pre-trained model is garbled in its entirety, preventing the evaluator from learning any details about the NN's architecture or parameters. The encrypted model is then executed using SFE, ensuring that both the user's input and the model remain private. The FPGA-based acceleration in GuardianMPC optimizes the execution of garbled instructions, reducing latency and improving computational efficiency compared to software-based secure inference frameworks [280].

By structuring the experimental setup to closely match real-world deployment conditions, GuardianMPC demonstrates its ability to facilitate secure NN computation while achieving an optimal balance between privacy and performance. The results obtained from this experimental configuration provide insights into the trade-offs associated with SFE and highlight the advantages of integrating hardware acceleration to enhance the efficiency of secure NN processing.

Benchmark Models

GuardianMPC is designed to support secure NN inference by evaluating models on garbled data while preserving both input privacy and function hiding. This ensures that a user can obtain classification results on their private input without revealing any information to the NN provider while also preventing the evaluator from learning any details about the NN architecture or its parameters. The ability to execute inference securely prevents adversarial manipulation and model inversion attacks, making GuardianMPC suitable for privacy-preserving NN applications.

To assess the performance of GuardianMPC, we tested it on multiple NN models, including multi-layer perceptrons (MLP) and CNN. These models were selected to evaluate the system's scalability and efficiency in executing both fully connected and convolutional architectures, which are fundamental to modern DL applications.

For the experimental evaluation, we considered the following models:

MLP: We used two benchmark MLP architectures widely adopted in secure computation literature. The first, referred to as BM1, consists of an input layer with 784 neurons, three hidden layers with 1024 neurons each, and an output layer with 10 neurons, trained on the MNIST dataset [96]. The second, BM2, is a shallower MLP used in SecureML [267], featuring a single hidden layer with 128 neurons and a square activation function [238]. These MLP architectures allow for direct comparison with previous secure computation frameworks, particularly those that do not support function hiding.

CNN: To evaluate GuardianMPC's ability to handle deeper networks with convolutional operations, we implemented two CNN architectures. The first, BM3, is a seven-layer CNN trained on CIFAR-10, commonly used in privacy-preserving inference benchmarks [318, 67, 238]. The second, LeNET-5, is a classical CNN architecture originally designed for digit classification on MNIST [221], also implemented in TinyGarble2 [173]. The use of CNNs allows us to analyze the efficiency of GuardianMPC in executing complex, large-scale NN with different types of layers.

Each of these benchmark models was evaluated in three different configurations to analyze the performance trade-offs between security and efficiency:

- **Baseline (Unprotected Inference):** The NN models were evaluated without any security measures, meaning that inference was performed

in plaintext without garbling or function hiding. This serves as a lower bound on execution time, as no cryptographic overhead is introduced.

- **TinyLEGO Framework (SFE-Based Inference):** The models were implemented using the TinyLEGO SFE framework [52], which allows privacy-preserving inference but does not support function hiding. This represents the standard approach for secure inference in prior works, where only the user's input is hidden.
- **GuardianMPC (PFE-Based Inference):** The models were implemented using GuardianMPC, which supports function hiding by applying PFE techniques. This prevents adversaries from extracting information about the NN structure or parameters, providing stronger security guarantees.

The evaluation results compare GuardianMPC with the other frameworks in terms of execution time, considering both the preparation phase (garbling and circuit setup) and the online phase (secure inference execution). The subsequent subsections provide a detailed breakdown of the execution time across different models, highlighting the computational efficiency of GuardianMPC and the trade-offs introduced by function hiding.

Execution Time

The evaluation of GuardianMPC's execution time is critical in understanding its efficiency in secure NN inference. This section presents a comprehensive analysis of the time complexity across different benchmark models, comparing GuardianMPC with existing frameworks such as TinyLEGO [52], MiniONN [238], and SecureML [267]. The results are divided into two primary phases: the preparation phase, which includes garbled circuit construction and OT setup, and the online phase, where encrypted inputs are processed, and the NN is evaluated.

BM1.

Table 6.5 presents a detailed comparison of execution time for BM1, an MLP-based model implemented in different frameworks. The results highlight that TinyLEGO achieves better performance in the preparation phase due to its less complex function-hiding mechanism. Specifically, TinyLEGO completes the preparation phase in 4188.42 ms, whereas GuardianMPC requires 6650.29 ms. This additional cost in GuardianMPC arises from the

Table 6.5: Comparison between the execution time of BM1 (the numbers in boldface indicate the best results).

Metric	**Baseline**	**TinyLEGO [52]**	**GuardianMPC**
Security	None	SFE	PFE
# of AND gates	N/A	2098	2098
Preparation Phase [ms]			
Construction	N/A	**3591.8**	5912.29
BaseOT	N/A	**579.03**	719.88
Random Generation	N/A	**17.59**	18.12
Total Preparation	N/A	**4188.42**	6650.29
Online Phase [ms]			
Communication	N/A	4219.72	**801.19**
Checking	N/A	**492.7**	608.94
Building	N/A	**6.09**	6.23
Evaluation	N/A	43.84	**3.26**
Total Online	**3.43**	4762.36	**1419.62**

integration of function hiding, which ensures that the model architecture and weights remain hidden from the evaluator.

Despite this increased preparation time, GuardianMPC significantly improves the online execution phase. The total online time of GuardianMPC is 1419.62 ms, which is considerably lower than TinyLEGO's 4762.36 ms. This advantage is due to GuardianMPC's optimized evaluation phase, which benefits from FPGA acceleration and efficient execution of garbled instructions. The evaluation step in GuardianMPC takes only 3.26 ms, compared to 43.84 ms in TinyLEGO, representing an improvement of approximately 13.44×. This acceleration stems from GuardianMPC's ability to execute garbled instructions in parallel, leveraging hardware optimizations that significantly reduce computation time.

BM2. Table 6.6 presents the execution time results for BM2, an MLP-based model with a single hidden layer. Compared to MiniONN [238], TinyLEGO, and GuardianMPC, the preparation phase in MiniONN is significantly faster at only 880 ms due to the absence of function hiding. However, MiniONN's online time is higher than GuardianMPC due to its reliance on SIMD-based optimizations rather than secure MPC principles.

GuardianMPC achieves the lowest online execution time of 425.91 ms, which is slightly higher than TinyLEGO but significantly lower than MiniONN. The evaluation phase in GuardianMPC takes only 2.8 ms, compared to TinyLEGO's 34.76 ms. The function-hiding mechanism in GuardianMPC introduces a slight computational overhead, but it is mitigated by hardware-

Table 6.6: Comparison between the execution time of BM2 (the numbers in boldface indicate the best results).

Approach	Security	Preparation [ms]	Online [ms]			Total [ms]
			Comm.	Evaluation	Total	
Baseline	None	N/A	N/A	N/A	**171.39**	**171.39**
MiniONN [238]	SFE	**880**	N/R	N/R	400	1280
TinyLEGO [52]	SFE	1961.24	70.57	34.76	392.39	2353.63
GuardianMPC	PFE	2323.96	**10.23**	**2.8**	425.91	2749.87

Table 6.7: Comparison between the execution time of BM3 (the numbers in boldface indicate the best results).

Approach	Security	Preparation [s]	Online [s]			Total [s]
			Comm.	Evaluation	Total	
Baseline	None	N/A	N/A	N/A	**9.72**	**9.72**
MiniONN [238]	SFE	472	N/R	N/R	72	544
EzPC [67]	SFE	N/R	N/R	N/R	N/R	265.6
TinyLEGO [52]	SFE	**913**	40.22	7.29	154	1067
GuardianMPC	PFE	1191	**23.73**	**1.18**	208	1399

based acceleration.

BM3: CNN with Seven Layers To assess the scalability of GuardianMPC, we evaluate a seven-layer CNN model (BM3), commonly used in secure inference studies such as Chameleon [318], EzPC [67], and MiniONN [238]. The execution time results are provided in Table 6.7.

GuardianMPC's preparation time for BM3 is 1191 seconds, which is slightly higher than TinyLEGO (913 seconds). However, the trade-off results in a significantly faster online execution time of 208 seconds, compared to TinyLEGO's 154 seconds. This improvement is due to GuardianMPC's efficient circuit execution methodology, where function hiding minimizes redundant operations, reducing bottlenecks in GC evaluations.

Compared to MiniONN, GuardianMPC incurs a higher preparation overhead but achieves a faster evaluation by leveraging optimized circuit parallelism. Notably, GuardianMPC reduces the evaluation time by 6.17× compared to TinyLEGO, confirming its efficiency in processing CNN architectures securely.

LeNET-5: CNN for MNIST Classification Table 6.8 details the execution time for the LeNET-5 CNN model, a well-established benchmark in

Table 6.8: Comparison between the execution time of LeNET-5 [221] (the numbers in boldface indicate the best results).

Approach	Security	Preparation [s]	Online [s]			Total [s]
			Comm.	Evaluation	Total	
Baseline	None	N/A	N/A	N/A	**1.73**	**1.73**
TinyGarble2 [173]	SFE	N/R	N/R	N/R	91.1	91.1
TinyLEGO [52]	SFE	**387.85**	37.02	2.61	49.23	419.95
GuardianMPC	PFE	429.23	**6.91**	**0.4**	36.95	466.25

secure NN inference [221]. This model is widely adopted in frameworks such as TinyGarble2 [173].

GuardianMPC's preparation time for LeNET-5 is 429.23 seconds, slightly higher than TinyLEGO's 387.85 seconds due to the additional security enhancements required for function hiding. However, GuardianMPC achieves a significantly reduced online execution time of 36.95 seconds, compared to 49.23 seconds in TinyLEGO.

More importantly, GuardianMPC reduces the evaluation time to just 0.4 seconds, which is 6.52× faster than TinyLEGO. This improvement is crucial for real-time applications where low-latency inference is required.

Evaluation Acceleration on FPGA

One of the key contributions of GuardianMPC is its ability to accelerate the evaluation phase of the LEGO protocol [280], which is crucial for oblivious inference. The hardware accelerator exploits hardware optimization techniques and the FPGA's parallel processing capabilities to achieve significant speedups in function evaluation. Unlike software-based secure computation frameworks, which rely on general-purpose CPUs for processing GC, GuardianMPC integrates a specialized execution unit on an FPGA to process garbled instructions in a highly parallelized manner.

A fundamental bottleneck in traditional secure computation is the evaluation of GC, particularly in the online phase where every AND gate requires evaluating a garbled truth table. The LEGO protocol improves efficiency by reducing the size of garbled tables; however, its reliance on serial execution of operations limits scalability. GuardianMPC addresses this by leveraging an FPGA-based processing unit that executes multiple garbled gates in parallel, significantly reducing execution time.

The acceleration impact is evident in Tables 6.5, 6.6, 6.7, and 6.8, where

GuardianMPC consistently achieves lower online evaluation times compared to TinyLEGO. Specifically, GuardianMPC accelerates the evaluation phase by 13.44× and 12.41× for BM1 and BM2, respectively. Similarly, for CNN benchmarks, the evaluation acceleration factors are 6.17× and 6.52× for BM3 and LeNET-5, respectively. These results indicate that GuardianMPC's FPGA-based evaluation mechanism significantly reduces computational overhead in function evaluation.

Several factors contribute to this acceleration:

- *Parallel Gate Evaluation:* Unlike CPU-based implementations that process gates sequentially, GuardianMPC's FPGA core enables concurrent evaluation of multiple garbled gates. This massively parallel execution reduces the latency of large circuits, making it particularly beneficial for deep NN.

- *Optimized Garbled Instruction Processing:* GuardianMPC translates NN operations into a sequence of MIPS instructions, which are garbled and stored in an optimized instruction memory. This structured approach minimizes redundant recomputation and ensures efficient retrieval of garbled labels.

- *Pipelined Execution:* The FPGA evaluator is designed with a pipelined architecture where different stages of garbled circuit evaluation execute simultaneously. This pipeline enables overlapping operations, reducing idle cycles and improving throughput.

- *Reduced Memory Bottleneck:* GuardianMPC implements an optimized memory interface that minimizes the number of memory accesses during garbled circuit evaluation. Instead of frequently fetching data from external memory, the FPGA-based evaluator efficiently processes garbled instructions within on-chip memory, reducing data transfer overhead.

These optimizations make GuardianMPC particularly suitable for privacy-preserving inference tasks, where minimizing the latency of secure computation is critical. By shifting the computationally intensive garbled circuit evaluation to FPGA hardware, GuardianMPC achieves substantial reductions in online execution time while maintaining function privacy. The results validate the effectiveness of FPGA acceleration, demonstrating its potential for large-scale privacy-preserving ML applications.

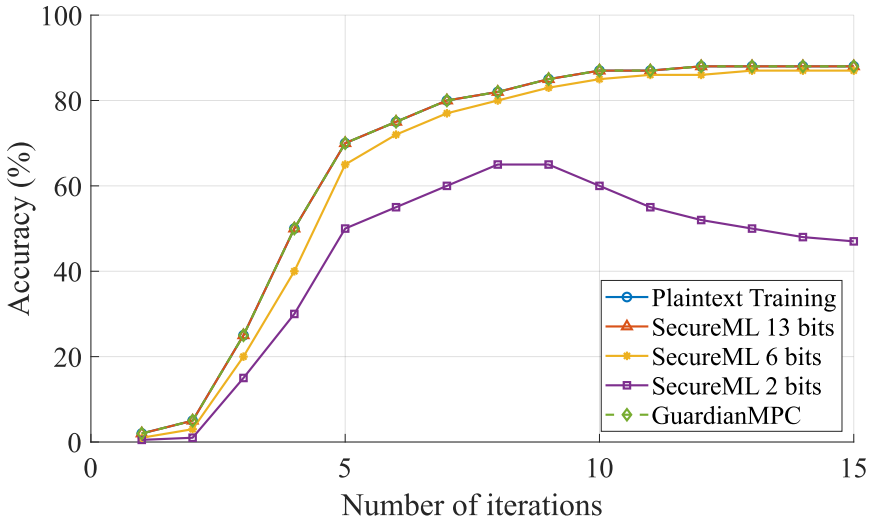


Figure 6.8: Comparison of accuracy over the first 15 iterations between plaintext training, SecureML [267] at various bit precisions (13, 6, and 2 bits), and GuardianMPC trained on MNIST [96] dataset.

Accuracy Evaluation

To evaluate the impact of GuardianMPC on NN accuracy, we conducted private training on the MNIST dataset [96] and compared the results against plaintext training performed in PyTorch [176], as well as against SecureML [267], which employs quantized fixed-point arithmetic to balance efficiency and security. The objective was to determine whether GuardianMPC introduces any accuracy degradation due to its function-hiding properties and secure computation mechanisms.

The experiment involved training an MLP with two hidden layers of 128 neurons each, using ReLU and square activation functions. The training was conducted over 15 epochs with a batch size of 128, matching the SecureML configuration to ensure a fair comparison. After completing the training process with GuardianMPC, the resulting model weights were extracted and applied to the same MNIST test set used for plaintext evaluation in PyTorch. The evaluation focused on whether GuardianMPC's secure computation affected the predictive performance of the NN.

Figure 6.8 presents a comparison of accuracy trends over the first 15 iterations for different settings. The figure shows the accuracy achieved in

plaintext training using PyTorch, which serves as the ideal benchmark. It also includes the accuracy results from SecureML at different bit precisions of 13-bit, 6-bit, and 2-bit fixed-point arithmetic. The GuardianMPC accuracy curve is also plotted to demonstrate its performance in privacy-preserving training. The comparison highlights that GuardianMPC maintains the same accuracy as plaintext training, unlike SecureML, which experiences varying degrees of accuracy loss depending on the bit precision used.

The accuracy loss in SecureML arises due to the use of quantized fixed-point arithmetic, which requires truncation and polynomial approximations for non-linear functions such as ReLU. At 13-bit precision, SecureML's accuracy remains close to plaintext training, but at 6-bit and especially 2-bit precision, significant accuracy degradation is observed. The results indicate that as lower bit precisions are used, the model struggles to approximate continuous activation functions accurately, leading to convergence issues and reduced classification performance. This highlights a fundamental trade-off between computational efficiency and model accuracy in privacy-preserving training frameworks.

GuardianMPC avoids these accuracy losses by preserving full-precision floating-point computations throughout the training process. Unlike SecureML, which relies on polynomial approximations for activation functions, GuardianMPC directly translates high-level NN operations into garbled MIPS instructions. This method ensures that operations such as matrix multiplications and activation functions are executed without approximations or truncations, thereby retaining the model's original predictive capabilities. Since GuardianMPC does not rely on fixed-point arithmetic, the trained model maintains accuracy identical to plaintext training without the need for special numerical adaptations.

Another advantage of GuardianMPC is its function-hiding property, which ensures that neither the model's structure nor its parameters are leaked during computation. Traditional secure computation schemes for training, such as SecureML, require function exposure to the evaluator for correct execution. In contrast, GuardianMPC garbles not only the model weights but also the instruction set, preventing adversarial inference about the model's architecture. This additional security layer does not come at the cost of accuracy, as demonstrated by the identical performance between GuardianMPC and plaintext training.

The ability of GuardianMPC to maintain accuracy while ensuring strong privacy guarantees makes it a suitable framework for privacy-preserving DL

applications. Unlike quantized approaches that sacrifice accuracy for performance, GuardianMPC provides robust function privacy without compromising predictive reliability. This is particularly important in sensitive applications such as medical diagnosis, financial analysis, and federated learning, where accuracy is paramount and secure model execution is required.

The results further emphasize the significance of maintaining full floating-point precision in secure computations. While some prior works in secure ML introduce approximations to improve computational efficiency, GuardianMPC ensures that all floating-point operations are faithfully represented in the secure computation process. This preserves the integrity of the trained model and eliminates concerns related to accuracy degradation, even when working with deep NN.

These findings demonstrate that GuardianMPC achieves a balance between security and accuracy. While many existing privacy-preserving frameworks introduce numerical approximations that affect accuracy, GuardianMPC retains the original model's precision, ensuring that privacy-preserving NN training does not compromise inference quality. The ability to securely train and evaluate NN without accuracy loss is a crucial advantage in deploying privacy-aware ML models in real-world applications.

6.4 Discussion

The Need for Secure Computation in Hardware Design

The field of hardware security has seen significant advancements due to the growing complexity of IC design and the globalization of semiconductor manufacturing. IP protection is a major concern as modern design flows involve multiple entities, including third-party foundries, design service providers, and verification vendors. This distributed model of hardware development exposes sensitive design assets to security threats, making it crucial to establish secure computational frameworks that allow collaboration without compromising proprietary information.

One of the fundamental challenges in securing hardware design is the necessity of protecting computation itself. Unlike traditional cryptographic techniques that ensure data confidentiality when stored or transmitted, hardware security demands that operations performed on sensitive data remain secure throughout the computation process. This requirement is particularly

important in EDA workflows, where IC designers rely on third-party tools for essential tasks such as synthesis, verification, and simulation. Designers need assurance that their circuits remain protected when processed by external EDA tools, while tool providers must safeguard their proprietary algorithms from being reverse-engineered or misused by adversarial users.

Traditional encryption-based solutions, including IEEE P1735, were introduced to address these security concerns by encrypting circuit designs before they are processed. However, the encryption used in these approaches does not protect computation itself, meaning that once decrypted for processing, the design becomes vulnerable to reverse engineering and potential IP theft. Moreover, several attacks have demonstrated that IEEE P1735 is susceptible to cryptanalysis techniques that allow adversaries to extract critical design details, undermining its effectiveness as a long-term security solution.

Limitations of Existing Approaches

Several methods have been proposed to protect hardware designs during computation, but each comes with inherent limitations. SFE has been explored as an alternative to encryption-based protection, enabling computations to be performed on encrypted data. While this technique enhances security, it does not provide function privacy, meaning that proprietary EDA tools remain exposed to adversarial analysis by users attempting to extract or replicate their underlying algorithms. Furthermore, many SFE-based solutions introduce significant computational overhead, making them impractical for large-scale IC design flows.

IEEE P1735, while widely adopted for securing IP within the EDA industry, has been shown to be ineffective against adversaries who exploit vulnerabilities in its encryption schemes. A major drawback of IEEE P1735 is its reliance on a trusted execution model, where it assumes that EDA tools themselves are secure and trustworthy. If an EDA tool is compromised, it can bypass encryption protections and gain unrestricted access to proprietary designs. This trust assumption is problematic, especially in modern hardware design workflows where third-party tool vendors may not always be fully trusted.

Secure computation techniques based on HE and MPC have also been explored for EDA applications. However, these approaches generally incur excessive performance costs due to the complexity of encrypted arithmetic op-

erations. Additionally, HE struggles with efficiently implementing non-linear operations, such as those required in synthesis and verification processes. These inefficiencies render HE impractical for real-world EDA applications that demand high-speed computations.

Garbled EDA: Privacy-Preserving Electronic Design Automation

With the increasing complexity of modern semiconductor design and the globalization of the chip manufacturing supply chain, ensuring the security and privacy of IP has become a critical challenge. The current landscape of EDA is dominated by proprietary tools and cloud-based computing services that, while powerful, introduce risks associated with IP theft, untrusted third-party access, and malicious modifications. Traditionally, hardware designers have relied on legal agreements and encrypted design files to safeguard their IP, but these approaches are insufficient in an era where sophisticated adversaries can exploit vulnerabilities in the design flow itself.

Garbled EDA provides an alternative paradigm by leveraging secure MPC techniques, specifically PFE, to protect IP throughout the design and verification phases. Unlike conventional encryption-based protections, which primarily safeguard data at rest and during transmission, Garbled EDA ensures that computation itself remains secure, preventing unauthorized access to both the designer's inputs and the proprietary algorithms used in EDA tools.

One of the fundamental advantages of Garbled EDA over traditional security measures, such as the IEEE P1735 standard for IP protection, is its resilience against CAD tool hacking. IEEE P1735 relies on encryption-based licensing schemes, which, while effective against casual piracy, are vulnerable to SCA, decryption exploits, and unauthorized modifications within the EDA software itself. In contrast, Garbled EDA eliminates the reliance on TEE by ensuring that all design operations are performed on encrypted representations of the circuit, preventing malicious actors from learning any meaningful information about the IP.

Garbled EDA also addresses a key limitation of SFE-based EDA approaches, which assume that function privacy is not a requirement. In an SFE setting, while the designer's inputs (e.g., standard cell libraries or layout constraints) remain private, the CAD tool's internal computations are exposed to the user. This transparency introduces the risk of reverse engineering proprietary synthesis, placement, and routing algorithms. By imple-

Table 6.9: Garbled EDA vs. existing methods.

Problems	IEEE P1735	SFE-based EDA	Garbled (PFE) EDA
Supports privacy of designer inputs (e.g., PDK)	✗	✓	✓
Safe from CAD/EDA tool hacking	✗	✓	✓
Handles untrusted CAD/EDA vendor	✗	✗/✓	✓
Supports CAD/EDA tool privacy	✗	✗	✓

menting PFE, Garbled EDA ensures that not only are the designer's inputs protected, but the EDA tool's functionality remains confidential, securing both parties involved in the computation.

The effectiveness of Garbled EDA in securing EDA workflows is summarized in Table 6.9, which compares its capabilities against IEEE P1735 and SFE-based EDA. Garbled EDA uniquely provides comprehensive protection against untrusted CAD tools, safeguarding both designer and tool vendor privacy while maintaining functional correctness.

By integrating PFE into the EDA workflow, Garbled EDA enables a new era of privacy-preserving hardware design. It allows IP owners to collaborate with untrusted third-party design tools without exposing their proprietary circuit designs, and it enables CAD tool vendors to protect their optimization algorithms from reverse engineering. These capabilities make Garbled EDA an essential advancement in the security of semiconductor design and verification.

GuardianMPC: Secure Computation for Privacy-Preserving Machine Learning

As ML becomes an integral part of modern computing, privacy and security concerns surrounding NN models have gained increasing attention. Traditional NN pipelines assume a trusted execution environment where both model parameters and input data are freely accessible to the computing infrastructure. However, in practical deployments, users frequently rely on third-party cloud services or outsourced computational resources, which introduces risks of model inversion, adversarial modifications, and backdoor insertion.

GuardianMPC addresses these concerns by implementing a secure computation framework based on PFE, ensuring that both the model structure and the input data remain confidential during inference and training. Un-

Table 6.10: Comparative analysis of various secure ML approaches.

Approach	Maintains Accuracy	Scalability	Real-time Performance	Model Independency
Homomorphic Encryption [125, 194]	No	Yes	No	Yes
Zero-Knowledge Proofs [222, 233]	Yes	No	No	Yes
Trigger Reconstruction [398]	Yes	No	No	Yes
Model Inspection [414]	Yes	No	No	No
GuardianMPC	Yes	Yes	Yes	Yes

like standard garbled circuit-based approaches, which expose function details to the evaluator, GuardianMPC extends privacy guarantees by garbling the model at the instruction level. This prevents adversaries from learning the architecture and internal parameters of the NN while still enabling efficient and accurate computations.

One of the main advantages of GuardianMPC is its ability to accelerate secure computation using FPGA-based hardware. SFE techniques often suffer from performance bottlenecks due to their reliance on cryptographic operations and extensive data exchanges. By leveraging FPGA parallelism, GuardianMPC reduces execution time for both inference and training, making privacy-preserving ML practical for real-world applications.

Table 6.10 compares GuardianMPC with alternative privacy-preserving approaches, including HE, ZKPs, and trigger reconstruction techniques. Unlike HE, which requires computationally expensive polynomial approximations for non-linear activation functions, GuardianMPC processes NN layers directly without introducing numerical inaccuracies. Compared to trigger reconstruction methods that attempt to detect and remove backdoors post hoc, GuardianMPC proactively prevents backdoor insertion by concealing the model structure throughout training and inference.

Chapter 7

Side-Channel Attacks Against Hardware Implementations

7.1 Motivation

SCA pose a critical threat to secure computation, allowing adversaries to extract sensitive information by exploiting unintended physical leakages such as power consumption, EM emissions, and timing variations [209, 210, 112]. Unlike cryptographic attacks that aim to break encryption schemes mathematically, SCA exploit the implementation of secure algorithms, often bypassing theoretical security guarantees. Such attacks have been extensively demonstrated against widely used cryptographic primitives [80] and secure MPC protocols [397], raising concerns about the real-world security of privacy-preserving computation techniques.

Timing attacks remain a significant concern in the context of GC and secure computation [209, 153]. Variations in execution time due to conditional branching, circuit depth, and memory access patterns create exploitable timing side-channels. Recent work has shown that even optimized garbled circuit constructions exhibit timing vulnerabilities, making it imperative to design countermeasures that eliminate timing discrepancies. The impact of such attacks extends to privacy-preserving ML, where adversaries may infer model parameters, input values, or decision boundaries by analyzing execution patterns [253].

Another major category of SCA is power and EM-based analysis, which allows adversaries to recover sensitive data by correlating power traces with

cryptographic operations [210, 6]. Secure computation frameworks implemented in hardware, particularly those using FPGA or custom accelerators, are susceptible to such attacks due to the predictability of switching activity in logic gates [257]. While masking and hiding techniques provide partial protection [247], they often introduce substantial performance overheads, motivating alternative countermeasures.

To address these vulnerabilities, recent advancements have proposed novel mitigation techniques that enhance side-channel resilience in secure computation frameworks. HWGN^2 introduces a side-channel-protected execution model for NN inference using GC, ensuring that both the architecture and weights remain confidential while mitigating power and EM leakage [156]. Meanwhile, Garblet extends secure MPC to chiplet-based architectures, distributing computations across multiple hardware units to reduce information leakage from a single compromised component [158]. These approaches, which will be discussed in detail later, provide complementary solutions for securing privacy-preserving computation against physical attacks.

7.2 Bake It Till You Make It: Heat-induced power leakage from masked NN

Secure hardware implementations rely on robust countermeasures against SCA, which exploit unintentional information leakage from physical devices to extract secret data [247, 210]. One of the most overlooked threats in secure computation is the impact of temperature variations on power leakage, which can undermine masking-based protections. While classical SCAs rely on passive monitoring of power consumption, EM radiation, or timing fluctuations [112, 335], recent studies have demonstrated that thermal variations themselves can serve as an attack vector, amplifying leakage in masked cryptographic implementations [253].

7.2.1 Heat-Induced Power Leakage in Secure Computation

The effect of temperature on side-channel security has long been studied, but its impact on masking countermeasures has received limited attention. As modern FPGA rely on complementary metal-oxide semiconductor (CMOS)

technology, their performance characteristics are highly temperature dependent [195, 150]. Increasing temperature alters key circuit parameters such as carrier mobility, threshold voltage, and propagation delay, all of which affect power consumption patterns [110, 106].

The primary sources of heat in FPGA-based accelerators are either external heat sources or internal heat generation from frequent memory operations [253]. While external heating techniques have been previously exploited for FIA [28], the recent Bake It Till You Make It [253] study introduced an adversarial model where internal heat generation alone is sufficient to induce side-channel leakage. By strategically increasing FPGA temperature through high-frequency memory write operations, an adversary can enhance the power consumption variations in masked NN implementations, thereby enabling first-order SCA that would otherwise be mitigated [253].

At the core of this phenomenon is the interplay between temperature, circuit delay, and power consumption. As demonstrated in prior work, temperature fluctuations impact the propagation delay of LUT and FF within FPGA [277, 10]. The variations in delay cause unintended power fluctuations, which, when measured over time, reveal statistically significant leakage traces in hardware implementations of masked computations [345, 91]. This contradicts the fundamental assumption of masking, which relies on independent power consumption of individual shares.

Understanding the impact of heat on side-channel security Masking is one of the most widely used countermeasures against SCAs, aiming to break the correlation between power consumption and sensitive data by representing secrets as randomized shares [247]. However, temperature-dependent changes in gate delays disrupt this assumption, leading to unintended leakage between masked shares [91].

Consider a first-order Boolean masking scheme, where a sensitive variable x is split into two uniformly random shares x_1 and x_2 such that $x = x_1 \oplus x_2$. Ideally, the power consumption of circuits processing x_1 and x_2 should be independent, ensuring that side-channel leakage remains minimal [247]. However, when operating under elevated temperatures, these circuits experience timing variations due to fluctuating threshold voltages [110]. These variations introduce glitches and timing misalignment between the masked shares, leading to observable power consumption differences, even at the first order [106, 407].

Our study in Bake It Till You Make It [253] experimentally demonstrated this phenomenon by analyzing the power traces of a masked NN accelerator implemented on an FPGA. The results indicated that masking fails to provide first-order security under high-temperature conditions, making NN accelerators vulnerable to practical attacks [253]. Given that temperature variations are inherent to real-world computing environments, this finding has significant implications for secure hardware design.

Implications for Secure Computation The existence of heat-induced power leakage poses a critical challenge for secure computation. Unlike conventional SCAs, where adversaries require specialized equipment to measure power fluctuations or inject faults [28], thermal SCA can be launched remotely by manipulating computational workloads [253]. This makes them particularly dangerous in multi-tenant FPGA cloud environments and secure DL accelerators, where multiple users share the same physical device.

Moreover, existing countermeasures designed to mitigate classical SCAs—such as dual-rail logic, randomized instruction scheduling, and DPA protections—are ineffective against thermal SCA [247, 335]. As shown in Bake It Till You Make It [253], masking-based protections alone cannot prevent leakage under temperature-induced variations. Therefore, addressing this new class of vulnerabilities requires rethinking secure hardware designs to incorporate thermal-aware mitigation strategies, such as dynamic voltage scaling, adaptive cooling mechanisms, and delay-compensated masking implementations [253, 91].

This study highlights the urgent need for temperature-resilient secure computation frameworks. The following sections will further explore internal heat generation mechanisms, experimental attacks, and mitigation techniques designed to counteract these newly discovered threats to secure hardware implementations.

7.2.2 Inducing Leakage through Internal Heat Generators

Given the above discussion, it is tempting to increase the temperature of the circuit to induce leakage. This can be achieved using a climate chamber to operate the device at higher temperatures, as demonstrated in prior work [91]. Nevertheless, an internal heat generator (HG) presents several ad-

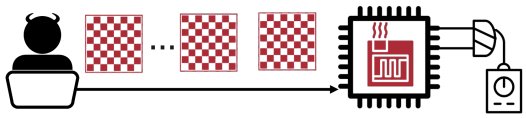


Figure 7.1: The adversary relies on the fact that at high temperatures, the power consumption associated with different shares is no longer independent of each other. In this regard, the adversary takes advantage of the memory allocated to store the inputs and, by writing alternating '0' and '1' patterns, attempts to increase the operating temperature of the FPGA and detect first-order leakage.

vantages, particularly its feasibility in remote attacks and cost-effectiveness. Even if the internal HG is unintentionally triggered, such as through high utilization due to normal operations, its effect on masking countermeasures is worth investigating. This is especially relevant for NN accelerators, which extensively utilize block RAMs (BRAM) for storing inputs and intermediate computations [73, 103]. The presence of masked NNs further amplifies BRAM utilization, making these devices highly susceptible to heat-induced side-channel leakage.

The core idea behind heat-induced leakage is to generate heat within the FPGA by flipping the input image, i.e., writing alternating '0' and '1' patterns into memory to toggle the corresponding BRAMs. This increases dynamic power consumption and, subsequently, chip temperature. The adversary, unaware of the NN's internal structure, simply feeds specially crafted inputs to the system at every clock cycle, as illustrated in Figure 7.1. This attack exploits an internal HG to make masked NN implementations vulnerable to side-channel analysis. Prior research [151, 4] has shown that one of the primary sources of heat generation in FPGA is the extensive use of BRAMs and FF pipelines, as well as frequent read/write operations within these components.

To evaluate the impact of such HGs on masked NNs, we consider ModuloNET [101], a NN accelerator that, like many others [99, 102], stores input images and activation function outputs within BRAMs. This architectural feature allows an adversary to generate heat by continuously feeding flipping images into the FPGA, forcing the BRAMs to repeatedly toggle. Although the adversary cannot directly manipulate the masked computations inside the FPGA, they can significantly impact the thermal environment by streaming input data with alternating bit patterns. The BRAM utilization in Modu-

loNET reaches 51.38% of the available Artix-7 FPGA resources [101], making it highly susceptible to thermal SCA.

To maximize the efficiency of the internal HG, an adversary must toggle multiple bits per cycle. This can be achieved by writing patterns of alternating '1's and '0's into BRAMs, ensuring that a large number of memory cells experience rapid transitions. Unlike prior HG techniques that rely on pipeline-based memory access [151, 4], this attack leverages parallel toggling, aligning with the design principles of NN accelerators.

The rationale behind exploiting an internal HG to break masking countermeasures stems from its impact on power consumption dependencies. The heat generated inside the FPGA results in increased power dissipation, disrupting the assumption that power consumption of individual masked shares remains independent. As highlighted in prior research [91], the power consumption of one function operating on a masked share can influence the power drawn by other simultaneous computations. This dependency is amplified at elevated temperatures, leading to increased side-channel leakage.

Empirical evidence from De Cnudde et al. [91] confirms that side-channel leakage from masked designs intensifies within a temperature range of 50°C to 70°C. While their study employed a climate chamber to induce this effect, we investigate whether similar conditions can be achieved internally through an adversary-controlled heat generation process. The attack follows a two-phase approach: first, the adversary repeatedly injects flipping images to heat the FPGA, and second, they capture power traces from the masked implementation under elevated temperatures to detect first-order leakage.

Comparison with the most relevant heat generation methods. De Cnudde et al. [91] were among the first to analyze the effects of extreme temperature on masking countermeasures. However, their approach relied on external heat sources, whereas we demonstrated that an internal HG can induce comparable leakage without external intervention. Similarly, the technique proposed by Alam et al. [11] leveraged write collisions in dual-port BRAMs to cause voltage drops and increase FPGA temperature. While effective for fault injection, this method requires dual-port BRAMs, limiting its applicability. In contrast, our approach operates on single-port BRAMs, making it widely applicable to various FPGA designs. More importantly, while prior work focused on FIA, our findings demonstrate that heat-induced power fluctuations can serve as a side-channel vulnerability, making masked

NNs susceptible to power analysis attacks.

The results from this study underscore the necessity of temperature-aware security designs for FPGA, particularly for masked computations. Traditional countermeasures designed for room-temperature operations may fail under elevated thermal conditions, necessitating new techniques to ensure resilience against heat-induced power leakage. Future work should explore mitigation strategies that dynamically adjust circuit timing or incorporate thermal management techniques to minimize side-channel leakage.

To effectively evaluate the impact of heat-induced side-channel leakage in masked NN, it is essential to validate the attack through experimental analysis. The previous section detailed how an adversary can manipulate input patterns to induce internal heat generation, leading to observable power fluctuations. However, demonstrating the feasibility of this attack requires a controlled experimental setup to measure leakage and assess its implications on masking countermeasures. In the following section, we outline the hardware and software configurations used to implement the attack, describe the methodology for inducing and measuring temperature-driven side-channel leakage, and present a comprehensive analysis of the collected data to highlight the vulnerabilities introduced by internal heat generation in secure computations.

7.2.3 Experimental Results

Measurement Setup

To evaluate the effectiveness of the proposed HG and its impact on first-order leakage, a masked NN is implemented on an Artix-7 FPGA, embedded within the CW305 board. The target design follows secure synthesis constraints to maintain share independence and prevent unintended optimizations that could influence power leakage. Specifically, LUT sharing is disabled, the placement of masked shares is enforced using `DONT_TOUCH` attributes, and logic optimization is disabled in Vivado 2021 [409].

To ensure a stable and noise-free power supply, a BK Precision 9130 low-noise DC power source delivers a consistent 1V voltage to the FPGA board. Power traces are captured using CW Husky and CW Lite, leveraging synchronous capturing to align data acquisition with the design's operational frequency. The segmentation feature of CW Husky enhances the trace collection speed by enabling batch captures in a single communication cycle. To

Table 7.1: Hardware resource allocation in masked NN implementation, evaluating BRAM-based heat generation.

Resource	Used	Utilization (%)	Available
LUT	15807	24.93	63400
FF	7782	6.13	126800
BRAM	**131**	**97.03**	135

reduce high-frequency noise, a mini-circuits SLP-30+ low-pass filter is placed between the CW305 and CW Husky.

The design operates at a frequency of 10 MHz, ensuring synchronization between the FPGA clock and the capture process. This mitigates potential timing inconsistencies such as jitter-related noise during trace collection.

Internal Heat Generation and Power Consumption To assess the effects of heat-induced leakage, an example masked NN accelerator is considered. Specifically, ModuloNET [101] is chosen due to its reliance on masking for security against first-order SCA. ModuloNET, like many other NN accelerators, utilizes BRAM for storing masked inputs and intermediate computations, making it particularly susceptible to adversarial heat generation through frequent memory accesses.

As part of the evaluation, the output of the PRNG is continuously monitored to ensure that cryptographic operations remain functionally correct and that any observed leakage is due to power fluctuations rather than computational errors.

Table 7.1 summarizes the FPGA resource utilization in the implementation. The high BRAM utilization of 97.03

This experimental setup provides a controlled environment for analyzing temperature-induced side-channel leakage, with a focus on masked NN that rely on memory-intensive operations. The following sections present the methodology, data collection, and observed leakage patterns.

Die Temperature Measurement

The CW305 target board is a customized board for effective side-channel evaluation; however, it lacks system monitoring sensors for real-time temperature assessment. To perform precise thermal measurements, we use the PYNQ-Z1 board with an Artix-7 FPGA (package FTG256), which provides

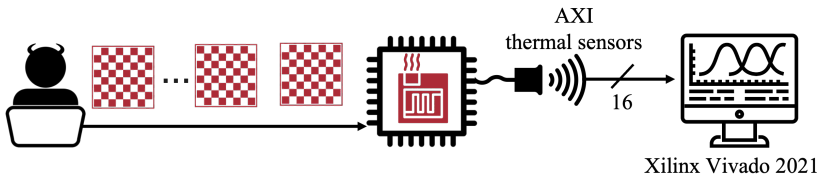


Figure 7.2: Experimental setup used to perform the thermal test.

access to XADC thermal sensors. The experimental setup for temperature evaluation is shown in Figure 7.2.

