SECURE OFFLOADING AND PARTITIONING
FOR HETEROGENEOUS CPU-FPGA SYSTEMS

By

MENGMEI YE

A dissertation submitted to the
School of Graduate Studies
Rutgers, The State University of New Jersey
In partial fulfillment of the requirements
For the degree of
Doctor of Philosophy
Graduate Program in Electrical and Computer Engineering

Written under the direction of

Sheng Wei

And approved by

_________________________________________________________________
_________________________________________________________________
_________________________________________________________________
_________________________________________________________________

New Brunswick, New Jersey
October 2021
ABSTRACT OF THE DISSERTATION

Secure Offloading and Partitioning
for Heterogeneous CPU-FPGA Systems

by MENGMEI YE

Dissertation Director: Sheng Wei

Heterogeneous CPU-FPGA systems have been shown to achieve significant performance gains in domain-specific computing. However, contrary to the huge efforts invested on the performance acceleration, the community has not yet investigated the security consequences due to incorporating FPGA into the traditional CPU-based architecture. In fact, the interplay between CPU and FPGA in such a heterogeneous system may introduce brand new attack surfaces if not well controlled. We develop a hardware isolation-based secure architecture, namely HISAtm, to mitigate the identified new threats. HISAtm extends the CPU-based trusted execution environment (TEE) to the heterogeneous FPGA components to enhance the security of the CPU-FPGA system. To securely offload the application to the FPGA part of HISAtm, we develop ApproVer to verify the security of the FPGA IP cores by applying an approximate computing-based verification mechanism. To help developers deploy applications on HISAtm, we develop TZSLICER to securely partition the software by meeting the security requirements and maintaining the original functionality. Furthermore, we explore another dimension of heterogeneity by extending HISAtm to a local-cloud system to support secure and privacy-preserving computations offloaded from the local user to the remote service provider.
ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor and dissertation director Dr. Sheng Wei for his invaluable supervision and continuous support during my senior-year undergraduate and Ph.D. study. I am very grateful that he provided much guidance to improve my research, presentation, and writing skills. Without his mentorship, support, and encouragement, I would not have completed my Ph.D. study, learned and worked on so many interesting research projects, and enjoyed my Ph.D. years so much.

I would also like to extend my deepest gratitude to my dissertation committee members, Dr. Yingying Chen, Dr. Saman Aliari Zonouz, and Dr. Shiqing Ma. I sincerely appreciate their excellent help and support. They provided many insightful research ideas and directions to improve my research works. Without their guidance, I would not have finished this dissertation.

Furthermore, many thanks to Dr. Rosario Cammarota (Intel Labs), Dr. Myra B. Cohen (Iowa State University), Dr. Athina Petropulu (Rutgers University), Dr. Byrav Ramamurthy (University of Nebraska–Lincoln), Dr. Witawas Srisa-an (University of Nebraska–Lincoln), Dr. Viswanathan Swaminathan (Adobe Research), Dr. Qiben Yan (Michigan State University), Dr. Bo Yuan (Rutgers University), Dr. He Zhu (Rutgers University), etc. I sincerely appreciate them offering me many opportunities and resources to attend conferences/workshops and work on many interesting research projects.

Also, I would like to thank my lab mates and coworkers, Deepak Nadig Anantha, Xianglong Feng, Akshay Ganal, Nianhang Hu, Nan Jiang, Huy Phan, Wenjie Qiu, Mehrdad Zaker Shahrak, Jonathan Sherman, Zhongze Tang, Ke Xia, Yi Xie, Zichen Zhu, etc., for the wonderful time spending on our projects and in the labs.

Some chapters of the dissertation are based on the publications listed in Acknowledgment of Previous Publications. I am very grateful for all the efforts from our coauthors and collaborators. They offered me much help and guidance on the experiments and paper
writing. A part of chapter 1 and chapter 2 is based on P1-P5 and P8. A part of chapter 3 is based on P4 and P9. A part of chapter 4 is based on P6 and P7. A part of chapter 5 is based on P10.

Special thanks to my family, especially to my parents and grandparents, for their love, support, and guidance. They have always been my role models. I love them so much.
# TABLE OF CONTENTS

Abstract .......................................................... ii

Acknowledgments .................................................. iii

List of Tables ...................................................... ix

List of Figures ...................................................... x

Chapter 1: Introduction .......................................... 1


2.1 CPU-FPGA Threat Models .................................. 7
  2.1.1 Overview of CPU-FPGA Threats ....................... 7
  2.1.2 Case Study ............................................... 8

2.2 Related Work .................................................. 10
  2.2.1 Intra-CPU and Intra-FPGA Security .................. 10
  2.2.2 Inter-CPU/FPGA Security ............................. 10
  2.2.3 Hardware Isolation Primitives ....................... 11

2.3 HISA Framework ............................................. 11
  2.3.1 Secure Data Flow ..................................... 13
Chapter 3: ApproVer: Runtime Hardware Security Verification Using Approximate Computing

3.1 Related Work
3.1.1 Security Verification Methods
3.1.2 Approximate Computing
3.2 Threat Model
3.3 ApproVer System Framework
3.4 Case Study on Video Motion Detection CPU-FPGA System
3.5 Experimental Evaluation
3.5.1 Experimental Setup
3.5.2 Effectiveness Evaluation
3.5.3 Efficiency Evaluation

Chapter 4: TZSLICER: Security-Aware Dynamic Program Slicing for Hardware Isolation
4.1 Threat Models ................................................................. 38
  4.1.1 Information Leakage Attacks ........................................ 39
  4.1.2 Data Falsifying Attacks ............................................. 39
4.2 Motivating Example ....................................................... 40
4.3 Related Work: Program Slicing .......................................... 41
4.4 TZSLICER System Framework ......................................... 41
  4.4.1 Taint Analyzer ....................................................... 42
  4.4.2 Program Slicer ...................................................... 43
  4.4.3 Slice Optimizer ...................................................... 44
4.5 Experimental Evaluation ............................................... 49
  4.5.1 Experimental Setup ................................................ 49
  4.5.2 Security Analysis ................................................... 49
  4.5.3 Quantitative Security Evaluation: TCB Size ...................... 50
  4.5.4 Performance Evaluation: World Switches ......................... 52
  4.5.5 Discussion: Trade-off Between Security and Performance ........ 52
4.6 EVOISOLATOR: Evolving Program Slices for TZSLICER .......... 53

Chapter 5: HybridTEE With Protective Perturbation-based Privacy Protection 58
  5.1 Background ............................................................. 62
    5.1.1 Image Recognition System ....................................... 62
    5.1.2 Privacy Issues in Image Recognition .......................... 63
  5.2 Related Work .......................................................... 64
    5.2.1 Privacy Preserving Image Recognition ......................... 64
5.2.2 Adversarial Attacks .......................... 65

5.3 Protective Perturbation .......................... 66
  5.3.1 System Overview .......................... 66
  5.3.2 Protective Perturbation Generation ............... 67
  5.3.3 Training the Perturbation Generator ............. 69

5.4 Experimental Evaluation ......................... 71
  5.4.1 Experimental Setup ........................ 71
  5.4.2 Effectiveness Evaluation ..................... 72
  5.4.3 Efficiency Evaluation ....................... 76

5.5 Limitation and Discussion ....................... 78

Chapter 6: Conclusion .................................. 79

Acknowledgment of Previous Publications .............. 80

References .............................................. 82
LIST OF TABLES

2.1 Reference IP cores, threat models, and security policy designs for security analysis. ................................. 21

2.2 Timing evaluation (μs) of output verifications. ................................................................. 23

3.1 Accuracy for NoTrojan IP detection by SAC_36 and TAC_20. ................................. 34

3.2 Precision and recall for STrojan IP detection by SAC_36 and TAC_20. .......... 34

3.3 Precision and recall for TTrojan IP detection by SAC_36 and TAC_20. ........ 34

4.1 Test programs adopted to evaluate TZSLICER. ......................................................... 49

4.2 Resource usage evaluation results represented by the number of lines of code for TZ-M, TZ-B, and TZ-L. The percentage value in parenthesis represent the relative savings from the baseline, as described in Equation 4.1. .......... 51

4.3 Resource usage evaluation results for TZ-L+ with different parameters. ...... 51

4.4 Evaluation of context switching overhead. The percentage value in parenthesis represent the relative savings from the baseline. ......................... 53

5.1 Target-model accuracy of the protective perturbation generator with and without auxiliary models. ................................................................. 74

5.2 Timing evaluation (ms) of generating one protected image from the generators, which execute in a CPU-only machine and a GPU-accelerated machine. ............................................................. 77
LIST OF FIGURES

2.1 Scope of the HISA work. .............................................. 8
2.2 Hardware system setup for the motion detection system using a Xilinx SoC with both CPU and FPGA (video sample courtesy of [19]). ................. 9
2.3 (a) Normal motion detection scenario without attacks; and (b) Motion detection results under CPU to FPGA or FPGA to CPU attacks, in which the moving objects are hidden in the background instead of being detected (video sample courtesy of [19]). ................................................. 9
2.4 CPU-FPGA threat model design for the motion detection system. ............ 10
2.5 Architecture of CPU-FPGA hardware isolation. ............................. 12
2.6 Realization of CPU-FPGA hardware isolation on ARM. ...................... 13
2.7 Secure/normal worlds communication flow. .................................... 13
2.8 Passive output verification via side channel analysis. .......................... 17
2.9 Proactive output verification via motion frame insertion. ....................... 19
2.10 Demonstration of timing side channel analysis. ............................... 19
2.11 Demonstration of proactive output verification for the motion detection IP (video sample courtesy of [19]). .................................................. 20
2.12 Evaluation of HISA isolation delay on the reference IP cores with access control and output verification. .................................................. 22
2.13 Power and energy evaluation for the XADC IP. ............................... 24

3.1 Threat model considered in this chapter: The untrusted 3PIP in the FPGA sends falsified results to the CPU. ........................................ 27
3.2 ApproVer system overview. 3PIP in the FPGA conducts the actual computation. ApproVer in the CPU requires two stages to detect if the 3PIP contains Trojans, namely approximate computation and security verification.