The XADC module enables 16-bit temperature readings via an analog-to-digital converter (ADC), which is accessed through the Xilinx Vivado 2021 local server. To communicate with the XADC module, we use the PYNQ-Z1 AXI streaming port, configured with a refresh period of 1 second. A key consideration in our experimental setup is avoiding heat generation from unintended sources. Previous studies have used an embedded MicroBlaze processor to store temperature data, but since this processor generates additional heat, we opted for a lightweight approach using Xilinx Vivado TCL scripting to log temperature data with minimal external interference.

To evaluate the impact of our HG, we conduct two sets of experiments:

- Providing a normal image input with minimal bit transitions.
- Feeding an alternating "0" and "1" pattern into memory to maximize dynamic power consumption.

Each experiment runs for 3,600 seconds, collecting temperature samples at 1-second intervals. To ensure unbiased results, we allow the FPGA to cool down for 60 minutes between experiments, following guidelines from prior thermal studies [151, 4].

The results of these experiments are illustrated in Figure 7.3. When processing a normal image, the FPGA temperature stabilizes at 42.1 °C. However, when executing the flipping input pattern, which induces frequent read/write operations in BRAM, the temperature rises sharply to 72.9 °C. This 30.8 °C increase confirms that our internal heat generation technique can significantly elevate the die temperature, creating an attack scenario where masking countermeasures may fail.

Validation Using Xilinx Power Estimator To further verify the accuracy of our experimental temperature measurements, we leverage the Xilinx

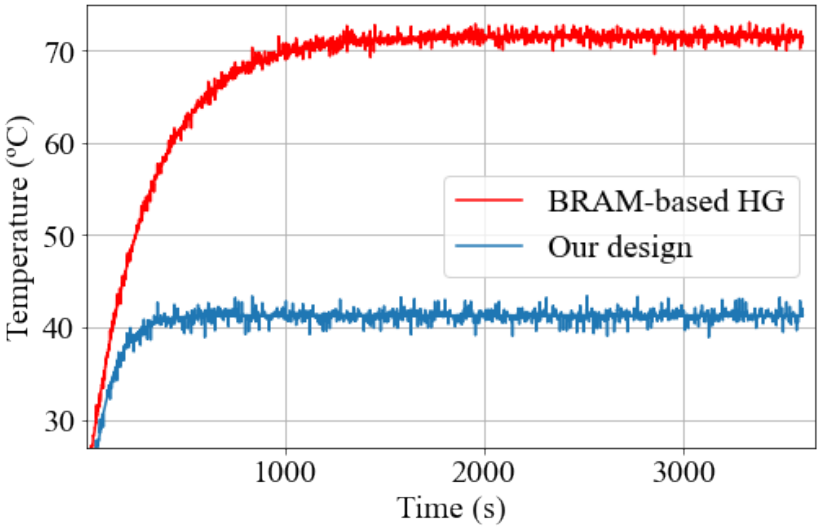


Figure 7.3: Measured temperature for ModuloNET when processing a normal image versus when executing the internal BRAM-based HG. The induced thermal increase demonstrates how internal computation alone can raise the die temperature, impacting masking security.

Power Estimator (XPE) tool [182], which models die temperature based on FPGA bitstream characteristics and ambient thermal conditions. When setting the ambient temperature to 25 °C, XPE estimates a die temperature of 40.2 °C, aligning closely with our hardware results (Figure 7.3). Furthermore, if we set the ambient temperature to 61.5 °C, XPE predicts a die temperature of 70 °C, further confirming that our heat generation method achieves comparable thermal conditions to an externally heated FPGA.

These results highlight the feasibility of inducing significant temperature fluctuations through controlled memory operations alone, without requiring external heat sources.

Temperature Influence on Delay of FPGA Components

The effectiveness of masking countermeasures relies on the assumption that power consumption of different shares remains independent. However, temperature variations introduce timing misalignments that can break this assumption. As discussed earlier, increased temperature alters propagation delays in FPGA components, which can cause masked shares to interact

Table 7.2: Propagation delay variations in FPGA components under different temperatures.

Temperature (°C)	Delay (ps)	
	Single FF	Chain of 10 LUTs
25	793	85
61.5	783	77

unexpectedly, leading to first-order leakage.

To quantify this effect, we conduct an experiment to measure the impact of temperature on two fundamental FPGA components:

- FF: Measuring clock-to-Q propagation delay.
- LUT: Measuring delay propagation through a chain of 10 inverters.

For the FF experiment, we measure the delay between the clock signal and the FF output, ensuring minimal routing delays by placing the FF close to the clock source. For the LUT chain, we use 10 inverters, since the delay of a single LUT is too small to measure accurately with our oscilloscope. The output pin transitions are recorded using a WavePro 254HD 2.5 GHz high-definition oscilloscope [381] with 20 GS/s sampling resolution.

The results of our delay measurements at 25 °C and 61.5 °C are presented in Table 7.2.

A key observation from these results is that both FF and LUT delays decrease as temperature increases. Specifically, the FF delay is reduced by 8 ps, and the LUT chain delay decreases by 10 ps when the temperature rises from 25 °C to 61.5 °C. This non-linear behavior confirms that thermal effects can systematically alter circuit timing, reinforcing the argument that temperature fluctuations introduce unintended dependencies in masked computations.

Implications for Side-Channel Security The observed variations in propagation delay raise significant concerns for secure hardware implementations. Since masking countermeasures rely on precise synchronization between shares, even minor timing discrepancies can create exploitable leakage. Our results demonstrate that temperature-induced changes in circuit timing can be systematically leveraged by an attacker to enhance side-channel leakage.

Moreover, while external voltage control techniques have been used to induce similar effects [128, 91], such approaches require adversarial access

to power management settings. In contrast, our findings highlight that temperature-induced leakage can be exploited passively by simply controlling input data patterns, making it a far more practical and stealthy attack vector.

7.2.4 Leakage Detection

After verifying the impact of giving flipping images on the operating temperature of the FPGA embodying ModuloNET, we investigate how the resulting temperature rise can affect the first-order leakage. The goal of experiments done in this regard is to understand whether a first-order secure design, i.e., our design of ModuloNET, exhibits first-order leakage if flipping images are fed into it and, thus, increases the operating temperature. For this purpose, we compare the t-scores calculated for traces collected from the design when providing normal (not flipping) and flipping images to BRAM-based input storage.

In these experiments, before collecting traces, we first wrote the "0" and "1" alternating patterns into the memory to reach a high die temperature. More precisely, after about 20 minutes (1,200s), the temperature becomes almost stable and reaches its maximum value. Note that an adversary interested in detecting the leakage needs neither this information nor access to the sensor/monitoring system, as she can simply give the flipping images for hours to ensure the die temperature is high.

Furthermore, we should highlight that we start with several tens of thousands of traces to see how many traces of the first-order leakage are detectable after capturing. In this regard, we collected 2M traces from the device in our experiments. Note that throughout this section, the number of traces refers to the total number of fixed and random traces, i.e., collecting 2M traces means that 1M fixed and 1M random traces are collected.

DPA Attack and Key Extraction

After detecting the leakage, the next question is whether the adversary can leverage the leakage to extract secrets. To answer this, we apply DPA. DPA stands as a prominent side-channel attack wherein attackers exploit power consumption patterns to extract secret information [210].

To investigate whether the leakage induced through our HG can be exploited, we launched four DPA attacks against ModuloNET. First, we exam-

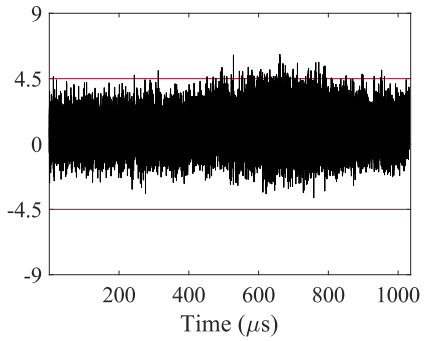


Figure 7.4: First-order leakage detection when the heat generator (HG) is enabled, showing t-score values exceeding the threshold after 2M traces.

ine whether a first-order DPA can break the security of ModuloNET when the PRNG is disabled, but without enabling the HG. This serves as a baseline to ensure that masking is correctly implemented. After confirming this, we analyze how enabling the HG can induce leakage exploitable by DPA. To further validate our results, we also conduct a second-order DPA to determine if increasing the number of traces allows a successful attack. The distinguisher used in all cases is Pearson correlation with a confidence level of 99.99% as suggested in [247].

We obtained the intermediate values from ModuloNET stored in layer BRAMs and registers between bias addition and masked output for the hidden and output layers. These attack vectors for launching DPA are chosen according to the points where the t-score exceeds the threshold, namely around $500\mu s$ for the hidden layer and $750\mu s$ for the output layer.

First-Order DPA with and Without PRNG and HG

We first evaluate ModuloNET with PRNG turned off to observe if a first-order DPA is feasible. As expected, with PRNG disabled, all masked values are unmasked, making first-order DPA highly successful. Our results indicate that the DPA correlation exceeds the threshold around $540\mu s$ for the hidden layer and $830\mu s$ for the output layer, confirming the vulnerability of the unprotected design [101].

We then analyze the scenario with PRNG enabled but without enabling the HG. Under these conditions, the correlation remains below the threshold, verifying the effectiveness of the masking protection against first-order

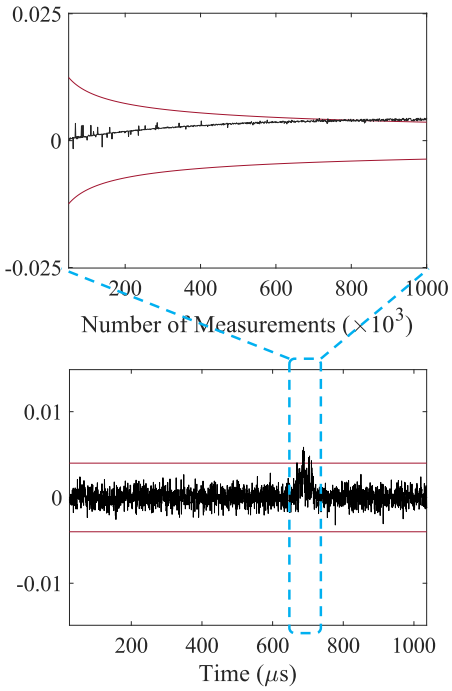


Figure 7.5: First-order DPA results on ModuloNET hidden layer with HG enabled and PRNG on, showing correlation peaks for successful key recovery.

DPA attacks even with up to $500K$ traces. This confirms that the properly masked ModuloNET implementation is resistant to first-order attacks in normal conditions.

Next, we investigate whether a second-order DPA can break the masking scheme under normal conditions (PRNG on, HG off). We follow the second-order DPA methodology proposed in [286, 257] and observe that $500K$ traces are sufficient to achieve a successful second-order attack. The leakage points identified in these results align closely with the time points identified through t-score analysis.

DPA with PRNG on and HG Enabled

After testing the design under normal conditions, we evaluate how enabling the HG affects its security. When the HG is turned on, we observe a significant increase in first-order leakage, making first-order DPA viable even with PRNG enabled. Our results demonstrate that an adversary can successfully

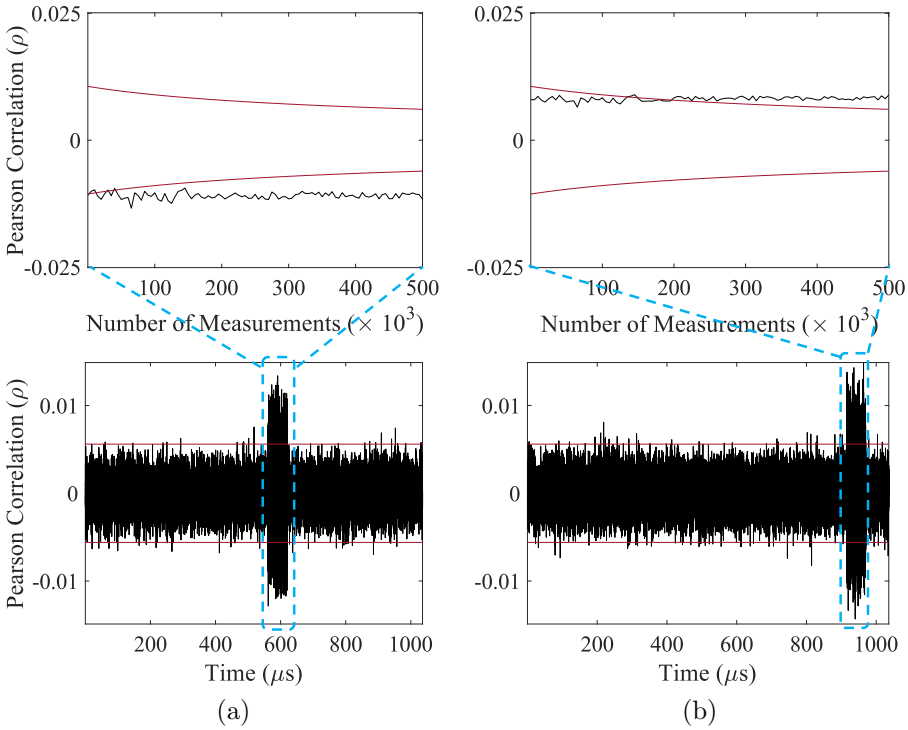


Figure 7.6: Results for the second-order DPA for $500K$ traces against the (a) hidden and (b) output layer of ModuloNET with HG on.

extract secret values with $880K$ traces. Compared to the case without HG, where no successful attack was possible, this highlights the effectiveness of heat-induced leakage in breaking masking protections.

7.2.5 Key Guesses and Attack Success Rate

To further analyze the effectiveness of our heat-based leakage exploitation, we evaluate the success rate (SR) of correct versus incorrect key guesses. We examine how well the first-order DPA attack distinguishes between correct and incorrect weight values for ModuloNET's hidden and output layers. Figure 7.7 illustrates the attack's effectiveness.

Figure 7.7 shows that the correlation for the correct key guess exceeds the threshold around $500\mu s$ for the hidden layer and $875\mu s$ for the output layer. Meanwhile, all incorrect key guesses remain below the threshold, con-

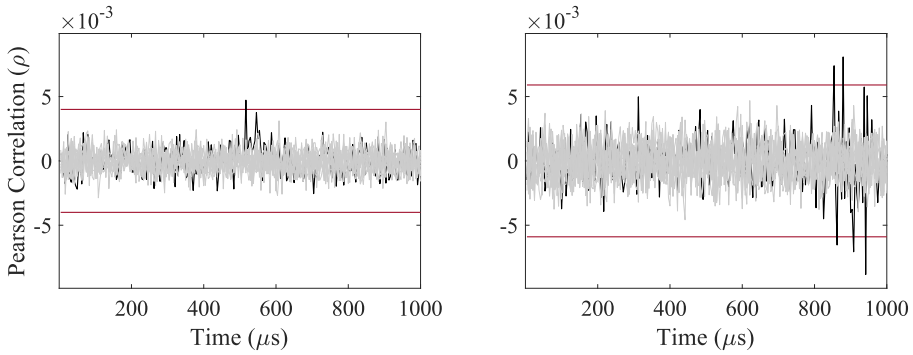


Figure 7.7: First-order DPA against ModuloNET with HG on at (a) hidden for $1M$ traces and (b) output layer for $500K$ traces (gray lines for wrong weight guesses and black line for correct weight guess).

firming that the attack correctly identifies the secret values. These results further validate that heat-induced leakage can be successfully leveraged to compromise masked NN implementations.

7.2.6 Implications for Secure Hardware Design

The findings presented in this work expose a fundamental limitation in the effectiveness of masking countermeasures when subjected to temperature-induced side-channel leakage. As modern FPGA and other hardware accelerators continue to be deployed in critical security applications, it is imperative to reassess their resilience against heat-induced vulnerabilities. This section discusses the broader implications of our results and outlines key considerations for designing robust hardware that can withstand temperature-induced side-channel threats.

One of the most significant takeaways from our study is the realization that masking techniques, while effective against conventional SCA, are highly susceptible to environmental variations such as temperature fluctuations. The core assumption that power consumption of masked shares remains independent is no longer valid when heat causes timing misalignment and glitch propagation [91]. This necessitates a paradigm shift in the design of cryptographic and ML accelerators, where security evaluations must extend beyond traditional SCA resistance and include environmental factors such as

heat and voltage fluctuations [210].

Another critical insight is that heat-induced leakage does not require adversarial access to voltage or clock controls, making it a more accessible attack vector than traditional fault injection techniques. This means that even remotely operated systems, such as cloud-hosted FPGA or edge computing devices, can be vulnerable if they execute memory-intensive workloads that naturally generate heat over time. The ability to induce leakage purely through workload manipulation represents a significant threat, as it allows adversaries to compromise security without requiring direct physical modifications [151, 4].

To mitigate the risks posed by heat-induced leakage, several countermeasures must be considered. One potential approach is adaptive frequency scaling and thermal monitoring, where hardware dynamically adjusts clock speed and voltage levels based on real-time temperature measurements. By detecting excessive temperature rises and adjusting operational parameters accordingly, designers can help prevent timing misalignment and glitch propagation that lead to first-order leakage. However, such mechanisms must be carefully designed to avoid introducing new side-channel vulnerabilities related to frequency scaling.

Another promising countermeasure is enhanced placement and routing constraints for masked implementations. Our results indicate that the physical placement of logic elements plays a crucial role in determining how heat affects masking security. By enforcing strict placement constraints that minimize temperature-induced timing variations, hardware designers can reduce the impact of environmental factors on side-channel security [345]. This approach is particularly relevant for FPGA, where designers have some degree of control over component placement.

Additionally, power-aware masking techniques should be explored to address heat-induced vulnerabilities. Unlike traditional masking, which assumes independent power consumption of shares, power-aware masking explicitly accounts for variations in power and timing caused by temperature fluctuations. This could involve incorporating randomized timing adjustments, dynamic voltage compensation, or alternative sharing schemes that are more resistant to environmental variations [195, 106].

From an architectural perspective, hardware-level countermeasures such as dedicated thermal management units and secure power distribution networks should be integrated into future security-critical processors and accelerators. These units can actively monitor and regulate temperature, ensur-

ing that masked computations remain secure even under extreme operating conditions. Furthermore, thermal isolation techniques—such as strategically placing security-critical components away from high-power regions—can help minimize the impact of localized heating effects.

The *Bake It Till You Make It* study [253] demonstrated how temperature fluctuations, even those generated internally by circuit components, can compromise masked implementations, leading to unexpected leakage. These findings underscore the challenges in securing hardware accelerators against SCA, even when conventional countermeasures like masking are in place.

To address these challenges, more robust approaches have been proposed, extending beyond masking and into the realm of cryptographic secure computation techniques. One such approach is HWGN^2 [156], which leverages SFE and GC to provide inherent resilience against power SCA. Unlike traditional masking, which relies on statistical independence of shares, HWGN^2 fundamentally transforms the execution model, preventing direct leakage of intermediate values. By adopting garbled circuit-based processing within a specialized hardware framework, HWGN^2 offers a scalable and efficient countermeasure, particularly for DL accelerators deployed in security-critical applications.

This shift from statistical countermeasures (masking) to cryptographic countermeasures (GC) represents a fundamental rethinking of side-channel resilience. The next section explores HWGN^2's design and implementation, highlighting its advantages in securing DL models against power SCA.

7.3 HWGN^2: Side-channel Protected Neural Network through Secure and Private Function Evaluation

7.3.1 Adversary Model

NN, particularly those deployed in security-sensitive applications, present valuable assets that require protection. These assets include the NN architecture, hyperparameters, and trained parameters, all of which are crucial for maintaining model integrity and accuracy [29]. Furthermore, in various real-world applications, such as medical diagnostics and defense-related systems, the input data itself contains highly sensitive information [261]. As a

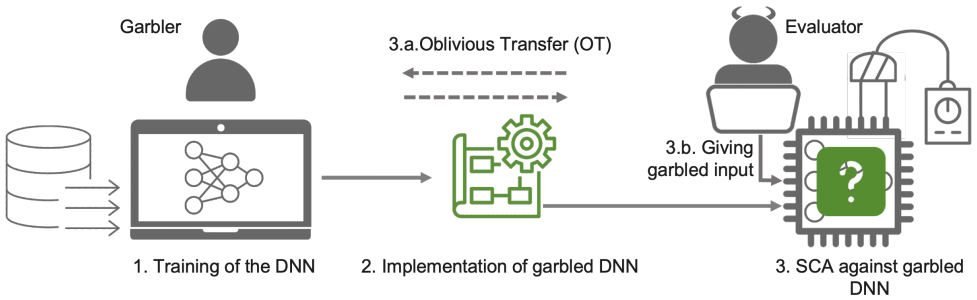


Figure 7.8: HWGN^2 framework: The process begins with training the NN as done for a typical DL task. The second step corresponds to the implementation of the garbled NN hardware accelerator along with running the OT protocol. The accelerator is delivered to the end-user, who attempts to collect the side-channel traces with the aim of extracting information on NN hardware acceleration (architecture, hyperparameters, etc.).

result, ensuring both the confidentiality of user inputs and the privacy of the NN itself is a critical requirement.

To frame our security discussion, we adopt the standard definitions of *security* and *privacy* from the SFE and PFE literature [35]. These definitions allow us to precisely characterize the attack vectors and security guarantees associated with HWGN^2.

HWGN^2 Adversary Model

In our work, we primarily consider the **HbC adversary model**, where the adversary plays the role of the evaluator (Bob), while the NN provider acts as the garbler (Alice) [226, 35] (see Figure 7.8). This adversary has access to the garbled circuit representation of the NN and attempts to extract secret information during the inference phase.

Following state-of-the-art security models [101, 102], HWGN^2 enforces a security framework where:

- The NN is trained offline by the garbler, who maintains full control over the model's hyperparameters and architecture.
- The hardware implementation of HWGN^2 strictly encompasses the **evaluator engine**, meaning that neither the garbling module nor encryption mechanisms are exposed on hardware.

- The evaluator operates with *garbled inputs* prepared offline, preventing direct exposure of sensitive intermediate computations.

7.3.2 Side-Channel Attack Scenario

A core security concern in hardware-based DL inference is vulnerability to power and EM SCA [334, 439]. The adversary in this setting can exploit physical leakage to infer hidden parameters, even without direct access to the model's architecture. The attack process typically follows these steps:

1. The evaluator supplies chosen inputs to the NN accelerator.
2. Power or EM traces are collected during inference, either via direct physical access or remotely.
3. These traces are analyzed using techniques such as DPA [210], Template Attacks [69], or CPA [56] to recover sensitive parameters.

7.3.3 HWGN^2 Countermeasures Against SCA

HWGN^2 mitigates the risk of SCA through SFE techniques, ensuring that intermediate computations never leak meaningful data. Unlike conventional masking approaches, which rely on statistical independence of shares, HWGN^2 employs:

- **Garbled circuit-based evaluation**, which obfuscates computation flows and eliminates deterministic power consumption patterns.
- **Constant-time execution models**, preventing adversaries from deducing information based on timing variations.
- **Structured memory access patterns**, reducing vulnerability to cache, memory-based SCA.

Unlike previous side-channel countermeasures for NN, HWGN^2 inherently supports resiliency against FIA and cache/memory-based attacks, further strengthening its applicability in real-world secure inference scenarios.

By adopting this adversary model and security framework, HWGN^2 ensures robust protection against both passive (HbC) and active (malicious) adversaries, making it a viable solution for privacy-preserving DL inference.

7.3.4 Core Architecture of HWGN^2

Garbled Circuit-Based Computation

At the heart of HWGN^2 lies a SFE approach, specifically utilizing GC to facilitate privacy-preserving inference. The garbling process ensures that intermediate computations do not leak sensitive information through power consumption, EM emissions, or other side-channel sources [35]. According to the garbling protocol G, the NN model is represented as a Boolean circuit consisting of logic gates, each of which is transformed into a garbled form. This process involves encoding the gate's truth table using encrypted labels, such that only the correct input keys can decrypt and reveal the correct output. The general process follows Yao's garbling scheme, as illustrated in Figure 4.1. First, the garbler, who is the NN provider, transforms the NN function $f = f_{NN}$ into a garbled circuit representation $(F, e, d) \leftarrow Gb(1^k, f)$. Next, the input values x are encoded into corresponding wire labels $(X_0^1, X_1^1, \ldots, X_n^0, X_n^1) \leftarrow e$. The garbled function F is then sent to the evaluator, who remains oblivious to the raw NN parameters. The evaluator interacts with the garbler through an OT protocol, where it receives the correct wire labels corresponding to its private input. The evaluator then computes the garbled function $y \leftarrow De(d, Ev(F, X))$ using the garbled circuit, ensuring that no intermediate values are exposed. This approach inherently prevents side-channel leakage, as every gate evaluation operates on encrypted values. Even if an attacker were to capture power traces or EM emissions, the randomized labels would obscure the actual data [226].

Instruction Set and Execution Model

HWGN^2 implements an efficient hardware execution model inspired by MIPS architecture, allowing a streamlined, instruction-based garbling approach while maintaining security. Each garbled circuit operation is mapped to a specific instruction, enabling direct execution on the hardware engine. These instructions define logical operations such as AND, OR, and XOR, as well as state transitions for gate evaluations. Unlike traditional garbled circuit implementations, which are often sequential, HWGN^2 supports a parallel processing model where multiple gates are evaluated simultaneously. This significantly reduces execution time while maintaining security [101]. Another critical aspect of HWGN^2 is its approach to oblivious memory access. To mitigate timing SCA, all memory accesses follow a structured access pat-

tern, preventing attackers from deducing sensitive information based on access timing [210]. Additionally, the garbler generates randomized encryption keys for each wire label, ensuring that the evaluator never sees a consistent pattern in circuit evaluation. These execution model enhancements make HWGN^2 highly efficient and resistant to side-channel threats.

Hardware-Software Co-Design

The efficient integration of hardware and software components is critical for achieving both security and performance in HWGN^2. The system architecture includes a dedicated garbling engine that pre-processes the NN before inference begins, reducing computational overhead compared to software-only garbling [439]. Since OT is a fundamental component of the protocol, HWGN^2 implements an optimized hardware OT module to reduce communication latency and improve efficiency. The execution engine maintains a hardware-protected state, ensuring that secret keys and intermediate values remain isolated from potential attackers. In addition to the hardware optimizations, the software layer of HWGN^2 includes a compiler that translates NN models into optimized garbled circuit representations. This step ensures that the circuit is minimal in size while maintaining security. To facilitate practical deployment, HWGN^2 is designed to be compatible with widely used ML libraries such as TensorFlow and PyTorch, allowing seamless integration into real-world applications. Although HWGN^2 primarily focuses on hardware-based garbled inference, it also supports hybrid execution models where certain operations can be offloaded to software for increased flexibility.

Security Advantages of HWGN^2

By combining garbled circuit-based computation with hardware acceleration and secure execution, HWGN^2 provides multiple layers of protection against side-channel threats. Since all computations are performed on encrypted labels, power consumption patterns do not correlate with the actual data, providing protection against power analysis attacks [69]. The structured execution model ensures that every inference operation runs in constant time, preventing attackers from deducing information based on variations in execution latency. Furthermore, the randomized nature of wire label encoding prevents adversaries from inferring NN parameters through EM leakage [56]. Unlike conventional masking approaches that can be bypassed by injecting

faults, HWGN^2 does not expose intermediate values, making fault-based attacks infeasible. The combination of these techniques ensures that DL inference remains both efficient and secure, addressing the limitations of prior countermeasures against SCA.

7.3.5 Side-Channel Resiliency Implementation and Evaluation

Concrete Implementation

When defining the PFE scheme $\mathcal{F}$, it is mentioned that $\mathcal{F}$ can securely and privately compute any function, which can be garbled by running the garbling scheme G. This fundamental principle enables the construction of a secure execution framework where computations can be securely outsourced without revealing sensitive information about the underlying function. In this context, the algorithm Π can be interpreted as a representation of the instruction set for a processor circuit, allowing $\mathcal{F}$ to be realized in practice by garbling the entire processor circuit and its corresponding instruction set [364].

A key distinction between HWGN^2 and previous works lies in the fundamental objective of the implementation. Unlike [401], which focused on optimizing the execution of an entire public MIPS program, our approach aims to securely execute a garbled instruction set while maintaining complete function privacy. This allows HWGN^2 to support privacy-preserving execution where the evaluator never gains knowledge about the structure of the underlying function. To demonstrate this, we present two different MIPS-based implementations, both serving as proof-of-concept prototypes that validate the effectiveness of garbled computation in mitigating side-channel vulnerabilities.

Although our implementations are based on MIPS architecture, they can be extended to support other processor architectures, such as ARM.

7.3.6 TinyGarble-based Implementation of HWGN^2

TinyGarble [362] is a widely used garbling framework that supports Yao's garbling scheme and leverages hardware synthesis tools to automatically generate Boolean circuits for secure computation. The advantage of this approach lies in its ability to automatically transform hardware descriptions

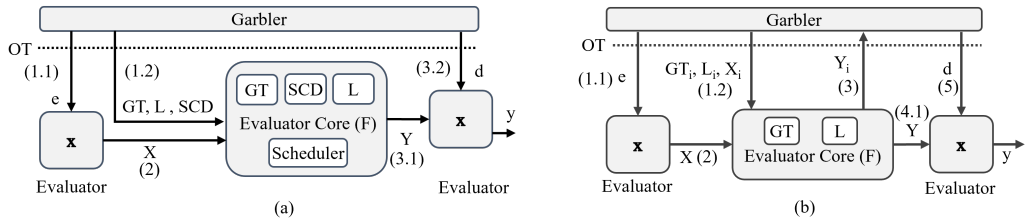


Figure 7.9: The execution flow of HWGN^2: (a) TinyGarble-based implementation [362] and (b) HWGN^2 with improved hardware resource utilization efficiency. Key elements: L: garbled labels, GT: garbled tables, e: encryption labels, d: decryption labels, x: evaluator's raw input, X: evaluator's garbled input, Y: garbled output, Y_i, X_i, GT_i, L_i: corresponding elements for the i^{th} sub-netlist, and SCD: circuit description used for mapping and evaluation.

into secure computation circuits, enabling efficient execution of garbled programs.

The main strengths of TinyGarble stem from its use of sequential circuit descriptions and several garbling optimizations, including Free-XOR optimization [214], which eliminates the need for encryption in XOR gates, reducing computational overhead; Row Reduction [276], which minimizes the size of garbled tables by removing redundant rows; and Fixed-Key Block Cipher Garbling [34], which accelerates garbling operations by using precomputed encryption keys.

Figure 7.9(a) illustrates the execution flow of HWGN^2 following the Tiny Garble-based implementation. The process is divided into multiple stages. First, the garbler generates encryption labels e and constructs the garbled circuit (GC) by producing garbled tables (GT), garbled labels (L), and a custom circuit description (SCD) mapping the GC to the function f. Second, the evaluator receives (GT, L, SCD, e) via a single OT interaction and derives the garbled input X from its raw input x. Third, the evaluator executes the garbled circuit sequentially using the TinyGarble scheduler, performing decryption and evaluation of each gate. Fourth, the garbler provides decryption labels d to enable the evaluator to decode the final output Y and obtain the raw output y.

This sequential execution model provides scalability but incurs high hardware resource utilization, particularly in memory and logic elements, due to the size of DL circuits [192]. To address these limitations, we introduce an

improved method for hardware resource efficiency.

7.3.7 HWGN^2 with Improved Hardware Resource Utilization Efficiency

A critical challenge in garbled inference is the trade-off between memory consumption and communication cost. To mitigate memory overhead while maintaining security, we adapt the TinyGarble2 framework [173], which partitions large circuits into sub-netlists for incremental evaluation.

Instead of sending all garbled tables and labels at once, HWGN^2 divides the computation into sub-netlists, each processed sequentially. This achieves significant reductions in memory usage, albeit at the cost of increased OT interactions. The trade-off is given by

$$M = \frac{N_{gate}}{4} \tag{7.1}$$

where N_{gate} is the total number of gates, and M determines the number of OT interactions required.

As illustrated in Figure 7.9(b), the execution proceeds as follows. First, the garbler sends one garbled table and its corresponding input labels per cycle instead of the entire circuit. Second, the evaluator processes each sub-netlist incrementally, returning its output to the garbler for decryption. Third, the final output is reconstructed after all sub-netlists are evaluated.

Since the garbler controls the scheduling and SCD mapping, the evaluator never learns the circuit topology, ensuring PFE.

7.3.8 Garbled MIPS Evaluator

To implement HWGN^2 on an FPGA, we designed a garbled MIPS evaluator, modifying the Plasma MIPS core [315]. Figure 7.10 illustrates its architecture.

The instruction handler receives garbled MIPS instructions, which are encrypted representations of MIPS operations. The garbled instructions are fetched, decoded, and executed without leaking information about their function. Each instruction corresponds to a securely garbled gate, ensuring privacy-preserving execution.

Figure 7.11 provides an example evaluation of a 2-bit adder using the garbled MIPS evaluator. Garbled instructions are processed step-by-step,

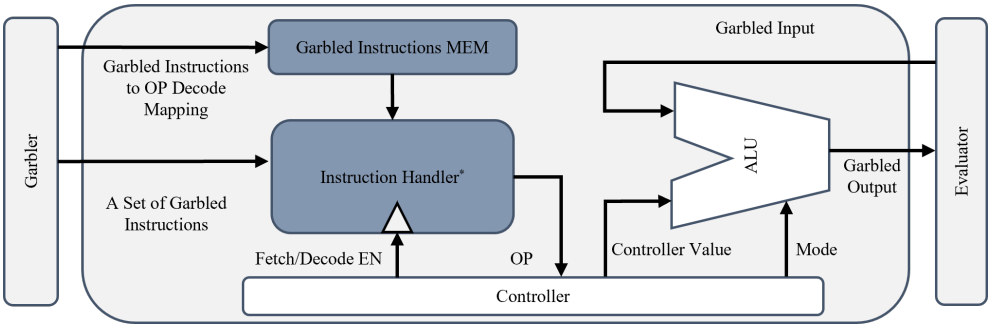


Figure 7.10: Garbled MIPS evaluator architecture, based on modifications to the Plasma MIPS core [315]. The modified instruction handler processes garbled instructions while ensuring complete privacy of the execution flow.

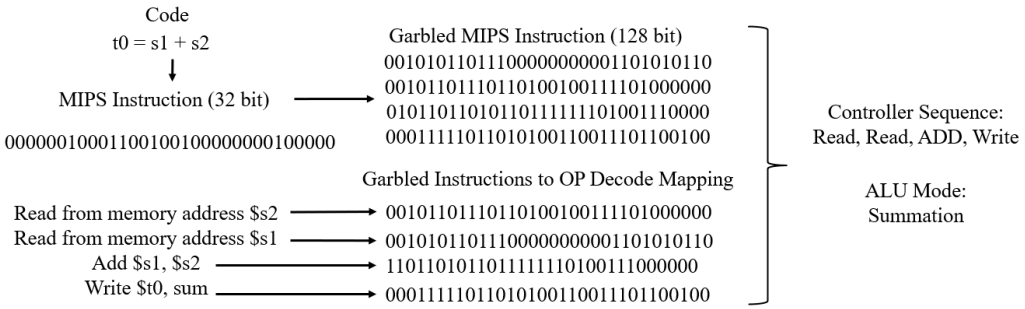


Figure 7.11: Example execution of a 2-bit adder using the garbled MIPS evaluator. The process involves fetching, decoding, and executing garbled instructions in a privacy-preserving manner.

maintaining execution privacy while ensuring correctness.

7.3.9 Hardware Implementation Resource Utilization

To assess the trade-off between communication cost, hardware resource utilization, and performance, we have synthesized the garbled evaluator in two different configurations. The first configuration corresponds to HWGN^2 with improved hardware resource utilization efficiency, capable of executing a single garbled instruction per OT interaction, as described in Section 7.3.7. The second configuration represents a fully functional HWGN^2 implementation based on the TinyGarble framework, capable of executing all 2345 garbled MIPS instructions with a single OT interaction, as explained in Section 7.3.6.

This dual-configuration approach allows us to comprehensively evaluate the impact of optimizing for either resource efficiency or execution speed.

For synthesis and implementation, we utilized Xilinx Vivado 2021 to generate the bitstream of our design. To ensure that the bitstream accurately represents the intended design without any modifications introduced by synthesis optimizations, we explicitly disabled the place-and-route optimization and applied the DONT-TOUCH attribute to critical modules. This guarantees that the logical and physical design remains exactly as specified at the RTL level, preventing unintended optimizations that could affect security and performance.

To evaluate HWGN^2, we implemented three distinct MLP architectures, each trained for digit classification tasks. The first architecture, BM1, consists of an input layer with 784 neurons, three hidden layers with 1024 neurons each, and an output layer with 10 neurons. This model has been widely used in prior secure computation research, with results reported in [327, 101, 364]. The second architecture, BM2, is a smaller MLP with an input layer of 784 neurons, two hidden layers containing 5 neurons each, and an output layer of 10 neurons. The third model, BM3, features an input layer of 784 neurons, three hidden layers with 6, 5, and 5 neurons respectively, and an output layer with 10 neurons. These models allow us to evaluate HWGN^2 across varying network complexities.

Hardware Resource Utilization and OT Cost Analysis

In HWGN^2, the processing of garbled instructions and input labels occurs over 32-bit data widths. To maintain a fair comparison, we include a 32-bit MAC unit [327] in the reported resource utilization. Notably, BoMaNET and ModulaNET do not rely on OT for input exchange, whereas RedCrypt uses two OT interactions, one for input and one for output. In contrast, HWGN^2 requires an additional M OT interactions, where M represents the number of sub-netlists. By setting the sub-netlist size to a single gate, and considering that every four gates translate to a garbled instruction, the total number of OT interactions is computed as:

$$M = \frac{N_{gate}}{4} \tag{7.2}$$

For BM1, which comprises $N_{gate} = 9380$ gates, this results in:

Table 7.3: Hardware resource utilization and OT cost comparison between approaches applied against BM1.

Approach	LUT	FF	OT Interaction
GarbledCPU [364]	21229	22035	2
RedCrypt [327] (One MAC Unit)	111000	84000	2
BoMaNET [102]	9833	7624	N/A
ModulaNET [101]	5635	5009	N/A
HWGN^2 (1 instruction per OT interaction)	1775	1278	2346

Table 7.4: Execution time and communication cost comparison between HWGN^2 and the state-of-the-art approaches for BM1. Results for [364] and HWGN^2 are based on an FPGA clock frequency of 20MHz. (N/R: not reported).

Approach	Time (Sec)	Communication (MB)
GarbledCPU [364]	1.74	N/R
RedCrypt [327]	0.63	5520
TinyGarble2 [173]	9.1	7.16
HWGN^2 (1 instruction per OT interaction)	3.25	12.39
HWGN^2 (Complete set of instructions per OT interaction)	0.68	619

$$2 + \frac{9380}{4} = 2346 \tag{7.3}$$

This accounts for 2 standard OT interactions plus 2344 additional OT interactions for processing garbled instructions.

7.3.10 Execution Time and Communication Cost Evaluation

To measure the computational overhead of HWGN^2, we deployed our implementation on a Xilinx Artix-7 FPGA (clocked at 20 MHz) for secure evaluation, while the garbler operated on an Intel Core i7-7700 CPU (3.60 GHz) with 16 GB RAM, running Linux Ubuntu 20. The execution time metric excludes offline preparation steps such as garbled circuit generation and label encryption, as these are performed prior to inference.

The results demonstrate that HWGN^2 achieves substantial improvements in execution efficiency and communication overhead compared to prior secure inference methods.

7.3.11 Side-Channel Evaluation

Side-channel Measurement Setup

HWGN^2 has been implemented on an Artix-7 FPGA device XC7AT100T with package number FTG256. To evaluate its resilience against power and EM SCA, we set up a dedicated side-channel measurement environment using a Riscure setup. The power and EM traces were captured using a LeCroy WavePro 725Zi oscilloscope, which has a high sampling rate and deep memory to facilitate high-resolution side-channel measurements.

To ensure accurate trace collection and prevent information loss, we configured our design frequency to 1.5 MHz while setting the oscilloscope sampling frequency to 127.5 MHz. This configuration ensured that for every clock cycle, we collected 85 sample points, providing a fine-grained view of power and EM variations during computation.

The reduction of the design frequency to 1.5 MHz was necessary to acquire high-resolution side-channel traces, but it also increased the execution time. The classification of a single input in the BM1 model required an execution time ranging between 3.25 and 4.73 seconds. Given that modern side-channel analysis techniques require millions of traces, collecting a sufficient dataset from BM1 alone would introduce a significant time overhead. To address this, we followed an approach similar to [101] and used a smaller MLP architecture, namely BM2, for trace collection.

BM1 is a deep multilayer perceptron (MLP) model that consists of an input layer with 784 neurons, three hidden layers each containing 1024 neurons, and an output layer with 10 neurons. This architecture was selected as it provides a realistic benchmark for evaluating side-channel resilience in DL inference. BM1 was trained on the MNIST dataset and was designed to serve as a representative test case for DL inference on secure hardware.

BM2 is a smaller MLP architecture, designed with reduced complexity for efficient side-channel trace collection. It consists of an input layer with 784 neurons, two hidden layers with 5 neurons each, and an output layer with 10 neurons. The reduced size of BM2 allowed for a significant reduction in execution time per inference, enabling the collection of a large number of traces in a practical timeframe. Using BM2, each classification was completed in 312 ms under HWGN^2 running at 1.5 MHz. Since HWGN^2 processes each instruction separately in a sequential manner, and given that NN exhibit repetitive computation patterns, we argue that leakage characteristics from

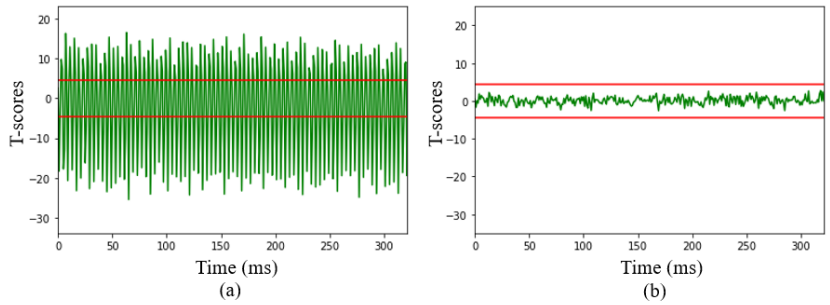


Figure 7.12: TVLA test results for BM2 implementation on (a) an unprotected MIPS core and (b) HWGN^2 with one instruction per OT interaction (computed for 10K traces).

BM2 can be extrapolated to larger architectures.

7.3.12 TVLA Test Evaluation of Power Side-Channel

To analyze the power side-channel leakage, we applied the TVLA test to two different implementations: an unprotected MIPS core (Plasma core from the OpenCores project [315]) and HWGN^2 with an execution capacity of one instruction per OT interaction.

Figure 7.12 shows the results of the TVLA test for both implementations. The t-scores were calculated using 10,000 captured traces, with 5,000 traces collected for fixed inputs and 5,000 traces collected for random inputs.

As seen in Figure 7.12, the unprotected MIPS core exhibits significant leakage, with t-scores exceeding the ± 4.5 threshold after only 10,000 traces. In contrast, the t-scores for HWGN^2 remain below the threshold, demonstrating that it does not leak sensitive information through power SCA.

To further validate the resilience of HWGN^2, we extended the experiment by capturing 2 million (2M) traces—1 million for fixed inputs and 1 million for random inputs. The results confirmed that the t-scores remained below the leakage threshold, even when analyzed over a large trace population.

The results in Figure 7.13 further confirm that HWGN^2 remains resilient against power SCA across different configurations, regardless of whether the instruction capacity per OT interaction is one instruction or a full instruction set.

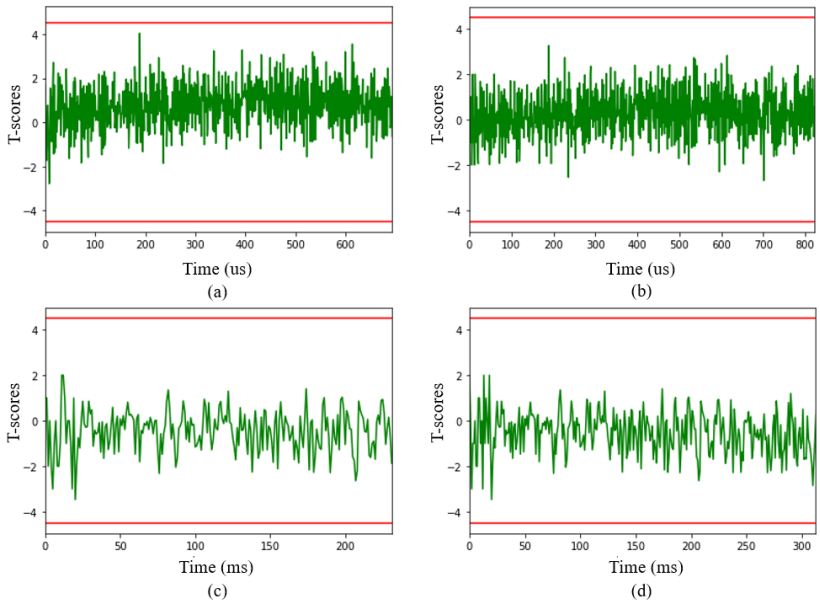


Figure 7.13: TVLA test results for HWGN^2 applied to (a) XNOR-based BM2 with full instruction set per OT, (b) BM2 with full instruction set per OT, (c) XNOR-based BM2 with one instruction per OT, and (d) BM2 with one instruction per OT (calculated for 2M power traces).

7.3.13 TVLA Test Evaluation of EM Side-Channel

EM side-channel analysis is another powerful attack vector, as demonstrated by Peeters et al. [295] and Standaert et al. [368]. Studies have shown that EM leakage can often provide more information than power leakage, making it a critical factor in evaluating hardware security.

To evaluate the EM leakage resistance of HWGN^2, we captured traces using an HP EM probe 125 (SN126 0.2mm) positioned near the FPGA surface. The TVLA results, shown in Figure 7.14, indicate that the t-scores for EM traces remain below the ± 4.5 threshold, confirming the EM leakage resilience of HWGN^2.

7.3.14 Architecture-Related Leakage Analysis

The ability of an adversary to extract structural information about a DL model through side-channel analysis was demonstrated in [29]. The authors showed that EM traces captured from an unprotected DL model running on

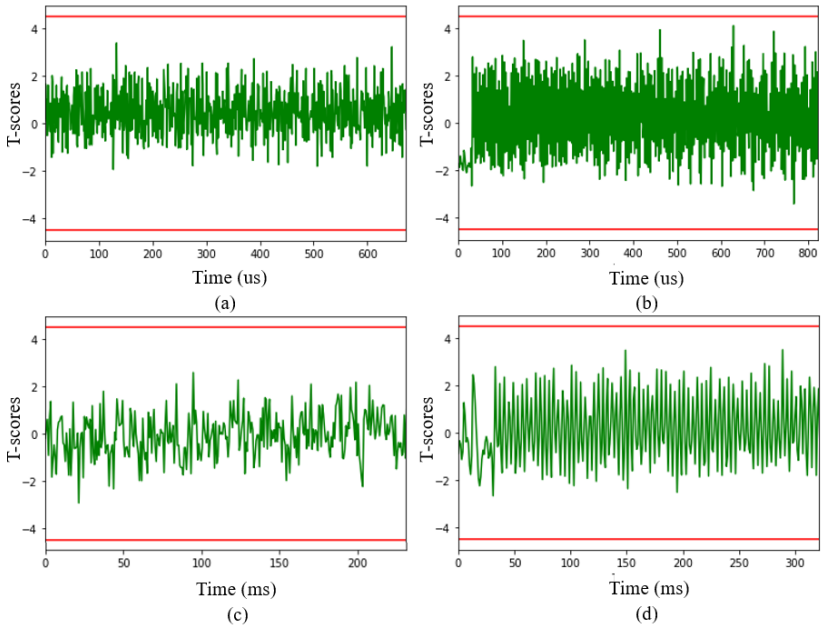


Figure 7.14: TVLA test results for HWGN^2 applied to (a) XNOR-based BM2 with full instruction set per OT, (b) BM2 with full instruction set per OT, (c) XNOR-based BM2 with one instruction per OT, and (d) BM2 with one instruction per OT (calculated for 2M EM traces).

an Atmel ATmega328P microcontroller revealed patterns that corresponded to the model's architecture. Specifically, they observed distinct signal patterns for different layers and neuron activations.

To examine whether HWGN^2 exhibits similar leakage patterns, we implemented the same BM3 model used in [29] and captured 100,000 EM traces. Figure 7.15 compares the captured traces.

The discussion so far has highlighted how HWGN^2 leverages SFE and PFE to provide side-channel-resilient NN execution. By employing GC and optimizing OT, HWGN^2 ensures that sensitive computations are protected from power and EM side-channel leakage. The focus has been on mitigating classical threats posed by an HbC adversary in a traditional computing setup, where a garbler and evaluator engage in secure computation over a well-defined protocol.

While HWGN^2 offers significant improvements in security for secure DL accelerators, one of its major limitations stems from the computational and communication overhead of GC, particularly in scenarios involving high-

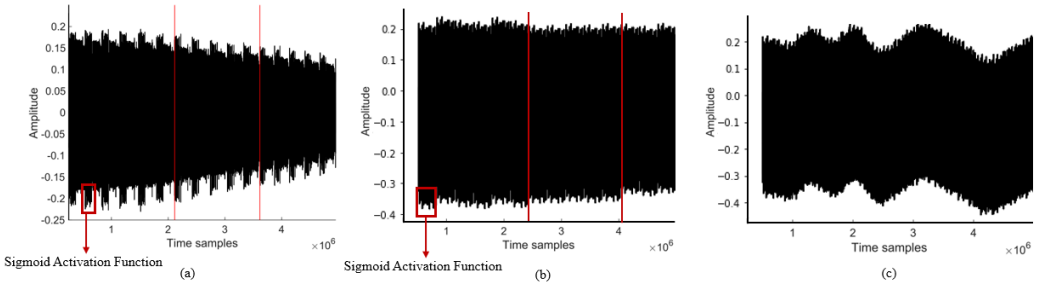


Figure 7.15: A randomly chosen EM trace pattern captured from BM3 implementation on (a) Atmel ATmega328P microcontroller [29], (b) FPGA with unprotected MIPS evaluator [315], and (c) HWGN^2. Red lines indicate where the unprotected evaluator starts processing the next layer.

throughput inference workloads. The reliance on a conventional execution environment—where the garbler and evaluator interact over a network—introduces latency bottlenecks that can limit scalability.

To further improve the efficiency of GC-based secure computation, we now shift our focus to **Garblet**, a framework that extends the concept of secure MPC to the domain of **chiplet-based architectures** [158]. Unlike HWGN^2, which assumes a single evaluator executing the GC-protected workload, **Garblet** distributes the garbling and evaluation process across physically distinct chiplets, thereby reducing communication overhead and improving performance through parallelism.

The key motivation behind **Garblet** is to leverage the **modular design of chiplet-based architectures** to **overcome the scalability limitations of traditional GC implementations**. In a chiplet-based system, individual chiplets can specialize in specific computational tasks—allowing **the garbler and evaluator to operate as distinct chiplets** within the same system-on-package (SoP) environment. This not only **reduces the cost of GC evaluation** but also enables more efficient hardware isolation, preventing untrusted chiplets from interfering with secure computations.

By integrating **custom hardware OT modules** and an **optimized evaluator engine**, **Garblet** enhances secure computation efficiency while minimizing communication between chiplets. Additionally, it introduces a **novel circuit decomposition technique** that partitions complex circuits into smaller, manageable subcircuits—enabling **parallel evaluation across multiple chiplets**. This approach fundamentally **reshapes the execution**

model of GC-based secure computation, offering a path toward **high-performance MPC implementations in next-generation computing platforms**.

In the following sections, we will explore the **Garblet framework**, its **chiplet-based implementation**, and the **practical considerations for deploying secure MPC in chiplet-based systems**.

7.4 Garblet: MPC for Protecting Chiplet-based Systems

7.4.1 Adversary Model in Chiplet-Based Secure Computation

We assume that chiplets are deployed to perform distributed computations, where multiple computational tasks are executed in parallel across different chiplets. This distributed architecture enables improved scalability and computational efficiency but also introduces new security concerns regarding data confidentiality and integrity.

At least one chiplet in the system is assumed to be trusted and is responsible for initiating and coordinating secure MPC. This trusted chiplet ensures that computations follow the prescribed cryptographic protocols while maintaining the privacy and integrity of the data being processed.

An adversary in this setting aims to compromise the security of the system by gaining access to untrusted chiplets. The attacker may attempt to corrupt at least one chiplet to achieve the following goals. The adversary can attempt to intercept data exchanged between chiplets, extracting sensitive user information or computational secrets. If an adversary gains control over an untrusted chiplet, they may attempt to tamper with the computations, introducing incorrect results or inferring hidden parameters of the processed model. The adversary may use SCA such as power analysis, EM leakage analysis, or memory access timing attacks to extract partial or full information about secret inputs or processed outputs.

To counteract these threats, one can rely on GC-based computation between chiplets, where security against a passive adversary is provably guaranteed. GC ensure that chiplets only process encrypted representations of data, preventing an adversary from learning useful information even if they gain

access to one of the untrusted chiplets. The garbled circuit approach also minimizes the risk of leakage through communication channels, as chiplets exchange only encrypted labels and not raw computational data.

In secure MPC, traditional adversary models include HbC adversaries and malicious adversaries. A semi-honest adversary follows the prescribed protocol steps without actively deviating from execution. However, while they do not alter the execution, they attempt to infer sensitive information by analyzing exchanged messages or intermediate results. Although this type of adversary does not disrupt computation, they still pose a significant risk in environments where data confidentiality is crucial. A malicious adversary, on the other hand, actively attempts to disrupt the protocol execution. They may manipulate input data, modify circuit structures, inject faults into the computation, or generate invalid GC that produce incorrect outputs. In the context of chiplet-based computation, a malicious adversary may tamper with inter-chiplet communication, introduce hardware Trojans in untrusted chiplets, or execute sophisticated FIA [226].

A particular concern in chiplet-based architectures is the risk of supply chain attacks, where an adversary compromises the design or manufacturing process of a chiplet before it is integrated into the final system. This risk necessitates stringent cryptographic safeguards, including techniques such as ZKPs and hardware attestation to ensure the authenticity of chiplets participating in computation.

Garblet mitigates these threats by implementing secure computation techniques such as GC, OT, and function privacy mechanisms. By executing computations within a structured chiplet-based environment, it ensures that even if one or more chiplets are compromised, no meaningful information about the processed data is revealed.

7.4.2 Methodology

In this section, we first elaborate on the client-server model and its mapping to Chiplet then the implementation of circuit decomposition using the reverse logic tracing approach [77], which divides the given circuit into sub-circuits. After that, each sub-circuit is assigned to dedicated engines for parallel processing. For securely and obliviously transferring inputs, we implement a novel dedicated hardware OT module. For GC generation, one can use any existing garbling engine, e.g., FASE [174]. We also create the very first evaluator engine on the chiplet to work efficiently with the garbling

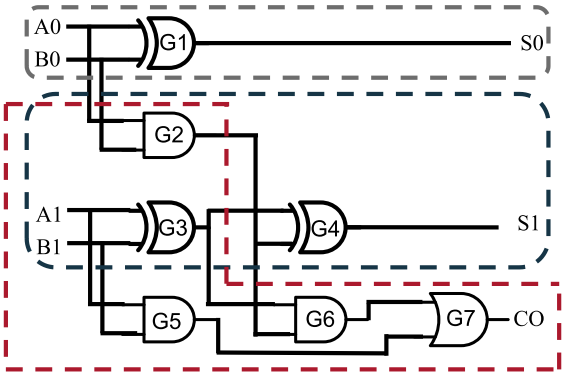


Figure 7.16: The sub-circuits of a two-bit adder corresponding to each output.

engine.

Client-Server Model and Chiplet Mapping

The design of Garblet leverages the traditional server-client model often used in secure computation frameworks. In this model, the garbler acts as the server and handles the majority of computationally intensive tasks, such as generating garbled tables and managing cryptographic keys. The evaluator, acting as the client, performs less computationally demanding tasks, primarily focused on decrypting the garbled tables and evaluating the circuit. This division of roles ensures that the evaluator's operations are optimized for speed, enabling real-time application scenarios, while the garbler focuses on heavy computation with higher resource requirements.

In Garblet, this client-server model is mapped to a chiplet-based architecture, where one chiplet functions as the garbler chiplet and the other as the evaluator chiplet. The garbler chiplet is equipped with dedicated hardware modules, including AES-based encryption units, a key management unit, and a pipelined garbling engine to efficiently handle the resource-intensive garbling process. In contrast, the evaluator chiplet is designed for low-latency operations, integrating modules for secure OT protocol execution and optimized evaluator engines.

Algorithm 11: Circuit Decomposition

Input: Circuit f with outputs O and inputs I
Output: Set of sub-circuits C, one for each output in O

Step 1: Extract Outputs and Inputs
Let $O \leftarrow$ `ExtractOutputs`(f) ; `// Extract output nodes`
Let $I \leftarrow$ `ExtractInputs`(f) ; `// Extract input nodes`

Step 2: Initialize Sub-Circuit Set
Initialize an empty set C to store sub-circuits for each output.;

Step 3: Reverse Logic Tracing
foreach *output* $o \in O$ **do**
    Initialize sub-circuit C_o for o;
    Call `ReverseTraverse`(o, f, C_o) ; `// Trace back to inputs`

`// Recursive tracing function`
Function `ReverseTraverse`(n, f, C_o)**:**
    if *n is an input node* **then**
        return ; `// Stop at primary input`
    Add n and its gate to C_o;
    foreach *input i of gate n* **do**
        `ReverseTraverse`(i, f, C_o) ; `// Recurse on gate inputs`

Step 4: Construct and Optimize Sub-Circuits
foreach $o \in O$ **do**
    $C_o \leftarrow$ `ConstructSubCircuit`(C_o) ; `// Compile traced gates`
    $C_o \leftarrow$ `OptimizeCircuit`(C_o) ; `// Minimize redundancy`

Return C ; `// Return the set of optimized sub-circuits`

Reverse Logic Tracing for Circuit Decomposition

Reverse logic tracing decomposes a circuit by backtracking from each output node to its input dependencies, capturing all gates, wires, and connections involved in the computation. This process is illustrated in Algorithm 11. The decomposition starts by selecting the primary outputs and performing a depth-first traversal (DFT) [77] in reverse, tracing the logic back to the primary inputs (PIs). During this traversal, each gate and input affecting the output is marked, forming a complete dependency tree. After constructing these dependency trees, each sub-circuit is compiled as an isolated block with all necessary components. The final step involves optimizing each sub-circuit for logic redundancy by simplifying or combining gates and paths that do not directly impact the output, thereby reducing complexity while maintaining full functionality.

By dividing circuits into sub-circuits using reverse logic tracing, our methodology facilitates parallel processing, critical for scaling to larger and more complex circuits. The number of sub-circuits generated increases linearly

with the number of circuit outputs, enabling finer granularity in workload distribution. While this increases pre-processing time slightly, it significantly enhances scalability by allowing multiple garbling and evaluator engines to operate concurrently.

As an example, Fig. 7.16 shows the sub-circuits of a two-bit adder, with each dashed area representing a sub-circuit responsible for a specific output bit. The gray dashed area is the `G1` gate connected to the `S0` output and `A0` and `B0` inputs. The dark blue dashed area is the sub-circuit of `G2, G3, and G4` gates connected to `S1` output and all the inputs. The third sub-circuit includes `G7, G6, G5, G3, and G2` connected to the `CO` output and all the circuit inputs.

7.4.3 Oblivious Transfer Implementation

The OT module enables secure transmission of data between the Garbler and Evaluator chiplets. We implemented a hardware-based 1-out-of-2 OT module, which comprises three primary blocks: the Key Generator, Random Selector, and Communication Interface, each designed in Verilog and synthesized onto the chiplets.

The Key Generator Block on the Garbler Chiplet uses a True Random Number Generator (TRNG) and key management unit to produce cryptographic keys for each input wire. The TRNG outputs are processed by a Von Neumann extractor [298] to ensure uniform distribution. For each input wire W_i, a random key K_i^0 is generated, and $K_i^1 = K_i^0 \oplus \delta$, where δ is a secure global offset. These keys are stored in dual-port BRAM, with retrieval managed by a control unit.

The Random Selector Block on the Evaluator Chiplet generates a random bit s_i for each wire, determining which key (K_i^0 or K_i^1) will be used. Each selection bit s_i is masked with a one-time pad r_i to form $m_i = s_i \oplus r_i$, then stored in a FIFO buffer. The masked bits m_i are sent to the Garbler Chiplet via the Communication Interface using a handshaking protocol.

The Communication Interface supports secure data exchange using high-speed protocols such as AXI and PCIe. Data channels use AXI4-Lite for control and AXI4-Stream for high-bandwidth data transfer, with data encrypted before transmission to ensure privacy. Dual-port BRAM buffers incoming and outgoing data, with a control FSM managing data flow to prevent conflicts.

OT Execution. The Garbler Chiplet initializes its TRNG and generates a

Algorithm 12: Evaluator Engine Implementation

Input: Garbled Circuit C, Garbled Tables T, Input Keys K_{in}
Output: Final Output Keys K_{out}

`InitializeMMU();`
Initialize dual-port BRAM for input keys and garbled tables;
Configure AES cores for decryption mode;

`ReceiveKeys(`K_{in}`);`
Receive the evaluator's input keys through the OT protocol and store them in BRAM;
Synchronize with Garbler Chiplet to ensure all keys are securely stored;

foreach *gate* G_i *in* C **do**
    if G_i *is XOR gate* **then**
        $K_{\text{out}} \leftarrow$ `XOR`$(K_{\text{in1}}, K_{\text{in2}})$; `// Use Free-XOR optimization`
    else
        $K_{\text{out}} \leftarrow$ `DecryptGate`(K_{in}, T_i) ; `// Decrypt using AES in decryption mode`
    Store K_{out} in BRAM for subsequent gate evaluations;
    `ManageMemory();`

`ManageMemory();`

Collect all output keys K_{out} corresponding to the circuit's primary outputs;
Concatenate the keys to form the final output;
`TransmitData(`*(Final Output)*`);`
Send the final output to the external evaluator for verification or further use;

unique global offset δ while the Evaluator Chiplet pre-loads the random bits r_i. The Garbler Chiplet generates keys K_i^0 and K_i^1 for each input wire and stores them. The evaluation chiplet generates selection bits s_i, masks them with r_i to produce $m_i = s_i \oplus r_i$, and sends them to the Garbler chiplet. The Garbler Chiplet computes the selected key $K_i^{s_i} = K_i^{m_i \oplus r_i}$ and transmits it to the Evaluator Chiplet, which stores the keys for evaluation.

7.4.4 Evaluator Engine Implementation

The literature suggested that a garbling engine can be modified to create an evaluator engine [174]; however, based on our experience, this task is more delicate than expected. For implementing the evaluator engine, one needs to implement input key handling, decryption logic, and memory management, as detailed in Algorithm 12. In our efficient and practical evaluator engine, only a single key K_i per input wire is needed. The hardware OT module transmits the evaluator's selected keys, which are stored in dual-port BRAM. In this way, for each wire W_i, $\text{BRAM}_{\text{Key}}[i] = K_i$. A synchronization unit ensures evaluation starts only after the secure storage of keys. Here, secure storage means that all keys are stored in the isolated memory and all buffers

are free, i.e., the handshake signal and acknowledge signal are both raised to 1; otherwise, the key values are accessible during the operations. The AES cores are reconfigured for decryption mode. For each gate G_i, the evaluator decrypts the garbled truth table entry E_i:

$$K_{\text{out}} = \text{AES}^{-1}(K_{\text{in}}, E_i) = \text{AES}^{-1}(K_{\text{in}}, \text{AES}(K_{\text{in}}, K_{\text{out}} \oplus R)),$$

where R is a random value. For XOR gates, the output key is directly computed: $K_{\text{out}} = K_{\text{in1}} \oplus K_{\text{in2}}$. This avoids decryption and reduces computational overhead. Memory management was optimized using a Memory Management Unit (MMU) that dynamically allocates addresses for sub-circuits being evaluated:

$$\text{MMU}_{\text{Address}}[i] = \begin{cases} \text{read}; & \text{if gate } G_i \text{ is ready for evaluation,} \\ \text{write}; & \text{if output } K_{\text{out}} \text{ is to be stored.} \end{cases}$$

The MMU minimizes data collisions and ensures efficient memory access.

We also optimized the communication interface between the Garbler and Evaluator engines. The AXI4-Stream interface was configured for direct memory access (DMA), supporting bulk data transfers of garbled tables and keys. Each packet is encrypted and includes a parity check P and 32-bit cyclic redundancy check (CRC) to verify data integrity: $P = \bigoplus_{i=1}^{n} \text{bit}_i$.

7.4.5 Sub-circuit Assignment: Advantages and Process

Efficient sub-circuit assignment and synchronization of the garbling and evaluation phases are essential for optimizing performance in our framework. The scheduler dynamically allocates encryption keys to the garbling engines and meticulously tracks the progress of each sub-circuit to prevent conflicts and ensure smooth parallel execution. After the garbling phase, the generated garbled tables are transmitted to the evaluator engines, where they are processed, and the results are evaluated to form the final output. Fig. 7.17 illustrates the high-level flow of sub-circuit assignment and its integration into the overall system. By distributing garbling tasks across multiple engines, the scheduler not only enhances computational efficiency but also contributes to the framework's security.

A crucial security benefit of this structured approach lies in Garblet's ability to enforce hardware-level isolation for security-critical tasks. By assigning different tasks, such as encryption and evaluation, to separate chiplets, we can physically separate sensitive operations (e.g., cryptographic key management) from non-critical ones (e.g., intermediate data storage and transmission). This separation limits the attack surface and reduces the risk of adversaries compromising critical components. For example, if an attacker gains access to a chiplet responsible for handling non-sensitive operations, such as managing communication between sub-circuits, they cannot directly manipulate or observe the encryption keys managed by a separate, isolated chiplet dedicated to a garbling engine. This hardware-level partitioning ensures that even if one component is compromised, the security of the overall computation remains intact, as sensitive operations are shielded from potential attacks. This robust isolation, combined with efficient sub-circuit assignment and synchronization, enables Garblet to perform secure computations with minimal performance overhead, providing both security and efficiency in a highly modular and scalable manner.

7.4.6 Chiplet-based GC Implementation Flow

To leverage the performance benefits of a chiplet-based implementation, we integrate a garbling engine, our custom hardware OT module, and our evaluator engine on chiplets. This integration enables us to perform complete GC without relying on external parties as opposed to, e.g., HostCPU in TinyGarble [362]. Fig. 7.18 illustrates the GC protocol distribution across two chiplets to reduce communication overhead and enhance secure computation efficiency. We utilize Xilinx UltraScale+ chiplets [181] that offer modular platforms with high-speed interfaces such as AXI [178] and PCIe [180]. The framework is implemented using Xilinx Vivado Design Suite [409] and Vitis Unified Software Platform [410]. These tools optimize communication latency between garbler and evaluator chiplets.

The garbling process is assigned to Chiplet A, while Chiplet B performs the evaluation. This separation allows parallel operation with minimal delays. Chiplet A garbles each Boolean gate in four main phases: (I) circuit representation and key generation, (II) gate garbling, (III) pipelined garbling, and (IV) inter-chiplet communication.

Circuit Representation and Key Generation. The function is represented as a Boolean circuit where each gate (`AND`, `OR`, `XOR`) corresponds to a

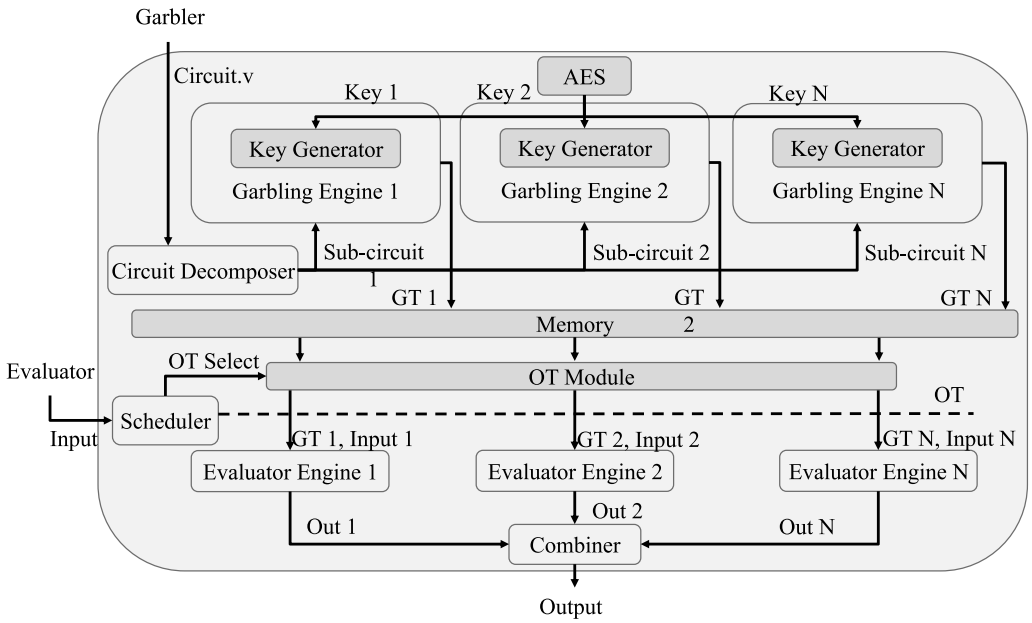


Figure 7.17: Sub-circuit assignment to garbling/evaluator engines.

logical operation in GC. The Key Generator module produces two cryptographic keys per gate's wire, one for '0' and one for '1';

Pipelined Garbling Process. Each gate is garbled in Chiplet A's garbling engine by creating a garbled truth table, where input wire keys are encrypted. The garbling engine is pipelined to garble one gate per clock cycle, ensuring continuous AES core operation. As gates are processed, garbled tables, inputs, and output keys are stored in Chiplet A's dual-port BRAM. A memory management wrapper manages read/write operations to prevent conflicts, enabling simultaneous garbling and data transmission to Chiplet B.

Inter-chiplet Communication. Communication between Chiplet A and Chiplet B is established using AXI and PCIe protocols to handle large data transfers efficiently. AXI enables direct memory access (DMA) for fast data transmission, while PCIe supports high-bandwidth communication to reduce delays in garbled table transfer. Chiplet B, configured as the evaluator, uses our HW 1-out-of-2 OT protocol to securely select its input keys. Upon receiving the garbled tables and keys, Chiplet B evaluates each gate using the garbled tables. XOR gates are evaluated without encryption due to Free-XOR optimization, while non-XOR gates are decrypted to reveal the correct output keys.

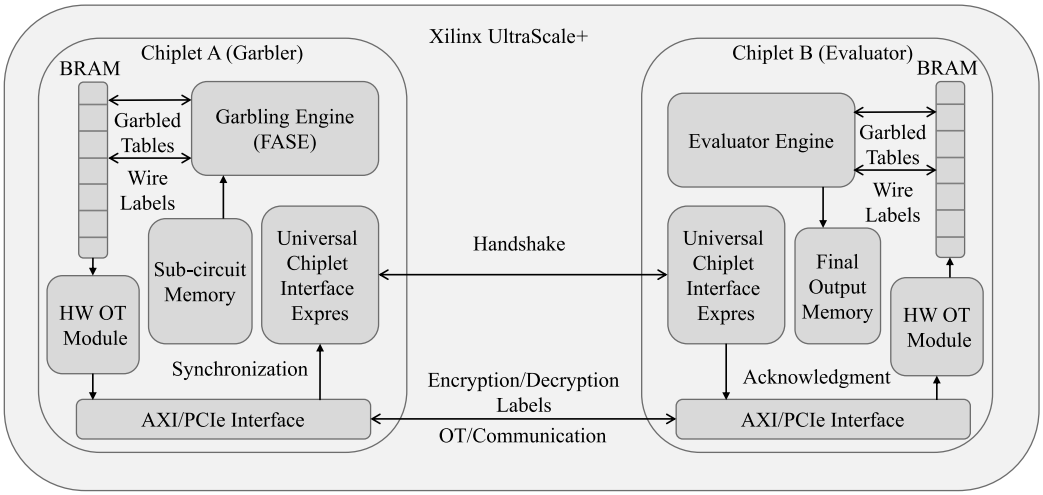


Figure 7.18: The flow of GC implementation on the chiplet-based system.

Evaluation and Synchronization. Dual-port BRAM in Chiplet B manages garbled tables and evaluation keys. Synchronization between Chiplets A and B is handled via a handshake protocol, ensuring Chiplet B only begins evaluation after receiving all required data. The Universal Chiplet Interface Express (UCIe) protocol optimizes synchronization, reducing delays and improving efficiency [180]. Once the evaluation is complete, Chiplet B decrypts the final garbled output to obtain the output in plaintext.

System Optimization. Parameters such as clock speed and inter-chiplet bandwidth are chosen for scalability to handle large computation efficiently [179, 178, 181].

Moreover, the chiplet-based architecture inherently supports scalability by enabling the integration of additional garbling and evaluator engines as required. By leveraging high-speed communication protocols such as UCIe and AXI4-Stream, the framework ensures that communication overhead remains manageable even with increasing computational demands. The modular design of the chiplets allows seamless addition of resources to scale the framework for more complex applications, such as DL inference or large cryptographic functions.

This makes the framework suitable for real-time applications where quick and secure computations are crucial.

Table 7.5: Hardware resource utilization: comparison between Garblet and implementations on monolithic FPGA [174, 156].

Resource	Garbling Engine		Evaluator Engine		
	FASE [174]	Garblet	Monolithic (Resource Efficient)	Monolithic (High Performance)	Garblet
LUT	31330	11729	1775	94701	5717
FF	11416	4103	1278	52534	2739
LUTRAM	553	93	N/R	N/R	78
BRAM	68.5	103	0	0	95
DSP	0	1	0	0	1

7.4.7 Experimental Results

Experimental Setup

We evaluated Garblet using common benchmark functions such as AES, multiplication, MAC, and an 8-bit adder cf. [174]. The experiments were conducted in two distinct scenarios to compare performance and hardware utilization. In the first scenario, a traditional server-client (personal computer (PC)-FPGA) setup was used, where a PC with an Intel Core i7-7700 CPU @ 3.60GHz, 16 GB RAM, and Linux Ubuntu 20 acted as the garbler, and the evaluator was implemented on an ARTIX7 FPGA board operating at a clock frequency of 20 MHz. This configuration served as the baseline for performance comparison. In the second scenario, Garblet was implemented on Vertex UltraScale+ chiplets, where the garbler and evaluator were deployed on separate chiplets. These chiplets were configured to achieve the maximum possible frequency and bandwidth, with high-speed transceivers operating at 32.75Gb/s and clock frequencies reaching up to 600 MHz.

To ensure consistency and minimize variability caused by external factors, all execution times reported represent the average of five independent runs. We explored the impact of resource allocation by testing configurations with a single pair of garbling and evaluator engines and then scaling up to multiple engines. This analysis allowed us to evaluate the trade-offs between hardware costs and performance gains in both resource-efficient and high-performance modes.

Hardware Resource Utilization Analysis

Table 7.5 compares the hardware resource utilization of Garblet with the implementations on monolithic FPGA as in [174, 156]. Below is a concise analysis highlighting the resource savings achieved by the Garblet.

Table 7.6: Hardware resource utilization of Garblet individual modules.

Resource	Garbling Engine	Scheduler	Key Generator	OT Module	Evaluator Engine	Combiner	Controller
LUT	11729	5739	8088	4182	5717	47	2071
FF	4103	3693	4270	2237	2739	13	1629
LUTRAM	93	0	0	21	78	0	0
BRAM	103	109	57	2	95	5	15
DSP	1	0	0	0	1	0	0

Table 7.7: Execution time cost (in μs): comparison of common benchmarks using baseline (not garbled), monolithic, and Garblet implementation.

Benchmark	**Baseline (μs)**	**Monolithic (μs)**	**Garblet (μs)**
Garbling Time			
Add_8_1	N/A	9.72	0.0173
Mult_1024_2048	N/A	3,910,000	30.1
MAC_32_1	N/A	828	9.31
AES_128_1	N/A	7,120	6.02
Evaluation Time			
Add_8_1	N/A	0.619	0.00312
Mult_1024_2048	N/A	4,180	9.21
MAC_32_1	N/A	99.1	3.02
AES_128_1	N/A	599	1.02
Communication Time			
Add_8_1	N/A	173,000	0.293
Mult_1024_2048	N/A	8,950,000,000	4,270
MAC_32_1	N/A	576,000	11.0
AES_128_1	N/A	4,910,000	219
Total Execution Time			
Add_8_1	0.017	1,730	**0.313**
Mult_1024_2048	257.17	8,960,000,000	**4,300**
MAC_32_1	43.85	577,000	**842**
AES_128_1	11.83	4,910,000	**226**

The Garblet's garbling engine reduces LUT utilization by 2.67× (from 31,330 to 11,729 LUTs) compared to FASE [174] due to its modular design and DSP offloading. The evaluator's LUT usage shows a 16.57× improvement over the implementation of a monolithic FPGA, demonstrating significant efficiency gains. The Garblet garbling engine also uses 4,103 FFs compared to 11,416 FFs in FASE, achieving a 2.78× reduction, primarily due to the efficient use of dual-port BRAMs, which minimizes the dependency on FF, LUTRAM, Garblet utilizes only 93 LUTRAMs compared to 553 in FASE, representing a 5.95× reduction attributed to the use of BRAMs for memory storage. The BRAM usage in the garbling engine increased by 1.5× (from 68.5 to 103 BRAMs), which is justified by the adoption of dual-port BRAMs for efficient data handling between garbling and evaluation engines. Each Garblet's garbling and evaluator engine incorporates 1 DSP, offload-

Table 7.8: Execution time comparison (in μs) between monolithic and Garblet implementation with one and three engines.

Metric	**2-bit Adder**	**Mult_1024_2048**
# Sub-circuits	3	2048
Garbling Time (μs)		
Monolithic	14	3,910,000
Garblet (One Engine)	0.0591	30.1
Garblet (Three Engines)	0.0377	12.9
Evaluation Time (μs)		
Monolithic	1.47	4,180
Garblet (One Engine)	0.00628	9.21
Garblet (Three Engines)	0.00412	4.66
Communication Time (μs)		
Monolithic	473,000	8,950,000,000
Garblet (One Engine)	0.522	4,270
Garblet (Three Engines)	0.849	6,190

Table 7.9: Execution time and peak memory cost of the circuit decomposition algorithm.

Benchmark	# Sub-circuits	Time (s)	Memory Peak (MB)
2-bit Adder	3	3.1	394
Mult_1024_2048	2048	65901	11387

ing specific computational tasks and further reducing LUT utilization. This trade-off is also justified by the significant reductions in computation time and enhanced scalability, which are critical for large-scale secure computations. Table 7.6 shows the resource utilization of each module. The garbling and evaluator engines can be instantiated multiple times as long as the platform supports the resource requirements.

Execution Time Cost

Effect of Communication Reduction.

We compared the execution time cost of common benchmark functions for implementations on baseline (not garbled), monolithic FPGA, and Garblet. The Garblet framework was tested in two setups: one with a single garbling and evaluator engine for resource efficiency, and another with multiple engines for better performance. Table 7.7 shows the execution times for benchmarks like AES, multiplication, MAC, and an 8-bit adder.

Garblet reduces communication costs by up to $59,000\times$ compared to monolithic implementations and improves performance by up to $5,500\times$ for benchmarks like the 8-bit adder. When compared to Baseline implementations, which do not involve multiple parties and therefore do not require

garbling, evaluation, or communication, Garblet introduces additional overhead for security. For the benchmarks, Garblet is about 15.65 times slower for the 8-bit adder, 257 times slower for multiplication, 745 times slower for MAC, and 47.8 times slower for AES.

Circuit Decomposition Execution Time: We evaluated the cost of our circuit decomposition algorithm on two benchmark functions: (I) a 2-bit adder and (II) `Mult_1024_2048`. The decomposition algorithm was implemented on the PC. Note that this pre-processing is performed offline (before running the GC protocol) and does not impact the framework's online performance. Multiple sub-circuits enable parallel computation, enhancing the framework's performance. Table 7.9 shows the execution time and peak memory cost of the circuit decomposition algorithm. As the number of sub-circuits increases, the execution time rises exponentially.