3.3 Demonstration of spatial/temporal approximation. The spatial approximation randomly selects a subset of tasks from every input job. The temporal approximation randomly selects a subset of jobs from all the inputs jobs.

3.4 The workflow of the video motion detection system (the video example is from [19]). The system requires three stages to complete the motion detection process, namely video pipeline, video processing, and AC-based security verification.

3.5 Video motion detection system setup (the video example is from [19]). The input monitor provides the input video to the Zynq SoC for the motion detection process. Then, the output monitor displays the detected motions.

3.6 F1-score comparison for SAC with different parameters (i.e., SAC_36, SAC_72, SAC_108, and SAC_144).

3.7 Time comparison between the accurate computation and the approximate computation conducted by the SAC/TAC with different parameters.

4.1 Sample C code for the technical discussion of TZSLICER, where array $x$ contains sensitive data that requires protection.

4.2 Sliced programs by directly leveraging the HISA framework.

4.3 TZSLICER system framework.

4.4 System dependency graph for taint analysis for the Function $M1$ in Figure 4.1. The shaded nodes represent the sensitive code (line numbers) identified by the taint analysis.

4.5 TZ-M & TZ-B Slicing.

4.6 TZ-L Slicing.

4.7 TZ-L+ Slicing using loop unrolling.

4.8 TZ-L+ variable renaming under data dependency.

4.9 Timing versus resource usage (normalized) under TZ-L and different unrolling parameters ($x = 2, 3, 4$) in TZ-L+.
5.9 Average target-model accuracy results on the protected images produced by the generators with the auxiliary models. The x-axis indicates auxiliary models. The y-axis indicates the average accuracy results (\%) of all the 8 target models. .................................................. 76

5.10 Average SSIM results on the protected images produced by the generators with the auxiliary models. The x-axis indicates auxiliary models. The y-axis indicates the average SSIM results. ............................... 77
CHAPTER 1
INTRODUCTION

Reconfigurable computing architecture, which combines both the general purpose application processor (i.e., CPU) and high performance programmable logic (i.e., field-programmable gate array, or FPGA), has gained rapid advancements recently and demonstrated superior performance and power gains [1, 2, 3, 4]. In particular, the CPU-FPGA architecture has been recently deployed in two types of computing systems: (1) CPU-FPGA based system on chips (SoCs), such as Xilinx Zynq [5], leverage the low power, high performance, and high flexibility brought by FPGAs to speed up the embedded computing tasks; and (2) CPU-FPGA based cluster and cloud systems, such as the recent commercial deployments in Amazon AWS [6] and Microsoft Azure [7], leverage FPGAs to accelerate computation-intensive and domain specific tasks.

Despite its strong potential in performance acceleration and the commercial deployment, the adoption of the CPU-FPGA architecture still faces significant challenges. One obvious obstacle is the burden of developing hardware oriented computation modules on FPGAs. Even though the problem is partially solved by the emerging high level synthesis tools, the community still heavily relies on FPGA third-party intellectual property (3PIP) cores that are delivered as black box designs. Under this context, the security of the IP cores becomes the top concern due to the following reasons. (1) The hardware IP cores are developed by third parties that are not fully trusted; (2) The service providers typically have limited resources to deploy separate hardware for the sensitive applications; and (3) Despite the traditional belief that hardware is more secure, the reconfigurable nature of the CPU-FPGA system may introduce new vulnerabilities exposed to potential remote attacks on hardware without the need of physical access. For example, the CPU-FPGA system could enable live deployments of hardware Trojans (on the FPGA part) and malware (on the CPU part),
which complicates the security measures and elevates the system security requirement to a new level.

The community has mostly focused on the performance acceleration and optimization of CPU-FPGA systems, where security has not been fully studied [3, 4]. In the security community, the prior works have intensively studied the security around CPU and FPGA systems, but they have been targeting CPU and FPGA systems separately, e.g., the former in the software security community [8, 9, 10] and the latter in the hardware security community [11, 12, 13, 14]. In particular, we focus on the following unsolved threat models between CPU and FPGA systems.

- **CPU to FPGA attack**, in which the software applications on the CPU side attempt to compromise the FPGA system, such as leaking confidential information; and

- **FPGA to CPU attack**, in which the third party IP cores on the FPGA side attempt to compromise the software applications on the CPU side, such as interrupting the software execution or falsifying the data.

Our key insight is to address the root cause of the vulnerabilities that the heterogeneous system components, namely CPU and FPGA, share the same system domain with direct access to each other. Our solution is two-fold. First, we adopt a CPU-driven hardware isolation mechanism to physically separate the system components, including the processes and data in the CPU domain and the third party IP cores in the FPGA domain, into a secure world and a normal world. The hardware isolation is accomplished at the physical bus level of the system, which ensures that secure world cannot directly access normal world, so that the sensitive components can be deployed in the secure world and become immune to the attack flows from the normal world. Second, we deploy a secure agent in the secure world, which enforces a set of security policies to block the illegal access from the attack flow (i.e., access control policies) and prevent sensitive information from being leaked (i.e., output verification policies). The security policies ensure that the secure
world resources and services are accessible only by legitimate processes or IP cores, while the attack flows in both directions between CPU and FPGA are blocked. We deploy the hardware isolation-based security mechanism as a new architectural framework, namely HISA.

Furthermore, there exist several challenges that we must address to effectively adopt HISA into practice. First, while in the past decades many existing research efforts have been targeting the 3PIP security problem under the context of traditional IC supply chain [14, 15, 16], we observe that the 3PIP security problem would become even more challenging in HISA. For example, FPGAs are designed to be flexible and capable of conducting dynamic partial reconfiguration [1, 17], which the attacker could leverage to covertly embed and reconfigure hardware Trojans at runtime [18]. Therefore, this situation pushes the requirement for security verification (e.g., hardware Trojan detection) from offline/static analysis to online/runtime monitoring.

Second, deploying the software programs in HISA is not straightforward. When sensitive information flows across multiple methods or in a much larger code region, securing the processing of such data can be quite complex. Furthermore, for legacy programs that have long been deployed, refactoring them to HISA may require significant efforts. The resulting software can also be inefficient or erroneous. For example, manually identifying possible weak links along a processing chain that a piece of sensitive data must flow through can lead to over-protection, resulting in waste of precious hardware resources as excessive code is being protected. On the other hand, if the data flow is complex, manual analysis can lead to under-protection, resulting in weak links that can be exploited by adversaries.

Third, when the programs are compute-intensive, it is challenging to execute them in a singular local end (e.g., a mobile device) due to the limited computational resources. Under this circumstance, the user might have to offload the computations to a service provider (e.g., a cloud platform) that are capable to execute such programs. Furthermore, to enhance the system security in this offloading solution, simply employing HISA to both the local
end and the cloud end is not enough. The local and cloud ends first require to build a trust between each other before conducting any computations. Then, the communication between the local end and the cloud end must always maintain secure and encrypted. In addition to the security, user privacy is significantly important to address in this local-cloud system. During offloading, the user data is sent to and accessed by the service provider. The data could be highly security/privacy-sensitive and should not always accessible by the service provider.

To resolve the aforementioned challenges, we target the following three research questions (i.e., RQs) for HISA:

- **Security-RQ:** How could we verify the security/correctness of FPGA-based applications in HISA at runtime?

- **Programming-RQ:** How could the software developer who has limited knowledge of security or hardware isolation techniques deploy the CPU programs into HISA?

- **Scalability-RQ:** How could HISA execute large applications, such as deep neural networks (DNNs)?

To address Security-RQ, we develop and deploy a runtime hardware security verifier to HISA, namely ApproVer, by applying approximate computing. The approximate computing algorithm employs two types of application-level approximations, namely spatial approximation and temporal approximation, to achieve the goals of runtime repeated execution and verification. Specifically, there are two requirements that our ApproVer must meet. From the security perspective, ApproVer needs to be employed in the (trusted) CPU component to verify security of the (untrusted) FPGA component. From the performance perspective, ApproVer must be highly efficient to capture the attacks with low false positive/negative rates. Based on the design, we conduct a comprehensive case study using a video motion detection application. Our empirical hardware evaluation justifies the premium security and performance of the proposed approach.
To address Programming-RQ, we develop a security-aware program slicing tool for HISA, namely TZSLICER, by leveraging dynamic analysis. The developers only need to prepare the source program, input data, sensitive (tainted) data, and resource constraints in terms of TEE memory capacity and communication overhead. Then, our TZSLICER tool could slice and deploy the program into HISA based on the given input information. There are three components in TZSLICER, namely Taint Analyzer, Program Slicer, and Slice Optimizer to generate the secure slice and normal slice in HISA. Specifically, the Taint Analyzer first generates a system dependency graph to analyze the program and propagate the computational flow based on the input data and tainted data. Then, the Program Slicer partitions the program based on the analysis results to generate and deploy the secure and normal slices in HISA. With the resource constraints, the Slice Optimizer optimizes the secure and normal slices to balance the TEE resource usage and communication overhead between the TEE (secure world) and the non-TEE (normal world).