7.4.8 Acceleration Using Multiple Garbling/Evaluator Engines

The Garblet benefits extend beyond communication cost reduction. Using multiple garbling and evaluator engines, along with circuit decomposition, enables parallel execution of computation tasks, significantly improving overall performance. We evaluated this performance gain by running the 2-bit adder and `Mult_1024_2048` benchmark functions using one and three garbling/evaluator engines. Table 7.8 shows the execution time cost comparison between monolithic Garblet with one and three engines. Using three engines significantly reduces garbling and evaluation times for both benchmark functions. For the 2-bit adder, garbling time decreased by 1.57×, and evaluation time reduced by 1.52×. Communication time also shows a reduction, demonstrating minimal overhead with multiple engines. For the `Mult_1024_2048` benchmark, the benefits of parallel processing are even more pronounced, with garbling time reduced by 2.34× and evaluation time by 1.98×. These results demonstrate that using multiple engines significantly accelerates framework performance while minimizing communication costs. Parallel processing of sub-circuits enables efficient handling of complex computations, making it a viable solution for time-sensitive applications.

Garblet provides a robust hardware-based secure computation framework that effectively mitigates side-channel vulnerabilities at the hardware level. By leveraging chiplet-based architectures, dedicated OT modules, and opti-

mized evaluator engines, Garblet ensures that critical operations such as key management and circuit evaluation remain isolated from adversarial influence [158]. This hardware-centric approach significantly reduces traditional side-channel leakage, making Garblet resilient against power and EM attacks [367, 101].

However, while Garblet secures execution on dedicated hardware, software-based garbled circuit (GC) implementations remain vulnerable to another class of attacks, timing SCA [42]. These attacks exploit variations in execution time to infer secret inputs, bypassing the cryptographic protections of GC protocols [209, 42].

This is precisely where Goblin comes into play, exposing timing side-channel vulnerabilities in software-based GC implementations [153]. Goblin demonstrates that despite the theoretical security guarantees of GC-based protocols, optimizations commonly used in software frameworks, such as free-XOR [214] and half-gates [429], introduce subtle timing variations that leak information about secret inputs. Unlike power or EM analysis, Goblin operates in a purely software domain, requiring only a single execution trace and no prior profiling of the system. This makes it a practical and highly effective attack against GC implementations deployed in cloud or distributed computing environments [317].

The timing leakage identified by Goblin highlights a critical security gap in software-based MPC implementations, even those built on secure cryptographic primitives. While hardware-based frameworks like Garblet can enforce constant-time execution due to their dedicated architectural design, software GC frameworks rely on compiler optimizations, memory access patterns, and runtime behavior, which can lead to unintended variations in execution time [143, 171].

In the next sections, we explore how Goblin exploits execution time variations, the adversary model it assumes, and its impact on widely-used GC frameworks. By understanding these vulnerabilities, we can identify potential countermeasures to further strengthen software-based secure computation frameworks against timing SCA.

7.5 Timing Side-Channel Attacks on Secure Computation

Timing side-channel vulnerabilities arise when the execution time of a software program exhibits dependencies on secret variables, allowing an adversary to infer confidential information based on variations in execution time. These side-channel leaks can be broadly classified into two main categories: instruction-related and cache-related timing channels.

The first category, instruction-related timing channels, occurs when the execution path, including the number and type of instructions executed, varies depending on secret-dependent conditions. In contrast, cache-related timing side-channels stem from the interaction between memory accesses and the cache subsystem, where differences in execution time are observed based on whether a memory access results in a cache hit or a miss. For example, while a cache hit may require only a few CPU cycles, a cache miss can impose significant delays, potentially spanning hundreds of cycles cf. [408].

An adversary aiming to exploit timing-based vulnerabilities can manually analyze the source code, examining execution paths for discrepancies in instruction execution time. A notable example of this can be seen in the timing attack against the TinyGarble framework [361], where unbalanced if-else branches led to observable execution time variations. By meticulously inspecting the code line by line, an attacker can identify and leverage such timing inconsistencies to extract sensitive information.

However, conducting manual timing analysis is a challenging and resource-intensive process, requiring both an in-depth understanding of the source code and the execution environment. To address this complexity, a variety of automated tools have been developed to systematically analyze and detect timing side-channel leaks in software implementations.

For this purpose, we employ a state-of-the-art tool that has been widely referenced in the literature for timing side-channel analysis [191], namely SC-Eliminator [408]. One of the key advantages of SC-Eliminator is its capability to analyze C/C++ source code, making it particularly relevant for evaluating existing garbling frameworks. This tool leverages the LLVM compiler infrastructure to perform static analysis, systematically identifying sensitive variables and pinpointing timing leaks based on their interactions with the program's execution structure. Given a program and a predefined list of secret inputs, SC-Eliminator efficiently assesses the timing vulnerabil-

ities associated with different execution paths, assisting in the mitigation of potential side-channel threats.

GC Tools. To investigate whether garbled circuit (GC) frameworks are susceptible to timing SCA, we analyzed five widely used open-source tools written in C and C++. These frameworks predominantly support the AES-NI (Advanced Encryption Standard New Instruction) set, which is designed to accelerate AES encryption operations on modern processors (for a detailed analysis of these tools, see [160]). Leveraging AES-NI has significantly improved the efficiency of GC computation, as it reduces the cost of performing AES encryptions, a critical operation in garbling and evaluating circuits.

One of the fundamental libraries in this space is JustGarble [34], a framework developed for garbling and evaluating Boolean circuits. Licensed under the GNU GPL v3, JustGarble does not inherently support communication or circuit generation, making it less suitable as a standalone general-purpose secure computation framework. However, its highly optimized design has made it a foundation for several other frameworks, including [362, 266, 194, 147, 144, 140].

The primary reason for JustGarble's efficiency lies in its ability to perform only one AES invocation per garbled gate, making it significantly faster than previous GC implementations [34]. This efficiency stems from JustGarble's use of cryptographic permutations, wherein fixed-key AES functions as a public random permutation [34]. While this assumption has been debated in prior works cf. [144, 147], JustGarble's theoretical underpinnings and superior performance have led to its widespread adoption in various secure computation and GC-based MPC frameworks [266, 140].

Recognizing the strengths of JustGarble, Songhori et al. [361, 362] introduced TinyGarble, a highly optimized and compact sequential garbling framework. TinyGarble extends JustGarble's capabilities by integrating circuit synthesis and compression techniques, making it directly applicable in practical MPC applications [160]. Its workflow consists of three primary steps: (1) converting a function defined in Verilog into a netlist representation, (2) translating the netlist into a custom SCD, and (3) securely evaluating the resulting Boolean circuit using a garbled circuit protocol.

Compared to JustGarble, TinyGarble represents a significant improvement as it incorporates recent protocol optimizations and circuit synthesis advancements. However, despite its enhanced flexibility and suitability for

hardware circuits, modifications introduced in TinyGarble have inadvertently led to timing side-channel vulnerabilities. These vulnerabilities, which arise from imbalanced execution paths.

In contrast to TinyGarble, which is based on Verilog, Obliv-C is an extension of C designed to support GC-based secure two-party computation [427]. Obliv-C extends the standard C programming language by introducing the `obliv` qualifier, which can be applied to variables and functions. This qualifier enforces strict typing rules, ensuring that secret values remain confidential unless explicitly revealed. Within an `obliv` block, all operations execute in a manner that conceals conditional branches and control flow dependencies, reducing leakage risks cf. [427, 426].

Beyond its inherent security properties, Obliv-C facilitates the development of modular secure computation libraries, making it a versatile tool for a wide range of privacy-preserving applications. It has been employed in various domains, including privacy-preserving linear regression [115], decentralized certificate authorities [193], collaborative ML models [382], secure classification of encrypted emails [148], and privacy-preserving stable matching protocols [98].

Apart from JustGarble, TinyGarble, and Obliv-C, we also examined two additional C++-based frameworks: EMP-toolkit [246] and ABY [95]. EMP-toolkit provides a comprehensive suite of MPC frameworks, supporting efficient execution of circuit-based protocols through integrated circuit generation and cryptographic libraries. Meanwhile, the ABY library is designed to enable seamless switching between multiple secure computation paradigms, including optimized implementations of Yao's GC, Boolean sharing, and arithmetic sharing. This flexibility allows developers to mix protocols to optimize computation cost and performance.

By evaluating these diverse GC frameworks, we aim to assess their susceptibility to timing SCA and understand how their architectural and optimization choices impact security. The following sections will delve deeper into potential vulnerabilities within these frameworks and discuss countermeasures to mitigate timing-related leaks.

Our Observations. As discussed earlier, we initiated our analysis by assessing the feasibility of mounting a timing SCA against the aforementioned GC frameworks. In such an attack scenario, an adversary seeks to exploit potential imbalances in conditional branching, particularly within `if-else`

Table 7.10: The number of leaky IF conditions (IF) in various frameworks (for a detailed report, refer to Appendix A).

Framework	IF
TinyGarble [361] (half-gate)	4
TinyGarble [361] (free-XOR)	7
JustGarble [187]	11
EMP-toolkit [246]	0
Obliv-C [426]	4
ABY [95]	0

statements. One primary concern arises from the fact that free-XOR and half-gate optimized Yao's GC protocols execute different operations when generating garbled inputs. If these sensitive operations are implemented using non-constant-time execution or branch-dependent assignments, they could introduce timing variations that leak secret-dependent information.

To evaluate this risk, we applied SC-Eliminator [408] to five widely used GC frameworks: TinyGarble [361], JustGarble [187], EMP-toolkit [246], Obliv-C [426], and ABY [95]. The results of this analysis, summarized in Table 7.10, reveal the number of conditional branches (`if-else` statements) flagged as potential sources of timing leakage.

Upon closer examination of the flagged branches, we identified instances where garbled input generation was performed in a secret-dependent manner. Specifically, within certain frameworks, unbalanced `if` statements were found in the garbled input computation, meaning the execution path varied based on the secret data. This finding strongly suggests that an attacker could successfully exploit these timing variations to infer sensitive information.

According to the results presented in Table 7.10, EMP-toolkit [246] and ABY [95] exhibited no leaky `if` statements. However, it is crucial to emphasize that while SC-Eliminator did not detect branch-based vulnerabilities in these frameworks, the absence of such findings does not categorically rule out other forms of timing-based attacks. For instance, vulnerabilities may still arise due to indirect sources such as memory access patterns or microarchitectural behaviors, which were beyond the scope of SC-Eliminator's static analysis.

Building upon these observations, we introduce Goblin, an attack framework that leverages timing side-channel leakage originating from existing

unbalanced `if` statements in GC frameworks. The following sections will elaborate on the methodology of Goblin and demonstrate its efficacy in exploiting timing vulnerabilities in practical settings.

7.5.1 Goblin and Its Building Blocks

Goblin operates through a structured sequence of steps designed to exploit timing side-channel vulnerabilities in GC frameworks. The primary steps of its attack flow are as follows:

(1) The first phase involves populating the cache with junk data using a dedicated junk generator (JG). This process is intended to force the eviction of the garbler's secret from the cache, thereby increasing the CPU core's access time when retrieving the global secret (R) from memory. By doing so, Goblin can capture execution timing variations corresponding to the evaluation of gates connected to input wires (i.e., gates located in the input layer). It is important to note that for certain GC frameworks, Goblin remains effective even without leveraging timing variations caused by cache effects, as further discussed in Appendix B.

(2) The second phase consists of measuring execution time on the CPU. Specifically, this includes recording the duration required to generate the garbler's token, which directly correlates with the input size. This timing data serves as a crucial element in uncovering variations that may reveal sensitive computations.

(3) In the final phase, Goblin reconstructs the garbler's secret (i.e., the garbler's input) by pre-processing the collected CPU cycle measurements and applying a clustering algorithm. This step enables the adversary to systematically infer confidential input values by analyzing subtle timing discrepancies.

By executing this sequence of operations, Goblin effectively exploits timing side-channels inherent in GC implementations, demonstrating the need for robust countermeasures against such vulnerabilities.

7.5.2 Our Eviction Method: Junk Generator

In our threat model, we assume that the server and computing parties operate independently, meaning that the adversary does not possess knowledge of the cache slice function or the victim's physical memory addresses. Consequently, employing a static eviction set and a fixed access pattern strategy is infeasible [141].

Additionally, implementing a dynamic eviction set with a static access pattern requires prior knowledge of the target's cache replacement policy, which is generally unavailable to the adversary [141]. Therefore, our Junk Generator (JG) follows a dynamic eviction set and dynamic access pattern strategy [141]. In essence, our JG extends the dynamic eviction methodology introduced in [141], improving its efficiency and adaptability.

Our attack shares similarities with Evict+Time attacks described in prior literature [297]. Specifically, JG continuously accesses memory by performing frequent read and write operations, similar to [285]. However, unlike prior approaches, where the adversary must first determine which critical memory regions are accessed during encryption, Goblin bypasses this step. Instead, it directly exploits the timing variations between garbling a bit value of "1" versus "0," allowing the adversary to infer input bits without additional analysis.

To amplify this timing difference, the JG algorithm recursively generates eviction sets and performs randomized memory accesses. While this approach requires multiple eviction tests, it operates with minimal system knowledge, enabling automated attacks on previously unknown architectures. Moreover, it is computationally more efficient than the static eviction set combined with a dynamic access pattern strategy [141].

Although cache eviction can be achieved by directly reading from a cache line [285], we opted to generate junk dynamically to circumvent CPU memory management constraints [141].

The Junk Generator (JG), as outlined in Algorithm 13, operates as follows. The iteration parameter n determines the number of cell indexes in the array that are summed and used to update another array cell. This process continues iteratively until reaching the last index, $(Size - 1)$. At this stage, JG generates new random values and repeats the process indefinitely, leading to cache disruption and potentially evicting critical data, such as the global parameter R utilized in free-XOR [214] and Half-Gates [428] optimizations.

While simple `For` loops could be employed for junk generation, we implemented a recursive function to allow indefinite junk creation, accommodating the unpredictable duration of the circuit garbling process.

7.5.3 Measuring Execution Time on CPUs

Once the Junk Generator (JG) enhances the variation in execution time based on input-dependent computations, the next step is to measure this time pre-

Algorithm 13: Junk Generator Algorithm

```
Input: Size = size of cache/64
Output: Junk ← Array[Size] and n ← 1
Function JG(n):
    while User Interrupt do
        if n == 1 then
            Seed ← t_time ;
            Junk[0...3] ← rand(Seed) ;
            n ← n + 1 ;                                  // Initiate recursive algorithm
            return JG(2) ;
        else if n == (Size − 1) then
            return JG(1) ;
        else if n ≠ (Size − 1) and n ≠ 1 then
            i ← n ;
            for i ≤ (Size − n − 1) do
                Junk[i + n + 1] ← Junk[i] + Junk[n] ;
            n ← n + 1 ;
            return JG(2) ;
```

cisely. According to Martin et al. [250], three primary sources can be utilized for measuring time without directly interfering with the execution of the target software cf. [243]: (1) Internal hardware-based time sources, such as timestamp counters that provide fine-grained cycle-level timing information. (2) External time sources, including external hardware components that trigger interrupts at specific intervals to measure elapsed execution time indirectly. (3) Virtual clock implementations, which leverage shared memory in multi-processor systems to construct a consistent clocking mechanism [297].

Without loss of generality, we focus on using an internal hardware time source, specifically the `rdtsc` instruction, which provides direct access to the CPU's timestamp counter (TSC). The `rdtsc` instruction is an x86 assembly command that reads the current value stored in the timestamp counter, which is updated with each CPU clock cycle. Due to its high granularity, the resolution of `rdtsc` is determined by the inverse of the CPU frequency, making it capable of measuring execution time with nanosecond precision on modern processors.

In general, the `TSC` register is shared across different user privilege levels [243], meaning it can be accessed under the following conditions: (1) A privileged or non-privileged user who has direct control over the CPU execution. (2) A service provider operating in a cloud environment where the processor is shared with multiple tenants, including the victim [250]. (3) A virtual machine (VM) user, either with privileged or non-privileged access,

who runs a process on a shared processor where the victim's computations are also being executed (e.g., cross-VM attacks) [243].

Consequently, an attacker may have different levels of access to the CPU running the garbling scheme. This access could occur directly on the same processor where the secure computation is performed, on a cloud provider's system where the execution environment is shared, or in a cross-VM setting where multiple tenants share CPU resources. The key distinction between privileged and non-privileged attackers is that the former can precisely control the garbler's execution and introduce intentional interruptions, whereas the latter cannot directly halt or manipulate execution flow. However, even an unprivileged attacker can still infer execution timing by monitoring when the garbling process begins or by leveraging external triggers, such as cache-based side-channel signals [343]. Once the adversary detects the start of the garbling process, they can align the collected timing traces accordingly, as demonstrated in prior works [235].

For the purposes of our attack, and in line with previous timing-based side-channel methodologies [263, 186], we assume that timing measurements are already aligned. To validate our approach, we have instrumented the source code of publicly available GC frameworks by inserting the `rdtsc` instruction before and after the core garbling function. By computing the difference between these timestamps, we accurately determine the execution time of the garbling process. This approach ensures precise timing measurements and serves as the foundation for further analysis of timing leakage in secure computation frameworks.

Resolution of timing measurements. The timestamps retrieved via the `rdtsc` instruction generally provide a resolution within the range of 1 to 3 CPU cycles on modern processors cf. [234]. For instance, on AMD processors up to the Zen microarchitecture, it is possible to achieve cycle-accurate resolution. However, more recent AMD architectures have significantly lowered the resolution, as the timestamp counter register is only updated every 20 to 35 cycles. A contrasting example can be seen in Intel Core $i7-7700$ processors, which were used in this study, where the `rdtsc` register is updated on every cycle [183].

Although one might assume that reducing the resolution of the timestamp counter makes mounting attacks more challenging, Goblin remains unaffected by this limitation. The reason is that Goblin primarily relies on measuring the difference between two consecutive readings rather than absolute timestamps. Since both readings share the same resolution, the attack remains

effective . Unlike other timing attacks that rely on repeated measurements and averaging due to variations in `rdtsc` resolution, Goblin does not require multiple executions. Instead, it directly exploits the variations observed from a single execution, leveraging the differences between timestamps.

Additionally, it is crucial to emphasize that Goblin is a single-trace attack. Due to the gate-by-gate execution model in garbled circuit (GC) frameworks, the timing differences captured from `rdtsc` naturally form a sequence of timestamps corresponding to individual gate operations. This granularity allows the attack to infer secret-dependent variations efficiently. Furthermore, while this study focuses on a timing attack using `rdtsc`, it is important to note that alternative methods for obtaining precise timing information could also be employed to achieve similar results.

7.5.4 Recovering Garbler's Input

Counting the Gates in the Input Layer

As outlined in our adversary model, we assume that the attacker is neither the garbler nor the evaluator, meaning they lack prior knowledge about the circuit structure, the number of input gates, or the gate types in the input layer. Goblin exploits this lack of knowledge by recovering such information through an analysis of the execution patterns of GC frameworks. Here, we describe how Goblin achieves this when JustGarble is employed as the garbling framework. JustGarble is particularly relevant due to its widespread adoption in various GC-based systems and its fundamental role in the core of several other garbling frameworks, including those evaluated in our study [362, 427].

Listing 1 provides a high-level description of JustGarble's primary functions. Within the listing, variables such as `NF`, `LF`, `GT`, `IF`, `INL`, `WL`, `GC`, and `OL` (defined in Lines 1–9) correspond to the number of fan-outs, the location of fan-outs, gate types, input fan-out values, initial input values, wire labels, the garbled circuit, and output labels, respectively.

According to JustGarble's protocol flow (as outlined in Listing 1), the first step in the garbling process involves constructing the garbler's labels for logical zero and one values (`IL`) using the `createNewWire` function (Listing 1, Line 5). Once these labels are initialized, the parser function—responsible for managing circuit information—is executed. Specifically, the `createInputLabels` function (Listing 1, Line 3) processes the Simple Circuit Description (`SCD`) file along with the `g_init` files, both of which store crucial details about the

```
1  def JustGarble(g_init, SCD):
2      NF, LF, GT = createNewWire(g_init, SCD)   #Pasrses the circuit, locate
       the fan-outs, and generates wire labels.
3      IF, INL = createInputLabels(NF, LF) #Fills tokens to input fan-outs (
       called twice per garbler input).
4      GC, OL, TT = garbleCircuit(IF, IFS, WL, GT) #Generates garbled tables
       and Garbled output tokens.
5  def createNewWire(g_init, SCD):
6      for i in SCD[0]: #first line of SCD, which contains the information
       about input layer gates
7          IF[i][0] = randomBlock();
8          IF[i][1] = xorBlocks(R, IF[i][0]);
9  def garbleCircuit(IFS, WL, GT):
10     R = AESEcbEncryptBlks(AES_Key)
11     if(IFS == known):
12         GC, OL = HalfGarbleGate(GT, IF)
13         return GC, OL
14     else: #(IFS == secret):
15         if(GT == XORGATE):
16             OL = XorBlock (IFS, R) #free-XOR optimization
17         else: #if(GT == ANDGATE)
18             mask1, mask2, mask3, mask4=AESEcbEncryptBlks(AES_Key,4)
19             #AND encryptions
20             OL = XorBlock(mask1 , mask2)
21             if (IFS == 1):
22                 OL = XorBlock(OL , R);
23         GC = [XorBlock(OL, mask3), XorBlock(OL, mask4)]
24     if(gate_location is in input_layer): #Generates associate garbler tokens
        to be transferred to Evaluator.
25         if(g_init == 0):
26             TT = IF;
27         else:
28             TT = xorBlocks(R, IF);
29     return GC, OL, TT
```

Listing 1: Protocol flow of primary functions of JustGarble.

circuit structure and the garbler's input values.

Through this parsing operation, the function extracts information about the circuit topology (`GT`) and determines the placement of input fan-in and fan-out gates (`LF` and `NF`), based on the data stored in the `g_init` file. For every input wire, `createInputLabels` is invoked twice—once to generate the label for the garbler and once for the evaluator. Consequently, the total number of calls to `createInputLabels` is twice the number of gates present in the input layer.

At this stage, Goblin begins counting the number of times `createInputLabels` is called, inferring that the number of input-layer gates is equal to half of the total function calls. Once this phase is completed, JustGarble proceeds to garble the circuit, executing the `garbleCircuit` function (Listing 1, Line 9) to process each gate, beginning with those in the input layer. Since the gar-

bler's and evaluator's inputs are first mapped onto these gates before the rest of the circuit is processed, Goblin is able to determine execution timing at this stage. By monitoring the execution cycles, Goblin effectively counts the CPU cycles associated with each gate in the input layer, thereby extracting critical insights about the garbler's input.

This method allows Goblin to infer secret-dependent operations in JustGarble-based GC frameworks without requiring direct access to the circuit description or runtime information. The next phase of Goblin involves leveraging this timing information to further refine the attack and recover the garbler's actual input values.

Goblin Against Free-XOR Optimization.

When the framework initiates the garbling process, it sequentially generates the output labels (`OL`) and constructs the corresponding garbled tables (`GT`) based on the order specified in the `SCD` file. Similar to many modern garbling frameworks, JustGarble employs the free-XOR optimization technique to efficiently generate garbler tokens for input values of 1. However, this optimization introduces a critical security weakness that Goblin exploits.

In particular, when free-XOR is enabled, the `GarbleCircuit` function (Listing 1, Line 9) entirely bypasses the instructions between Line 11 and Line 14 in Listing 1. Consequently, regardless of whether an input is public or secret, the framework first verifies the gate type (`GT`) and treats all inputs as secret. If the gate type corresponds to an `XOR`-based operation—including gates categorized as `INV`, `XOR`, and `XNOR` in GC protocols—the function computes the output label by simply XORing the labels for logical values 0 and 1 (Listing 1, Line 16). In contrast, for non-XOR gates, such as `AND`, `OR`, and their negated forms, the function constructs the output label by performing a series of cryptographic encryptions (Listing 1, Lines 18 to 22).

A closer inspection of the garbling function reveals that in the final stage of encryption (Listing 1, Line 14 and between Lines 25 and 28), if the garbler's input value is "1", the function executes an additional encryption, performs an extra memory access, and carries out one additional XOR operation. This results in a measurable dependency between execution time and input values, a vulnerability that Goblin is designed to exploit.

More specifically, when garbling `AND`-type (non-XOR) gates—including `AND/NAND`, `OR/NOR`, `ANDN`, `ORN`, `NANDN`, and `NORN`—the presence of an unbalanced `if` condition results in a longer execution time for cases where the input value is "1". The primary cause of this time discrepancy is the increased number of memory accesses associated with these operations.

Goblin capitalizes on this execution time variance to infer the garbler's input value. If the global constant R remains available in the L1 cache, the timing difference between garbling input values 0 and 1 is relatively minor, often overshadowed by the overhead of encryption itself. However, Goblin amplifies this difference using its Junk Generator (JG). The JG process actively fills the cache with junk data in parallel with the execution of the `createNewWire` function (Listing 1, Line 5). This forced cache eviction forces the CPU to reload R from RAM into the L1 cache, artificially increasing the execution time discrepancy between generating tokens for input values 0 and 1.

To maximize the effectiveness of the JG, Goblin first determines the exact CPU core and thread executing the garbling process by calling the `LSCPU` instruction. Subsequently, it attempts to assign the JG task to the same thread executing the garbling function. If direct assignment to the same thread is not possible, the JG is at least scheduled on the same CPU core to maximize cache contention.

It is important to note that no special privileges or elevated access rights are required to execute the JG alongside the garbling process. The JG's primary operation involves filling the shared L3 cache, which is accessible to all processes running on the same processor. However, when assigned to the same core as the garbling process, JG operates even more efficiently by filling the L1 and L2 caches first, thereby accelerating cache evictions and reducing measurement errors. This strategic placement further amplifies the timing differences exploited by Goblin, increasing the precision and SR of the attack.

Goblin's Attack on Half-Gate Optimization. Although JustGarble does not inherently support half-gate optimization, later frameworks such as TinyGarble and Obliv-C have incorporated it. Despite this addition, Goblin remains effective against these frameworks. When half-gate optimization is enabled, the function `HalfGarbleGate` (refer to Listing 2) is invoked by `GarbleGate`.

In cases where the input value (`IF`) is zero and the gate type (`GT`) corresponds to `ANDGATE`, the function skips the garbling operation and directly assigns a constant value to `OL`, thereby reducing execution time compared to the garbling process for an input value of one or other gate types. However, when the input value is one, the function proceeds with encryption (Listing 2, line 11), introducing an asymmetric execution path due to the conditional `if` statement. This creates a dependency between execution time and the

```
1  def HalfGarbleGate(GT, IF):
2      R = AESEcbEncryptBlks(AES_Key)
3      mask1, mask2 = AESEcbEncryptBlks(AES_Key,2)
4      if(IF[0] == 0):
5          if(GT == ANDGATE):
6              OL = mask1 #XorBlock(mask1, 0)
7          else: #if(GT == XORGATE):
8              OL = XorBlock(mask1, IF[1])
9      if(IF[0] == 1):
10         if(GT == XORGATE):
11             OL = mask1 #XorBlock(mask1, 0)
12         else: #if(GT == ANDGATE):
13             OL = XorBlock(mask1, R)
14     GC = XorBlock(OL, mask2)
15     if(gate_location is in input_layer): #Generates associated garbler
       tokens to be transferred to Evaluator.
16             if(g_init == 0):
17                 TT = IF;
18             else:
19                 TT = xorBlocks(R, IF);
20     return GC, OL, TT
```

Listing 2: HalfGarbleGate function flow.

input value, making it susceptible to timing side-channel analysis.

As with free-XOR optimization, Goblin exploits the execution time variations caused by the unbalanced `if` conditions present in Listing 2, specifically on lines 3 and 8.

Following this stage, the remaining steps are not relevant to Goblin, as they do not reveal any further information about the secret (i.e., the garbler's input). The extracted information is already sufficient to mount the attack. Consequently, Goblin can proceed to the offline phase to complete the attack.

Pre-processing the Acquired CPU Cycles. As previously discussed, when utilizing free-XOR optimization, an attacker anticipates a noticeable difference in the CPU cycle count between `INV`, `XOR`, and `XNOR` gates versus other gate types, such as `AND/NAND`, `OR/NOR`, `ANDN`, `ORN`, `NANDN`, and `NORN` gates.

This substantial discrepancy arises because, in free-XOR optimization, XOR-type gates are garbled using a simple XOR operation, which incurs minimal computational cost and requires only a few CPU cycles. In contrast, garbling other gate types, such as `AND`, necessitates additional operations such as memory reads/writes and cryptographic key generation. These additional steps introduce extra memory accesses, significantly increasing the overall CPU cycle count. This phenomenon is evident when analyzing the computational structure of these gates.

When applying clustering techniques to infer the garbler's input in a non-profiled manner, this discrepancy causes the gate types themselves to dominate the clustering centroids rather than the input values. To mitigate this issue, Goblin first categorizes CPU cycle measurements into subgroups corresponding to the distinct gate types present. Specifically, it separates `AND`-type gates (`AND/NAND`, `OR/NOR`, `ANDN`, `ORN`, `NANDN`, and `NORN`) from `XOR`-type gates (`INV`, `XOR`, and `XNOR`, collectively referred to as `XOR` gates) based on the median CPU cycle values.

Following this, each subgroup undergoes z-score normalization to standardize CPU cycle distributions across different gate types. Finally, the normalized data is recombined while preserving the original execution order of captured CPU cycles. By normalizing cycle counts, Goblin effectively reduces the disparity between XOR and AND-type gate computations, improving the SR of the attack.

The pre-processing step becomes more intricate when half-gate optimization is enabled. Our analysis reveals that in addition to XOR gates displaying significantly lower CPU cycle counts, OR/NOR gates exhibit a distinct, disproportionate increase in execution time. This behavior stems from the inherent structure of gates whose truth tables contain an odd number of ones, such as `AND`, `NAND`, `OR`, and `NOR`.

Fundamentally, these gates can be expressed as:

$$G : (v_a, v_b) \rightarrow (\alpha_a \oplus v_a) \wedge (\alpha_b \oplus v_b) \oplus \alpha_c$$

where v_a and v_b represent logical input values, while α_a, α_b, and α_c are predefined constants [428].

For an AND gate, all α values are set to zero, whereas for an OR gate, they are set to one. Consequently, it is unsurprising that CPU cycle measurements for garbling OR/NOR gates form an entirely distinct cluster from the rest.

Additionally, another notable observation is that garbling OR/NOR gates with an input of "0" generally takes longer than garbling AND/NAND gates with an input of "1". This timing discrepancy further reinforces Goblin's capability to differentiate secret-dependent computations, thereby facilitating precise key recovery.

Therefore, unlike the free-XOR optimization case where AND/NAND and OR/NOR gates can be classified under the same category, distinguishing between AND/NAND gates with an input of "0" and OR/NOR gates with an input of "1" becomes challenging. This overlap leads to inaccurate clustering,

as the algorithm incorrectly groups both types into a single cluster, even though they should be separated based on their distinct input values.

To address this issue, Goblin introduces an additional data scaling technique before normalization, ensuring that the patterns align with those of other gate types (i.e., higher CPU cycles for input 1). The first step, similar to the free-XOR scenario, involves partitioning the collected CPU cycles $\{c_i\}_{i=1}^{n}$ into subsets corresponding to different gate types: XOR/XNOR, AND/NAND, and OR/NOR.

To achieve this, Goblin calculates the 66^{th} percentile of all elements in $\{c_i\}_{i=1}^{n}$ and assigns those exceeding this threshold to the OR/NOR subset, denoted as c_{OR}. The remaining elements are then further divided into AND/NAND and XOR/XNOR subsets using the same methodology as in the free-XOR case. Specifically, the larger elements from the remaining set $\{c_i\}_{i=1}^{n} \setminus c_{OR}$ are assigned to the AND/NAND subset (c_{AND}), determined by the median value. The remaining values are allocated to the XOR/XNOR subset.

Once the subsets are defined, Goblin applies a transformation of the form $t_i = ac_i + b$ for all $c_i \in c_{OR}$, where the coefficients a and b are computed as:

$$a = \frac{\text{Max}(c_{AND}) - \bar{c}_{AND}}{\text{Max}(c_{OR}) - \bar{c}_{AND}}, \quad b = \bar{c}_{AND} - a \cdot \bar{c}_{OR},$$

where $\text{Max}(\cdot)$ represents the maximum value in a subset, and $\bar{c}$ denotes the average of the subset. This transformation ensures that the CPU cycle distribution for OR/NOR gates aligns more closely with the AND/NAND category.

After this transformation step, normalization is performed following the same procedure as in the free-XOR case, ensuring consistency in data representation and improving clustering accuracy.

Extracting garbler's input through clustering. Once the pre-processed data is ready, Goblin proceeds with the clustering algorithm to identify each garbler's input bit.

Goblin employs clustering for two primary reasons: (1) there is no clear distinction in CPU cycle counts between XOR and XNOR gates when processing input values of zero and one, leading to an overlap, and (2) there is no predefined reference for CPU cycles corresponding to input zero and one for each gate, requiring a comparative analysis between collected values.

Since Goblin applies normalization to the CPU cycle data, the dominance

of gate types in centroid formation is eliminated. As a result, Goblin clusters the CPU cycles into just two categories, each corresponding to an input value of zero or one, regardless of the gate type.

To extract the input bits, Goblin tracks the maximum CPU cycle value, $\text{Max}(\{c_i\}_{i=1}^{n})$, before normalization. Once the clustering process concludes, all members of the cluster containing this maximum value are labeled as "1", indicating that the garbler's input bit is "1". Consequently, the other cluster comprises c_i values corresponding to the garbler's input bit "0".

7.5.5 Performance Metric

Let $\mathbf{c}_i$ represent a leakage measurement, specifically the number of CPU cycles, corresponding to a garbler input $x = x_1 \cdots x_n$, where n denotes the number of bits associated with n wires forming the garbler's input to the circuit. For example, in a garbled 128-bit AES design, $n = 128$.

To assess the effectiveness of our attack, we compute its SR in recovering the garbler's input from a *single* trace $\{c\}_i^n$. Since Goblin operates as a non-profiling attack, it does not rely on a leakage profile, distinguishing it from profiling attacks that require pre-collected leakage distributions.

For classification, we employ the k-means clustering algorithm as a distinguisher, where each observation c_i is assigned to either cluster p_0 or p_1, representing input bit x_i being "0" or "1", respectively. The SR is formally defined as:

$$\text{SR} := \sum_{j \in \{0,1\}} \sum_{i=1}^{n} \Pr(c_i \in p_j \mid x_i = j). \tag{7.4}$$

In essence, SR quantifies the proportion of correctly recovered bits out of the total n bits in the garbler's input. This definition aligns with the standard evaluation methodology in side-channel analysis literature [369]. Specifically, we consider the SR of order 1, which represents the probability that the correct key is ranked first.

7.5.6 Experimental Results

We evaluated the JustGarble, TinyGarble, and Obliv-C frameworks, all publicly available through their respective GitHub repositories [187, 361, 426]. The garbler and evaluator codes were executed on two separate systems running Linux Ubuntu 20, each equipped with 16 GB of memory and an Intel

Core i7-7700 CPU operating at 3.60 GHz. These systems were interconnected via a LAN cable.

Since the garbling process can access the global secret value R at any point during execution, we ensured that the Junk Generator (JG) was initiated as soon as the garbling process began. This maximized the likelihood that the CPU would fetch R from RAM into the L1 cache. The garbling process's start time was identified using non-privileged CPU instructions that indicate active applications on each core. Additionally, to enhance effectiveness, we assigned the JG to the same CPU core responsible for generating GC on the garbler system.

To collect execution traces consisting of multiple time stamps, we employed the `rdtsc` instruction, as detailed in Section 7.5.3. The collected CPU cycle data was subsequently analyzed using the k-means clustering algorithm implemented in MATLAB 2021.

Results for Benchmark Functions

To assess the effectiveness of Goblin, we targeted widely-used benchmark functions, including 128-bit AES, 288-bit SHA3, 256-bit Multiplier, 128-bit Summation, and 128-bit Hamming, garbled using JustGarble [187], TinyGarble [362], and Obliv-C [426]. Additional results for various input sizes can be found in Section 7.5.8.

For evaluating the attack's performance, we calculated the SR by applying different garbler inputs and analyzing the results. Due to the infeasibility of testing all possible input combinations (e.g., a 256-bit Multiplier requires 2^{256} attack runs), we randomly selected 1000 inputs for testing Goblin. For each selected input, a single trace containing multiple timestamps was captured.

In the k-means clustering algorithm, centroids were initialized at 100 different starting values, and the final clustering result was determined based on the least within-cluster sum of point-to-centroid distances.

Figure 7.19 illustrates the SR when free-XOR or half-gate optimization is enabled. The red lines within the box plots represent the average SR achieved against each benchmark function.

As observed in Figure 7.19(a), the attack demonstrated a higher SR against the AES benchmark compared to, for instance, the 256-bit Multiplier. This outcome can be attributed to three primary factors:

First, since only 1000 inputs were tested, variations in results are expected due to statistical fluctuations.

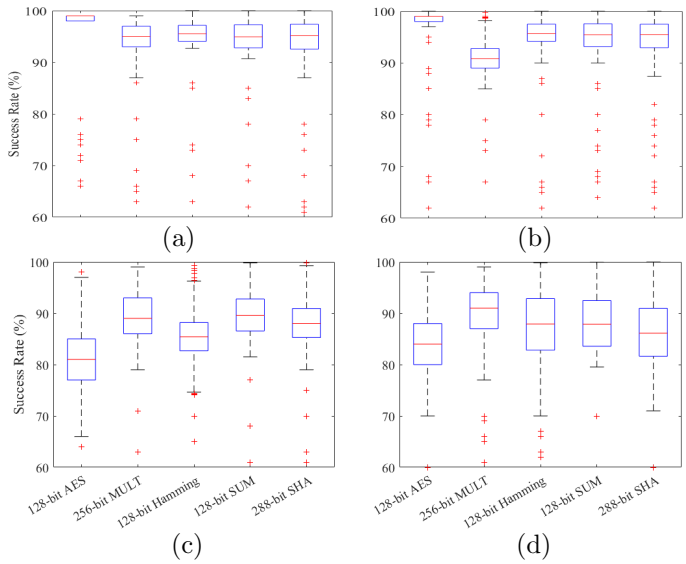


Figure 7.19: SR of Goblin for 1000 randomly chosen inputs applied to GC generated by TinyGarble [362] with (a) free-XOR, (b) half-gate optimizations, (c) JustGarble [187], and (d) Obliv-C [426].

Second, the input layer of the 256-bit Multiplier contains a higher proportion of XOR gates compared to AES. This makes the attack more challenging, as XOR gates exhibit only a subtle difference in execution time between input values "0" and "1".

Third, it is important to note that Goblin operates as a non-profiling, single-trace attack. This means that for each gate—and consequently for each input bit—only a single timing measurement is obtained. As a result, an increase in the number of input bits enhances Goblin's ability to accurately determine their values.

Compared to Figure 7.19(a), Figure 7.19(b), which corresponds to the half-gate optimization, exhibits an overall reduction in SR across the same benchmark functions. This decrease can be attributed to the increase in the number of gate types that need to be identified for the same number of input bits and observations.

Nevertheless, even for circuits incorporating diverse gate types, such as AES, Goblin achieved an average SR exceeding 90%, demonstrating that variations in gate types do not significantly degrade its effectiveness (see Appendix D).

The presence of outliers in Figure 7.19 is primarily due to the imperfect

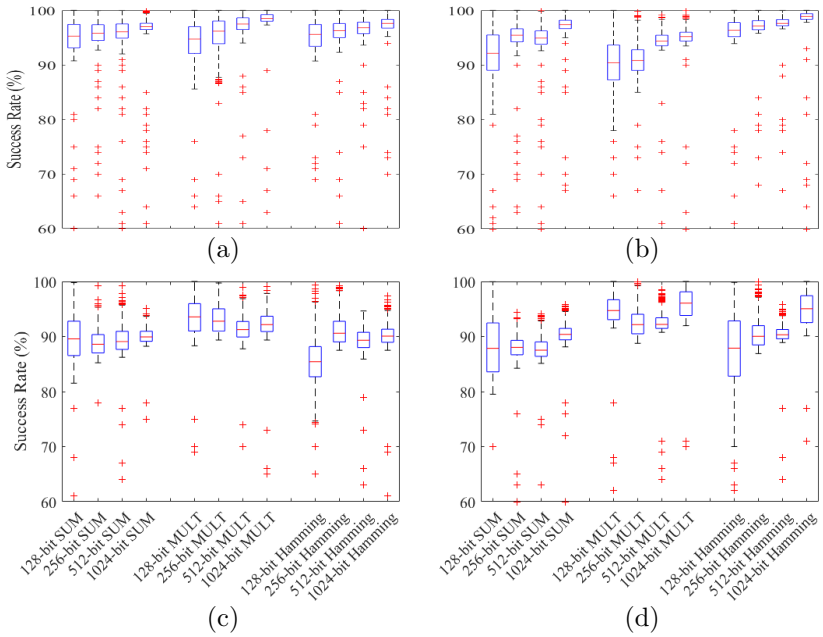


Figure 7.20: SR of Goblin against benchmark functions for a range of input bits garbled by TinyGarble [361] with (a) only free-XOR optimization, (b) half-gate protocol, (c) JustGarble [187], and (d) Obliv-C [426] for 1000 randomly chosen inputs.

process of filling the L3 cache with junk. When R remains available in the L1 cache of the garbler's core, the execution time difference between garbler token generation for input "0" and "1" diminishes.

However, such outliers are infrequent, occurring in only 11 out of 1000 experiments, indicating that the Junk Generator (JG) introduces only a minor error. Even in these rare cases, Goblin successfully revealed the garbler's input with an SR ranging between 60% and 100%.

7.5.7 Scalability of Goblin

To evaluate the scalability of Goblin, we launched the attack against three benchmark functions—MULT, SUM, and Hamming—across a range of input sizes from 128 to 1024. The results are presented in Figure 7.20, where Figure 7.20(a) and Figure 7.20(b) illustrate the outcomes for free-XOR and half-gate optimizations, respectively.

As observed in Figure 7.20(a), increasing the input size consistently im-

proves the minimum and average SR across nearly all cases. This enhancement can be attributed to the fact that larger input sizes provide Goblin with a broader dataset for clustering, resulting in more observations to compare against each other.

Similar to previous experiments, outliers are present in Figure 7.20. A natural question arises: Is it possible to launch Goblin without employing the Junk Generator (JG) to reduce the number of outliers? To address this, we conducted experiments and found that for JustGarble [187] and Obliv-C [426], the SR drops significantly—approaching 50%—when JG is disabled, due to the minimal execution time difference between garbler's input "0" and "1."

However, for TinyGarble [361], it is indeed possible to achieve high SR without utilizing JG (see Appendix B). This is primarily due to TinyGarble's implementation, which generates tokens for garbler input in an input-dependent manner. That being said, enabling JG still enhances Goblin's average SR, offering a trade-off between performance and attack feasibility.

7.5.8 Impact of the Number of Traces

In the previous experiments, we evaluated the effectiveness of Goblin by selecting 1000 random inputs, as capturing CPU cycles for all possible inputs is both impractical and infeasible. This limited selection introduces variance in the results.

To further investigate this, we analyzed the impact of increasing the number of CPU cycle traces. We collected traces after feeding powers of ten (ranging from 10 to $100,000$) random inputs into the 128-bit SUM, Hamming, and MULT benchmark functions—three benchmarks that exhibited relatively high variance in our earlier results (see Figure 7.19). Figure 7.21 illustrates the SR of Goblin as a function of the number of CPU cycle traces.

As evident from the figure, increasing the number of traces consistently improves the SR of Goblin. For larger trace counts, SR variance diminishes, with the average stabilizing around 97% for all cases except for the 128-bit MULT benchmark. This deviation can be attributed to the variation in gate types, as previously discussed.

It is important to note that Goblin is a single-trace attack, meaning each trace is processed independently. Thus, while increasing the number of traces does not improve the effectiveness of individual attacks, it reduces overall variance in the results. Consequently, to obtain a more precise assessment

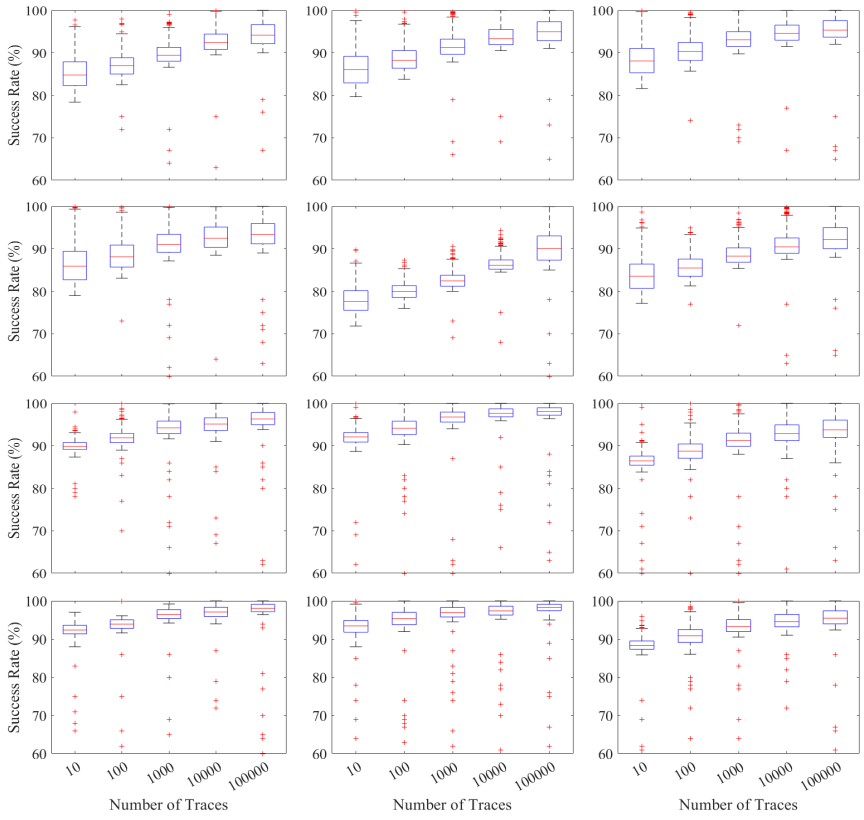
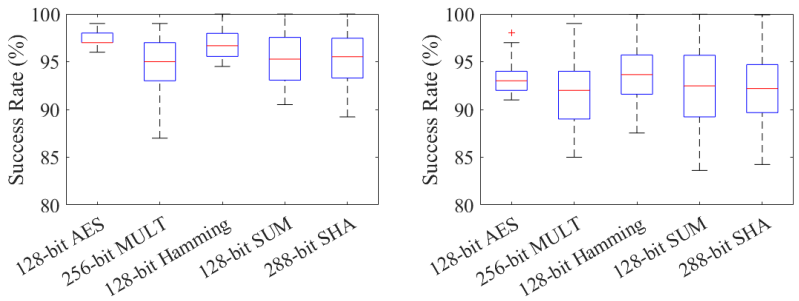


Figure 7.21: SR of Goblin against (a) 128-bit SUM, (b) 128-bit Hamming, and (b) 128-bit MULT for a range of 10-100,000 randomly chosen inputs (first to last row: JustGarble [187], Obliv-C [426], TinyGarble [361] with free-XOR, and with half-gate optimizations).

of Goblin's effectiveness, it is recommended to evaluate a larger number of traces.

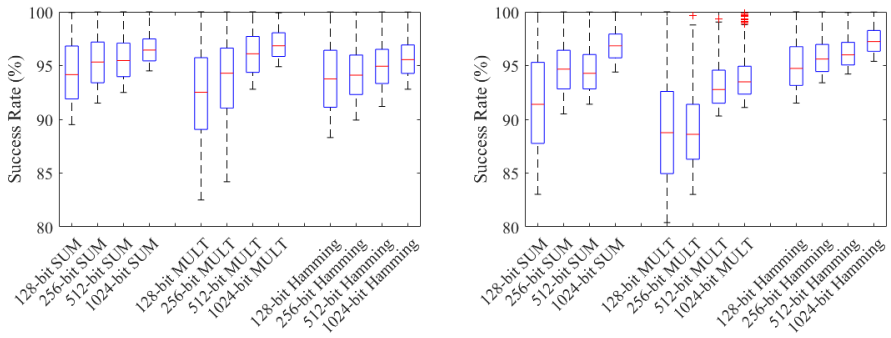
We initially limited our experiments due to the time-intensive nature of collecting traces across all benchmark functions. However, comparing results for 1000 and 100,000 traces reveals that the improvement in the average SR is marginal, further validating Goblin's effectiveness even with a moderate number of traces.

To evaluate the impact of an implementation where not all timing side-channel vulnerabilities are addressed, we executed Goblin against TinyGarble



(a) (b)

Figure 7.22: SR of Goblin for 1000 randomly chosen inputs given to GC garbled by TinyGarble [362] when (a) only free-XOR or (b) half-gate optimization is enabled and JG is disabled.



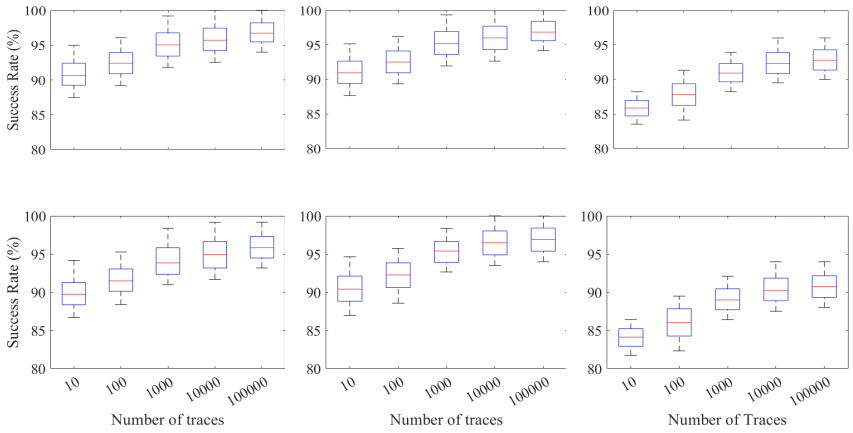
(a) (b)

Figure 7.23: SR of Goblin against MULT, SUM, and Hamming benchmark functions for a range of inputs garbled by TinyGarble [361] when (a) only free-XOR optimization, (b) half-gate protocol is enabled, and JG is disabled.

with the Junk Generator (JG) disabled.

Figure 7.22 presents the results of Goblin against TinyGarble with JG disabled. Even without JG, Goblin successfully reveals the garbler's input with an average SR exceeding 95%, which is only slightly lower than when JG is enabled.

To further investigate this, we executed Goblin against MULT, SUM, and Hamming benchmarks, varying input sizes from 128 to 1024 bits while keeping JG disabled. The results, illustrated in Figure 7.23, show that SR



(a) (b) (c)

Figure 7.24: SR of Goblin against 128-bit (a) SUM, (b) Hamming, and (c) MULT. CPU cycle traces captured from 10-100, 000 randomly chosen inputs when JG is disabled. (Top: TinyGarble [361] with only free-XOR, Bottom: with half-gate optimization).

increases as the benchmark input size grows.

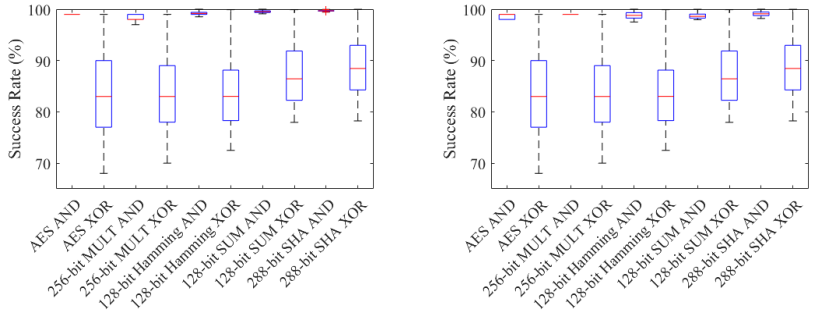
Additionally, Goblin was tested against MULT, SUM, and Hamming modules without JG to assess its effectiveness in the absence of artificial cache eviction. Figure 7.24 shows the SR results for 128-bit (a) SUM, (b) Hamming, and (c) MULT benchmarks using CPU cycle traces collected from 10 to 100, 000 randomly selected inputs. These results demonstrate that Goblin can extract garbler input from an insecurely implemented framework even without leveraging JG.

To examine the influence of gate types in the input layer on the SR, we counted the number of XOR and AND gates in the input layers of AES and 256-bit MULT. These two benchmark functions exhibit significant variation in results, as shown in Figure 7.19. Table 7.11 provides a detailed breakdown of the gate types present in the input layers of AES and 256-bit MULT benchmark functions.

Moreover, the AND gate category includes AND/NAND, OR/NOR, ANDN, ORN, NANDN, and NORN gates, while the XOR gate category consists of INV, XOR, and XNOR gates, as described in Section 7.5.4. It is evident from Table 7.11 that AND gates dominate the AES input layer (75% of input layer gates), whereas the input layer of the 256-bit MULT consists of an

Table 7.11: Type of the gates in the input layer of the AES and 256-bit MULT modules.

	AES		256-bit MULT	
	Percentage (%)	Count	Percentage (%)	Count
AND gates in input layer	75	96	50	256
XOR gates in input layer	25	32	50	256



(a) (b)

Figure 7.25: SR of Goblin computed separately for AND and XOR input gates of 128-AES, 256-bit MULT, 128-bit Hamming, 128-bit SUM, and 288-bit SHA modules with (a) free-XOR and (b) half-gate optimization.

equal proportion of XOR and AND gates. This discrepancy explains why the results for these two benchmark functions differ. The key reason lies in the fact that determining the inputs given to XOR gates is inherently more challenging compared to AND gates.

To further analyze this effect, we separately calculated the SR of Goblin when applied to AND and XOR gates. Figure 7.25 presents the results for launching Goblin against 128-AES, 256-bit MULT, 128-bit Hamming, 128-bit SUM, and 288-bit SHA modules, similar to Figure 7.19, where the results for AND and XOR gates were aggregated.

As observed in Figure 7.25, Goblin consistently achieves an average SR close to 100% when targeting AND gates. However, when attacking XOR gates, the SR varies within a range of 65% to 100% across different benchmark functions. This aligns with the results presented in Figure 7.19, where the difference between the mean values of CPU cycles collected for input "0" and input "1" is significantly larger for AND gates than for XOR gates. The

close similarity in the mean values of CPU cycles for XOR gates indicates that Goblin has a lower SR in distinguishing XOR gate inputs.

These findings reinforce the observation that Goblin achieves a higher SR when attacking modules with a larger proportion of AND gates in their input layers.

7.6 Discussion

The field of secure computation is rapidly evolving, driven by the need to balance efficiency, security, and scalability in privacy-preserving applications. Advances in GC, MPC, and hardware-based secure inference have introduced promising optimizations, yet they also open new attack surfaces. The works examined in this chapter highlight key challenges and opportunities in the design of secure hardware accelerators and cryptographic protocols, focusing on mitigating SCA, improving performance, and ensuring robust security guarantees.

GC have long been a cornerstone of SFE, offering a way for multiple parties to compute over encrypted data. However, optimization techniques such as free-XOR and half-gates, while significantly reducing computation and communication overhead, introduce vulnerabilities that can be exploited by SCA. The Goblin attack demonstrates how timing variations in GC frameworks can leak sensitive information about the garbler's input [153]. By analyzing execution time differences using ML-assisted clustering, Goblin successfully recovers private inputs with high accuracy. Similarly, heat-induced SCA have emerged as a significant threat to secure NN accelerators [253]. Unlike traditional power analysis, which requires precise voltage measurements, this attack manipulates FPGA components to generate heat, inducing data-dependent leakage even in first-order masked implementations. Both of these attacks underscore a fundamental challenge in secure computation, optimizations that reduce computational complexity often introduce unintended side effects that adversaries can exploit.

At the hardware level, solutions such as HWGN^2 [156] demonstrate the feasibility of implementing GC directly on FPGA accelerators to mitigate side-channel leakage. By leveraging custom-designed SFE modules, HWGN^2 significantly reduces logical and memory resource requirements while maintaining resistance to both power and EM SCA. This approach aligns well with chiplet-based secure computation frameworks such as Garblet [158], which

seek to minimize communication overhead in MPC by distributing garbling tasks across multiple chiplets. These solutions highlight the increasing role of hardware in secure computation, emphasizing the need for tightly integrated cryptographic protocols and secure microarchitectures.

While each of these works presents innovative approaches to improving security and efficiency, they also reveal new trade-offs and open challenges. For instance, while chiplet-based MPC drastically reduces communication costs, the presence of untrusted chiplets or interposers raises concerns about data integrity and leakage. One potential countermeasure is the integration of TEE within secure chiplets to enforce stricter access control policies [224]. Similarly, post-quantum cryptographic techniques, such as lattice-based MPC [296], could provide stronger security guarantees against future adversaries. In the case of timing attacks against GC, countermeasures include adopting constant-time execution strategies [207] and introducing artificial noise in the garbling process [95]. However, these solutions often come at the cost of increased computation, making trade-off analysis crucial for real-world implementations.

The threat posed by heat-induced leakage is particularly concerning because it leverages inherent physical properties of hardware rather than software vulnerabilities. One possible mitigation is the use of dynamic thermal management (DTM) techniques [151] to regulate temperature by dynamically redistributing workloads. Additionally, security-aware memory allocation techniques [114] can optimize BRAM usage to prevent excessive heat buildup. However, such approaches may introduce performance bottlenecks, requiring further research into hybrid solutions that combine architectural and software-level defenses.

A key insight from these works is that secure computation techniques must be evaluated not only for their cryptographic soundness but also for their susceptibility to SCA. The trade-off between security and efficiency is evident in all the examined works, whether in GC optimizations, chiplet-based MPC, or secure hardware accelerators. A recurring theme is the need for hybrid approaches that integrate multiple security mechanisms to achieve robust protection. For example, while HWGN^2 provides resilience against power and EM attacks, it could be further strengthened by incorporating oblivious RAM (ORAM) techniques to prevent access pattern leakage. Likewise, HE-based secure inference, though computationally expensive, presents an alternative path that eliminates the need for GC altogether [194]. Future research should explore hybrid architectures that merge GC, HE, and ORAM

techniques to achieve optimal security-performance trade-offs.

Beyond technical countermeasures, secure computation must also consider broader system-level implications. As hardware becomes increasingly integral to cryptographic protocols, ensuring resistance to physical attacks such as fault injection and DPA is paramount. Developing formal security verification methodologies for hardware implementations will be critical in preventing unexpected vulnerabilities. Moreover, the role of compilers and secure microarchitectures in mitigating timing and power-based SCA requires deeper investigation.

In conclusion, the works discussed in this chapter collectively illustrate the rapid advancements and persistent challenges in secure computation. While significant strides have been made in optimizing MPC, protecting against SCA, and leveraging hardware accelerators for secure inference, these solutions often introduce new trade-offs that must be carefully analyzed. The future of secure computation lies in designing resilient architectures that seamlessly integrate cryptographic techniques with secure hardware implementations. By bridging the gap between theoretical security and practical deployment, we can pave the way for more robust privacy-preserving technologies in real-world applications.

The field of secure computation is no longer just about cryptography, it is about designing secure systems from the ground up. Moving forward, cross-disciplinary collaboration between cryptographers, hardware designers, and system architects will be essential in achieving truly secure and efficient computation.

Chapter 8

Fault Injection Attacks Against Hardware Implementations

8.1 Motivation

ML, particularly deep NN, has become integral to various domains, including image recognition, fraud detection, natural language processing, and even drug discovery [432, 248]. As with other ML applications, NN-based systems typically operate in two main phases: training and inference. During the training phase, a substantial amount of data is utilized to optimize the model parameters, whereas, in the inference phase, the trained NN is applied to new inputs to generate predictions. This work focuses on inference tasks performed at the edge, where current solutions either require clients to send potentially sensitive data to cloud servers or mandate that model owners store their proprietary NN models on clients' devices.

The latter approach is gaining prominence due to the increasing demand for edge computing, which mitigates network congestion by enabling local computation near users, thereby reducing the communication overhead associated with accessing HPC resources [161]. However, this paradigm shift introduces new security risks, including the possibility of IP infringement and the exposure of model parameters, which could compromise the business interests of NN owners or reveal sensitive information about the training data [248]. In light of these risks, physical attacks have been launched against edge devices that host NN models [53, 29, 307, 170, 54, 377, 254]. Secure inference has emerged as a countermeasure, enabling the client and

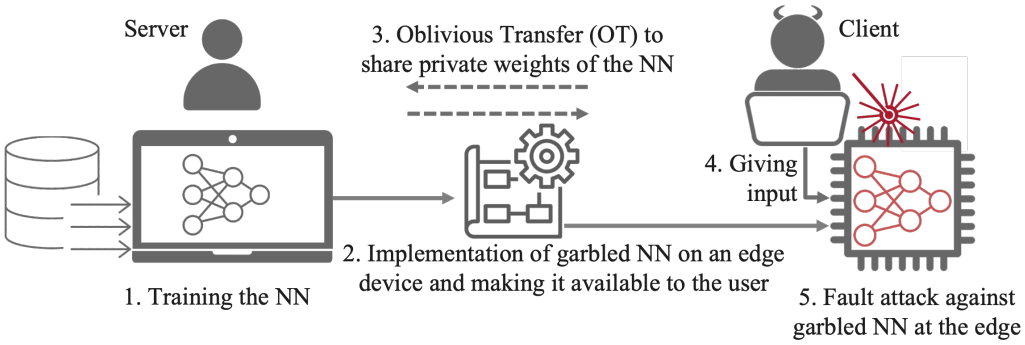


Figure 8.1: Overview of our attack scenario. The client has physical access to the device at the edge running the garbled NN to perform inference. The server represents the NN owner whose private inputs are NN weights.

the NN owner to interact in a way that allows the client to obtain the prediction result while ensuring that neither party gains unauthorized access to the other's private data (input privacy) [222].

To achieve secure inference, extensive research has been dedicated to secure MPC, particularly secure two-party computation [222, 317, 173, 327, 328, 24, 75, 156, 333, 72]. Due to their competitive performance in terms of online latency and accuracy, MPC-based NN inference protocols continue to gain traction. While these protocols provide strong security guarantees (as illustrated in Figure 8.1), implementation vulnerabilities may still be exploited. This raises a critical question: Are implementation-level attacks beyond the adversary models typically considered in MPC protocols?

One might argue that such attacks are already addressed within existing adversary models in secure two-party computation. In the context of passive adversaries, the notion of side-channel (or side-information) leakage has been acknowledged in prior work [35]. It has been recognized that privacy is rarely absolute [35], meaning that some information may inadvertently leak during protocol execution. This includes circuit size, structure, and even certain computational details [35, 133, 230]. Although the inputs of participating parties should remain undisclosed, recent work has challenged this assumption [223, 153]. Specifically, passive SCA in these studies have demonstrated that private data can be extracted by analyzing power consumption or execution time variations [223, 153].

When considering active implementation attacks, i.e., FIA, the traditional MPC literature has primarily interpreted faults as corruption affecting either

the circuit or the inputs [32, 38, 31, 21, 227, 108]. Unlike accidental hardware failures, Byzantine faults arise from adversarial actions that aim to compromise the integrity or confidentiality of the computation [227]. Such attacks can be used to either extract secret inputs or manipulate the computation to produce incorrect results, thereby violating privacy and correctness guarantees.

To counteract faults, several cryptographic safeguards have been proposed, including notions of robustness and fair execution [32, 134]. Fairness ensures that either all parties receive the correct output or none do [215, 279], while robustness (or guaranteed output delivery) prevents adversaries from disrupting protocol execution [227]. Despite these measures, the question remains: Is there a fault injection attack that can compromise input privacy without violating fairness or correctness guarantees—one that remains effective even in robust, fair MPC protocols?

8.2 FaultyGarble: Fault Attack on Secure MPC NN Inference

8.2.1 Fault Injection Attacks: Techniques and Impact

Fault injection (FI) attacks have emerged as a powerful class of active implementation attacks that can compromise cryptographic protocols, secure computation frameworks, and hardware security primitives. Unlike passive attacks, which rely on observing side-channel leakages such as timing, power, and EM emissions, FI attacks actively perturb the normal execution of a device by introducing transient or permanent faults. These faults, if carefully crafted, can lead to exploitable behavior, such as incorrect computations, bypassing security mechanisms, or even recovering secret information.

FI attacks have been studied extensively in cryptographic implementations, including block ciphers [28], public-key cryptosystems [49], and protocols used in secure MPC [227]. These attacks can be broadly categorized into transient faults, which are temporary and do not cause permanent damage to the hardware, and permanent faults, which alter the system's functionality indefinitely. Depending on the precision and capabilities of an attacker, fault injection methods include clock glitches [355], voltage manipulation [340], EMFI [339], and LFI [378]. Each of these methods has different implications for secure computation frameworks, which often rely on specific hardware

and cryptographic protections to maintain confidentiality and integrity.

Secure computation protocols such as GC [419] and HE [119] are susceptible to FI attacks when deployed in real-world hardware. These attacks can violate core security properties such as correctness and fairness by inducing errors in the evaluation of encrypted circuits. In particular, FI attacks against secure inference in DL applications, where privacy-preserving models are deployed on edge devices, present an alarming security risk [53]. The following subsections provide a deeper look into FI attacks against secure computation and possible countermeasures.

8.2.2 Fault Injection and Active Attacks Against Secure Computation

Active fault attacks have posed serious threats to secure computation protocols, particularly when adversaries can target the execution of cryptographic primitives within TEE, hardware accelerators, or MPC frameworks. These attacks are particularly relevant when adversaries have partial access to a device's physical environment or can manipulate its execution remotely.

Techniques of Fault Injection in Secure Computation

Various fault injection techniques have been developed to exploit vulnerabilities in secure execution environments:

- *Clock and Voltage Glitches*: These involve perturbing the device's clock or voltage supply to cause incorrect instruction execution [390]. Cryptographic algorithms often rely on precise execution of modular arithmetic, and even a single-bit fault can leak information about secret keys [49].

- *Laser and Electromagnetic Fault Injection*: These are precise FI methods used to target specific regions of a chip, altering its computation at the transistor level [339, 378]. Laser FI has been extensively used against smart cards and TEEs.

- *Rowhammer and Microarchitectural Faults*: These exploit hardware vulnerabilities in memory to induce bit flips that can compromise secure execution [205]. Such attacks have been demonstrated against cryptographic keys stored in memory [313].

- *Software-based Fault Injection*: This class of attacks does not require physical access but manipulates execution through privilege escalation or manipulating exception handling [234]. Attacks such as Spectre and Meltdown exploit speculative execution to read protected memory, bypassing secure execution environments [207].

Impact on Secure Computation

FIA compromise secure computation in several ways:

- *Confidentiality Violations*: In MPC, GC rely on encryption to ensure input privacy [35]. Faults injected into key-generation or OT can reveal cryptographic keys, allowing attackers to recover private inputs.
- *Incorrect Computation and Fairness Violations*: In secure inference, FI attacks can lead to incorrect predictions, breaking correctness guarantees [222].
- *Function Privacy Violations*: By targeting Boolean circuits used in secure computation, attackers can infer the structure and topology of the underlying function being evaluated [133].
- *Denial-of-Service (DoS) Attacks*: An adversary can use FI to cause protocol failures, effectively halting computations in MPC-based systems.

These threats highlight the necessity of robust protection mechanisms to safeguard secure computation protocols from FI attacks.

8.2.3 Protection Against Fault Injection Attacks

Developing countermeasures against FIA requires a combination of hardware-based and software-based approaches. Depending on the attacker model and available security assumptions, different protection mechanisms can be deployed.

Hardware-Based Countermeasures

- *Redundant Execution*: Many secure processors implement instruction duplication or error detection codes (EDC) to verify computations. This approach is widely used in TEEs such as Intel SGX [81].

- *Glitch-Resistant Circuit Design*: Secure cryptographic hardware often integrates fault detection logic to prevent clock or voltage glitches from affecting execution [390].
- *Physical Shields and Sensors*: High-security environments, such as those used in military applications, integrate tamper detection circuits that erase sensitive data upon physical manipulation [339].

Software-Based Countermeasures

- *Algorithmic Error Detection*: Cryptographic implementations can include integrity checks, such as randomized encoding to detect incorrect computations [126].
- *Constant-Time Execution*: Preventing software-based FI requires eliminating control-flow dependencies that attackers could exploit to manipulate execution timing [207].
- *Fault-Resilient Secure Computation Protocols*: MPC frameworks can integrate ZKPs to ensure that results are computed correctly even in adversarial conditions [134].

Although countermeasures exist, defending against high-precision FI attacks, such as LFI, remains an open challenge. Future research must focus on combining formal verification with hardware security primitives to mitigate active threats while maintaining performance.

The discussion so far has highlighted the dangers of FI attacks against secure computation, particularly within secure inference frameworks. While MPC-based inference has been extensively studied for passive side-channel leakage, active fault injection remains an underexplored yet critical threat vector.

FaultyGarble presents the first known LFI attack against garbled circuit implementations of secure NN inference. Unlike traditional FI attacks that target cryptographic implementations, FaultyGarble exploits transient errors in garbled circuit evaluation, allowing an adversary to extract model parameters while maintaining functional correctness. This presents a significant shift in the understanding of MPC security, as FaultyGarble demonstrates that even maliciously secure MPC implementations are not inherently protected against FI attacks.

The next section provides an in-depth exploration of FaultyGarble, detailing its attack methodology, experimental results, and implications for the future of secure inference.

8.2.4 Adversary Model

Adversary Model in MPC. Secure MPC protocols consider different adversary models, with some frameworks assuming that both the client and the NN owner adhere to the protocol, following an HbC model. In contrast, a *malicious* adversary may deviate arbitrarily from the prescribed protocol [226]. Such deviations can include manipulating circuit execution or introducing corrupted inputs to extract sensitive information about the other party's private data [227]. A range of fault-based corruptions—whether applied to input values or the circuit structure—has been explored in MPC literature [32, 38, 31, 21, 227, 108]. If an NN owner is found engaging in malicious activity, the consequences can be severe due to public accountability [222, 226]. Conversely, clients, who maintain control over their execution environment, may exploit this control to act maliciously and extract NN model parameters in an attempt to replicate the proprietary model.

The malicious behavior of both parties has been analyzed in prior work [63, 233], particularly in the context of enforcing *input consistency* for the client. A major focus of malicious adversarial models has been the ability to construct an incorrect or manipulated garbled circuit, which is typically an attack strategy associated with the party responsible for circuit generation—i.e., the NN owner in our case. While existing literature has explored these threats, it has largely overlooked alternative attack vectors that could be exploited by a malicious client.

Our Adversary Model. In our threat model, we assume a client-server architecture, where a malicious client aims to extract the proprietary weights of the NN model stored on the server cf. [389, 317, 222]. Consistent with previous studies on secure MPC-based NN inference, we assume that both the client and server are aware of the NN configuration. Furthermore, the client either possesses prior knowledge of the processor architecture or is capable of profiling the system to identify points of interest. This assumption is reasonable, as the processor hardware itself is not protected by MPC, even though the NN model parameters remain confidential to the server.

Additionally, we consider an adversary capable of performing physical FIA, such as LFI, against the edge device executing the garbled NN. The

client, holding knowledge of its own input and access to the NN's output, exploits intrinsic characteristics of maliciously secure inference protocols. Specifically, in many implementations, inference is performed in a layer-wise manner on a general-purpose processor, as observed in frameworks such as [222, 317, 362, 363, 364, 212, 262, 213]. By injecting faults into key instructions, the attacker alters their behavior and incrementally reconstructs the NN model's weights using a divide-and-conquer approach.

Our attack specifically targets NNs composed of alternating linear (fully connected, convolutional, etc.) and non-linear ReLU layers, common building blocks of NN. Existing MPC-related countermeasures fail to prevent this attack, as they primarily focus on securing computations against incorrect circuits rather than protecting against malicious behavior introduced by a client who follows the protocol but exploits physical fault injection techniques.

8.2.5 Methodology

Model Extraction Attacks

One of the first known approaches to extracting the weights of NN models was introduced by Carlini et al. [62], targeting unprotected NNs that are not integrated into any secure MPC protocol. Their method strategically selects input values with minor variations and examines how these small changes influence the outputs. By identifying key transition points where the behavior of the ReLU activation function shifts, the attack gradually uncovers information about the model's parameters. Repeating this process across different layers enables the reconstruction of the entire NN with high accuracy and significantly fewer queries compared to conventional extraction methods.

Building on this idea, Lehmkuhl et al. [222] extended the attack to secure inference protocols that employ additive SS under the HbC adversary model. Like the work of Carlini et al., this attack exploits the sequential layer-by-layer evaluation of NNs, a common characteristic of secure inference protocols. The attack begins at the last layer of the NN and proceeds backward toward the first, where at each stage, the adversary slightly alters their input shares provided to intermediate layers. This process, known as **malleation**, involves introducing small controlled perturbations to the intermediate layer inputs, which, in turn, affect the final output. By analyz-

ing these changes, the adversary gradually deduces the actual model weights, as the final output is revealed in plaintext at the conclusion of the protocol.

While both attacks focus on extracting NN weights efficiently, they exploit different vulnerabilities. The attack in [62] primarily leverages the linear region of the ReLU function, whereas [222] takes advantage of weaknesses in the protocol's implementation to manipulate ReLU into behaving linearly. Despite targeting different aspects of secure and non-secure NN inference, both techniques underscore the inherent risks posed by structured model extraction attacks.

Fault Injection for Model Extraction

Our proposed attack leverages fault injection as a cryptanalytic tool to compromise garbled NN protocols. Specifically, we target NN models incorporating the ReLU activation function, where inference is executed in a layer-by-layer manner. The focus is on fully connected NN architectures, where for consistency and comparison purposes, the linear layers do not include additive bias terms, similar to the assumption in [222].

A key distinction between our approach and the attack in [222] lies in the nature of the intermediate values processed during inference. In unprotected interactive NN protocols, intermediate computations involve floating-point representations, whereas in garbled NN inference engines, all values are strictly binary, i.e., either "0" or "1," due to the inherent constraints of GC [35]. This binary representation significantly simplifies the attack, as explained in the following section.

Extracting the Last Layer's Weights

Figure 8.2 illustrates the interaction between the NN owner and the malicious client during the evaluation of the k^{th} layer in a garbled NN. The attack begins by targeting the last layer's weights, i.e., when $k = \ell$. To initiate the attack, the client deliberately sets its input to an all-zero vector, $x = \{0\}^n$.

Standard Evaluation Process. At this stage, the client receives an array of encrypted input labels

$$X = (X_1^{0,1}, X_2^{0,1}, \ldots, X_n^{0,1}),$$

which correspond to $x = \{0\}^n$ (see Figure 8.2). The client then proceeds with the evaluation process as prescribed by the protocol, computing each

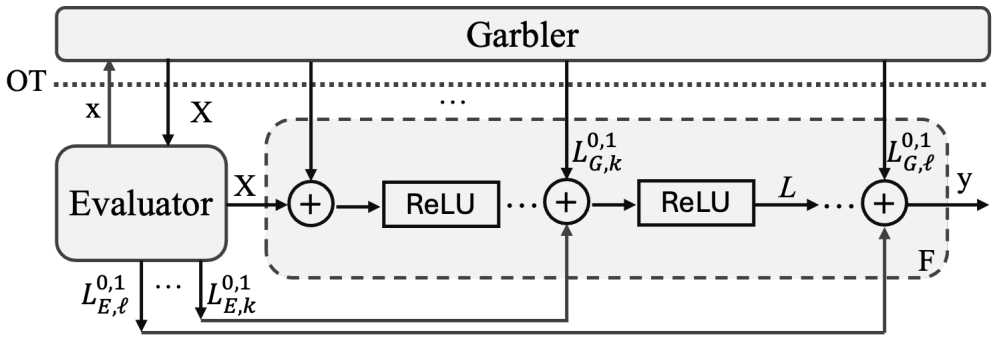


Figure 8.2: A high-level flow of an iterative GC-based NN inference. $L_{G,k}^{0,1}$ and $L_{E,k}^{0,1}$: garbler's and client's labels for k^{th} layer ($1 \leq k \leq \ell$). x and X: client's raw and garbled inputs received via OT; y: client's raw outputs; L: the intermediate layer garbled output.

layer iteratively until reaching the final layer:

$$M_\ell(\texttt{ReLU}(M_{\ell-1}(\cdots \texttt{ReLU}(M_1(x))))),$$

where $M_k := (\texttt{AND}, \texttt{XOR})$ represents the operations within the linear layer, consisting of an `AND` followed by an `ADD` operation. The last layer, $M_\ell \in \mathbb{R}^{m \times t}$, maps t inputs to m output classes. Figure 8.2 illustrates the execution of the last linear layer for a single neuron.

To compute the output of the j^{th} neuron in the final layer, the garbler's label $L_{G,\ell}^{0,1}$, which represents the layer's weights, is multiplied by the client's label $L_{E,\ell}^{0,1}$, the output of the previous layer:

$$Y_j = L_{G,\ell}^{0,1} L_{E,\ell}^{0,1}, \quad 1 \leq j \leq m,$$

where m is the number of neurons in the last layer. At the end of this process, the client holds the garbled output vector $Y = [Y_1, Y_2, \ldots, Y_m]$. Using the decryption label d received from the garbler, the client decrypts Y to obtain the raw output values:

$$(y_1, y_2, \ldots, y_m) = De(d, Y).$$

Weight Recovery via Fault Injection. Under normal circumstances, the decrypted outputs y_j ($1 \leq j \leq m$) do not reveal information about the last layer's weights, w_ℓ, since the NN inputs and biases are set to zero. However,

by injecting a fault that flips the AND operation in the final layer to an XOR, the decrypted output simplifies to:

$$y_j = w_\ell \oplus 0 = w_\ell.$$

Since the client knows that its input was set to zero ($x = \{0\}^n$), observing the decrypted output directly reveals the last layer's weights. This fault-injection technique enables the malicious client to extract critical model parameters, bypassing the security guarantees of the garbled NN protocol.

Extracting Intermediate Layers' Weights

After successfully recovering the weights of the last layer, such as w_ℓ for a given neuron, the next step is to extract the weights of the intermediate layers. The extraction process follows a similar approach but introduces an additional step: forcing all ReLU activation functions (AFs) after the targeted intermediate layer to behave linearly, effectively turning them into buffers. Once the last layer weights, w_ℓ, have been obtained, the attacker proceeds to extract the weights layer by layer from $M_{\ell-1}$ down to M_1.

Fault Injection for Weight Extraction. As in the last layer attack, the client sets its input to $x = \{0\}^n$ and evaluates the NN model honestly. If the target is the k^{th} layer, the computation follows:

$$M_k(\texttt{ReLU}(M_{k-1}(\cdots \texttt{ReLU}(M_1(x))))),$$

where $1 \leq k \leq \ell - 1$. Each layer consists of a linear transformation, M_k, followed by a non-linear activation function, $\texttt{ReLU} = (1 - \texttt{MSB}(x)) \cdot x$, as illustrated in Figure 8.2.

To retrieve the weights of layer M_k, the attacker injects a fault into the AND operation within M_k, replacing it with an XOR, similar to the attack on the last layer. The resulting faulty transformation is denoted by M'_k. The intermediate layer output L is computed as follows:

$$L = \texttt{ReLU}(M'_k(L^{0,1}_{G,k}, L^{0,1}_{E,k})) = \texttt{ReLU}(L^{0,1}_{G,k} \oplus L^{0,1}_{E,k}).$$

Expanding the ReLU function's bitwise operation results in:

$$L = (1 - \texttt{MSB}(L^{0,1}_{G,k} \oplus L^{0,1}_{E,k})) \cdot (L^{0,1}_{G,k} \oplus L^{0,1}_{E,k}).$$

Since the ReLU function outputs zero for negative values, the attacker injects an additional fault into ReLU to force it to behave linearly, effectively converting it into a buffer. Instead of applying its standard transformation:

$$(1 - \texttt{MSB}(x)) \cdot x,$$

the faulty ReLU operates as:

$$(1 \,\texttt{OR}\, \texttt{MSB}(x)) \cdot x.$$

This transformation simplifies to:

$$(1 \cdot x) = x,$$

ensuring that the ReLU function acts as a buffer rather than a non-linear operation.