Furthermore, we address Scalability-RQ by extending the singular TEE system in HISA to a dual TEE system, namely HybridTEE. This extension provides a platform for users to securely offload the large applications to a remote service provider. HybridTEE contains a local TEE for the users and a cloud TEE for the service provider, which ensures that the computations are only executed in the TEEs for security protection and the private data only stays on the local device for privacy protection. For the applications that cannot be split and must be fully executed in the cloud, we design a proactive privacy protection method for the users. In this study, we target a deep learning-based image recognition system and add protective perturbations to the input images to protect the privacy-sensitive information in the images. Specifically, the protected images with the perturbations are not visible to human vision to defend against the privacy leakage. In the meantime, the original target model running in the cloud can still perform the requested tasks on the protected images with high accuracy.

The remainder of the dissertation is organized as follows. First, we show the detailed
design and evaluation for HISA in chapter 2. Second, we address Security-RQ by developing ApproVer in chapter 3. Third, we address Programming-RQ by designing TZSLICER in chapter 4. Fourth, we address Scalability-RQ by extending HISA to a local-cloud system with proactive privacy protection in chapter 5. Finally, we conclude the dissertation in chapter 6.
CHAPTER 2
HISA: HARDWARE ISOLATION-BASED SECURE ARCHITECTURE FOR CPU-FPGA EMBEDDED SYSTEMS

2.1 CPU-FPGA Threat Models

2.1.1 Overview of CPU-FPGA Threats

Given that both the CPU and the third party FPGA IP cores often process critical data, it is not surprising that the attackers have the incentive to break into either system for the purpose of stealing certain confidential data or compromising the sensitive operations. There are two types of attack flows that we focus on in this chapter, as outlined in Figure 2.1:

**CPU to FPGA Attack.** In a CPU to FPGA attack, the attacker attempts to access the memory blocks (e.g., BRAMs) on the FPGA side, which store secret data, from a compromised software application on the CPU side. Typically the attacker would issue a two-step threat flow: (1) Infer the memory address of the secret data in BRAM using data scan pattern analysis; and (2) Access the inferred memory blocks by directly conducting memory access (DMA) or monitoring the data transfer on the bus line.

**FPGA to CPU Attack.** In an FPGA to CPU attack, the attacker first compromises the FPGA design flow by embedding a maliciously modified IP core, namely hardware Trojan [15], into the FPGA. Once triggered, such a hardware Trojan could proactively access the secret data on the CPU side to either leak or falsify the data, both of which cause security consequences concerning secret data or sensitive operations. In order to hide the Trojan from being detected, attackers typically employ extremely rare triggering conditions that are only known by themselves and difficult to be covered by regular functional test cases.
2.1.2 Case Study

To better illustrate the two CPU-FPGA threat models, we implement a prototype system of a security-sensitive surveillance camera, which has an on-device motion detection module to detect the objects in motion in the captured video in real time, as demonstrated in Figure 2.2 (video sample courtesy of [19]). Figure 2.2 shows the hardware system setup using Xilinx Zynq-7000 ZC702 SoC. The SoC has a CPU component that contains two ARM cores and an FPGA component that contains a Xilinx FPGA board. We employ the CPU part on the board to provide basic interface to receive the video frames from the HDMI card, and we deploy a motion detection IP core based on the Gaussian Mixture Model (GMM) [20, 21] in the FPGA part to conduct real time motion detection based on the received video frames.

As shown in Figure 2.3(a) and Figure 2.3(b), there is a remarkable white area to indicate the moving objects on the road. We further implement a threat model based on the prototype system. The outcome caused by the threat model is shown in Figure 2.3(b), where the moving object is hidden in the background if there is no motion detected. The attack scenario is a “replay attack” triggered by either of the two CPU-FPGA attack flows shown in Figure 2.4: (1) CPU to FPGA attack, where a malware in the CPU system accesses BRAM via DMA, scans the memory blocks for the output video frame location, and replaces the target frame with one of the pre-recorded motion-free frames; and (2) FPGA to CPU attack,
where a hardware Trojan embedded in the motion detection IP maliciously replaces the video output frames. Both of the attacks compromise the shared data holder (i.e., the BRAM) between the CPU and FPGA, which poses significant security challenges. Existing countermeasures based on encrypting the data in the memory [8, 22] or isolating different software applications, information flow, or IP cores [9, 10, 13, 14] do not suffice to prevent such attacks, as the former poses overwhelming overhead for this real-time video processing application, and the latter does not prevent attacks across CPU and FPGA boundaries.

Figure 2.2: Hardware system setup for the motion detection system using a Xilinx SoC with both CPU and FPGA (video sample courtesy of [19]).

Figure 2.3: (a) Normal motion detection scenario without attacks; and (b) Motion detection results under CPU to FPGA or FPGA to CPU attacks, in which the moving objects are hidden in the background instead of being detected (video sample courtesy of [19]).
2.2 Related Work

2.2.1 Intra-CPU and Intra-FPGA Security

The software security community focuses on protecting sensitive data and services from potential attacks within the CPU system, using a variety of security mechanisms, such as encryption-based [8, 22] and isolation-based approaches [9, 10]. The hardware security community effectively addressed the trust and integrity issues within the FPGA system, such as hardware Trojan detection [15], information flow tracking/isolation [12, 13], and isolation between multiple IP cores [14]. However, neither threads of research have considered the interplay between CPU and FPGA and thus the new security challenges brought by the emerging heterogeneous system.

2.2.2 Inter-CPU/FPGA Security

More recently, there have been several research efforts related to the Inter-CPU/FPGA security issues. Jacob et al. employ a malicious IP core to break the secure boot process of a CPU-FPGA SoC [23]. Olson et al. develop “Border Control” [24], an OS-level solution to ensure that the accelerators are respecting the access permissions enforced by the page table. Similarly, MMU or IO-MMU based techniques [25] and the CAPI interface [26] require a trusted OS to manage and enforce the security. HISA is orthogonal to these related works in
that (1) It targets the security of the CPU-FPGA systems at runtime instead of the boot-up stage; and (2) It provides a hardware-level solution other than at the OS level, which does not require the OS or upper-level software to be in the trusted computing base (TCB) and thus provides an additional and strong layer of security to the CPU-FPGA systems.

2.2.3 Hardware Isolation Primitives

Isolation or sandboxing has been an important security principle adopted in many security-sensitive systems, such as hypervisor-based virtual machines [27] for secure cloud computing. Recently, hardware-based isolation mechanisms, such as Intel SGX [28] and ARM TrustZone [29], have been developed to provide lower level, more resilient isolation primitives. Intel SGX is based on memory encryption and does not support peripheral devices like FPGAs. ARM TrustZone functions at the physical bus level, which could be adopted to protect on-chip peripheral devices. Recently, there have been related works that employ ARM TrustZone to secure software applications [9, 10, 30, 31] and hardware components [11, 32, 33]. The existing ARM Trusted Firmware [34] implements the generic functionality of TrustZone without customized security policies. HISA differentiates from these existing research efforts by providing a universal CPU-FPGA security framework with well-defined security policies, which support a wide range of applications.

2.3 HISA Framework

The CPU-FPGA hardware isolation primitive is inspired by the most natural way of protecting valuable assets, which is to physically isolate them from the potential malicious attacks. As shown in Figure 2.5, with hardware isolation we split the runtime environment of a CPU core into two isolated “worlds”, namely the secure world and the normal world. Under the context of the CPU-FPGA heterogeneous system, each world contains both the CPU component and the FPGA component. The hardware isolation primitive ensures that, while isolated from the secure world, the normal world components do not have access to
the secure world resources. However, the secure world applications can still access normal world resources, leading to a secure one-way communication between the two isolated environments. Figure 2.5 demonstrates the four communication flows between the CPU and FPGA components in the secure and normal worlds, where only the two communications originated from the secure world will pass through, and those from the normal world will be blocked by the hardware isolation primitive.

With the recent advancement in CPU security features, such as ARM TrustZone [29], the hardware isolation can be realized at the physical bus level and enforced via strictly controlled CPU context switch, which is secure against software-level attacks [9]. However, the prior research and practice on CPU-enabled hardware isolation technologies involve only the protection of software data/code [10] or OS kernels [9], while the isolation of hardware (e.g., FPGA) components in a CPU-FPGA heterogeneous system has not been fully studied.

Figure 2.6 illustrates our realization of extending the CPU-based hardware isolation to the FPGA system using ARM TrustZone [29]. The ARM-based CPU-FPGA system, such as that in Xilinx Zynq SoC [5], involves a CPU system and an FPGA system connected via an AXI Interconnect. We enable the security checking feature at the AMBA bus port on
the AXI interconnect using the Xilinx Vivado tool, which ensures that the AMBA bus port will enforce the security property set by its non-secure (NS) bit while granting accesses. Since the NS bit of the AMBA bus port (i.e., AWPROT[1] for writing and ARPROT[1] for reading) is set to 0 by default (i.e., the property is “secure”), it will only grant accesses to the secure world applications (with NS bit being 0), which technically achieves the secure physical isolation.

### 2.3.1 Secure Data Flow

Based on the CPU-FPGA hardware isolation primitive, we deploy the FPGA IP cores into the secure world, as shown in Figure 2.7, so that it is isolated from the potentially malicious
normal world applications.\(^1\) The separation of the two worlds causes the system to execute in two independent modes with completely isolated logical and physical components and resources. At runtime, the CPU determines the schedule of switching between the two execution modes, and it employs a secure monitor, running in the secure world, to execute the switching upon receiving a secure monitor call (SMC) from either world. To enable the communication between the two worlds, we deploy two blocks of shared memory in the normal world for storing the inputs and outputs to and from the secure world. This is required as the system must support the regular functionality of computation using the IP core under protection, as long as the service request is legitimate and the output does not leak confidential information. To support the security verification, we deploy a secure agent in the secure world, which is in charge of invoking the IP core services on behalf of the normal world application.