Propagation of Extracted Weights. Once the ReLU activation function has been bypassed, the intermediate layer output simplifies to:

$$L = (1 \,\texttt{OR}\, \texttt{MSB}(L_{G,k}^{0,1} \oplus L_{E,k}^{0,1})) \cdot (L_{G,k}^{0,1} \oplus L_{E,k}^{0,1}),$$

which reduces to:

$$L = L_{G,k}^{0,1} \oplus L_{E,k}^{0,1}.$$

This output is then propagated through the remaining layers, where the faulty ReLU ensures that no additional non-linearity is introduced:

$$Y_j = M_\ell(M_{\ell-1}(\cdots M_{k+1}(L_{G,k}^{0,1} \oplus L_{E,k}^{0,1}))).$$

Since the weights of the layers $M_\ell, \ldots, M_{k+1}$ were already extracted in previous steps, their values are known. Thus, after decrypting the final output, the extracted NN model weights can be reconstructed as:

$$y_j = w_\ell w_{\ell-1} \cdots w_{k+1} \cdot (w_k \oplus 0) = \underbrace{w_\ell w_{\ell-1} \cdots w_{k+1}}_{\text{Known values}} w_k.$$

By iteratively repeating this process, the attacker can systematically extract the weights of all intermediate layers.

8.2.6 Fault Injection in Garbled NN Inference Engines

The implementation of GC-based inference engines on general-purpose processors, such as MIPS [198] or ARM [18], provides notable advantages in terms of practicality and efficiency. Leveraging these well-established architectures enhances secure computation by ensuring compatibility with existing hardware ecosystems while benefiting from optimized software co-design approaches [160]. Moreover, the availability of advanced development tools and compiler optimizations further simplifies integration and accelerates deployment.

To clearly illustrate our fault injection attack, we focus on one of the most widely used general-purpose GC implementations based on the MIPS I architecture [362, 364]. However, it is important to emphasize that our attack is not restricted to MIPS-based implementations; it can also be applied to other architectures with comparable instruction sets. We begin by identifying possible fault injection points that can be exploited to achieve the desired attack functionality. Following this, we select a specific fault injection target as a practical demonstration of our attack methodology.

Program Counter (PC). During instruction execution, the processor begins with the Instruction Fetch (IF) stage, where the PC holds the address of the current instruction and supplies it to the instruction memory. The instruction located at this address is then retrieved, and the PC is incremented by 4 to point to the next instruction in sequence. A fault injection attack at this stage could manipulate the PC value, forcing the processor to fetch an unintended instruction, such as executing an `XOR` operation instead of an `AND` operation during the linear layer computation.

Decoded Instruction. In the second stage, Instruction Decode (ID), the fetched instruction is decoded to determine the operation it specifies. One of the critical fields in this stage is the `func` register, which determines the exact operation to be performed. An attacker can inject a fault at this stage by modifying the `func` register, altering the intended instruction and compelling the processor to execute a different operation.

Register Read (RR). In the third stage, the processor reads the values stored in the source registers (rs and rt) from the register file, which contains 32 general-purpose registers. In conventional, unprotected NN evaluation protocols, an attacker can manipulate the register values to arbitrary numbers. However, in the context of GC, the intrinsic encryption of data prevents the attacker from setting registers to specific known values, limiting direct

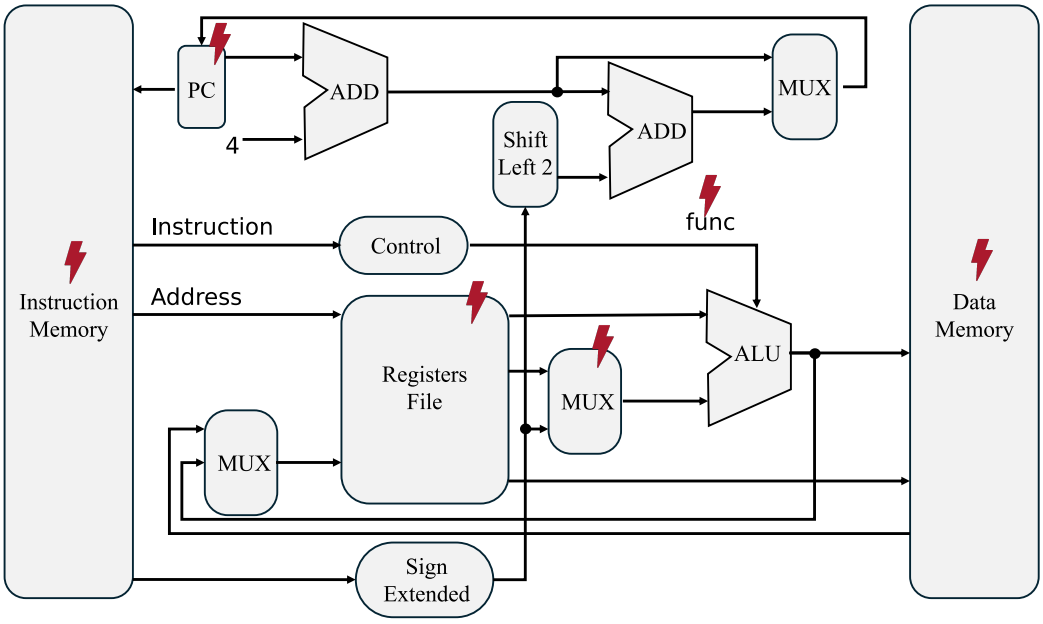


Figure 8.3: A high-level representation of a general-purpose processor architecture, adapted from [362], illustrating possible fault injection points. Here, `func` refers to the function code (6 bits) used in an R-Type register operation.

manipulation [35].

Arithmetic Logic Unit (ALU). In the Execution (EX) stage, the ALU performs the operation specified by the decoded instruction. The ALU Control unit determines which operation should be performed based on the `func` field and opcode. For example, if the instruction specifies an `ADD` operation, the ALU adds the source register values and stores the result. Fault injection at this stage could alter ALU logic or the control flow, allowing an attacker to modify computations, for instance, forcing a ReLU activation function to behave linearly as a simple buffer.

Destination Register. In the Memory Access (MEM) stage, memory is accessed only if required; for R-type instructions, this step is typically bypassed. Finally, in the Write Back (WB) stage, the result from the ALU operation is written into the destination register. Similar to the register read stage, in unprotected NN evaluation, an attacker could tamper with the destination register values. However, in a GC-based protocol, the garbled data format prevents setting specific known values, restricting direct exploitation.

Table 8.1: ALU function register value in MIPS I architecture

Function	Binary Code	Function	Binary Code
NOTHING	6'b000000	OR	6'b000101
ADD	6'b000001	AND	6'b000110
SUBTRACT	6'b000010	XOR	6'b000111
LESS_THAN	6'b000011	NOR	6'b001000
LESS_SIGNED	6'b000100		

Among the potential fault injection points, we focus on decoded instruction faults as a demonstrative example. However, this does not exclude the possibility that an attacker could exploit other fault injection locations to achieve a similar objective.

8.2.7 Fault Injection in the Decoded Instruction of the NN Model

Our attack consists of two key steps: (1) modifying the operation of the linear layer by changing `AND` to `XOR` to extract the last layer weights and propagate intermediate layer weights forward, and (2) forcing the ReLU activation function to behave linearly, effectively converting it into a buffer instead of performing its usual non-linear operation. This second step ensures that ReLU does not alter negative intermediate weight values to zero, which is essential for extracting weights from intermediate layers. Figure 8.4 provides a high-level overview of the control signals, ALU operations, and the specific fault injection point targeted in our attack.

To carry out the first phase of our attack—altering the ALU function from `AND` to `XOR`—we inject a fault into the `func` register during the execution of the final linear layer, as illustrated in Figure 8.4. This fault alters the ALU's functionality, ensuring that the intended weight values propagate to the last layer. Table 8.1 lists the ALU functions in the MIPS I architecture, providing insight into the available operations and their respective binary representations.

Extracting the last layer's weights As shown in Table 8.1, the register value for the `AND` operation, $6'b000110$, differs by only one bit from that of `XOR`, $6'b000111$ (specifically, the least significant bit (LSB)). To carry out the attack, the client monitors the instruction decode path and identifies the

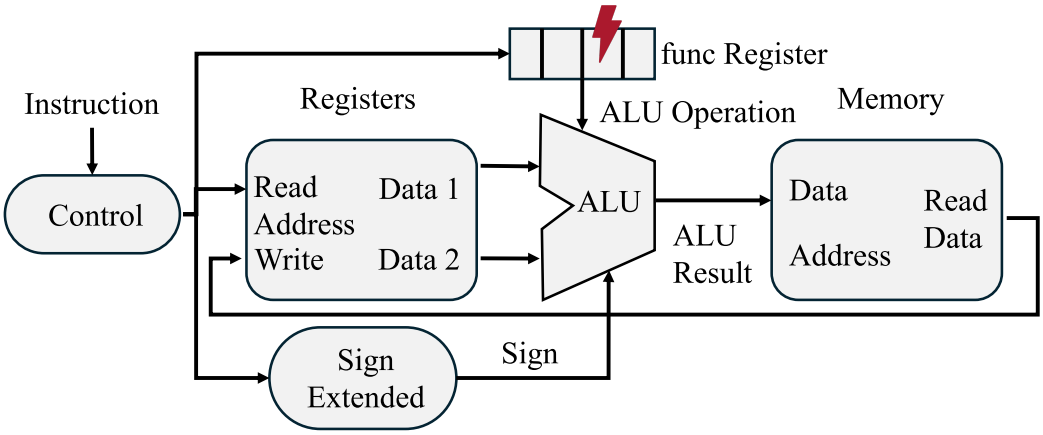


Figure 8.4: A high-level illustration of the control signals, ALU procedure, and the location of our fault attack.

location of the `func` register on the die. At the onset of the execution time window corresponding to the last layer, the client targets the first bit of the `func` register. By injecting a fault at this location, the operation is altered from `AND` to `XOR`, ensuring that the last layer weights propagate to the output.

Extracting intermediate layers' weights To recover the weights of intermediate layers, the attack proceeds in two steps: first, modifying the operation of the target intermediate layer from `AND` to `XOR`—similar to the approach used for the last layer—and second, forcing all subsequent ReLU functions to behave as linear buffers. ReLU in the GC framework is implemented as $(1 - \texttt{MSB}(x))\texttt{AND}\, x$. This implementation translates into two MIPS I instructions: `SUB $result, $Constant_1, $MSB;` followed by `AND $ReLUOutput, $result, $x;` The goal of the attacker is to modify the first operation so that instead of computing $1 - \texttt{MSB}(x)$, it computes $1\texttt{OR}\texttt{MSB}(x)$. This change transforms the ReLU function from $(1-\texttt{MSB}(x))\ \texttt{AND}\, x$ to $1\ \texttt{AND}\, x$, effectively turning it into a buffer since the `OR` of any value with 1 is always 1, and `AND` with 1 retains the input value.

To achieve this, the client modifies the `func` register value from `SUB` ($6'b000010$) to `OR` ($6'b000101$). This requires flipping the three least significant bits of the `func` register ($010 \rightarrow 101$). After injecting the fault, the processor first computes the `OR` of `Constant_1` and $\texttt{MSB}(x)$, which always results in `Constant_1`. Next, it performs the `AND` operation between `Constant_1` and x, yielding x as the output. By repeating this process for each neuron,

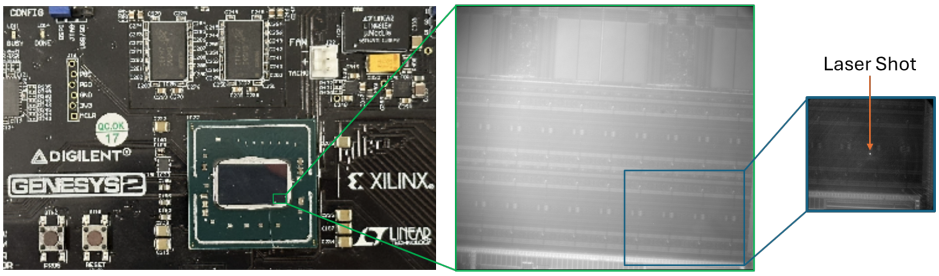


Figure 8.5: Iterative magnification of the device under the AlphaNOV setup (from left to right): the Genesys2 board and the die shown is the Kintex 7 FPGA with the heatsink removed; the middle image depicts the die using the 20X lens to show the corner where the FF for fault is placed; Lastly, the rightmost image is captured using the 50X lens, illustrating the fault injection at the point of interest (the white dot corresponds to the laser shot).

the attacker systematically extracts all intermediate layer weights through multiple faults and queries.

8.2.8 Experimental Setup

For the LFI experiment, we utilized a Genesys 2 development kit featuring an AMD/Xilinx Kintex 7 (XC7K325T-2FFG900C) FPGA, fabricated using 28 nm technology. The FPGA is housed in a flip-chip package. To access the silicon backside, the fan and heat spreader were removed, exposing the die without any further modifications, such as silicon polishing (see Figure 8.5). Throughout all experiments, the FPGA core was supplied with 1.0 V and operated at a clock frequency of 200 MHz.

8.2.9 Laser Fault Injection Setup

The LFI and near-infrared (NIR) microscopy were conducted using the ALPhANOV S-LMS [15] system. This setup integrates an optical microscope with a precision-controlled laser for fault injection. The microscope includes a camera system for imaging and a lens mounted on an XYZ motorized stage, allowing precise focus on targeted regions. Two magnification lenses were employed during the experiments: a 20X lens with standard resolution (NA=0.6) and a field of view of 480 $\times$ 380 μm, and a 50X ultra-high resolution lens (NA=0.7) with a field of view of 190 $\times$ 150 μm. The system

is controlled via dedicated software, which facilitates navigation using an IR view of the die and allows for precise control of the XYZ stage and camera settings.

To capture real-time images of laser injections, the image integration time was set to 0.1 ms, with a frame rate of 60 Hz, matching the display refresh rate. The laser source employed in the experiment was a High Pulse Performance PDM laser with a wavelength of 1064 nm. For fault injection, the laser operated with a peak current of 1 A, a pulse width of 250 ns, and a repetition frequency of 100 kHz. The laser was controlled via the ALPhANOV software, which provided real-time monitoring of laser shots within the targeted fault regions."

8.2.10 Results

8.2.11 Complexity of the Attack: Number of Faults and Queries

In this section, we analyze the number of queries and fault injections required to extract the weights of commonly used NN models, as examined in prior works [222, 62]. To extract the weights of the last layer, the client requires only a single fault per weight. Since $L_{G,\ell}^{0,1}$ corresponds to the label of zero, and multiplying zero by any value results in zero, only the faulty neuron's weight—where the `AND` operation is altered to `XOR`—propagates to the output. Thus, the number of faults required to extract the last layer's weights is $\#\text{faults} = p_\ell$, where p_ℓ denotes the number of parameters in the last layer.

Extracting weights from intermediate layers follows a similar approach but requires additional fault injections. First, a fault is injected into the linear computation of the target layer to propagate the target weight to the input of `ReLU`. Then, another fault is injected into the `ReLU` function of the target layer to disable its non-linearity, effectively making it a buffer. Additionally, all neurons in subsequent layers from the target layer up to the last layer, i.e., $M_k, M_{k+1}, \ldots, M_\ell$, require faults in their linear computation to ensure weight propagation. Each of these layers' `ReLU` functions must also be modified to behave as buffers, requiring further fault injections.

The total number of faults required to extract an entire NN model is

Table 8.2: Comparison of query and fault complexity between our attack, [222], and [62]. The number of faults applies only to our attack. Unlike [222], which targets an HbC-secure inference engine, and [62], which attacks unprotected models, our attack is mounted against a maliciously secure NN inference engine.

Network Dimensions	# Parameters	# Queries			# Faults
		[222]	[62]	Ours	
784-128-1	100,480	100,480	$2^{21.5}$	100,480	200,832
784-32-1	25,120	25,120	$2^{19.2}$	25,120	50,208
10-10-10-1	210	210	2^{16}	210	610
10-20-20-1	620	620	$2^{17.1}$	620	1,820

calculated as follows:

$$\#\text{faults} = (\underbrace{\ell(\ell-1)/2}_{\texttt{ReLU}} + \underbrace{\ell(\ell-1)/2}_{\texttt{AND}\rightarrow\texttt{XOR}}) \cdot (p - p_\ell) + p_\ell = O(\ell^2 p),$$

where p is the total number of parameters in the NN model, p_ℓ represents the number of parameters in the last layer, and ℓ is the number of layers in the NN. Furthermore, the number of queries remains equivalent to the number of model parameters, as each query provides an input to the NN. Thus, the query complexity of our attack is comparable to state-of-the-art model extraction attacks, despite targeting a more secure scheme designed to withstand malicious adversaries.

Table 8.2 compares our attack against GC-based NN inference secured against a malicious adversary to [222], which targets an HbC-secure inference engine, and [62], which attacks an unprotected NN model. Notably, while [222] evaluates NN inference with secret-sharing-based linear layers and GC-based ReLU functions, our attack considers a fully GC-based NN model. Unlike [222], which requires modifying the client's share in a secret-sharing-based linear layer, our attack directly injects faults into a GC-based architecture.

Since [222] and [62] do not involve fault injections, the number of faults is only relevant to our attack. Additionally, our attack requires up to 30× fewer queries than [62], while matching the query complexity of [222], despite the fact that our target is a *maliciously secure* NN model, whereas [222] attacks an *HbC-secure* model. This result highlights that a client can successfully extract a model secured against malicious adversaries without incurring any

Figure 8.6: Simulation of the `alu_func` register during the execution of a neuron in the *first* hidden layer (blue: execution window of a neuron, purple: execution of multiplication and summation per connected input, yellow: execution of the ReLU function).

additional query overhead. However, unlike the previous attacks, our method requires fault injections something that the client is assumed to be capable of performing.

8.2.12 Simulation Results

To assess the effectiveness of our attack against garbled NN inference engines, we first simulate the fault injection's impact. For all experiments, we introduced single-bit faults. We implemented a proof-of-concept multi-layer perceptron (MLP), hereafter referred to as the *target model*, featuring two hidden layers with five neurons each, a final layer with ten neurons, and an input layer with five inputs. This configuration aligns with benchmark MLPs from prior studies [156, 317, 267].

To evaluate the target model, we used the GC Lite_MIPS implementation from [361]. This setup allows us to observe the input labels and output results to analyze the effects of fault injection. To compile the MLP into MIPS I instructions, we employed the GNU compiler collection (GCC) [366], following the recommendations of [362]. Fault injection was simulated using SystemVerilog Assertions (SVA) in Vivado Suite 2023 [411]. The simulation was conducted with a clock period of 50 ns, and the `alu_func` register cycled through the ALU functions listed in Table 8.1, ranging from 0 to 8.

Figure 8.6 illustrates the simulation results of the `alu_func` register while executing a neuron in the first hidden layer. The blue rectangle marks the execution window of a single neuron, encompassing both multiplication operations and the ReLU activation. The purple rectangle highlights the execution

Figure 8.7: Simulation of the `alu_func` register during the execution of a neuron in the last layer (blue: execution window of a neuron, purple: execution of multiplication and summation per connected input).

of a single multiplication operation (`AND` $= 6'b000110$) followed by summation (`ADD` $= 6'b000001$). Meanwhile, the yellow rectangle represents the ReLU execution, which consists of a subtraction operation (`SUB` $= 6'b000010$) followed by an addition operation (`ADD` $= 6'b000001$).

As seen in Figure 8.6, the execution of a single neuron in the first hidden layer requires 12 clock cycles: Two clock cycles per input connection (totaling 10 cycles for 5 inputs) and two additional clock cycles for ReLU execution.

Thus, the execution of a single neuron in the hidden layer requires 12×50 ns $= 600$ ns. Since the target model consists of two hidden layers, each containing five neurons, this process repeats 10 times, leading to a total execution time of 6000 ns.

Figure 8.7 presents the simulation of the `alu_func` register while executing a neuron in the last layer of the target model. The blue rectangle represents the execution window of the neuron, while the purple rectangle highlights a single multiplication (`AND` $= 6'b000110$) followed by summation (`ADD` $= 6'b000001$).

According to Figure 8.7, the execution time for each neuron in the last layer is determined as follows: Each neuron is connected to five intermediate outputs. The execution requires two clock cycles per connection. Consequently, each neuron computation spans 10 clock cycles, corresponding to 500 ns.

Given that the last layer contains 10 neurons, the total execution time for this stage amounts to $500 \times 10 = 5000$ ns.

Figure 8.8: Simulation of the `alu_func` register during the computation of a neuron in the last layer after fault injection (blue: execution window of a neuron, purple: altered data register due to fault injection, orange: value of `alu_func` after fault injection).

Fault Injection in the Last Layer

To illustrate the effectiveness of our attack, we focus on the second neuron in the last layer and demonstrate how fault injection modifies the `alu_func` register. The results are shown in Figure 8.8. The orange rectangle highlights the effect of the fault injection on the `alu_func` register. By injecting a fault into the LSB of `alu_func`, its value changes from 6 to 7, effectively altering the executed instruction from `AND` $= 6'b0000110$ to `XOR` $= 6'b0000111$.

The purple rectangle in Figure 8.8 depicts the modified output data following the fault injection. To verify the attack's impact, we retrieve the garbled output label of the target neuron (second neuron in the last layer). We intentionally set the weight of this neuron to 1 to analyze how the core processes the `AND` and `XOR` operations on the label 0 and weight 1. Prior to the attack, as shown in Figure 8.7, the core produces the output "$0x092d4010$," which corresponds to the garbled label of 0, computed as $0\texttt{AND}1 = 0$, based on the output retrieved from [361]. After injecting the fault, as depicted in Figure 8.8, the core generates "$0x0241200c$," which, according to [361], corresponds to the garbled label of 1. This result stems from performing the `XOR` operation on input 0 and weight 1, i.e., $0 \oplus 1 = 1$. This confirms that our attack successfully extracts the weight of the second neuron in the last layer.

To extract the remaining neuron weights in the last layer, the same fault injection process can be applied at the same fault location but at different execution time frames.

Figure 8.9: Simulation of the `alu_func` register during the computation of a neuron in the first *intermediate layer* (blue: execution window of a neuron; purple: modified data register due to fault injection; orange: modified value of `alu_func` after fault injection).

Fault Injection in the Intermediate Layers

The attack on intermediate layers follows the same fault injection technique used in the last layer but requires an additional step: forcing the ReLU functions in all subsequent layers to behave linearly, effectively turning them into buffers. To demonstrate this process, we target the first neuron in the first hidden layer, with the simulation results shown in Figure 8.9.

The orange rectangle in Figure 8.9 highlights the fault injection location within the `alu_func` register during the execution of the ReLU function. Following the fault injection, the instruction $\texttt{SUB} = 6'b000010$ is altered to $\texttt{OR} = 6'b000101$. As a result, the ReLU function, initially defined as $(1 - \texttt{MSB}(x))\,\texttt{AND}\,x$, is modified to $(1\,\texttt{OR}\,\texttt{MSB}(x))\,\texttt{AND}\,x$. Since the OR operation with 1 always results in 1, the expression simplifies to $1\,\texttt{AND}\,x = x$. This modification forces the ReLU function to act as a buffer, allowing the neuron's output to pass through unchanged.

To assess the impact of this alteration, we intentionally set the intermediate layer weights to -1 and all inputs to 1. In this setup, the neuron output should be -5 since it aggregates five inputs. Normally, the ReLU function would take this negative value and output 0. However, after fault injection, we observe a change in the LSB of the ReLU output.

If the ReLU output remains 0, the corresponding LSB retains the label for that value. Conversely, if the ReLU outputs a negative value, the LSB transitions to 1, following the two's complement representation used by the core: $5 = 0b101$ and $-5 = 0b011$.

As shown in Figure 8.9, the data register's LSB changes to "$0x024340c4$," which corresponds to the label of 1 according to the labels generated by the

tool [52]. In contrast, without any fault injection, the LSB of the data register remains at "$0x0affffffa$," which corresponds to the label of 0, as illustrated in Figure 8.6 and confirmed by [361]. This validates that the fault injection successfully modifies the ReLU function into a buffer.

By systematically applying this process—injecting faults into both the linear layer and ReLU functions—an adversary can extract the entire NN model weights.

Laser Fault Injection Results

The setup utilized the same MIPS implementation with the NN program stored in the program memory. The FF responsible for storing the opcode is located in slice X1Y138 of the Kintex 7 FPGA. This slice is positioned near one of the corners of the FPGA, as shown in Figure 8.5.

To localize and target the fault injection site, we initially used a 20X lens to navigate to the corner of the FPGA. A 50X lens was then employed for precise focusing on the target slice before irradiating the laser. The primary goal of this experiment was to induce a fault in the opcode FF for the last layer, flipping a bit to alter the instruction from `AND` to `XOR`. If successful, this modification would reproduce the behavior observed in the simulated fault injection.

To ensure that the induced fault affected only the intended bit without disrupting other circuit components, we implemented flag outputs to monitor different stages of the fault and verify the correctness of the output. A dedicated flag was assigned to track the targeted opcode bit, ensuring that the fault occurred as intended. Additionally, we included an output that compared the ALU's result with the expected faulty value after the fault injection. To confirm that the processor continued functioning correctly, we also monitored the execution of subsequent instructions.

The experiment successfully triggered all three output indicators, demonstrating that the fault injection led to the correct extraction of weights from the last layer. The faults were transient, meaning that the affected register was overwritten with the correct instruction values in the subsequent clock cycle(s) or after resetting the processor. Furthermore, no degradation in the NN's accuracy was observed post-attack, confirming that the NN model remained intact after the fault injection. The attack achieved a success rate of 1, meaning that every fault attempt successfully flipped the targeted bit.

Using the same methodology, LFI can be extended to extract weights

from other layers as well.

8.3 Discussion

Limitations of Cut-and-Choose Against Fault Attacks

Maliciously secure GC frameworks primarily rely on the cut-and-choose mechanism to mitigate the risk of a corrupt garbler generating fraudulent circuits [230, 232]. The core idea behind cut-and-choose is that instead of evaluating a single garbled circuit, the evaluator receives multiple GC and randomly selects a subset to be opened for verification. If the selected circuits pass the integrity check, the remaining circuits are assumed to be correctly constructed, and the protocol proceeds with their evaluation.

Despite the robustness of cut-and-choose in preventing a dishonest garbler from manipulating the computation, it fails to address adversarial behavior from the evaluator. In particular, the cut-and-choose mechanism is designed to ensure correctness but does not inherently prevent FIA that modify the evaluation process. Since the evaluator is responsible for executing the GC, a malicious client can still inject faults at the evaluation stage without violating the principles of maliciously secure GC.

Our attack exploits this shortcoming by selectively injecting faults into the garbled circuit evaluation phase. Even though the cut-and-choose protocol enforces the correctness of the GC before evaluation, it does not prevent an adversary from targeting specific gates or altering the execution of arithmetic operations within the garbled circuit. By introducing faults at carefully chosen locations, the client can extract the model weights while adhering to the framework's defined security guarantees. This highlights a fundamental gap in current maliciously secure GC frameworks—while they are designed to prevent incorrect circuit generation, they do not offer protection against active physical attacks, such as LFI, that occur during execution.

Vulnerabilities in Other General-Purpose Processors

While our attack is demonstrated on a MIPS-based GC inference engine, it is not limited to this architecture. Other widely used general-purpose processors, such as ARM [18], x86 [185], PowerPC [17], SPARC [360], and RISC architectures [292], follow similar instruction execution cycles. The fetch-decode-execute cycle is a common principle across these architectures,

where control signals dictate the execution of arithmetic and logical operations within the processor's ALU.

In our attack, we leveraged the vulnerability in the instruction decoding phase, where control signals are stored in specific registers (e.g., the `func` register in MIPS). This vulnerability is not unique to MIPS; all modern processors contain similar control registers that determine ALU functionality. If an adversary identifies the location of these control registers through side-channel analysis or reverse engineering, they can manipulate instruction execution in a similar manner.

In ARM architectures, the ALU operations are determined by opcode encodings that are stored in the processor pipeline [363]. Fault injection could modify these opcodes to change arithmetic operations in secure computation frameworks. Similarly, in x86 processors, execution relies on microcode interpretation [185], and targeted fault injection could disrupt branch predictions or micro-operation scheduling, leading to erroneous computation results. RISC architectures, which prioritize efficiency through a reduced instruction set, could also be vulnerable if an instruction register is modified to change the control flow of the program [292].

Given the commonalities across these architectures, the attack methodology presented in this paper can be extended to other platforms. Any general-purpose processor used to evaluate GC or execute secure computations is potentially vulnerable to this form of fault injection.

Potential Countermeasures Against Fault Injection Attacks

To mitigate FIA against GC-based inference engines, several countermeasures can be considered. These countermeasures fall into different categories, each with trade-offs in terms of security, computational overhead, and practicality.

One effective countermeasure is the adoption of PFE to protect the execution logic. In the PFE setting, not only is the data protected, but the function itself is also obfuscated [133]. By extending GC protocols to support PFE, the attacker loses visibility into how operations are performed, making fault injection significantly harder to implement. Several implementations of PFE have been proposed, such as GarbledCPU [364], GarbledEDA [157], and HWGN^2 [156]. These frameworks garble both data and execution logic, thereby preventing an attacker from knowing when or where to inject faults. However, PFE significantly increases computational complexity and memory requirements, making it a less practical solution for resource-constrained

environments.

An alternative approach involves leveraging fault-tolerant maliciously secure GC frameworks. Some interactive secure computation frameworks have been designed to be resilient against faults. TinyLEGO [108] and MiniLEGO [109] introduce XOR-homomorphic commitments that ensure the integrity of computed values throughout the garbled circuit. By enforcing consistency across circuit evaluations, these frameworks mitigate FIA by making it difficult for an adversary to selectively manipulate individual gate evaluations. In these frameworks, each gate's output is linked to the integrity of the entire computation path. A single incorrect modification would propagate errors throughout the circuit, making it infeasible to selectively inject faults without triggering detectable inconsistencies. However, the downside of these methods is their significant communication and computation overhead.

Another promising countermeasure is the adoption of hardware-based protections to increase fault resistance. Instruction set hardening can increase the HD between opcode representations, making single-bit fault injections ineffective. For example, ensuring that the opcode for `XOR` differs from `AND` by multiple bits would require an attacker to inject multiple simultaneous faults [199]. Error detection mechanisms such as parity checks and Hamming codes can be implemented to validate instruction execution and flag anomalies [27]. Additionally, introducing randomized execution timing can prevent precise timing-based FIA. By varying instruction execution times unpredictably, an attacker cannot accurately time fault injections, reducing the attack's reliability [27].

Redundant execution is another potential defense mechanism. This method involves executing the same computation multiple times and comparing results. Dual execution, where two identical computations are run and their results are compared at checkpoints, can help detect inconsistencies caused by faults. A related technique is diversity-based execution, which involves using different instruction sequences or randomized memory layouts to prevent predictable attack points [406]. Control flow integrity (CFI) mechanisms can also be employed to enforce strict execution flow policies and detect unauthorized modifications [384].

Future Research Directions

While our study demonstrates the vulnerability of GC-based NN inference engines to fault injection, further research is needed to explore efficient PFE

implementations that minimize computational overhead. Additionally, hybrid secure computation models that combine GC with HE or oblivious RAM (ORAM) could offer stronger resilience against active adversaries. Another promising avenue is the development of hardware-based security mechanisms that integrate fault detection and mitigation directly into processor architectures. These open questions highlight the need for more robust countermeasures to protect secure computation frameworks from evolving adversarial threats.

Chapter 9

Discussion and Future Work

9.0.1 Lessons Learned from Secure Hardware Implementations

Extensive research in secure computation has revealed that cryptographic security alone is insufficient when implemented on physical hardware. While most theoretical frameworks operate under the assumption that adversaries interact with the system exclusively through defined cryptographic interfaces, actual attackers frequently exploit underlying physical properties such as timing discrepancies, power consumption patterns, and fault injection techniques. These observations expose a critical disconnect between theoretical security assurances and practical robustness. Throughout this work on secure MPC and hardware-based protections, several key lessons have emerged that are likely to shape the trajectory of future secure computation systems.

One of the most fundamental insights is the disparity between theoretically proven security and its practical realization. Secure computation protocols like GC and OT offer strong cryptographic guarantees under idealized assumptions. However, when these protocols are instantiated on hardware, new vectors of attack often emerge. The FaultyGarble [155] attack clearly demonstrated that even sophisticated MPC-based systems can be undermined by inducing faults during the computation process [155]. In this case, LFI was successfully employed to extract private model parameters from a secure inference pipeline, illustrating that physical attacks can circumvent the expected security provided by cryptographic mechanisms. This emphasizes the necessity for secure computation frameworks to incorporate both

algorithmic and physical-layer protections.

Another critical takeaway involves the perennial trade-off between efficiency and security. Secure computation frameworks are inherently resource-intensive, frequently demanding substantial computational power and communication bandwidth. While numerous optimizations have been proposed to mitigate these costs, such optimizations can inadvertently introduce vulnerabilities. The Goblin [153] framework exemplified this trade-off by showing that software acceleration significantly boosts inference performance, yet potentially introduces unintended side-channel leakage [153]. Thus, future designs must seek a delicate balance between efficiency and uncompromised security.

SCA remain one of the most persistent and evolving threats to secure hardware implementations. While traditional cryptographic countermeasures like masking and blinding offer partial mitigation, they are not foolproof. The HWGN^2 [156] framework demonstrated that even cryptographically protected inference processes can leak sensitive information through power analysis, EM emissions [156]. These results underline the importance of augmenting cryptographic protections with hardware-level defenses such as noise injection, randomized instruction scheduling, and voltage balancing, to achieve meaningful resistance against SCAs.

In addition to mitigating attacks, addressing performance limitations is crucial for the practical deployment of secure computation. The GuardianMPC [154] study delivered a comprehensive evaluation of the cost-performance trade-offs in existing frameworks, concluding that many current approaches impose excessive overheads that hinder their viability for large-scale applications [154] . The findings underscore that while maintaining strong security remains essential, performance bottlenecks must also be resolved to make secure computation frameworks accessible and scalable.

Thermal side-channel attacks represent a new and emerging threat landscape. The Bake It Till You Make It [254] attack highlighted that heat-induced variations in power consumption can be exploited to retrieve sensitive information, such as cryptographic keys and model parameters [254]. In contrast to conventional power analysis, which relies on high-resolution measurements, thermal SCAs exploit gradual temperature changes over time, rendering them more difficult to detect and counteract. This novel threat emphasizes the need for secure computation frameworks to account for environmental parameters like heat dissipation and to develop temperature-aware countermeasures.

The transition toward chiplet-based architectures introduces additional complexity in secure computation. Classical security models generally presume a monolithic and trusted processor environment. In contrast, modern systems increasingly adopt multi-chiplet designs, where individual components may be only partially trusted or even compromised. The Garblet [158] framework exemplified how secure computation must evolve in this context to maintain data confidentiality and integrity across distributed and potentially untrusted hardware components [158]. In such architectures, adversaries could inject faults, tamper with intermediate results, or intercept inter-chiplet communication, necessitating the development of cryptographic protocols capable of maintaining robustness in heterogeneous and adversarial environments.

Beyond protecting data privacy, secure computation must also address IP protection. The GarbledEDA [157] framework demonstrated that secure multiparty computation techniques can be integrated into EDA flows to safeguard proprietary circuit designs during processes like simulation, verification, and testing [157]. As the semiconductor industry becomes increasingly globalized, protecting design assets at every stage is vital. However, privacy-preserving verification techniques come with their own challenges, particularly in terms of scalability. Continued improvements to MPC efficiency will be necessary to make IP-protecting solutions both secure and commercially viable.

A final overarching lesson is that secure computation is deeply interdisciplinary by nature. Realizing truly secure and efficient implementations requires collaborative innovations across cryptography, hardware security, machine learning, and systems engineering. As adversaries adopt increasingly advanced strategies, defenders must integrate expertise across these domains to develop resilient and adaptive security mechanisms. The collective insights gained from GuardianMPC [154], FaultyGarble [155], HWGN^2 [156], Goblin [153], Bake It Till You Make It [254], Garblet [158], and GarbledEDA [157] provide a strong foundation for guiding future research, ensuring that secure computation evolves from a theoretical construct to a practical and enforceable security standard in real-world systems.

9.0.2 Future Directions in Secure and Private Implementation of MPC

While secure MPC has made remarkable progress in enabling privacy-preserving computation, several key challenges still hinder its widespread deployment in real-world systems. As adversaries develop increasingly sophisticated attack vectors and hardware architectures evolve toward more complex and distributed designs, future secure computation frameworks must address several pressing concerns. These include reducing computational and communication overhead, improving robustness against side-channel and fault injection attacks, enhancing scalability for chiplet-based designs, and integrating cryptographic protocols with hardware-assisted security mechanisms. This section outlines several promising research directions that can guide the development of next-generation secure computation frameworks.

To ensure secure MPC continues to evolve from a theoretical construct into a practical, deployable technology, it must include interdisciplinary innovation that covers cryptography, systems design, and physical security. The following subsections detail key points for enhancing efficiency, robustness, and adaptability in future MPC implementations, particularly as secure computing applications expand across cloud infrastructure, edge devices, and AI inference pipelines.

9.0.3 Overcoming Computational and Communication Overhead

Despite considerable advancements in MPC efficiency, computational and communication overhead continues to be a primary barrier to large-scale deployment. Protocols based on GC, HE, and OT introduced high costs, making them less suitable for latency-sensitive or resource-constrained applications such as secure machine learning or privacy-preserving cloud services [267, 319, 202]. As such, continued research is needed to develop optimized MPC techniques that maintain strong security guarantees while significantly reducing operational costs.

A promising direction involves the use of hardware acceleration to enhance the performance of secure computation. The GuardianMPC [154] framework, for instance, demonstrated how integrating dedicated hardware modules, such as optimized OT engines, can effectively reduce both processing and communication overhead during secure inference [154]. These find-

ings show that hardware-software co-design—where cryptographic primitives are implemented directly in specialized accelerators—can offer considerable performance benefits without sacrificing privacy.

Building on this idea, future research should explore the design and implementation of specialized hardware platforms, including FPGAs and ASICs, to accelerate cryptographic operations while maintaining strict security properties. Such platforms may also support scalable reconfiguration, which would enable MPC frameworks to dynamically adapt performance characteristics to different application demands.

Another crucial area for improvement is minimizing the round complexity of MPC protocols. Many current schemes involve multiple rounds of communication, leading to undesirable latency, particularly in high-latency or bandwidth-constrained environments. Techniques such as function-independent preprocessing, batched operations, and communication pattern optimization offer viable paths to reducing this bottleneck [202]. These techniques are particularly important for scenarios like distributed learning, where network latency can reduce throughput [245].

Additionally, hybrid models that combine TEEs [81] with cryptographic MPC protocols may provide a more practical balance between efficiency and security by selectively offloading sensitive operations to trusted hardware components. This combination allows secure MPC systems to bypass expensive cryptographic operations when a minimal trusted base is available, enabling secure computation to scale into resource-constrained devices at the edge.

9.0.4 Stronger Resilience Against Fault and Side-Channel Attacks

To provide trustworthy and tamper-resilient results, secure computation frameworks must evolve to effectively counter physical-layer threats such as fault injection and SCA. While cryptographic protocols offer rigorous privacy guarantees at the algorithmic level, real-world hardware deployments are susceptible to implementation-level vulnerabilities [28]. These include variations in power consumption, electromagnetic emissions, timing behavior, and externally induced faults [104, 117, 28, 153]. Such attacks often bypass mathematical guarantees by targeting the physical substrate that executes secure computation.

Recent findings, such as the FaultyGarble [155] attack, have shown that even sophisticated MPC-based frameworks can be compromised via LFI. In these scenarios, adversaries introduced controlled faults during secure inference to leak internal model parameters [155]. These observations highlight a critical need for future MPC systems to move beyond passive cryptographic protection and incorporate active defenses that ensure the integrity of the computation process.