With the completion of the data flow as shown in Figure 2.7, the normal agent successfully invokes the IP service and obtains the response. However, during this process, both the IP core and the data it processes are never exposed to potential malicious access from the normal world, and all the interactions related to the protected hardware and data are conducted by the secure agent that is trustworthy. In addition, the secure agent serves as a security gateway to examine and filter the data requests from the normal world applications, to further improve the security of the IP core.

**Secure Monitor.** The secure monitor plays an essential role in the entire secure data flow, as it controls the context switching between secure and normal worlds, where completely different and isolated memory blocks and registers are used. During each world switch, the monitor detects the source and destination worlds, saves the states of the source world, and restores the states of the destination world.

**Secure Agent.** The secure agent resides in the secure world, and it is the only entity

---

\(^1\)For the ease of discussion, here we describe the communication flow using the CPU to FPGA attack as an example (i.e., FPGA IP under protection). The same framework and flow can be applied to the FPGA to CPU attack as well.
that has direct access to the IP core under protection, which is activated immediately after the secure monitor completes the world switching operation, as shown in Figure 2.7. The secure agent is in charge of the following three tasks: (1) Receiving the original IP service request with inputs and passing it over to the target IP core; (2) Delivering the output of the IP service call to the shared memory in the normal world; and (3) Issuing SMC to request the switching from the secure world to the normal world.

**Normal Agent.** The normal agent serves as the access point of the hardware isolation framework to the end users. It coordinates the service call and data communications between the normal world and the secure world. Due to the fact that the normal agent resides in the normal world, it does not have direct access to the secure world data/services. Instead, the normal agent invokes the service call indirectly via an SMC to the secure monitor that in turn switches the CPU mode to the secure world.

**Shared Memory.** The shared memory is the medium that enables the secure communications between the normal world and the secure world. The hardware isolation ensures that there is no direct communication channel between the two worlds. However, since both the secure world and normal world have access to the normal world resources, it is possible to convert a memory block in the normal world as the shared memory and thus the communication channel.

### 2.3.2 CPU-FPGA Security Policies

The CPU-FPGA hardware isolation framework only provides the fundamental support for creating the isolated environment without the enforcement of security. Obviously, it is not practical to disable all accesses to the secure world, as the IP core must allow normal accesses issued by non-malicious users and applications. Therefore, we deploy access control policies at the secure agent to check the legitimacy of the service request, for both the CPU to FPGA and FPGA to CPU flows, to block only the malicious or suspicious requests. Furthermore, for attacks that leak or falsify security/privacy-sensitive data or
operations and that are not possible to identify at the incoming access control phase, we conduct output verification at the secure agent prior to delivering the results from the secure world to the normal world.

**Access Control Policy.** Upon receiving a request from the normal world, the secure agent first examines whether the requesting agent has the privilege for the service request using a pre-defined, application-specific access control policy. The access control serves two purposes: (1) It eliminates unauthorized service requests to the IP core, possibly issued by a malicious attacker attempting to gain access to the secret data/service hosted by the IP core; (2) For legitimate service accesses, it keeps track of the frequency of such requests and schedules them according to the resource and computation capability of the secure service, which not only ensures efficient resource usage but also prevents potential denial of service attacks. In summary, the access control policy ensures the security and integrity of the protected IP core at the entry point of the secure world.

**Output Verification Policy.** Upon completing the service request, the secure agent examines the correctness of the results before delivering them to the shared memory. There are two reasons why we enforce this policy. First, the hardware isolation technology only ensures isolation but lacks security and integrity verification for the data and service deployed in the secure world, which may cause security concerns under the events that malicious software and hardware services have been deployed in the secure world, such as a malware or hardware Trojan. Second, for data and services that are subject to reliability issues due to random or environmental effects, the additional verification ensures the correctness of the results.

### 2.4 Case Study: Securing IoT Smart Camera

In this section, we close the loop in the case study by leveraging HISA to counter the replay attack in the IoT security camera targeting the FPGA to CPU attack flow. To employ HISA, we first place the motion detection IP and the corresponding frame buffers into the secure
world of HISA. Then, our focus is on the design of application-specific output verification policies for the secure agent to enforce. In particular, we develop both passive and proactive output verification policies to prevent the replay attack. The former monitors representative signals from the system in a non-intrusive manner and captures suspicious behavior; The latter employs a “design for trust” paradigm to facilitate the security verification.

![Diagram of motion detection](image)

**Figure 2.8:** Passive output verification via side channel analysis.

### 2.4.1 Passive Output Verification

Given the difficulty in detecting the FPGA to CPU hardware Trojan via normal tests, our solution is to adopt a timing side channel-based approach to differentiate the attack and normal scenarios. Our intuition is that there is a clear difference in execution time between the normal motion detection operation and the one infected with hardware Trojans. The normal motion detection module involves loading video frames from the memory, executing the motion detection algorithm, and delivering the results to the output video buffer. Suppose $Tc_i$ represents the time duration of copying the $i^{th}$ frame, and $Tp_i$ represents the time duration of processing the $i^{th}$ frame, the normal motion detection time $Tn_i$ for the $i^{th}$ frame can be calculated as Equation 2.1.

$$Tn_i = Tc_i + Tp_i$$ (2.1)
In the FPGA to CPU attack, once the output frame without motion has been recorded, the hardware Trojan just needs to copy the recorded frame to the output frame buffer and, therefore, the total time duration under attack, $T_{a_i}$, can be represented in Equation 2.2.

$$T_{a_i} = T_{c_i} \quad (2.2)$$

We assume that $T_{p_i}$ is significantly high as it is consumed by sophisticated computer vision algorithms for motion detection [20] and, therefore, the FPGA to CPU attack can be captured by comparing the $T_{n_i}$ and $T_{a_i}$ values. In our system design, we place a timer in the surveillance video system to measure the system execution time for the normal case (i.e., motion detection) and the attack case (i.e., frame copy) as shown in Figure 2.8. We start/stop the timer once the motion detection module begins/ends with the processing of the corresponding video frames. By monitoring the measured timing values, we are able to observe the difference in time when the hardware Trojan is activated.

**2.4.2 Proactive Output Verification**

In proactive output verification, we leverage the fact that the replay attack outputs the same pre-recorded video frames for all the different input frames. As shown in Figure 2.9, our proactive countermeasure is to insert artificial motion patterns in a sampled set of input frames and check the corresponding output frames periodically for the inserted motion. In our video pipeline, the secure agent generates motion patterns and adds them into the input frames every $k$ seconds. The inserted patterns will be captured by the motion detection module and labeled with white spots in the output frame, which can be as small as invisible to human. Then, the output verification in HISA can check the pixel values in the specific motion-inserted region to determine if they have been labeled in white. Under a replay attack, the output frame will not contain any motion including the intentionally inserted ones, which serves as an indicator for the presence of FPGA to CPU attack.
2.4.3 Effectiveness of Security Policies

Figure 2.10 shows the distribution of timing results with time measurements of 1245 frames in the attack case and 1000 frames in the normal case. We observe a large gap between the timing in the two cases (a more than 100% difference). The results indicate that the side channel-based output verification can achieve zero false positives/negatives in detecting the replay attack. Furthermore, we demonstrate the effectiveness of the proactive verification in Figure 2.11, which shows that, in the non-attack scenario, the output frame contains additional motion objects that are inserted by the defense mechanism. Note that we intentionally design large-size motion objects only for the demonstration purpose, which can be made as small as invisible in a real setting.

Figure 2.10: Demonstration of timing side channel analysis.
2.5 Experimental Evaluation

2.5.1 Experimental Setup

**HISA Implementation.** We implement HISA on a Xilinx Zynq-7000 all programmable SoC [5], namely the ZC702 board. It is equipped with an ARMv7-A CPU in the Zynq processing system (PS) and a programmable logic (PL) subsystem (i.e., FPGA), which are connected by the AMBA Interconnect. The ARMv7-A processor has dual 667MHz ARM Cortex-A9 cores, which supports the ARM TrustZone feature for hardware isolation. Since HISA is a CPU-based security framework, it does not introduce additional hardware area and resource overhead to the SoC. Also, the security of the secure agent and the secure monitor is ensured by the secure boot process supported by ARM TrustZone [29].

**Reference IP Implementation.** For a comprehensive security and performance evaluation, we adopt four different custom IP cores, as shown in Table 2.1, which cover a variety of application domains, such as computing (i.e., Multiplier), cryptography (i.e., RSA), signal processing (i.e., XADC), and video processing (i.e., Motion Detection). For each IP core, we develop a set of threat models that involves both CPU to FPGA and FPGA to CPU attacks. Among them, Denial of Service (DoS) is a representative CPU to FPGA attack,
in which a malicious CPU application attempts to issue an excessive amount of requests to the IP core to block its service to the legitimate users. The information leakage attack is a representative FPGA to CPU attack, in which an information leakage hardware Trojan [11] is embedded in the IP core that may leak sensitive data via a covert channel without being identified by the software applications. In addition, we consider several special threat models targeting on certain IP cores, such as replay attack for the motion detection IP.

Table 2.1: Reference IP cores, threat models, and security policy designs for security analysis.

<table>
<thead>
<tr>
<th>Custom IP</th>
<th>Threat Model</th>
<th>Access Control</th>
<th>Output Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier [35]</td>
<td>Denial of Service*</td>
<td>Caller Identification</td>
<td>Sampled Redundancy Check</td>
</tr>
<tr>
<td></td>
<td>Information Leakage†</td>
<td>Caller Access Control</td>
<td></td>
</tr>
<tr>
<td>RSA [36]</td>
<td>Denial of Service*</td>
<td>Caller Identification</td>
<td>Information Flow</td>
</tr>
<tr>
<td></td>
<td>Information Leakage†</td>
<td>Caller Access Control</td>
<td></td>
</tr>
<tr>
<td>XADC sensor [37]</td>
<td>Denial of Service*</td>
<td>Caller Identification</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>Information Leakage†</td>
<td>Caller Access Control</td>
<td></td>
</tr>
<tr>
<td>Motion Detection [21]</td>
<td>Data Falsifying Attack*†</td>
<td>Secure Isolation</td>
<td>Side Channel Inspection</td>
</tr>
<tr>
<td></td>
<td>Information Leakage*†</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* CPU to FPGA attack; † FPGA to CPU attack.

**Reference Security Policy Implementation.** For our evaluation purpose, we implement two access control policies:

- **Caller Identification**, where the secure agent examines the identity of the caller that is requesting the IP core service and filters out unauthorized accesses via either a “blocklist” (i.e., blocks pre-recorded malicious applications) or a “allowlist” (i.e., only allows for the pre-recorded trusted applications).

- **Caller Access Control**, where the secure agent records the caller access history (e.g., number of access times and access patterns) and blocks the access upon capturing suspicious patterns (e.g., excessive number of accesses) [32].

Furthermore, we implement the following output verification policies:
• Sampled Redundancy Check, in which we partially repeat the computation by the IP core at the software level. For example, for a multiplier IP, we sample and re-compute certain bits of the multiplication and check for correctness. An advanced version will be elaborated in the next chapter.

• Information Flow Verification, in which the hardware isolation framework tracks the information flow by inserting certain check points in the IP core. Then, it checks the correctness of the information flow instead of that of the end results to detect information leakage.

Then, we deploy the access control and/or the output verification policies for each custom IP. For example, the XADC IP adopts the caller identification and access control policies. The multiplier IP adopts sampled redundancy check for the last digit of the multiplication results. The RSA IP employs information flow verification to check whether it has been indeed invoked to differentiate the normal request from the malicious information leakage attack.

![Multiplier Isolation Delay](image1.png)  
![XADC Isolation Delay](image2.png)  
![RSA Isolation Delay](image3.png)  

Figure 2.12: Evaluation of HISA isolation delay on the reference IP cores with access control and output verification.

2.5.2 Performance Overhead Evaluation

We first measure the world switching delay for three of the IP cores, as shown in Figure 2.12. For each IP, we collect a group of delay values while increasing the number of times to invoke the IP service. In these experiments, HISA conducts one world switch from the
normal world to the secure world, executes the computation tasks, and switches back to the normal with output to the shared memory. We observe that the hardware isolation delays per 10 IP service calls are below 10 μs, which indicates a very low overhead.

Furthermore, to evaluate the timing overhead introduced by the output verification policies in the motion detection IP, we measure the execution time of the passive verification for each frame with a total number of 871 frames. The distribution of timing results is shown in the first column of Table 2.2. We observe that the side channel analysis takes sub-micro second, which is negligible compared to the motion detection time that is hundreds of milliseconds. Also, Table 2.2 shows the timing overhead of the proactive verification method, which is at hundreds of milliseconds, similar to motion detection time for each frame. Considering that this process will be carried out every \( k \) seconds (e.g., \( k=2 \)), the overhead is in an acceptable range to meet the real time requirement.

<table>
<thead>
<tr>
<th>Table 2.2: Timing evaluation (μs) of output verifications.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Passive Verification</td>
</tr>
<tr>
<td>Minimum</td>
</tr>
<tr>
<td>First Quartile</td>
</tr>
<tr>
<td>Median</td>
</tr>
<tr>
<td>Third Quartile</td>
</tr>
<tr>
<td>Maximum</td>
</tr>
</tbody>
</table>

2.5.3 Power and Energy Overhead Evaluation

We further evaluate the power/energy consumption of HISA considering that power/energy is a limited resource in embedded systems. We run the TI fusion power measuring tool [38] while invoking the reference IP cores for power measurements on both the PS (CPU) and PL (FPGA) components on the SoC. Figure 2.13 shows the detailed power/energy evaluation results of the XADC IP. Similar to the performance evaluation reported in [33], we compare three cases: (1) Regular XADC, which is the baseline IP core without any
security protection; (2) Isolated XADC, which is the HISA protected XADC IP; and (3) Encrypted XADC, which protects the IP by encrypting the sensor data using AES-CBC.

In the power results for both PS and PL (i.e., the left two figures), we observe that HISA reaches the peak power with a higher frequency than the baseline (i.e., regular XADC). However, the increased frequency is noticeably lower than the encrypted XADC. Also, for the same workload assigned, HISA finishes the execution earlier than the encrypted XADC, which further saves the energy consumption. These observations are reflected into the energy results (i.e., the right two figures).

![XADC PL-Internal Power and Energy Evaluation](image1)

![XADC PS-Internal Power and Energy Evaluation](image2)

Figure 2.13: Power and energy evaluation for the XADC IP.
CHAPTER 3
APPROVER: RUNTIME HARDWARE SECURITY VERIFICATION USING APPROXIMATE COMPUTING

In this chapter, we aim to address Security-RQ for HISA by developing a runtime FPGA security verification mechanism, namely ApproVer. Although there exist several solutions aiming to verify the security of IP cores, they either rely on the availability of a golden chip (e.g., [39]), which is not feasible in reality, or conduct only offline/static verification (e.g., [40]) which does not fulfill the runtime security requirement. The key insight of our ApproVer in addressing the runtime hardware security challenge is inspired by the runtime redundancy-based verification [41, 42] used in fault-tolerant system designs, where one or multiple repeated executions of the key algorithms or operations are conducted to approve or disapprove the potentially faulty primary execution. While the redundancy-based verification works effectively for fault-tolerant systems, there are two major challenges to apply it to the security domain. (1) Security Challenge: Under the security context the repeated executions must be conducted on a separate trusted component, other than the component under verification, as otherwise the repeated executions would also be contaminated with the attack and eventually result in false negatives in the security verification; and (2) Performance Challenge: The repeated executions may cause significant performance overhead and thus easily compromise the premium runtime performance benefit brought by the IP core.

We address both challenges by developing ApproVer that is a security verification framework employing approximate computing for HISA. In particular, we address the security challenge by deploying the verification process (i.e., the repeated executions) in the CPU component, which is separate from the FPGA (i.e., where the IP core resides) and considered as trustworthy. Furthermore, to address the performance challenge, we employ
approximate computing and develop an approximate software version of the IP core on the CPU for the runtime repeated executions. The key rationale behind this design is that the computation accuracy required by the repeated executions (i.e., for security verification) is orders of magnitude lower than that required by the IP core under verification. It is because a malicious security attack would typically dramatically change the results of the system, which can be effectively and efficiently captured by an inaccurate but fast approximate execution.

3.1 Related Work

3.1.1 Security Verification Methods

Currently there are two main categories of techniques for security verification against untrusted software and hardware computations. The first category, namely verifiable computing, examines the correctness of the computation by requiring a proof associated with the computation results [43, 44, 45]. However, it is challenging to reduce the performance overhead between the prover (i.e., the party that conducts the computation) and the verifier (i.e., the party that verifies the correctness) in the verification protocol. The second category examines the information flow of the computation to ensure that the data and program execution are routed and executed in the expected manner [16, 40, 46]. The existing information flow tracking techniques are mostly static (i.e., at compilation time instead of runtime), which are not sufficient to defend against the hardware security threats discussed in this research.

3.1.2 Approximate Computing

Approximate computing has been leveraged to trade computation accuracy for performance acceleration and energy/resource savings [47]. In addition, there have been research works that employ approximate computing for authentication and information protection in embedded systems [48, 49]. The approximation strategies include hardware-level approximation
such as critical path reduction [50] and precision control [51], as well as software-level approximation such as loop perforation [47].

To the best of our knowledge, ApproVer is the first use of approximate computing to verify 3PIP security in the CPU-FPGA architecture like HISA. Compared to the verifiable computing mechanisms, the IP cores as provers in the proposed verification process do not need to provide any additional protocol-based proofs. In addition, our method focusing on runtime results-based verification is orthogonal to the recent CPU-FPGA security work aiming to prevent the attacks using hardware and system level techniques [52, 53].

### 3.2 Threat Model

Figure 3.1 shows the threat model we target in this chapter, where a hardware Trojan in the untrusted 3PIP manipulates the system functionality and sends falsified results to the CPU (i.e., FPGA to CPU attack). We consider two types of hardware Trojans, namely the spatial Trojan (or STrojan) and the temporal Trojan (or TTrojan), which falsify the computation results in the spatial and temporal manners, respectively.

![Figure 3.1: Threat model considered in this chapter: The untrusted 3PIP in the FPGA sends falsified results to the CPU.](image)

Specifically, in HISA, the CPU requests the FPGA to compute a certain number of jobs, where each job consists of multiple independent iterative tasks. The STrojan compromises a subset of tasks in every requested job, and the TTrojan compromises a subset of jobs completely (i.e., all the tasks in these targeted jobs are compromised). For example, in an encryption IP that encrypts messages, we assume each message is a job and each word in the message is a task. The IP with the STrojan could return the falsified encrypted messages with incorrectly encrypted words in each of them. The IP with the TTrojan could return
the encrypted messages where all the words in a subset of the messages are incorrectly encrypted. Note that other possible attacks, such as side channel attacks [54, 55], are out of the scope for this chapter.