To mitigate these threats, future secure computation architectures must adopt built-in fault detection and fault tolerance mechanisms. Strategies such as redundant execution [300], dual-rail logic [323], randomized instruction scheduling [58], and majority voting [30] can help identify and neutralize fault-induced errors. Hardware-level monitoring units can also play a role by detecting abnormal power or thermal signatures indicative of fault attempts [254]. By embedding these mechanisms into secure computation frameworks, the system gains the ability to detect tampering at runtime and potentially recover from or halt faulty executions.

In parallel, addressing side-channel leakage remains a complex challenge. The HWGN^2 [156] framework introduced PFE techniques that obscure both inputs and functional structure, effectively reducing leakage from timing and power analysis [156]. These techniques represent a promising step toward minimizing leakage without incurring excessive computational cost.

Future research should focus on generalizing and optimizing such countermeasures to support a wide range of secure computation tasks. Additionally, dynamic runtime techniques such as circuit randomization [3] and operand shuffling [395] can further strengthen resistance to SCAs. On the hardware side, approaches such as power equalization, current flattening, and artificial noise injection can mask data-dependent variations [247]. Combining these techniques in a layered fashion will be critical to defending against increasingly advanced adversaries capable of performing high-resolution measurements and machine learning-assisted SCA analysis.

Ultimately, resilience against fault and side-channel attacks will require coordinated innovations across circuit design, compiler-level transformations, and cryptographic protocol engineering. The integration of these disciplines will be essential to achieving robust, real-time secure computation capable of resisting not only passive observation but also active physical tampering.

9.0.5 Scalability in Chiplet-Based Architectures

The growing adoption of chiplet-based architectures introduces new dimensions of complexity and risk in the design of secure computation systems. Unlike monolithic processors, which execute computation within a single trusted boundary, chiplet-based systems are composed of multiple discrete hardware units connected through high-speed interconnects. These components may originate from different vendors, vary in trustworthiness, and operate under differing security assumptions. This heterogeneity challenges the foundational assumptions of many traditional MPC protocols, which often presume a unified and fully trusted computational platform [303].

The Garblet framework offered a pioneering solution to this challenge by demonstrating how GC-based MPC could be efficiently distributed across multiple chiplets while preserving security guarantees and significantly reducing communication overhead [158]. This distribution enables higher parallelism and scalability, making it well-suited for complex workloads such as privacy-preserving neural network inference. However, scaling MPC across untrusted chiplets raises several unresolved issues that require focused research.

One critical challenge is ensuring that data transmitted between chiplets remains secure. Cross-chiplet data leakage, replay attacks, and fault propagation are genuine risks when computation is split across physically separate units. Secure inter-chiplet communication protocols must therefore include authentication [2], integrity checking [177], and encryption mechanisms [387] that operate at both the hardware and protocol levels [8].

Furthermore, adversarial chiplets can potentially deviate from expected computations or leak sensitive intermediate values. Addressing this requires the development of maliciously-secure MPC protocols and verification mechanisms capable of detecting and mitigating malicious behavior at runtime. Techniques such as circuit-level consistency checks [9], redundancy across chiplets [1], and secure multi-hop data routing [121] may help minimize trust assumptions in heterogeneous chiplet-based environments.

Future chiplet-based secure computation platforms must integrate security at the microarchitectural level by embedding hardware root-of-trust modules [13], implementing secure boot processes [372], and enabling fine-grained access control [271] between chiplets. These features will be vital for maintaining isolation and integrity in multi-chip designs, particularly in scenarios involving third-party components or off-the-shelf accelerators [271].

Moreover, as chiplet ecosystems continue to expand, there is a growing need for scalable resource management, load balancing, and task scheduling frameworks that are aware of both performance constraints and security risks. Ensuring consistent execution while distributing workloads across potentially untrusted compute units requires new orchestration techniques that tightly couple system-level scheduling with MPC security guarantees [8].

In summary, scalable secure computation in chiplet-based systems demands a rethinking of assumptions around trust boundaries, communication integrity, and architectural uniformity. By developing cryptographic protocols, runtime monitors, and microarchitectural primitives tailored to distributed heterogeneous computing, the MPC community can better prepare for the next wave of secure, modular hardware platforms.

Bibliography

[1] Secrop: Secure cluster head centered multi-hop routing protocol for mobile ad hoc networks. *Security and Communication Networks*, 9(16), 2016.

[2] Physical layer security: Authentication, integrity and confidentiality. *arXiv:2001.07153*, 2020.

[3] Generation of all randomizations using circuits. *Annals of the Institute of Statistical Mathematics*, 75:683–704, 2022.

[4] Andreas Agne, Markus Happe, and Marco Platzner. Seven recipes for setting your fpga on fire–a cookbook on heat generators. In *2014 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 1–7. IEEE, 2014.

[5] Gianluca Agosta and Jean-Michel Paris. Hardware authentication for secure chiplet integration. *IEEE Design & Test*, 39(5):70–85, 2022.

[6] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi. The em side—channel(s). In *Cryptographic Hardware and Embedded Systems (CHES 2002)*, volume 2523 of *Lecture Notes in Computer Science*, pages 29–45. Springer, 2002.

[7] Faizan Ahmad and Kazim Shamsi. Hardware security challenges and opportunities in chiplet-based architectures. *IEEE Transactions on Secure Systems*, 38(2):230–248, 2022.

[8] Aikata Aikata, Ahmet Can Mert, Sunmin Kwon, Maxim Deryabin, and Sujoy Sinha Roy. Reed: Chiplet-based accelerator for fully homomorphic encryption. *arXiv preprint arXiv:2308.02885*, 2023.

[9] D. Airehrour, J. Gutierrez, and S. K. Ray. A survey of secure routing protocols in multi-hop cellular networks. *IEEE Communications Surveys & Tutorials*, 20(4):3182–3203, 2018.

[10] MD Abdullah Alam, Domenic Forte, and Mark M Tehranipoor. Recycled fpga detection using ring oscillator patterns. In *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 43–48. IEEE, 2016.

[11] Md Mahbub Alam, Shahin Tajik, Fatemeh Ganji, Mark Tehranipoor, and Domenic Forte. RAM-Jam: Remote temperature and voltage fault attack on FPGAs using memory collisions. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 101–110, 2019.

[12] Faisal Ali and Neha Gupta. Modular integration of chiplets for ai accelerators. *ACM Transactions on Design Automation of Electronic Systems*, 27(3):1–20, 2022.

[13] Usman Ali, Hamza Omar, Chujiao Ma, Vaibhav Garg, and Omer Khan. Hardware root-of-trust implementations in trusted execution environments. Cryptology ePrint Archive, Paper 2023/251, 2023.

[14] Abdulaziz Alkanhal, Shams Shakib, and Krzysztof Gaj. An efficient ip protection technique for hardware accelerators using logic obfuscation and encryption. *IEEE Transactions on Emerging Topics in Computing*, 10(1):196–209, 2019.

[15] AlphaNOV. S-LMS. `[Online]https://www.alphanov.com/en/products-services/single-laser-fault-injection` [Accessed: Mar.5, 2024], 2023.

[16] Thales Alves and Felicita Felton. Trustzone: Integrated hardware and software security. In *ARM Technical Report*, 2004.

[17] Apple Computer, Inc. and IBM Corporation and Motorola, Inc. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann Publishers, 2 edition, 1995.

[18] ARM ARM. Architecture reference manual. *ARMv7-A and ARMv7-R edition*, 2012.

[19] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 535–548. ACM, 2013.

[20] C. Ashokkumar, Ravi Prakash Giri, and Bernard Menezes. Highly efficient algorithms for aes key retrieval in cache access attacks. In *2016 18th International Symposium on High Assurance Systems Engineering (HASE)*, pages 142–149. IEEE, 2016.

[21] Yonatan Aumann and Yehuda Lindell. Security in the presence of active and passive adversaries. *Journal of Cryptology*, 23:281–345, 2010.

[22] Ali Bahmani and Yousuf Ahmad. Trusted execution in heterogeneous multi-chip modules (mcms). *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(2):321–338, 2023.

[23] Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. Fault injection techniques on cryptographic devices. In *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 3–4. IEEE, 2011.

[24] Marshall Ball, Brent Carmer, Tal Malkin, Mike Rosulek, and Nichole Schimanski. Garbled neural networks are practical. *Cryptology ePrint Archive*, 2019.

[25] Marshall Ball, Tal Malkin, and Mike Rosulek. Garbling gadgets for boolean and arithmetic circuits. In *Proc. of the 2016 ACM SIGSAC Conf. on Computer and Comm. Security*, pages 565–577, 2016.

[26] Endre Bangerter, David Gullasch, and Stephan Krenn. Cache games—bringing access-based cache attacks on aes to practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505. IEEE, 2011.

[27] Hagai Bar-El, Hovav Shacham, Eli Biham, David Boneh, and Bart Preneel. The sorcerer's apprentice guide to fault-tolerant processing. In *Proceedings of the IEEE Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 65–75. IEEE, 2006.

[28] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, 2012.

[29] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. CSI NN: Reverse engineering of neural network architectures through electromagnetic side channel. In *28th USENIX Security Symp. (USENIX Security 19)*, pages 515–532, 2019.

[30] Stefan Baumgartner, Mario Huemer, and Michael Lunglmayr. Efficient majority voting in digital hardware. *arXiv:2108.03979*, 2021.

[31] Donald Beaver. Secure multiparty computation can be efficient. *Advances in Cryptology–CRYPTO'91*, pages 306–318, 1991.

[32] Donald Beaver, Shafi Goldwasser, and Silvio Micali. Secure multiparty protocols and zero-knowledge proof systems tolerating a faulty minority. *Journal of Cryptology*, 4:117–136, 1989.

[33] George Becker, Jim Cooper, Elke De Mulder, Gilbert Goodwill, Joshua Jaffe, Gary Kenworthy, Timofei Kouzminov, Andrew Leiserson, Mark Marson, and Pankaj Rohatgi. Test vector leakage assessment (tvla) methodology in practice. In *International Cryptographic Module Conference (ICMC)*, volume 1001, page 13. SN, 2013.

[34] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *2013 IEEE Symp. on Security and Privacy*, pages 478–492. IEEE, 2013.

[35] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *Proc. of the 2012 ACM Conf. on Computer and Comm. security*, pages 784–796, 2012.

[36] Aner Ben-Efraim, Yehuda Lindell, and Eran Omri. Optimizing semi-honest secure multiparty computation for the internet. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 578–590, 2016.

[37] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing (STOC)*, pages 1–10. ACM, 1988.

[38] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems with constructive proofs for symmetric-key cryptography. *SIAM Journal on Computing*, 48(3):792–811, 2019.

[39] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III 39*, pages 701–732. Springer, 2019.

[40] Yoshua Bengio. Learning deep architectures for ai. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2009.

[41] Sarah Benhani and Vasileios Zografos. Secure interconnects for chiplet-based architectures. *IEEE Transactions on Secure Computing*, 19(5):1023–1035, 2022.

[42] Daniel J. Bernstein. Cache-timing attacks on aes. Technical Report 2005/102, Cryptology ePrint Archive, 2005. https://eprint.iacr.org/2005/102.

[43] Vishal Bhat and Kunal Mehta. Zero-trust architectures for chiplet-based systems. *ACM Transactions on Secure Systems*, 20(4):87–102, 2021.

[44] Arnab Bhattacharyya, Vipul Goyal, Aayush Jain, and Peter S. Steenkiste. Secure computation with minimal interaction, revisited. In *Advances in Cryptology – CRYPTO 2021*, pages 659–689. Springer, 2021.

[45] Swarup Bhunia and Ashish Mehta. Future trends in secure chiplet integration. *IEEE Design & Test*, 41(1):23–39, 2023.

[46] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In Burt Kaliski, editor, *Advances in Cryptology – CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525. Springer, 1997.

[47] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus

Nielsen, Kurt Nielsen, Jakob Pagter, Michael Schwartzbach, and Tomas Toft. *Secure Multiparty Computation Goes Live*. Springer, 2009.

[48] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1175–1191, 2017.

[49] Dan Boneh, Richard A DeMillo, and Richard J Lipton. On the importance of checking cryptographic protocols for faults. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 37–51, 1997.

[50] Shekhar Borkar and Andrew A Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[51] Gilles Brassard, Claude Crépeau, and Jean-Marc Robert. All-or-nothing disclosure of secrets. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, volume 263 of *Lecture Notes in Computer Science*, pages 234–238. Springer, 1987.

[52] L Braun and W Zakarias, R. Tinylego framework. `[Online]https://github.com/AarhusCrypto/TinyLEGO` [Accessed Sep.28, 2023], 2019.

[53] Jakub Breier, Shivam Bhasin, David Jap, Kayalvizhi Khoo, and Simon Tinguely. Practical fault attack on deep neural networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2204–2206, 2018.

[54] Jakub Breier, Dirmanto Jap, Xiaolu Hou, Shivam Bhasin, and Yang Liu. Sniff: Reverse engineering of neural networks with fault attacks. *IEEE Trans. on Reliability*, 2021.

[55] Franjo Brglez, David Bryan, and Krzysztof Kozminski. Combinational profiles of sequential benchmark circuits. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1929–1934, 1985.

[56] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *Intrl. WKSP on cryptographic hardware and embedded systems*, pages 16–29. Springer, 2004.

[57] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[58] Christopher James Buehler. An instruction scheduling algorithm for communication-constrained microprocessors. Master's thesis, Massachusetts Institute of Technology, 1998.

[59] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security)*, pages 991–1008, 2018.

[60] Megha Byali, Harsh Chaudhari, Arpita Patra, and Ajith Suresh. Flash: Fast and robust framework for privacy-preserving machine learning. *Cryptology ePrint Archive*, 2019.

[61] Claude Carlet, Louis Goubin, Emmanuel Prouff, and Matthieu Rivain. Higher-order masking schemes for s-boxes. *Cryptology ePrint Archive*, 2012:203, 2012.

[62] Nicholas Carlini, Matthew Jagielski, and Ilya Mironov. Cryptanalytic extraction of neural network models. In *Annual Intrl. Cryptology Conf.*, pages 189–218. Springer, 2020.

[63] Henry Carter, Charles Lever, Patrick Traynor, and Kevin R. B. Butler. Secure outsourced garbled circuit evaluation for mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 289–306. USENIX Association, 2016.

[64] Esteban Castro, Swarup Bhunia, and Domenic Forte. Breaking ieee p1735 cryptography using side-channel attacks. In *Proceedings of the 2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 83–93, 2020.

[65] Rajat Chakraborty and Swarup Bhunia. Trusted chiplet integration: Challenges and solutions. *IEEE Transactions on Secure Hardware Design*, 28(1):55–72, 2020.

[66] Nishanth Chandran, Divya Gupta, Sai Lakshmi Bhavana Obbattu, and Akash Shah. {SIMC}:{ML} inference secure against malicious clients at {Semi-Honest} cost. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1361–1378, 2022.

[67] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. Ezpc: Programmable and efficient secure two-party computation for machine learning. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 496–511. IEEE, 2019.

[68] Nishanth Chandran, Aayush Jain, Rafail Ostrovsky, and Amit Sahai. Threshold fully homomorphic encryption and secure computation. In *Advances in Cryptology – EUROCRYPT 2020*, pages 501–531. Springer, 2020.

[69] Suresh Chari, Josyula R Rao, and Pankaj Rohatgi. Template attacks. In *Intrl. WKSP on Cryptographic Hardware and Embedded Systems*, pages 13–28. Springer, 2002.

[70] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. Trident: Efficient 4pc framework for privacy preserving machine learning. *arXiv preprint arXiv:1912.02631*, 2019.

[71] Hao Chen, Qing Wang, and Ning Wu. V2c: A verilog to c translation tool for hardware/software co-design. In *Proceedings of the 2013 IEEE International Symposium on Circuits and Systems*, pages 120–123, 2013.

[72] Tiancheng Chen, Xiao Wang, Yu Wang, and Mingsheng Zhang. Secureml: Privacy-preserving machine learning. In *USENIX Security Symposium*, pages 555–572, 2020.

[73] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 609–622. IEEE Computer Society, 2014.

[74] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv preprint arXiv:1712.05526*, 2017.

[75] Yu Chen, Lei Zhang, Tingting Wu, and Xiaodong Du. Fobnn: Fully oblivious binarized neural networks for privacy-preserving machine learning. *IEEE Transactions on Information Forensics and Security*, 2024.

[76] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*, pages 409–437. Springer, 2017.

[77] To-Yat Cheung. Graph traversal techniques and the maximum flow problem in distributed computation. *IEEE Transactions on Software Engineering*, SE-9(4):504–512, 1983.

[78] Seung Geol Choi, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. On the security of the "free-xor" technique. In *Theory of Cryptography Conference*, pages 39–53. Springer, 2012.

[79] Josh Daniel Cohen-Benaloh and Michael de Mare. Efficient broadcast time-stamping. *Technical Report, Clarkson University*, 1987.

[80] Jean-Sébastien Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In *Cryptographic Hardware and Embedded Systems - CHES 1999, First International Workshop, Worcester, MA, USA, August 12-13, 1999, Proceedings*, pages 292–302. Springer, 2000.

[81] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[82] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, 2015.

[83] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A secure and optimally efficient multi-authority election scheme. In *Annual International Cryptology Conference (CRYPTO)*, pages 103–118. Springer, 1997.

[84] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael. In *First Advanced Encryption Standard (AES) Conference*, 1998.

[85] Anders Dalskov, Daniel Escudero, and Marcel Keller. Fantastic four:{Honest-Majority}{Four-Party} secure computation with malicious security. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2183–2200, 2021.

[86] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology–CRYPTO 2012: 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, pages 643–662. Springer, 2012.

[87] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology – CRYPTO 2012*, pages 643–662. Springer, 2012.

[88] Saurabh Das and Anil Kumar. Design challenges and future prospects of chiplet architectures. *Journal of Computer Engineering*, 20(1):88–102, 2021.

[89] Saurabh Das, Jian Zhang, and Yu Wang. Chiplet-based architecture for next-generation computing systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(7):1894–1906, 2022.

[90] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. Chet: an optimizing compiler for fully-homomorphic neural-network inferencing. In *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation*, pages 142–156, 2019.

[91] Thomas De Cnudde, Maik Ender, and Amir Moradi. Hardware masking, revisited. *IACR Trans. on Cryptographic Hardware and Embedded Systems*, pages 123–148, 2018.

[92] Roy De Maesschalck, Dominique Jouan-Rimbaud, and Denny L Massart. On mahalanobis distance classification. *Chemometrics and Intelligent Laboratory Systems*, 50(1):1–18, 2000.

[93] E. De Mulder, B. Wyseur, F.-X. Standaert, and I. Verbauwhede. A practical introduction to hardware security. In *Fault Analysis in Cryptography*, pages 1–19. Springer, 2013.

[94] Amine Dehbaoui, Karim Tobich, Jean-Max Dutertre, and Assia Tria. Electromagnetic fault injection: Towards a fault model on a 32-bit microcontroller. In *IEEE Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 7–15, 2012.

[95] Daniel Demmler, Thomas Schneider, and Michael Zohner. Aby-a framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[96] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE signal processing magazine*, 29(6):141–142, 2012.

[97] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[98] Jack Doerner, David Evans, and Abhi Shelat. Secure stable matching at scale. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1602–1613, 2016.

[99] Zhiwei Du, Rodrigo Guerra, Shuangchen Li, Linghao Song, Xiaochen Guo, and Yuan Xie. Shidiannao: Shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 92–104. ACM, 2015.

[100] Richard Dubes and Anil K Jain. *Cluster Analysis and Applications*. Prentice-Hall, 1980.

[101] Anuj Dubey, Afzal Ahmad, Muhammad Adeel Pasha, Rosario Cammarota, and Aydin Aysu. Modulonet: Neural networks meet modular arithmetic for efficient hardware masking. *IACR Trans. on Cryptographic Hardware and Embedded Systems*, pages 506–556, 2022.

[102] Anuj Dubey, Rosario Cammarota, and Aydin Aysu. Bomanet: Boolean masking of an entire neural network. In *2020 IEEE/ACM Intrl. Conf. On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2020.

[103] Anuj Dubey, Rosario Cammarota, and Aydin Aysu. Maskednet: The first hardware inference engine aiming power side-channel protection. In *2020 IEEE Intrl. Symp. on Hardware Oriented Security and Trust (HOST)*, pages 197–208. IEEE, 2020.

[104] Ferhat Erata, TingHung Chiu, Anthony Etim, Srilalith Nampally, Tejas Raju, Rajashree Ramu, Ruzica Piskac, Timos Antonopoulos, Wenjie Xiong, and Jakub Szefer. Systematic use of random self-reducibility against physical attacks. *arXiv preprint arXiv:2405.05193*, 2024.

[105] Brian Everitt, Sabine Landau, Morven Leese, and Daniel Stahl. *Cluster Analysis*. John Wiley & Sons, 2011.

[106] Ilya M Filanovsky and Aiman Allam. Mutual compensation of mobility and threshold voltage temperature effects with applications in cmos circuits. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 48(7):876–884, 2001.

[107] Renato Fogaça, Adriano Pereira, and Sergio Lima. Chiplet integration and packaging: Current trends and challenges. *Microelectronics Journal*, 138:105987, 2023.

[108] Tore Kasper Frederiksen, Thomas P Jakobsen, Jesper Buus Nielsen, and Roberto Trifiletti. Tinylego: An interactive garbling scheme for maliciously secure two-party computation. *Cryptology ePrint Archive*, 2015.

[109] Tore Kasper Frederiksen, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Peter Sebastian Nordholt, and Claudio Orlandi. Minilego: Efficient secure two-party computation from general assumptions. In *Advances in Cryptology–EUROCRYPT 2013: 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques,*

Athens, Greece, May 26-30, 2013. Proceedings 32, pages 537–556. Springer, 2013.

[110] Jean Pierre Gag, Christophe Paillard, Bruno Robisson, and Philippe Maurine. Temperature impact on cmos logic gates. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(5):791–801, 2012.

[111] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 251–261, 2001.

[112] Karla Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In *Cryptographic Hardware and Embedded Systems – CHES 2001*, pages 251–261. Springer, 2001.

[113] Yansong Gao, Bao Gia Doan, Zhi Zhang, Siqi Ma, Jiliang Zhang, Anmin Fu, Surya Nepal, and Hyoungshick Kim. Backdoor attacks and countermeasures on deep learning: A comprehensive review. *arXiv preprint arXiv:2007.10760*, 2020.

[114] David Garcia, Gustavo Mariscal, Ramesh Karri, and Michail Maniatakos. Optimized memory allocation for side-channel protected implementations on fpga. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 45–54. ACM, 2019.

[115] Adrià Gascón, Phillipp Schoppmann, Borja Balle, Mariana Raykova, Jack Doerner, Samee Zahur, and David Evans. Privacy-preserving distributed linear regression on high-dimensional data. *Proc. Priv. Enhancing Technol.*, 2017(4):345–364, 2017.

[116] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8:1–27, 2018.

[117] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. Ecdsa key extraction from mobile devices via nonintrusive physical side channels. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1475–1490, 2018.

[118] Daniel Genkin, Adi Shamir, and Eran Tromer. Rsa key extraction via low-bandwidth acoustic cryptanalysis. In *Annual Cryptology Conference*, pages 444–461. Springer, 2014.

[119] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford University, 2009. `crypto.stanford.edu/craig`.

[120] Craig Gentry. Fully homomorphic encryption using ideal lattices. *STOC*, pages 169–178, 2009.

[121] Mohammed Gharib, Ali Owfi, and Soudeh Ghorbani. Kpsec: Secure end-to-end communications for multi-hop wireless networks. *arXiv:1911.05126*, 2019.

[122] Tarun Ghose and Rahul Singh. Beyond monolithic chips: The rise of chiplet architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 30(12):3215–3230, 2022.

[123] Rajarshi Ghosh and Sudhakar Prasad. Chiplet-based architectures for cloud computing and ai workloads. *IEEE Cloud Computing*, 8(2):56–64, 2021.

[124] Benedikt Gierlichs, Lejla Batina, Bart Preneel, and Ingrid Verbauwhede. Mutual information analysis: a generic side-channel distinguisher. *International workshop on cryptographic hardware and embedded systems*, pages 426–442, 2008.

[125] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *Intrl. Conf. on machine learning*, pages 201–210. PMLR, 2016.

[126] Christophe Giraud. Differential fault analysis on aes key schedule and countermeasures. *International Conference on Smart Card Research and Advanced Applications (CARDIS)*, pages 1–13, 2004.

[127] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 15:315–323, 2011.

[128] David R Gnad and Mehdi B Tahoori. Analysis of power side-channel leakage in fpga hardware. *IEEE Transactions on Computers*, 66(3):496–509, 2016.

[129] Oded Goldreich. *The Foundations of Cryptography - Volume 2*. Cambridge University Press, 2004.

[130] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 218–229. ACM, 1987.

[131] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM*, 43(3):431–473, 1996.

[132] Shafi Goldwasser, Yael Kalai, Raluca Ada Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 555–564, 2013.

[133] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. One-time programs. In *Annual Intrl. Cryptology Conf.*, pages 39–56. Springer, 2008.

[134] Shafi Goldwasser, Yael Tauman Kalai, and Guy N Rothblum. Knowledge assumptions and their applications to cryptographic protocol design. *SIAM Journal on Computing*, 48(3):730–773, 2019.

[135] Shafi Goldwasser, Michael P Kim, Vinod Vaikuntanathan, and Or Zamir. Planting undetectable backdoors in machine learning models. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 931–942. IEEE, 2022.

[136] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1982.

[137] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing (STOC)*, pages 291–304. ACM, 1985.

[138] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[139] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 2672–2680, 2014.

[140] Adam Groce, Alex Ledger, Alex J Malozemoff, and Arkady Yerukhimovich. Compgc: Efficient offline/online semi-honest two-party computation. *Cryptology ePrint Archive*, 2016.

[141] Daniel Gruss, Clément Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in javascript. *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, pages 300–321, 2016.

[142] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *arXiv preprint arXiv:1708.06733*, 2017.

[143] Shay Gueron and Vlad Krasnov. Fast and side-channel resistant AES-GCM using AVX instructions. In *2011 IEEE Symposium on Security and Privacy Workshops*, pages 19–25. IEEE, 2011.

[144] Shay Gueron, Yehuda Lindell, Ariel Nof, and Benny Pinkas. Fast garbling of circuits under standard assumptions. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 567–578, 2015.

[145] Ujjwal Guin, Ratnesh Singh, and Mark Tehranipoor. Counterfeit detection in chiplet-based systems: Challenges and solutions. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(7):1239–1250, 2018.

[146] Chun Guo, Jonathan Katz, Xiao Wang, Chenkai Weng, and Yu Yu. Better concrete security for half-gates garbling (in the multi-instance setting). In *Annual International Cryptology Conference*, pages 793–822. Springer, 2020.

[147] Chun Guo, Jonathan Katz, Xiao Wang, and Yu Yu. Efficient and secure multiparty computation from fixed-key block ciphers. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 825–841. IEEE, 2020.

[148] Trinabh Gupta, Henrique Fingler, Lorenzo Alvisi, and Michael Walfish. Pretzel: Email encryption and provider-supplied functions are compatible. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 169–182, 2017.

[149] Shai Halevi and Jing Zhang. Homomorphic encryption for secure ai inference in multi-chip modules. *IEEE Transactions on Secure Computing*, 28(1):120–139, 2020.

[150] Hisashi Hanamura, Hiroshi Kitagawa, Akira Tsunoda, Noboru Asahi, and Hiroyuki Sugiura. Operation of cmos circuits at cryogenic temperatures. *IEEE Journal of Solid-State Circuits*, 21(4):619–625, 1986.

[151] Markus Happe, Andreas Agne, Christian Plessl, and Marco Platzner. Eight ways to put your fpga on fire–a systematic study of heat generators. In *2012 IEEE International Symposium on Field-Programmable Custom Computing Machines*, pages 199–206. IEEE, 2012.

[152] Tyler Harvey, Jonathan Koppel, and Daniel Koppel. Thermanator: Thermal residue-based post factum attacks on keyboard password entry. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2018.

[153] Mohammad Hashemi, Domenic Forte, and Fatemeh Ganji. Time is money, friend! timing side-channel attack against garbled circuit constructions. In *International Conference on Applied Cryptography and Network Security*, pages 325–354. Springer, 2024.

[154] Mohammad Hashemi, Domenic J Forte, and Fatemeh Ganji. Guardianmpc: Backdoor-resilient neural network computation. *IEEE Access*, 2025.

[155] Mohammad Hashemi, Dev Mehta, Kyle Mitard, Shahin Tajik, and Fatemeh Ganji. Faultygarble: Fault attack on secure multiparty neural network inference. *Cryptology ePrint Archive*, 2024.

[156] Mohammad Hashemi, Steffi Roy, Domenic Forte, and Fatemeh Ganji. Hwgn 2: Side-channel protected nns through secure and private function evaluation. In *Security, Privacy, and Applied Cryptography Engineering: 12th International Conference, SPACE 2022, Jaipur, India, December 9–12, 2022, Proceedings*, pages 225–248, 2022.

[157] Mohammad Hashemi, Steffi Roy, Fatemeh Ganji, and Domenic Forte. Garbled eda: Privacy preserving electronic design automation. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pages 1–9, 2022.

[158] Mohammad Hashemi, Shahin Tajik, and Fatemeh Ganji. Garblet: Multi-party computation for protecting chiplet-based systems. *Cryptology ePrint Archive*, 2025.

[159] Trevor Hastie, Robert Tibshirani, Jerome H Friedman, and Jerome H Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, volume 2. Springer, 2009.

[160] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. Sok: General purpose compilers for secure multi-party computation. In *2019 IEEE symposium on security and privacy (SP)*, pages 1220–1237. IEEE, 2019.

[161] William Hatcher and Wei Yu. A survey of deep learning approaches for network security. *IEEE Communications Surveys & Tutorials*, 21(1):18–43, 2018.

[162] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2017.

[163] Ehsan Hesamifard, Hassan Takabi, Mehdi Ghasemi, and C. Battaglino. Privacy-preserving machine learning as a service. *Proceedings on Privacy Enhancing Technologies*, 2018(3):123–142, 2018.

[164] Benjamin Hettwer, Stefan Gehrer, and Tim Güneysu. Applications of machine learning techniques in side-channel attacks: a survey. *J. of Cryptographic Engineering*, 10(2):135–162, 2020.

[165] Benjamin Hettwer, Johannes Heyszl, and Georg Sigl. Differential cluster analysis. *International Conference on Smart Card Research and Advanced Applications*, pages 19–35, 2019.

[166] Annelie Heuser and Michael Zohner. Intelligent machine homicide. In *Intrl. WKSP on Constructive Side-Channel Analysis and Secure Design*, pages 249–264. Springer, 2012.

[167] Sanghyun Hong, Nicholas Carlini, and Alexey Kurakin. Handcrafted backdoors in deep neural networks. *Advances in Neural Information Processing Systems*, 35:8068–8080, 2022.

[168] Md Hoque and Redwan Hasan. Hardware trojans in chiplet architectures: Threats and countermeasures. *ACM Transactions on Embedded Computing Systems*, 20(6):1–24, 2021.

[169] Gabriel Hospodar, Benedikt Gierlichs, Elke De Mulder, Ingrid Verbauwhede, and Joos Vandewalle. Machine learning in side-channel analysis: A first study. *J. of Cryptographic Engineering*, 1(4):293, 2011.

[170] Xiaolu Hou, Bin Wang, Xiapu Xu, Patrick Lin, and Xiaosong Du. Security analysis of deep learning models against side-channel attacks. *IEEE Transactions on Information Forensics and Security*, 15:2119–2131, 2020.

[171] Licheng Hu, Jinyi Wang, Mingjie Liang, and Patrick Schaumont. TimeCrypt: Efficient timing channel protection for hardware and software enclaves. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 161–171. IEEE, 2019.

[172] Yan Huang, Jonathan Katz, Vladimir Kolesnikov, Ranjit Kumaresan, and Alex J. Malozemof. Amortizing garbled circuits. In *Annual Cryptology Conf.*, pages 458–475. Springer, 2014.

[173] Siam Hussain, Baiyu Li, Farinaz Koushanfar, and Rosario Cammarota. Tinygarble2: Smart, efficient, and scalable yao's garble circuit. In *Proc. of the 2020 WKSP on Privacy-Preserving Machine Learning in Practice*, pages 65–67, 2020.

[174] Siam U Hussain and Farinaz Koushanfar. Fase: Fpga acceleration of secure function evaluation. In *2019 IEEE 27th Annual Intrl. Symp.*

on Field-Programmable Custom Computing Machines (FCCM), pages 280–288. IEEE, 2019.

[175] IEEE Standards Association. IEEE Standard for Encryption and Management of Electronic Design Intellectual Property (IP). `https://standards.ieee.org/ieee/1735/5774/`, 2016. IEEE Standard P1735-2016.

[176] Sagar Imambi, Kolla Bhanu Prakash, and GR Kanagachidambaresan. Pytorch. *Programming with TensorFlow: solution for edge computing applications*, pages 87–104, 2021.

[177] Kazuya Imamura, Kazuhiko Minematsu, and Tetsu Iwata. Integrity analysis of authenticated encryption based on stream ciphers. In *Provable Security (ProvSec 2016)*, pages 257–276. Springer, 2016.

[178] Xilinx Inc. Axi reference guide for ultrascale+ fpga architecture, 2021. Accessed: 2025-02-28.

[179] Xilinx Inc. Ultrascale architecture and product data sheet: Overview, 2021. Accessed: 2025-02-28.

[180] Xilinx Inc. Ultrascale+ fpga chiplet-based design and communication protocols, 2021. Accessed: 2025-02-28.

[181] Xilinx Inc. Ultrascale+ fpga product selection guide, 2021. Accessed: 2025-02-28.

[182] Xilinx Inc. Xilinx power estimator (xpe) user guide. `https://www.xilinx.com/products/design-tools/power/xpe.html`, 2024.

[183] Intel Corporation. Intel® Core™ i7 Processors. `[Online]https://www.intel.com/content/www/us/en/products/details/processors/core/i7.html` [Accessed: Oct.16, 2024], 2017.

[184] Intel Corporation. Intel optimization manual. `https://software.intel.com/en-us/articles/intel-optimization-manual`, 2020.

[185] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 2022. Volume 1: Basic Architecture.

[186] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, cross-vm attack on aes. In *Research in Attacks, Intrusions and Defenses - RAID 2014*, volume 8688 of *Lecture Notes in Computer Science*, pages 299–319. Springer, 2014.

[187] irdan. Justgarble framework. `[Online]https://github.com/irdan/justGarble` [Accessed Jan.30, 2023], 2014.

[188] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In *Advances in Cryptology – CRYPTO 2003*, pages 145–161. Springer, 2003.

[189] Matthew Jagielski, Giorgio Severi, Niklas Pousette Harger, and Alina Oprea. Subpopulation data poisoning attacks. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3104–3122, 2021.

[190] Anil K. Jain, M. Narasimha Murty, and Patrick J. Flynn. Data clustering: A review. *ACM Computing Surveys (CSUR)*, 31(3):264–323, 1999.

[191] Jan Jancar. The state of tooling for verifying constant-timeness of cryptographic implementations. `[Online]https://neuromancer.sk/article/26` [Accessed: Oct.16, 2024], 2021.

[192] Kimmo Järvinen, Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. Garbled circuits for leakage-resilience: Hardware implementation and evaluation of one-time programs. In *Intrl. WKSP on Cryptographic Hardware and Embedded Systems*, pages 383–397. Springer, 2010.

[193] Bargav Jayaraman, Hannah Li, and David Evans. Decentralized certificate authorities. *arXiv preprint arXiv:1706.03370*, 2017.

[194] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. {GAZELLE}: A low latency framework for secure neural network inference. In *27th USENIX Security Symp. (USENIX Security 18)*, pages 1651–1669, 2018.

[195] Ankur Kalra and Mukesh Kumar Sharma. Effect of temperature variation on cmos delay and leakage. *International Journal of VLSI design & Communication Systems*, 4(4):1, 2013.

[196] Seny Kamara and Mariana Raykova. Secure outsourced computation in a multi-tenant cloud. In *International Conference on Financial Cryptography and Data Security (FC)*, pages 26–44. Springer, 2012.

[197] Lars Kamm, Rainer Biehl, Matthias Dankar, and Jean-Pierre Hubaux. Secure multiparty computation for genomics: Private set intersection and population stratification. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2015(1):133–147, 2015.

[198] Gerry Kane. *mips RISC Architecture*. Prentice-Hall, Inc., 1988.

[199] Ramesh Karri, Jiang Hu, Prabhat Mishra, and Yier Jin. Fault-based attack detection in cryptographic hardware. *IEEE Computer*, 43(11):76–82, 2010.

[200] Jonathan Katz and Lior Malka. Constant-round private function evaluation with linear complexity. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 556–571. Springer, 2014.

[201] Leonard Kaufman and Peter J Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley & Sons, 2009.

[202] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Motion: A framework for mixed-protocol secure computation. *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2020(2):173–190, 2020.

[203] Joe Kilian. Founding crytpography on oblivious transfer. In *Proc. of the annual ACM Symp. on Theory of computing*, pages 20–31, 1988.

[204] Jaehun Kim, Stjepan Picek, Annelie Heuser, Shivam Bhasin, and Alan Hanjalic. Make some noise. unleashing the power of convolutional neural networks for profiled side-channel analysis. *IACR Trans. on Cryptographic Hardware and Embedded Systems*, pages 148–179, 2019.

[205] Yoongu Kim, Ross Daly, Jeremie Kim, Onur Mutlu, and Vijay Seshadri. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[206] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[207] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. *Communications of the ACM*, 63(7):93–101, 2020.

[208] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*, pages 19–37, 2019.

[209] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[210] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology - CRYPTO 1999, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, pages 388–397. Springer, 1999.

[211] Vladimir Kolesnikov and Sai Mohan Kumaresan. Improved ot extension for transferring short secrets. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology – CRYPTO 2015*, volume 9216 of *Lecture Notes in Computer Science*, pages 54–70. Springer, 2015.

[212] Vladimir Kolesnikov, Mike Rosulek, and Ni Trieu. Duplo: Efficient multi-party computation benchmarking and its application to hardware design. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 509–526. ACM, 2017.

[213] Vladimir Kolesnikov, Mike Rosulek, and Ni Trieu. Practical multi-party private set intersection from symmetric-key techniques. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1257–1272. ACM, 2018.

[214] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free xor gates and applications. In *Intrl. Colloquium on Automata, Languages, and Programming*, pages 486–498. Springer, 2008.

[215] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. {SWIFT}: Super-fast and robust {Privacy-Preserving} machine learning. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2651–2668, 2021.

[216] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 1097–1105, 2012.

[217] Naveen Kumar and Divya Gupta. Efficient secure multiparty computation: Theory, practice, and applications. *ACM Computing Surveys*, 54(2):1–36, 2021.

[218] Eyal Kushilevitz and Rafail Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 364–373, 1997.

[219] Chun-Hao Lai, Jishen Zhao, and Chia-Lin Yang. Leave the cache hierarchy operation as it is: A new persistent memory accelerating approach. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pages 1–6, 2017.

[220] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[221] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proc. of the IEEE*, 86(11):2278–2324, 1998.

[222] Ryan Lehmkuhl, Pratyush Mishra, Akshayaram Srinivasan, and Raluca Ada Popa. Muse: Secure inference resilient to malicious clients. In *USENIX Security Symposium*, pages 2201–2218, 2021.

[223] Itamar Levi and Carmit Hazay. Garbled-circuits from an sca perspective: Free xor can be quite expensive... *Cryptology ePrint Archive*, 2022.

[224] Shuang Li, Yannan Wu, Xuan Zhao, Yinan Bao, and Wei Zhang. Chiplet: A new paradigm in high-performance secure computing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(12):4314–4327, 2020.

[225] Y Lindell and B Pinkas. A proof of yao's protocol for secure two-party computation. eccc report tr04-063. In *Electronic Colloquium on Computational Complexity (ECCC)*, 2004.