3.3 ApproVer System Framework

![ApproVer system overview](image)

Figure 3.2: ApproVer system overview. 3PIP in the FPGA conducts the actual computation. ApproVer in the CPU requires two stages to detect if the 3PIP contains Trojans, namely approximate computation and security verification.

To defend against the aforementioned threat model, we design a hardware security verification mechanism by using approximate computing (AC), namely ApproVer. In the proposed mechanism, we assume that the CPU is trusted and benign, but the FPGA is not trusted. Figure 3.2 shows the CPU-FPGA system overview with the proposed framework. ApproVer residing in the CPU contains two stages, namely approximate computation and security verification. In the approximate computation stage, we propose two approximation strategies, namely spatial approximation and temporal approximation to convert the input jobs to approximated jobs. Then, the software computation processes the approximated jobs and sends the completed jobs to the verification component in the security verification stage. In the meantime, the IP in the FPGA sends the completed jobs by the actual computation to the verification component in the CPU. Furthermore, the verification component compares the approximate results with the actual results and generates the Trojan detection results.

**Spatial/Temporal Approximation.** As shown in Figure 3.3, there are \( m \) input jobs,
where each job includes \( n \) tasks, i.e., \( n \times m \) tasks in total. The spatial approximation randomly selects \( x \) different tasks from each job and generates \( m \) approximated jobs, i.e., \( x \times m \) tasks in total. For the temporal approximation, \( y \) out of the \( m \) jobs are selected with each containing all the \( n \) tasks.

Figure 3.3: Demonstration of spatial/temporal approximation. The spatial approximation randomly selects a subset of tasks from every input job. The temporal approximation randomly selects a subset of jobs from all the inputs jobs.

\[ f_i = \frac{IP_i}{AC_i} \]  

(3.1)

\[ I = \begin{cases}  
\frac{n}{x}, & \text{spatial approximation} \\
1, & \text{temporal approximation} 
\end{cases} \]  

(3.2)

**Security Verification.** To detect whether the IP contains hardware Trojans, the verification component calculates the factor value \( f_i \) for each job as shown in Equation 3.1, where \( IP_i \) indicates the results of the completed \( i \)th job generated by the IP, and \( AC_i \) indicates the completed \( i \)th job generated by the software computation. The verification component checks the difference between the factor value \( f_i \) and the ideal factor value \( I \) shown in
Equation 3.2. For the spatial approximation, \( n \) and \( x \) indicate the number of tasks in each input job and the number of tasks in each approximated job, respectively. For the temporal approximation, since the selected jobs have the same tasks in the corresponding input jobs, the ideal factor \( I \) is 1. If the difference between \( f_i \) and \( I \) is larger than a threshold, the verification component will report the IP is malicious. For the special case of \( AC_i = 0 \), we consider it as an invalid verification.

### 3.4 Case Study on Video Motion Detection CPU-FPGA System

![Diagram of Video Motion Detection System]

Figure 3.4: The workflow of the video motion detection system (the video example is from [19]). The system requires three stages to complete the motion detection process, namely video pipeline, video processing, and AC-based security verification.

Based on the proposed framework, we conduct a case study on a CPU-FPGA based video motion detection system. As shown in Figure 3.4, the system involves three stages, namely video pipeline, video processing, and AC-based security verification. The implementation of the video pipeline stage is based on [56] and [57]. First, the system loads the video input via an HDMI-in module. The VDMA module (aka., video direct memory access) transfers the video into the input block of the DDR memory. In the video processing stage, the third-party video motion detection IP reads the input memory block and detects if the video contains any motions. Specifically, the IP identifies the background and the foreground in the input video frame, and conducts the pixel-by-pixel matching between the foreground
and the background to calculate the pixel differences, which represents motions. After the video processing stage, the IP loads the processed video into the output block of the DDR memory. Then, the VDMA transfers the video output via the HDMI-out module.

The output monitor in Figure 3.4 shows that the benign IP successfully detects the pedestrian (labelled with red rectangles) as motions (i.e., white pixels). In the STrojan scenario, the malicious IP removes a subset of detected motions to mislead the system into ignoring partial actual motions. In the TTrojan scenario, the malicious IP removes all the motions in certain video frames, equivalent to disabling the motion detection feature during the playback of those frames. To detect the Trojans at runtime, ApproVer is activated when the IP is generating the motion detection results. It obtains the number of motions generated by the IP and compares it with the approximate computing results to decide whether the IP is malicious.

We implement the video motion detection CPU-FPGA system on a Xilinx Zynq-7000 SoC ZC702 board that contains dual ARM processors and FPGA as shown in Figure 3.5. In the FPGA, there are HDMI-in/out module (via a FPGA mezzanine card, namely FMC), VDMA, and video motion detection IP. ApproVer is executed by the ARM processors. The laptop provides the video input, and the PC monitor displays the video output with the detected motions.

3.5 Experimental Evaluation

We apply ApproVer into the CPU component of the video motion detection system to defend against the STrojan and TTrojan embedded in the FPGA component of the system. We evaluate the effectiveness and efficiency of the proposed method on a video input with 400 frames from [19]. Specifically, we aim to answer the following questions.

1. Effectiveness: Can our ApproVer distinguish between benign IPs and malicious IPs successfully?
2. Efficiency: Can our approximate computation achieve high performance at runtime?

3.5.1 Experimental Setup

In the FPGA, we develop the following IPs in the motion detection system. (1) NoTrojan IP: The IP is benign and conducts accurate computation. Note that it is only to evaluate if our ApproVer identifies the NoTrojan IP as a benign IP correctly, and we do not rely on this IP to detect Trojans, i.e., no golden Trojan-free model is required in our verification approach. (2) STrojan IP: The IP hides a certain number of pixels in the output frames. In the experiments, it removes the upcoming motions after more than 5000 motions in the frame. There are 353 out of 400 frames that are compromised by the STrojan. (3) TTrojan IP: The IP hides all the motions in a certain number of frames. In the experiments, there are 142 frames compromised by the TTrojan (i.e., from the 30th frame to the 90th frame, and from the 180th frame to the 260th frame).

In the CPU, ApproVer treats each frame as a job and each pixel in a frame as a task. There are two approximation strategies. (1) Spatial approximation (SAC), which processes one pixel in every \( x \) pixels (represented as SAC_{\times}); and (2) Temporal approximation
(TAC), which processes accurate computation for one frame in every \( y \) frames (represented as TAC\(_y\)). The motion detection algorithm adopted by the software is the same as that implemented in the FPGA IP. For security verification, the verification component calculates the factor value for each frame and decides whether the IP is benign or malicious.

3.5.2 Effectiveness Evaluation

We adopt the following four metrics to quantify the effectiveness of the proposed approach. Note that our ApproVer only identifies if the IP is malicious and does not identify which Trojan is embedded in the IP:

- **True positive (TP):** ApproVer successfully identifies the malicious IP.
- **True negative (TN):** ApproVer successfully identifies the benign IP.
- **False positive (FP):** ApproVer misidentifies the benign IP as the malicious IP.
- **False negative (FN):** ApproVer misidentifies the malicious IP as the benign IP.

We first evaluate whether ApproVer detects the NoTrojan IP correctly by calculating its accuracy based on Equation 3.3, which measures the occurrence rate of all the TN outcomes, since the TP and FN do not exist in the NoTrojan IP. Table 3.1 indicates the accuracy when we apply SAC\(_{36}/\)TAC\(_{20}\) with different thresholds. The threshold is a percentage of the ideal factor value to distinguish between the benign IP and the malicious IP. In the experiments, we observe that ApproVer never misidentifies the NoTrojan IP as the malicious IP unless the threshold is set as 100\% for SAC. The threshold 100\% indicates that the mechanism does not tolerate any difference generated from the approximate computation and the actual computation, which is why the accuracy decreases under this threshold.

\[
\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \quad (3.3)
\]
Furthermore, we evaluate whether ApproVer is able to detect the Trojans in the IP. Since the TN and FP do not exist in the IPs with Trojans, we calculate the precision and recall to comprehensively show the two aspects of the accuracy as shown in Equation 3.4. (1) Precision: In all of the frames where ApproVer identifies the Trojan as active, how many of them are truly compromised by the Trojan; and (2) Recall: In all of the frames where the Trojan is truly active, how many of them are correctly identified by ApproVer.

\[
\text{precision} = \frac{TP}{TP + FP} \\
\text{recall} = \frac{TP}{TP + FN}
\]  

(3.4)

Table 3.1: Accuracy for NoTrojan IP detection by SAC_36 and TAC_20.

<table>
<thead>
<tr>
<th>Threshold</th>
<th>25%</th>
<th>50%</th>
<th>75%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAC Accuracy</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.97</td>
</tr>
<tr>
<td>TAC Accuracy</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.2: Precision and recall for STrojan IP detection by SAC_36 and TAC_20.

<table>
<thead>
<tr>
<th>Threshold</th>
<th>25%</th>
<th>50%</th>
<th>75%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAC Precision</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Recall</td>
<td>0.87</td>
<td>0.92</td>
<td>0.94</td>
<td>0.95</td>
</tr>
<tr>
<td>TAC Precision</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Recall</td>
<td>0.94</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.3: Precision and recall for TTrojan IP detection by SAC_36 and TAC_20.

<table>
<thead>
<tr>
<th>Threshold</th>
<th>25%</th>
<th>50%</th>
<th>75%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAC Precision</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.95</td>
</tr>
<tr>
<td>Recall</td>
<td>0.96</td>
<td>0.96</td>
<td>0.96</td>
<td>0.96</td>
</tr>
<tr>
<td>TAC Precision</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Recall</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.2 shows the precision and recall results for SAC_36/TAC_20 with different thresholds when they defend against STrojan. We observe that, when ApproVer identifies an IP as malicious, the IP is always truly embedded with Trojans (i.e., the precision is always
However, with different thresholds ApproVer ignores 5% to 13% cases when the Trojan is active. We argue that increasing the threshold is helpful to increase the detection sensitivity and reduce the errors. Table 3.3 shows the precision and recall for SAC_36/TAC_20 with different thresholds against TTrojan. We observe that the TAC perfectly identifies the IP with TTrojan (i.e., the precision and recall are always 1s). For the SAC, there are 4% error rates for the recall, because the SAC misidentifies the malicious IP as a benign IP when the hidden motions are relatively small. Also, increasing the threshold to 100% compromises the precision of SAC for the same reason as in the NoTrojan experiments.

$$ F_1\text{-score} = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} $$

(3.5)

Figure 3.6: $F_1$-score comparison for SAC with different parameters (i.e., SAC_36, SAC_72, SAC_108, and SAC_144).

After the experiments of SAC_36/TAC_20 with different thresholds, we fix the threshold as 85% that performs well with the high accuracy in all the aforementioned experiments. Then, we change the parameters of SAC/TAC to discuss how the parameters affect the detection results. In the experiments, we notice that the parameters only affect the results for SAC against STrojan/TTrojan. To illustrate the precision and recall at the same time, we calculate the $F_1$-score [58] as shown in Equation 3.5. Figure 3.6 shows the $F_1$-score results of SAC with different parameter values against the STrojan and TTrojan. We observe that
higher SAC parameter values increase the computation error and reduce the F1-score. Also, the decreasing trend is approximately linear.

Figure 3.7: Time comparison between the accurate computation and the approximate computation conducted by the SAC/TAC with different parameters.

### 3.5.3 Efficiency Evaluation

Furthermore, we evaluate the efficiency of the SAC and TAC. Figure 3.7 shows the time consumption of the approximate computation stage, since it is the primary overhead in the proposed mechanism. In Figure 3.7(a), we notice that the SAC consumes at least 71% less time compared to the accurate computation conducted by the benign IP in the FPGA. Also, as shown in Figure 3.7(b), the TAC consumes at least 48% less time than the accurate computation conducted by the IP. The results indicate that the SAC and TAC are able to verify the security with low performance overhead.
CHAPTER 4
TZSLICER: SECURITY-AWARE DYNAMIC PROGRAM SLICING FOR HARDWARE ISOLATION

In this chapter, we develop TZSLICER to address Programming-RQ to improve usability of HISA. TZSLICER could automatically identify code statements in a program that must be protected based on a sensitive, must-protect variable list provided by developers. As such, the inputs of TZSLICER are a program that contains sensitive data and a list of variables that must be protected. TZSLICER analyzes the program and applies taint analysis of sensitive variables. It then analyzes the taint analysis results and performs program slicing to extract statements that must be executed in isolation. The extracted statements are then used to form new methods that are executed in isolation. The remaining code from the original program is then instrumented so that code statements that make calls to the protected methods can be inserted. The outputs of TZSLICER include a “normal world” program that executes without hardware protection and a “secure world” program that executes under protection.

A major challenge in the design of TZSLICER is the fact that the capacity of the “secure world” must be kept low. It is because all the “secure world” resources are considered as part of the trusted computing base (TCB), which, once compromised, would propagate the security threats to the entire system. As having been a consensus in the security community, minimizing the size of TCB is an important requirement to security design [59]. We address this challenge by developing multiple program slicing schemes targeting the method (i.e., TZ-M), code block (i.e., TZ-B), and code line (i.e., TZ-L) levels. While TZ-L (i.e., the finest granularity in slicing) achieves the minimum TCB size, it introduces an increased number of context switches between the two worlds. To further address this issue, we develop an advanced version of TZ-L (i.e., TZ-L+) that conducts loop unrolling and rescheduling on the results of TZ-L to optimize the context switching overhead.
We evaluate the effectiveness and efficiency of TZSLICER using 7 real-world applications that perform common tasks in embedded systems, such as signal processing, cryptographic operations, and statistical computations. Our evaluation results indicate that TZSLICER can successfully achieve the program slicing objectives for hardware isolation and meet the security requirement. Also, the multi-level slicing and the loop unrolling mechanisms are capable of achieving controllable TCB sizes and context switching times, respectively, which are essential to accommodate for various application requirements.

Furthermore, we improve the feasibility of TZSLICER by proposing EVOISOLATOR in section 4.6. EVOISOLATOR is a self-adaptive slicing tool for HISA, which applies genetic algorithms to evolve program slices. As an extension of TZSLICER, it could adapt to the scenario where the constraints of security, resource usage, and context switching overhead keep updating at runtime. The details of TZSLICER and EVOISOLATOR are elaborated below.

4.1 Threat Models

We target two common types of threat models that have been used against embedded systems: information leakage attacks and data falsifying attacks. Both attacks have the potential of compromising the security or privacy of the critical data and programs in the embedded systems and must be prevented. Throughout the discussions in this chapter, we employ a concrete example as shown in Figure 4.1 to introduce the threat models, the countermeasures, as well as the design and implementation of TZSLICER. Without loss of generality, we assume that array $x$ contains sensitive data that is subject to security or privacy breaches under the two threat models. The program involves typical code structures (e.g., loops and branches), data structures (e.g., arrays), and operations (e.g., assignments and arithmetic computations) that commonly appear in real world C programs.
4.1.1 Information Leakage Attacks

Information leakage is one of the major forms of threats that compromise the security and privacy of critical systems [60], wherein the adversary breaches into the system, identifies the secret data, and leaks the data via an overt or covert channel. To date, there are a variety of effective methods to issue information leakage attacks, such as malware [61], memory access pattern inference [8], and hardware Trojans [15]. In the example shown in Figure 4.1, an information leakage attack can be conducted by the attacker hacking into the program call stacks involving lines 5, 8, and 14 via debugging and reading the array \( x \) values.

4.1.2 Data Falsifying Attacks

Different from the information leakage attacks that read critical data from the system, data falsifying attacks aim to modify the critical data, which may cause system malfunction or denial of services. In particular, in security-sensitive systems, such as authentication or monitoring systems, the falsified data can cause the system to bypass the otherwise mandatory security verification procedures and result in severe security compromises [62]. In the example shown in Figure 4.1, the attacker can issue a data falsifying attack directly

```
1 void M1(bool flag) {
2     if(flag) {
3         for (i = 0; i < 20; i++) {
4             b[i] = b[i] + 1;
5             y[i] = x[i] + a[i] + 1;
6             b[i] = a[i] + 2;
7             a[i] = a[i] + 1;
8             x[i] = x[i] + a[i];
9         }
10     }
11     else {
12         for (i = 0; i < 20; i++) {
13             b[i] = a[i] + b[i];
14             y[i] = x[i] + a[i];
15         }
16     }
```

```
1 int M2(int m, int n) {
2     int s = 0;
3     s = *p + *q;
4     return s;
5 }
```

```
1 void main() {
2     bool flag = true;
3     int m = 5;
4     int n = 6;
5     M1(flag);
6     int s = M2(m, n);
7 }
```
by manipulating the array $x$ values (i.e., at lines 5, 8, and 14) or indirectly by compromising the variable $flag$ (i.e., at line 2). Both cases falsify the critical data $x$ and thus endanger the security of the entire system.

### 4.2 Motivating Example

Figure 4.2 shows the most straightforward way to deploy the target program to the HISA framework and gain immediate security benefits. The entire program is embedded in the secure world and, consequently, no accesses to any part of the program can be achieved without going through the security verifications at the secure agent.

![Figure 4.2: Sliced programs by directly leveraging the HISA framework.](image)

Based on the example shown in Figure 4.2, we note that the developer must create two separate programs: the secure app and the normal app, and deploy them into the two worlds in order to leverage the HISA framework. This significantly increases the complexity of software development. In particular, the new hardware isolation architecture requires the developer to determine how to split the data and code based on their security properties, which is challenging for the general software developers who do not have security background or deep understanding about the underlying hardware framework. Furthermore, we note that the straightforward code deployment shown in Figure 4.2 results in a big size TCB (i.e., the size of the secure world) in the HISA framework, which negatively impacts
security. In this chapter, we aim to address these challenges by developing TZSLICER and EVOISOLATOR, automatic security-aware program slicing and optimizing techniques to achieve security guarantee while maintaining a small TCB.

4.3 Related Work: Program Slicing

Program Slicing [63] is a commonly used approach in software engineering, wherein a software program is partitioned into multiple pieces that satisfy certain slicing criteria to facilitate software testing, debugging, and maintenance. There are more than 30 types of program slicing methods that have been developed [63] to satisfy a variety of conditions, which fall under two basic categories: static slicing [64, 65] and dynamic slicing [66, 67]. Static slicing extracts slices from a program based on the static analysis without considering any specific input data. As such, they can suffer from imprecision that can lead to a large amount of code that must be protected in the context of our application. Dynamic slicing, on the other hand, considers the execution context and thus generates much smaller and more precise slices than static slicing. However, the generated slices are incomplete as the quality of the slices depends on the quality of the test inputs. Test inputs that provide higher code coverage would be likely to produce more slices.

4.4 TZSLICER System Framework

Figure 4.3 shows the system architecture of the TZSLICER framework, which automatically generates the secure and normal slices required by HISA for traditional software developers who do not have deep knowledge about the hardware isolation techniques applied in HISA. To achieve this goal, we develop two system components for TZSLICER: the Taint Analyzer and the Slice Optimizer. The Taint Analyzer determines the propagation of sensitive information (i.e., the user tainted variables) throughout the program following the system dependency graph (SDG) [64]. Based on the taint analysis results, TZSLICER deploys all the sensitive components into the secure world and the rest into the normal world. After
that, the Slice Optimizer further reduces two types of system runtime overhead introduced by the sliced programs, namely the resource overhead and the communication overhead, by refining the deployed code in the two slices.