[226] Yehuda Lindell. Fast cut-and-choose-based protocols for malicious and covert adversaries. *J. of Cryptology*, 29(2):456–490, 2016.

[227] Yehuda Lindell. Secure multiparty computation. *Communications of the ACM*, 64(1):86–96, 2020.

[228] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Advances in Cryptology-EUROCRYPT 2007: 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Barcelona, Spain, May 20-24, 2007. Proceedings 26*, pages 52–78. Springer, 2007.

[229] Yehuda Lindell and Benny Pinkas. A proof of security of yao's protocol for two-party computation. *J. of cryptology*, 22(2):161–188, 2009.

[230] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. *Journal of cryptology*, 25:680–722, 2012.

[231] Yehuda Lindell, Benny Pinkas, Nigel P Smart, and Avishay Yanai. Efficient constant-round multi-party computation combining bmr and spdz. *J. of Cryptology*, 32(3):1026–1069, 2019.

[232] Yehuda Lindell and Ben Riva. Cut-and-choose yao-based secure computation in the online/offline and batch settings. In *Advances in Cryptology–CRYPTO 2014: 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II 34*, pages 476–494. Springer, 2014.

[233] Yehuda Lindell and Ben Riva. Blazing fast 2pc in the offline/online setting with security for malicious adversaries. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 579–590, 2015.

[234] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take a way: Exploring the security implications of amd's cache way predictors. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 813–825, 2020.

[235] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. Platypus: Software-based power side-channel attacks on x86. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 355–371. IEEE, 2021.

[236] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security)*, pages 973–990, 2018.

[237] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*, pages 605–622. IEEE, 2015.

[238] Jian Liu, Mika Juuti, Yao Lu, and Nadarajah Asokan. Oblivious neural network predictions via minionn transformations. In *Proc. of the 2017 ACM SIGSAC Conf. on computer and Comm. security*, pages 619–631, 2017.

[239] Shuo Liu, Jian Liu, and Neil Zhenqiang Gong. Feature inference attacks against model partitioning. *IEEE Symposium on Security and Privacy (S&P)*, pages 2386–2402, 2022.

[240] Yuntao Liu, Dana Dachman-Soled, and Ankur Srivastava. Mitigating reverse engineering attacks on deep neural networks. In *2019 IEEE Computer Society Annual Symp. on VLSI (ISVLSI)*, pages 657–662. IEEE, 2019.

[241] Zhiyuan Liu, Haoran Zhao, and Qiang Feng. Heterogeneous chiplet integration: Trends and prospects. *IEEE Transactions on Electron Devices*, 70(2):1084–1096, 2023.

[242] Joe Loughry and David A Umphress. Information leakage from optical emanations. *ACM Transactions on Information and System Security (TISSEC)*, 5(3):262–289, 2002.

[243] Yangdi Lyu and Prabhat Mishra. A survey of side-channel attacks on caches and countermeasures. *Journal of Hardware and Systems Security*, 2(1):33–50, 2018.

[244] Pieter Maene, Bart Coppens, and Ingrid Verbauwhede. Hardware-based fault injection attacks and countermeasures. *IEEE Transactions on Secure Computing*, 15(4):1431–1444, 2017.

[245] Kiwan Maeng and G Edward Suh. Approximating relu on a reduced ring for efficient mpc-based private inference. *arXiv preprint arXiv:2309.04875*, 2023.

[246] A.J Malozemoff, X Wang, and J Katz. Emp-toolkit framework. `[Online]https://github.com/emp-toolkit` [Accessed Jan.30, 2023], 2022.

[247] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007.

[248] Zoltán Ádám Mann, Christian Weinert, Daphnee Chabal, and Joppe W Bos. Towards practical secure neural network inference: the journey so far and the road ahead. *ACM Computing Surveys*, 56(5):1–37, 2023.

[249] Heiko Mantel, Alexander Weber, and Daniel Taha. Using pca to enhance side-channel evaluations. *Journal of Cryptographic Engineering*, 7(4):339–357, 2017.

[250] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 118–129. IEEE, 2012.

[251] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[252] Frank McKeen, Ilya Alexandrovich, Isaac Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. Innovative instructions and software model for isolated execution. *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, pages 1–10, 2013.

[253] Dev M Mehta, Mohammad Hashemi, David S Koblah, Domenic Forte, and Fatemeh Ganji. Bake it till you make it: Heat-induced power leakage from masked neural networks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(4):569–609, 2024.

[254] Rahul Mehta, Philip Karanja, Fawad Ahmad, Sanu Mohan, and Umit Ogras. Bake: Breaking ai-based key extraction with fault injection attacks. *IEEE Transactions on Dependable and Secure Computing*, 2023.

[255] Fabio Merli, Klaus Schindler, David Oswald, and Christof Paar. Side-channel attacks on cryptographic software: A systematic analysis of the major software countermeasures against side-channel attacks. *IEEE Transactions on Information Forensics and Security*, 8(1):67–81, 2013.

[256] Abdullah Mesbah, Saurabh Bagchi, and Anand Raghunathan. Investigation of power side-channel leakage in hardware masking schemes. In *Proceedings of the 54th Annual Design Automation Conference (DAC)*, pages 1–6. ACM, 2017.

[257] Thomas S. Messerges. Using second-order power analysis to attack dpa resistant software. *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 238–251, 2000.

[258] Gabriel Miranda, Saurabh Das, and Yu Wang. Chiplet-based architectures: Opportunities and challenges. *IEEE Design & Test*, 39(3):56–65, 2022.

[259] Ilya Mironov, Omkant Pandey, Omer Reingold, and Salil Vadhan. Computational differential privacy. In *Advances in Cryptology - CRYPTO 2011*, pages 126–142. Springer, 2011.

[260] Pratyush Mishra, Xiao Wang, Chang Liu, Matt Fredrikson, and Anupam Datta. Delphi: A cryptographic inference service for neural networks. In *USENIX Security Symposium*, pages 2505–2522, 2020.

[261] Sparsh Mittal, Himanshi Gupta, and Srishti Srivastava. A survey on hardware security of DNN models and accelerators. *J. of Systems Architecture*, 117:102163, 2021.

[262] Fan Mo, My T. Thai, Mehdi Nematollahi, Anupam Chattopadhyay, Prateek Saxena, and Jian Liu. Haac: Hardware-assisted efficient and scalable secure inference with homomorphic encryption. *CoRR*, abs/2301.12994, 2023.

[263] Ahmad Moghimi and Berk Sunar. Cachezoom: How SGX amplifies the power of cache attacks. In *Cryptographic Hardware and Embedded Systems - CHES 2017*, volume 10529 of *Lecture Notes in Computer Science*, pages 69–90. Springer, 2017.

[264] Payman Mohassel and Peter Rindal. Fast and secure three-party computation: The garbled circuit approach. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 591–602. ACM, 2014.

[265] Payman Mohassel, Peter Rindal, and Mike Rosulek. Fast and secure three-party computation: The garbled circuit approach. In *ACM Conference on Computer and Communications Security (CCS)*, pages 591–608, 2018.

[266] Payman Mohassel, Mike Rosulek, and Ye Zhang. Fast and secure three-party computation: The garbled circuit approach. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 591–602, 2015.

[267] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE symposium on security and privacy (SP)*, pages 19–38. IEEE, 2017.

[268] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. Universal adversarial perturbations. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1765–1773, 2017.

[269] Amir Moradi, Markus Kasper, and Christof Paar. Em side-channel attacks on commercial contactless smartcards using low-cost equipment. *Financial Cryptography and Data Security*, 15:79–99, 2016.

[270] Maria Mushtaq, Muhammad Asim Mukhtar, Vianney Lapotre, Muhammad Khurram Bhatti, and Guy Gogniat. Winter is here! a decade of cache-based side-channel attacks, detection & mitigation for RSA. *Information Systems*, 92:101524, 2020.

[271] Mohammed Nabeel, Mohammed Ashraf, Satwik Patnaik, Vassos Soteriou, Ozgur Sinanoglu, and Johann Knechtel. 2.5d root of trust: Secure system-level integration of untrusted chiplets. `https://arxiv.org/abs/2009.02412`, 2020. Accessed: 2025-04-17.

[272] Ram Nagarajan and Pranav Pillai. Hardware root-of-trust architectures for chiplet-based security. *IEEE Transactions on Embedded Systems Security*, 18:51–68, 2022.

[273] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML)*, pages 807–814, 2010.

[274] Aoi Nakamoto. W-shield: Protection against cryptocurrency wallet credential stealing. In *Workshop on Security and Privacy in E-Commerce 2018*, pages 71–107, 2018.

[275] Moni Naor and Benny Pinkas. Efficient oblivious transfer protocols. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 448–457. Society for Industrial and Applied Mathematics, 2001.

[276] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *Proceedings of the 1st ACM Conference on Electronic Commerce*, pages 129–139, 1999.

[277] Ajay Nayak and Ramesh Lalgudi. Accurate delay measurement in logic circuits using tester-per-pin timing measurement capability. *IEEE Transactions on Components and Packaging Technologies*, 25(4):705–712, 2002.

[278] Paarth Neekhara, Shehzeen Hussain, Prakhar Pandey, Shlomo Dubnov, Julian McAuley, and Farinaz Koushanfar. Universal adversarial perturbations for speech recognition systems. *arXiv preprint arXiv:1905.03828*, 2019.

[279] Lucien KL Ng and Sherman SM Chow. Sok: Cryptographic neural-network computation. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 497–514. IEEE, 2023.

[280] Jesper Buus Nielsen and Claudio Orlandi. Lego for two-party secure computation. In *Theory of Cryptography Conference*, pages 368–386. Springer, 2009.

[281] Jesper Buus Nielsen, Thomas Schneider, and Roberto Trifiletti. Constant round maliciously secure 2pc with function-independent preprocessing using lego. *Cryptology ePrint Archive*, 2016.

[282] Valeria Nikolaenko, Siddharth Jaggi, Sriram Rajamani, Mario Schapira, Amit Sahai, and Srinath Setty. Privacy-preserving ridge regression on hundreds of millions of records. In *IEEE Symposium on Security and Privacy (SP)*, pages 334–348, 2013.

[283] Monique Ogburn, Claude Turner, and Pushkar Dahal. Homomorphic encryption. *Procedia Computer Science*, 20:502–509, 2013.

[284] Claudio Orlandi, Peter Scholl, and Sophia Yakoubov. The rise of paillier: homomorphic secret sharing and public-key silent ot. In *Advances in Cryptology–EUROCRYPT 2021: 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17–21, 2021, Proceedings, Part I 40*, pages 678–708. Springer, 2021.

[285] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' Track at the RSA Conference*, pages 1–20. Springer, 2006.

[286] Elisabeth Oswald, Stefan Mangard, Norbert Pramstaller, and Vincent Rijmen. A practical second-order dpa attack on aes. *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, pages 192–205, 2006.

[287] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, pages 506–519, 2017.

[288] Rakesh Patel and Kumar Ramesh. Physically unclonable functions (pufs) for chiplet authentication. *IEEE Transactions on Secure Hardware*, 34:23–40, 2023.

[289] Vivek Patil and Amit Bhardwaj. Chiplet-based architectures in next-generation high-performance computing. *IEEE Transactions on Computers*, 71(8):2234–2247, 2022.

[290] Arpita Patra and Divya Ravi. On the exact round complexity of secure three-party computation. In *Advances in Cryptology – CRYPTO 2019*, pages 425–458. Springer, 2019.

[291] Arpita Patra and Ajith Suresh. Blaze: Blazing fast privacy-preserving machine learning. Cryptology ePrint Archive, Paper 2020/042, 2020. `https://eprint.iacr.org/2020/042`.

[292] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 2 edition, 1998.

[293] Daniel Pawlowski and Richard Thompson. Power-efficient computing with chiplets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):3123–3135, 2018.

[294] Mark Pawlowski and Xiaolong Liu. Zero-trust architectures for multi-chip security. *IEEE Transactions on Secure Systems*, 30(4):899–913, 2022.

[295] Eric Peeters, François-Xavier Standaert, and Jean-Jacques Quisquater. Power and electromagnetic analysis: Improved model, consequences and comparisons. *Integration*, 40(1):52–60, 2007.

[296] Chris Peikert. A decade of lattice cryptography. *Foundations and Trends in Theoretical Computer Science*, 10(4):283–424, 2016.

[297] Colin Percival. Cache missing for fun and profit, 2005.

[298] Yuval Peres. Iterating von neumann's procedure for extracting random bits. *The Annals of Statistics*, pages 590–597, 1992.

[299] Benny Pinkas, Thomas Schneider, Nigel P Smart, and Stephen C Williams. Secure two-party computation is practical. In *Intrl. Conf. on the theory and application of cryptology and information security*, pages 250–267. Springer, 2009.

[300] Jesse Pool, Ian Sin Kwok Wong, and David Lie. Relaxed determinism: Making redundant execution on multiprocessors practical. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '07)*, 2007.

[301] Michael O. Rabin. How to exchange secrets by oblivious transfer. Technical Report TR-81, Aiken Computation Laboratory, Harvard University, 1981. Unpublished manuscript.

[302] Farhan Rahman and Di Wang. Secure boot and firmware verification in chiplet-based systems. *IEEE Transactions on Embedded Security*, 15:321–336, 2020.

[303] Md Tauhidur Rahman and [Co-authors]. Security implications of chiplet-based architectures. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, pages 1001–1015, 2022.

[304] Shoaib Rahman and Amit Kumar. Secure execution models for chiplet-based computing. *ACM Transactions on Secure Hardware*, 14(3):1–19, 2021.

[305] Jeyavijayan Rajendran, Hao Zhang, Cheng Zhang, Garrett S Rose, Ruben Pino, and Ramesh Karri. Security analysis of logic obfuscation. In *Proceedings of the 50th Annual Design Automation Conference*, pages 1–9, 2013.

[306] Jeffrey Ramey and John Clark. Chiplet-based processors: A paradigm shift in computing. *IEEE Spectrum*, 58(4):44–51, 2021.

[307] Samuel Ramjam, Bin Tan, and Philip Johnson. Fault injection and detection for deep neural networks. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2020.

[308] Anusha Ranganathan, Thomas Schneider, and Oleksandr Tkachenko. Hardware Oblivious Transfer accelerator for Privacy-Preserving Protocols. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1484–1489. IEEE, 2021.

[309] Rangharajan V Rao, Hiroshi Kodama, Tatsuya Okuda, Young-Joo Woo, and Hiroshi Satoh. Heterogeneous integration for performance and reliability: From fundamentals to applications. *Microelectronics Reliability*, 125:114327, 2021.

[310] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow2: Practical 2-party secure inference. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 325–342, 2020.

[311] Pratik Ravi and Joshua Shen. Adaptive security mechanisms for dynamic reconfiguration of chiplets. *IEEE Transactions on Secure Architectures*, 40(3):210–225, 2022.

[312] Rehan Raza and Shahid Ahmed. Chiplet-based security: Fault injection and countermeasures. *IEEE Transactions on Circuits and Systems II*, 68(9):2143–2154, 2021.

[313] Kaveh Razavi, Erik Bosman, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. *USENIX Security Symposium*, pages 1–18, 2016.

[314] Tao Ren and Fang He. Trusted execution frameworks for chiplet-based processors. *Journal of Secure Computing*, 28:15–29, 2021.

[315] Steve Rhoads. Plasma - most mips i(tm) opcodes. `https://opencores.org/projects/plasma`, 2001. Last accessed: Jan. 30, 2023.

[316] M. Sadegh Riazi, Siam U Hussain, and Farinaz Koushanfar. Chiplet-based secure processing: Challenges and opportunities. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 123–130. IEEE, 2021.

[317] M Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin Lauter, and Farinaz Koushanfar. {XONN}:{XNOR-based} oblivious deep neural network inference. In *28th USENIX Security Symp. (USENIX Security 19)*, pages 1501–1518, 2019.

[318] M Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *Proc. of the 2018 on Asia Conf. on computer and Comm. security*, pages 707–721, 2018.

[319] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. Privpy: General and scalable privacy-preserving computation with secure multi-party computation. In *USENIX Security*, pages 585–602, 2019.

[320] Mohammad Sadegh Riazi, Amit Sahai, and Samee Zahur. Ram-sc: Faster garbled circuits for ram programs. In *International Conference on Theory and Practice of Public Key Cryptography*, pages 94–124. Springer, 2019.

[321] Riscure. Side-channel and fault injection attacks: Real-world threats and countermeasures. Technical report, Riscure Security Research, 2021.

[322] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[323] M. Robert, P. Maurine, and A. Razafindraibe. Design and analysis of dual-rail circuits for security applications. *IEEE Transactions on Computers*, 54(4):438–451, 2005.

[324] Daniel Rosenberg and Jeremy Lau. Modular computing with chiplets: A new design paradigm. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 69(3):1012–1023, 2022.

[325] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[326] Masoud Rostami, Farinaz Koushanfar, and Ramesh Karri. Hardware security: Threat models and metrics. *Proceedings of the IEEE*, 102(8):1283–1295, 2013.

[327] Bita Darvish Rouhani, Siam Umar Hussain, Kristin Lauter, and Farinaz Koushanfar. Redcrypt: Real-time privacy-preserving deep learning inference in clouds using fpgas. *ACM Trans. on Reconfigurable Technology and Systems (TRETS)*, 11(3):1–21, 2018.

[328] Bita Darvish Rouhani, M Sadegh Riazi, and Farinaz Koushanfar. Deepsecure: Scalable provably-secure deep learning. In *Proc. of the 55th annual design automation Conf.*, pages 1–6, 2018.

[329] Jeyavijayan Roy, Farinaz Koushanfar, and Igor L Markov. Ending piracy of integrated circuits. *Computer*, 41(10):30–38, 2008.

[330] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[331] Ahmed Salem, Michael Backes, and Yang Zhang. Don't trigger me! a triggerless backdoor attack against deep neural networks. *arXiv preprint arXiv:2010.03282*, 2020.

[332] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO-54: 54th Annual IEEE/ACM Intrl. Symp. on Microarchitecture*, pages 238–252, 2021.

[333] Tim Sander, Marcel Keller, and Alexander Taylor. Dash: Efficient privacy-preserving machine learning using oblivious computation. *ACM Transactions on Privacy and Security*, 2023.

[334] Falk Schellenberg, Dennis RE Gnad, Amir Moradi, and Mehdi B Tahoori. An inside job: Remote power analysis attacks on fpgas. In *2018 Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, pages 1111–1116. IEEE, 2018.

[335] Werner Schindler, Karl Lemke, and Christof Paar. A stochastic model for differential side channel cryptanalysis. *IACR Cryptology ePrint Archive*, 2005:54, 2005.

[336] Werner Schindler, Kerstin Lemke, and Christof Paar. A stochastic model for differential side channel cryptanalysis. In *Intrl. WKSP on Cryptographic Hardware and Embedded Systems*, pages 30–46. Springer, 2005.

[337] J-M Schmidt, Christof Paar, Johannes Heyszl, and Georg Sigl. Optical side-channel attacks. In *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 77–84. IEEE, 2009.

[338] Johannes Schmidt, Michael Hutter, Werner Schindler, and Thomas Plos. Optical fault induction attacks on secure integrated circuits. In *12th International Workshop on Cryptographic Hardware and Embedded Systems – CHES 2007*, pages 2–12. Springer, 2007.

[339] Julian Schmidt, Thomas Plos, Christoph Helfmeier, and Markus Bochum. Optical side-channel attacks. *Springer Journal on Hardware Security and Trust*, 4(2):92–103, 2011.

[340] Thomas Schneider and Amir Moradi. Dpa, leakage estimation, and leakage resilience. *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 57–69, 2004.

[341] Thomas Schneider and Amir Moradi. A leakage model for standard side-channel attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2015(2):238–256, 2015.

[342] Thomas Schneider, Andy Rupp, and Emmanuel Prouff. Efficiently masking aes for hardware and software implementations. In *Cryptographic Hardware and Embedded Systems - CHES 2016*, pages 119–140. Springer, 2016.

[343] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 587–600, 2018.

[344] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24. Springer, 2017.

[345] Paul Sedcole, Peter YK Cheung, Wayne Luk, and Thomas Becker. Within-die delay variability in 90 nm fpgas and beyond. *Field-Programmable Custom Computing Machines*, 2006:129–138, 2006.

[346] Benjamin Seifert and Ali Khosravi. Efficient side-channel attack countermeasures for chiplet-based architectures. *IEEE Transactions on Information Forensics and Security*, 17:1205–1219, 2022.

[347] Paul Selmke, Martijn R. B. Schut, Marc Fyrbiak, and Christof Paar. Electromagnetic fault attacks: Concrete results. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2016 Workshop on*, pages 3–13. IEEE, 2016.

[348] Adeel Shahid and Rehman Khan. Boot-time security for heterogeneous chiplet architectures. *Journal of Trusted Computing*, 13:92–105, 2021.

[349] John Shalf. The future of computing beyond moore's law. *Philosophical Transactions of the Royal Society A*, 378(2166):20190061, 2020.

[350] Abhi Shelat and Chih-Hao Shen. Two-output secure computation with malicious adversaries. In *Advances in Cryptology–EUROCRYPT 2011: 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings 30*, pages 386–405. Springer, 2011.

[351] Hanif D Sherali and Cihan H Tuncbilek. A squared-euclidean distance location-allocation problem. *Naval Research Logistics (NRL)*, 39(4):447–469, 1992.

[352] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *IEEE Symposium on Security and Privacy (SP)*, pages 3–18, 2017.

[353] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. Prime+Probe 1, JavaScript 0: Overcoming browser-based Side-Channel defenses. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2863–2880, 2021.

[354] Sergei Skorobogatov. *Semi-Invasive Attacks: A New Approach to Hardware Security Analysis*, volume 21 of *Springer Series in Advances in Information Security*. Springer, 2010.

[355] Sergei Skorobogatov. Semi-invasive attacks—a new approach to hardware security analysis. *Technical Report UCAM-CL-TR-630, University of Cambridge*, 2010.

[356] Sergei P. Skorobogatov. Low temperature data remanence in static ram. *University of Cambridge, Computer Laboratory*, 2002.

[357] Sergei P. Skorobogatov. Optical fault injection attacks. *Cryptographic Hardware and Embedded Systems (CHES)*, pages 2–12, 2010.

[358] Sergei P. Skorobogatov and Ross J. Anderson. Semi-invasive attacks – a new approach to hardware security analysis. In *Cryptographic Hardware and Embedded Systems – CHES 2005*, pages 1–23. Springer, 2005.

[359] Liwei Song, Xinwei Yu, Hsuan-Tung Peng, and Karthik Narasimhan. Universal adversarial attacks with natural triggers for text classification. *arXiv preprint arXiv:2005.00174*, 2020.

[360] Paul Song. *SPARC Architecture, Assembly Language Programming, and C*. Prentice Hall, 1999.

[361] E Songhori, H Siam, and S Riazi. Tinygarble framework. `[Online]https://github.com/esonghori/TinyGarble` [Accessed Jan.30, 2023], 2019.

[362] Ebrahim M Songhori, Siam U Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. Tinygarble: Highly compressed and scalable sequential garbled circuits. In *2015 IEEE Symp. on Security and Privacy*, pages 411–428. IEEE, 2015.

[363] Ebrahim M Songhori, M Sadegh Riazi, Siam U Hussain, Ahmad-Reza Sadeghi, and Farinaz Koushanfar. Arm2gc: Succinct garbled processor for secure computation. In *Proc. of the 56th Annual Design Automation Conf. 2019*, pages 1–6, 2019.

[364] Ebrahim M Songhori, Thomas Schneider, Shaza Zeitouni, Ahmad-Reza Sadeghi, Ghada Dessouky, and Farinaz Koushanfar. Garbled-cpu: A mips processor for secure computation in hardware. In *2016 53nd ACM/EDAC/IEEE Design Automation Conf. (DAC)*, pages 1–6. IEEE, 2016.

[365] Wenting Zheng Srinivasan, PMRL Akshayaram, and Popa Raluca Ada. Delphi: A cryptographic inference service for neural networks. In *Proc. 29th USENIX Secur. Symp*, pages 2505–2522, 2019.

[366] Richard M. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection (GCC)*. Free Software Foundation, Boston, MA, USA, 1998.

[367] François-Xavier Standaert. How (not) to use welch's t-test in side-channel security evaluations. In *Intrl. Conf. on smart card research and advanced applications*, pages 65–79. Springer, 2018.

[368] François-Xavier Standaert and Cédric Archambeau. Using subspace-based template attacks to compare and combine power and electromagnetic information leakages. In *Intrl. WKSP on Cryptographic Hardware and Embedded Systems*, pages 411–425. Springer, 2008.

[369] François-Xavier Standaert, Tal G Malkin, and Moti Yung. A unified framework for the analysis of side-channel key recovery attacks. In *Annual Intrl. Conf. on the Theory and Applications of Cryptographic Techniques*, pages 443–461. Springer, 2009.

[370] François-Xavier Standaert, Olivier Pereira, Yu Yu, Jean-Jacques Quisquater, Moti Yung, and Elisabeth Oswald. Leakage resilient cryp-

tography in practice. In *Towards Hardware-Intrinsic Security*, pages 99–134. Springer, 2010.

[371] Hugo Steinhaus. Sur la division des corps matériels en parties. *Bulletin de l'Académie Polonaise des Sciences*, 4:801–804, 1956.

[372] Franz-Josef Streit, Florian Fritz, Andreas Becher, Stefan Wildermann, Stefan Werner, Martin Schmidt-Korth, Michael Pschyklenk, and Jürgen Teich. Secure boot from non-volatile memory for programmable soc architectures. `https://arxiv.org/abs/2004.09453`, 2020. Accessed: 2025-04-17.

[373] Pramod Subramanyan, Shubhajit Ray, and Sharad Malik. Evaluating the security of logic encryption algorithms. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 137–143. IEEE, 2015.

[374] Jingjing Sun, Xiaoxiao Xu, and Minghao Li. Advancing interconnect technology for chiplet architectures. *Journal of Microelectronics*, 102:56–69, 2022.

[375] Xiaoqiang Sun, F Richard Yu, Peng Zhang, Zhiwei Sun, Weixin Xie, and Xiang Peng. A survey on zero-knowledge proof in blockchain. *IEEE network*, 35(4):198–205, 2021.

[376] Yung-Chang Sun, Tsung-Yu Tsai, and Yi-Ching Wu. Heterogeneous integration technology for advanced computing. *Micromachines*, 9(11):562, 2018.

[377] Shahin Tajik, Pascal Sasdrich, and Amir Moradi. Artificial intelligence security: Fault injection against neural networks. In *IEEE International Conference on Artificial Intelligence Security (AISec)*, 2022.

[378] Shahin Tajik and Jean-Pierre Seifert. Power glitch attacks on fpga-based systems: Survey and new results on a secure sram-based puf. In *International Conference on Smart Card Research and Advanced Applications (CARDIS)*, pages 115–130, 2017.

[379] Sijun Tan, Brian Knott, Yuan Tian, and David J Wu. Cryptgpu: Fast privacy-preserving machine learning on the gpu. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1021–1038. IEEE, 2021.

[380] Zhi Tang, Qian Wang, Peng Liu, and Xiapu Luo. Power side channels in usb chargers. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1105–1121. IEEE, 2017.

[381] Teledyne LeCroy. *WavePro 254HD High Definition Oscilloscope User Guide*. Teledyne LeCroy, 2022. User Manual.

[382] Lu Tian, Bargav Jayaraman, Quanquan Gu, and David Evans. Aggregating private sparse learning models using multi-party computation. In *NIPS Workshop on Private Multi-Party Machine Learning*, 2016.

[383] Meng Tian and Qian Zhao. Security challenges in chiplet-based systems. *IEEE Embedded Systems Letters*, 11(2):56–59, 2019.

[384] Niek Timmers, Alessandro Barenghi, Francesco Regazzoni, Elena Dubrova, Cristiano Giuffrida, and Herbert Bos. Controlling pc on arm using fault injection. In *Proceedings of the 2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 25–35. IEEE, 2016.

[385] Kris Tiri and Ingrid Verbauwhede. A dynamic and differential cmos logic with signal independent power consumption to withstand differential power analysis on smart cards. In *Proceedings of the 28th European Solid-State Circuits Conference (ESSCIRC)*, pages 403–406. IEEE, 2005.

[386] M. Caner Tol, Saad Islam, Andrew J. Adiletta, Berk Sunar, and Ziming Zhang. Don't knock! rowhammer at the backdoor of dnn models. *arXiv preprint arXiv:2110.07683*, 2021.

[387] M. Toorani and A. A. Beheshti. An elliptic curve-based signcryption scheme with forward secrecy. *arXiv:1005.1856*, 2010.

[388] Randy Torrance and Dick James. The state-of-the-art in ic reverse engineering. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 363–381. Springer, 2009.

[389] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction APIs. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 601–618. USENIX Association, 2016.

[390] Michael Tunstall, Debdeep Mukhopadhyay, and Luca Breveglieri. Differential fault analysis of aes and fault countermeasures. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 112–124, 2011.

[391] Thinh Van and Xue Zhang. Secure enclaves for chiplet-based ai accelerators. *IEEE Transactions on Secure Systems*, 26(6):1011–1024, 2021.

[392] Wim Van Eck. Compromising emanations: Eavesdropping risks of computer displays. *Computers & Security*, 4(4):269–286, 1985.

[393] Kush R. Varshney, Julia Rogers, Peder Olsen, Roman M. Garnett, Jeffrey W. Scovell, and Katy Börner. Privacy-preserving data sharing in public health: Methods for secure multiparty computation. *Annual Review of Public Health*, 38:439–459, 2017.

[394] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 5998–6008, 2017.

[395] Arunkumar Vijayakumar, Vinay C. Patil, Daniel E. Holcomb, Christof Paar, and Sandip Kundu. Physical design obfuscation of hardware: A comprehensive investigation of device- and logic-level techniques. *arXiv:1910.00981*, 2019.

[396] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. Falcon: Honest-majority maliciously secure framework for private deep learning. *arXiv preprint arXiv:2004.02229*, 2020.

[397] Alex Wagner, Yehuda Lindell, Orr Dunkelman, and Ran Canetti. Scapi: The secure computation api. In *International Conference on Security and Cryptography for Networks (SCN)*, pages 515–535. Springer, 2018.

[398] Bolun Wang, Yuanshun Yao, Shawn Shan, Huiying Li, Bimal Viswanath, Haitao Zheng, and Ben Y Zhao. Neural cleanse: Identifying and mitigating backdoor attacks in neural networks. In *2019*

IEEE Symposium on Security and Privacy (SP), pages 707–723. IEEE, 2019.

[399] J Wang and P Patel. Universal chiplet interconnect express (ucie): A standard for chiplet ecosystems. *IEEE Micro*, 43(1):14–24, 2023.

[400] Ke Wang, Yang Yang, Lin Lin, and Wei Wang. Exclusive cache performance optimization for emerging workloads. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, pages 1–8. IEEE, 2019.

[401] Xiao Wang, S. Dov Gordon, Allen McIntosh, and Jonathan Katz. Secure computation of mips machine code. In *European Symp. on Research in Computer Security*, pages 99–117. Springer, 2016.

[402] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 21–37, 2017.

[403] Alexander Warnecke, Julian Speith, Jan-Niklas Möller, Konrad Rieck, and Christof Paar. Evil from within: Machine learning backdoors through hardware trojans. *arXiv preprint arXiv:2304.08411*, 2023.

[404] Samuel Weiser, Mario Werner, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Sgxjail: Defeating enclave malware via confinement. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 151–162. ACM, 2018.

[405] Carolyn Whitnall and Elisabeth Oswald. Robust profiling for DPA-style attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 3–21. Springer, 2015.

[406] Markus G. J. Witteman, Jasper van Woudenberg, and Erik Bakker. Secure programming in the presence of fault attacks. In *Proceedings of the 2008 International Conference on Embedded Systems and Applications (ESA)*, pages 1–7, 2008.

[407] Gary Workman. *Physical defects and their modeling*. Springer, 1998.

[408] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. Eliminating timing side-channel leaks using program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 15–26, 2018.

[409] Inc. Xilinx. v2021.1. `[Online]https://www.xilinx.com/products/design-tools/vivado.html` [Accessed: Oct.16, 2024], 2021.

[410] Inc. Xilinx. v2022.1. `[Online]https://docs.xilinx.com/v/u/en-US/ug1416-vitis-documentation.html` [Accessed: Oct.16, 2024], 2022.

[411] Inc. Xilinx. *Vivado Design Suite User Guide*, 2023.

[412] Guowen Xu, Xingshuo Han, Tianwei Zhang, Shengmin Xu, Jianting Ning, Xinyi Huang, Hongwei Li, and Robert H Deng. Simc 2.0: Improved secure ml inference against malicious clients. *IEEE Transactions on Dependable and Secure Computing*, 2023.

[413] Ling Xu and Peng Zhao. Secure monitoring and attestation in heterogeneous chiplets. *IEEE Transactions on Secure Computing*, 32(1):99–117, 2023.

[414] Xiaojun Xu, Qi Wang, Huichen Li, Nikita Borisov, Carl A Gunter, and Bo Li. Detecting ai trojans using meta neural analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 103–120. IEEE, 2021.

[415] Xuan Xu, Jeyavijayan Rajendran, Hao Zhang, and Ramesh Karri. A novel bypass attack and bdd-based tradeoff analysis against all known logic locking attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 189–210. Springer, 2017.

[416] Mengjia Yan and Moinuddin K. Qureshi. New attacks and defense for encrypted-address cache. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 360–373. ACM, 2019.

[417] Andrew C Yao. Protocols for secure computations. *23rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 160–164, 1982.

[418] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Foundations of Computer Science (FOCS)*, pages 162–167, 1986.

[419] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symp. on Foundations of Computer Science (sfcs 1986)*, pages 162–167. IEEE, 1986.

[420] Fan Yao, Adnan Siraj Rakin, and Deliang Fan. Deephammer: Depleting the intelligence of deep neural networks through targeted chain of bit flips. *arXiv preprint arXiv:2003.13746*, 2020.

[421] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 719–732, 2014.

[422] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium*, pages 719–732, 2014.

[423] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. *Proceedings of the 23rd USENIX Security Symposium*, pages 719–732, 2014.

[424] Bei Yu, Yibo He, Mei Li, Jun Li, Lei Wang, Renjie Song, and Qiang Xu. Hardware intellectual property protection: Techniques and countermeasures. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 25(2):1–27, 2020.

[425] Jing Yuan, Wei Zhang, and Shankar Kumar. Secure design methodologies for heterogeneous chiplet integration. *Journal of Hardware and Systems Security*, 5(2):89–107, 2021.

[426] S Zahur, G Kerneis, and G Necula. Obliv-C secure computation compiler. `[Online]https://github.com/samee/obliv-c` [Accessed Feb.2, 2023], 2018.

[427] Samee Zahur and David Evans. Obliv-C: A language for extensible data-oblivious computation. *Cryptology ePrint Archive*, 2015.

[428] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 220–250. Springer, 2015.

[429] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole: Reducing data transfer in garbled circuits using half gates. In *Advances in Cryptology – EUROCRYPT 2015*, pages 220–250. Springer, 2015.

[430] Chengliang Zhang, Suyi Li, Junzhe Xia, Wei Wang, Feng Yan, and Yang Liu. Batchcrypt: Efficient homomorphic encryption for cross-silo federated learning. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC 2020)*, 2020.

[431] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*, 2016.

[432] Jian Zhang, Xiaofei Wang, and Fengjun Zhang. Privacy-preserving machine learning: Applications and challenges. *IEEE Security & Privacy*, 19(2):14–23, 2021.

[433] Wei Zhang and Hong Li. Chiplet security vulnerabilities and mitigation strategies. *IEEE Design & Test*, 40(2):112–127, 2023.

[434] Yao Zhang, Mingyu Gao, and Jason Cong. Hardware-accelerated secure function evaluation with partial garbled circuits. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, page partial pagination—please confirm from proceedings. IEEE, 2011.

[435] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *ACM Conference on Computer and Communications Security (CCS)*, pages 305–316, 2012.

[436] Yu Zhang, Farinaz Koushanfar, and Siddharth Garg. Em attacks on machine learning algorithms: A systematic study of modern neural networks. *IEEE Transactions on Information Forensics and Security*, 15:2542–2556, 2020.

[437] Haochen Zhao and Xiao Wang. Silent Oblivious Transfer and Efficient Private Set Intersection. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 785–802. IEEE, 2020.

[438] Li Zhao, Ravi Iyer, Srihari Makineni, Don Newell, and Liqun Cheng. Ncid: a non-inclusive cache, inclusive directory architecture for flexible and efficient cache hierarchies. In *Proceedings of the 7th ACM international conference on Computing frontiers*, pages 121–130, 2010.

[439] Mark Zhao and G Edward Suh. Fpga-based remote power side-channel attacks. In *2018 IEEE Symp. on Security and Privacy (SP)*, pages 229–244. IEEE, 2018.

[440] Xuanqiang Zhao, Benchi Zhao, Zihan Xia, and Xin Wang. Information recoverability of noisy quantum states. *Quantum*, 7:978, April 2023.

[441] Zirui Neil Zhao, Adam Morrison, Christopher W Fletcher, and Josep Torrellas. Last-level cache side-channel attacks are feasible in the modern public cloud. *arXiv preprint arXiv:2405.12469*, 2024.

[442] Zhaoji Zhou, Hongbo Li, and Weiqiang Wang. Design and analysis of side-channel attack based on crosstalk. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(12):2846–2857, 2019.

[443] Zhiqiang Zhou, Xin Wang, and Chenchen Wu. Magnetic sensor based side-channel attack on cryptographic hardware. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 984–989. IEEE, 2019.

[444] Tianqi Zhuang, Meng Shen, Aniello Castiglione, and Gang Wang. Security vulnerabilities and risks in the ieee p1735 standard for protecting electronic-design intellectual property. In *Proceedings of the 2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 151–161, 2019.

.1 A detailed report of leaky IF conditions

Table 1 contains details of leaky IF conditions in each function of TinyGarble [361], EMP-toolkit [246], Obliv-C [427], and ABY [95].

Table 1: A detailed report of leaky IF conditions (IF) of every function call in JustGarble [34], TinyGarble [361] with half-gate and free-XOR optimization, EMP-toolkit [246], Obliv-C [427], and ABY [95].

Framework	Function	IF	Framework	Function	IF
TinyGarble (half-gate) [361]	GarbledLowMem	0	JustGarble [187]	createNewWire	0
	GarbledGate	2		TRUNCATE	0
	ParseInitInputStr	0		TRUNC_COPY	0
	RemoveGarbledCircuit	0		getNextId	0
	HalfGarbleGateKnownValue	0		getFreshId	0
	NumOfNonXor	0		getNextWire	0
	HalfGarbleGate	2		createEmptyGarbledCircuit	0
	InvertSecretValue	0		removeGarbledCircuit	0
	XorSecret	0		startBuilding	0
	OutputBN2StrLowMem	0		finishBuilding	2
	RandomBlock	0		extractLabels	0
	Total	4		garbleCircuit	8
TinyGarble (free-XOR) [361]	GarbledLowMem	2		blockEqual	0
	GarbledGate	5		mapOutputs	0
	ParseInitInputStr	0		createInputLabels	0
	RemoveGarbledCircuit	0		randomBlock	0
	NumOfNonXor	0		xorBlocks	0
	XorSecret	0		findGatesWithMatchingInputs	1
	OutputBN2StrLowMem	0		Total	11
	RandomBlock	0	EMP-toolkit [246]	HalfGateGen	0
	Total	7		parse_party_and_port	0
Obliv-C [427]	yaoGenerateGate	3		NetIO	0
	yaoGenrRevealOblivBits	0		Total	0
	yaoGenrFeedOblivInputs	1	ABY [95]	YaoSharingInit	0
	yaoKeyNewPair	0		BooleanCircuit	0
	yaoSetBitAnd	0		init_aes_key	0
	yaoSetBitOr	0		ceil_divide	0
	yaoSetBitXor	0		clean_aes_key	0
	yaoFlipBit	0		EncryptWire	0
	yaoSetHashMask	0		EncryptWireGRR3	0
	yaoSetHalfMask	0		PrintKey	0
	yaoSetHalfMask2	0		PrintPerformanceStatistics	0
	yaoKeyDouble	0		XOR_DOUBLE_B	0
	Total	4		Total	0