![Figure 4.3: TZSLICER system framework.](image)

4.4.1 Taint Analyzer

To indicate the security requirement of the application for program slicing, the developer provides three data blocks as the inputs to TZSLICER: (1) A set of tainted variables that are the source of security sensitivities; (2) A set of test input vectors that provide a full coverage of the functional test to enable dynamic program slicing; and (3) The resource constraints that determine the key parameters in the slicing optimization.

Based on the user inputs, we employ TaintGrind, a dynamic taint analysis framework [68] to determine the propagation of taints in the SDG [64]. The originally tainted variables combined with those determined by the propagation analysis represent the critical program components that require protection. Depending on the granularity of the taint analysis, we develop the following three tainting methods.

- Method-level Tainting (TZ-M), which taints the program at the function level and generates sensitive functions and non-sensitive functions;

- Block-level Tainting (TZ-B), which further taints the internal code blocks (e.g., branches)
within the functions and generates sensitive and non-sensitive blocks; and

- Line-level Tainting (TZ-L), in which each line of the program is labeled as either sensitive or non-sensitive based on the taint propagation.

Figure 4.4 shows the taint analysis procedure and results of TZ-L for the function M1 presented in Figure 4.1, where Array x is the user tainted variable. We observe that the taint on x is propagated to 3 lines of code, including lines 5, 8, and 14, which are deemed sensitive and subject to the information leakage and data falsifying threats discussed in section 4.1. The taint analysis fully identifies the attack surfaces that have the potential of exposing the sensitive data.

Figure 4.4: System dependency graph for taint analysis for the Function M1 in Figure 4.1. The shaded nodes represent the sensitive code (line numbers) identified by the taint analysis.

4.4.2 Program Slicer

The results of the taint analysis form two sets of program components: (1) The code that is tainted, either originally by the developer or by the propagation in the SDG; and (2) The code that is not tainted. By the definition of taint under this context, Component (1) indicates the sensitive component that composes the secure slice, and Component (2) forms the normal slice, which are targeted to deploy in the secure world and the normal world, respectively.

We develop a Program Slicer that conducts the slicing and deployment of the two program slices based on the results obtained from the Taint Analyzer. An important
design principle of the Program Slicer is to ensure that the sliced programs are functionally equivalent to the original program. Such a principle cannot be achieved by directly slicing the original program into the secure and normal slices, because there exist communications between the secure and normal slices that are now split into two separate and non-concurrent CPU modes. To address the missing communication problem, the Program Slicer injects world switching code into the two slices where the communication with the other slice is needed. As discussed in the HISA framework, the communication between the two worlds is through the shared memory and, therefore, the injected world switching code involves two parts:

- Shared memory access, in which the program slice loads or stores data to or from the shared memory, enabling the communication with the other slice; and

- World switching, in which the program slice in one world issues a secure monitor call that causes the secure monitor in the HISA framework to switch the CPU mode to the other world.

Upon accomplishing the memory access and the world switching operations, both the shared data and the CPU mode are switched from the source world to the destination world, which concludes one complete world switching. In summary, the program slicing methods achieve secure world resource savings with the consequence of possibly increased communication overhead in certain cases. After slicing, the information flow of the sliced programs remains the same as the original program.

4.4.3 Slice Optimizer

To address the system resource challenge, we employ dynamic taint analysis that is dependent upon the specific executions of the sliced programs, which has been shown to achieve significantly smaller slices than static slicing [69]. The dynamic taint analysis method leverages the user provided test input vectors and propagates the taints only if the corresponding
code exists in the execution path determined by the test input vectors. Consequently, the secure world only hosts sensitive data and code that will be executed at system runtime, which reduces a significant amount of resource usage as compared to the static method. Figure 4.5 and Figure 4.6 demonstrate the complete program slicing process following the code example in Figure 4.1, corresponding to the three tainting methods TZ-M, TZ-B, and TZ-L.

• TZ-M Slicing. As shown in Figure 4.5, TZ-M Slicing applies the TZ-M tainting method, which places the tainted function $M1$ into the secure world and the non-tainted $M2$ function into the normal world. Compared to the baseline slicing method in Figure 4.2, TZ-M Slicing reduces the resource overhead of the secure world by the size of the non-tainted $M2$ function. Consequently, since $M2$ is now in the normal world, a world switch together with shared memory accesses are required to maintain the original program flow and functionality.

• TZ-B Slicing. As shown in Figure 4.5, TZ-B Slicing leverages TZ-B tainting that further carves the unexecuted “else” branch in $M1$ out of the secure world, which can be achieved by considering the dynamic $flag$ value provided by the user ($flag = 1$). As can be
observed from Figure 4.5, TZ-B Slicing achieves direct resource savings in the secure world (by the size of the “else” branch in $M1$) without increasing the communication overhead.

• TZ-L Slicing. As shown in Figure 4.6, TZ-L Slicing further reduces the secure world resource usage by tainting into the level of each line of code within the $M1$ function. Based on the TZ-L tainting method demonstrated in Figure 4.4, the lines 5, 8, and 14 are tainted and migrated to the secure world. After the slicing, in order to maintain the original program flow, the sliced programs must conduct 2 round-trip switches as shown in Figure 4.6.

![Figure 4.6: TZ-L Slicing.](image)

To reduce the communication overhead, we employ a dynamic scheduling mechanism, which manipulates the order of the instruction execution at the runtime of each world to reduce the number of communications between the two worlds. To achieve this goal, the dynamic scheduling mechanism involves two systematic approaches, namely loop unrolling and code reordering. The loop unrolling scheme applies to the sliced program with loop structures, which fully or partially decomposes the loops and enables the merge of multiple iterations to reduce the number of world switches. The code reordering mechanism further reorders and merges the code within each iteration to reduce communications under the constraints of data dependencies.
Communication Optimization 1: Loop Unrolling. Similar to the benefits obtained in other domains [70, 71], loop unrolling in the context of TZSLICER has the potential of breaking the restricted boundaries enforced by the iterations, which enables flexible and thus less frequent world switches. Figure 4.7 demonstrates the benefit of loop unrolling using the same example in Figure 4.1 while unrolling 2 iterations of the loop at a time. We observe that although the number of switches per iteration remains 4, the total number of iterations is reduced by half and, therefore, the total number of switches is reduced by half after unrolling. Without loss of generality, we define an unrolling parameter \( x \) to represent the number of iterations being unrolled. In other words, the total number of switches is reduced by \( x \) times in the ideal case.

![Secure World](image1)

![Normal World](image2)

Figure 4.7: TZ-L+ Slicing using loop unrolling.

The applicability of loop unrolling is restricted by several factors. First, the loop must have been split partially into the secure and partially into the normal world with SMCs for communication. Second, the code in the next iteration must be predictable; in other words, there are no branch statements present in the loop. Third, there are no data dependencies in the loop body, as otherwise it is not possible to execute the code line of next iteration right after the one in the current iteration. For the programs that do not meet the first two requirements, we downgrade TZ-L+ to TZ-L, while for those that do not meet the third requirement, we employ a variable renaming technique to eliminate the dependency.
Communication Optimization 2: Variable Renaming. In the cases of data dependency, we employ a variable renaming approach to eliminate the dependency and maintain the capability of unrolling and scheduling. The variable renaming approach works in the following two steps:

- **Step 1: Dependency Detection**, wherein we find variables that are subject to data dependencies, including read after write (RAW), write after read (WAR), and write after write (WAW) in the current iteration. Figure 4.8 shows an example (extracted from the DAXPY test program discussed in section 4.5), where line #4 has an RAW dependency on lines #2 and #3. Consequently, the original loop unrolling method will fail as the lines #2 and #3 cannot be re-scheduled to the current iteration due to the data dependency.

- **Step 2: Variable Renaming**, Once a data dependency is detected, we rename the variables subject to the dependency by expanding the variable name with iteration numbers, as shown in Figure 4.8. The renaming keeps the variables from being overwritten and thus eliminates the dependency.

![Figure 4.8: TZ-L+ variable renaming under data dependency.](image-url)
4.5 Experimental Evaluation

4.5.1 Experimental Setup

We adopt 7 real-world C programs to evaluate TZSLICER, as shown in Table 4.1. The programs involve the functionalities of signal processing, cryptography, and statistical computations, where there exist sensitive variables that are subject to information leakage and/or data falsifying attacks. Table 4.1 also summarizes the statistics of the programs. We observe that the programs cover a diverse set of test cases with varying lines of code (LoC), branch statements, loops, and functions, which collectively provide a comprehensive test set for evaluating different slicing methods supported by TZSLICER.

Table 4.1: Test programs adopted to evaluate TZSLICER.

<table>
<thead>
<tr>
<th>Test Cases</th>
<th># Lines</th>
<th>Branches</th>
<th>Loops</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>83</td>
<td>3</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Sobel_Filter</td>
<td>121</td>
<td>8</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>Matrix_Multiplication</td>
<td>26</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>AES_KeyExpansion</td>
<td>81</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Linear_Regression</td>
<td>40</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Shift_Cipher</td>
<td>57</td>
<td>8</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>DAXPY</td>
<td>33</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

4.5.2 Security Analysis

Technically, TZSLICER is capable of defending against the aforementioned information leakage and data falsifying attacks benefiting from the strong hardware bus-level isolation provided by HISA. This is possible because attackers are now restricted from accessing the resources deployed in the secure world based on the user-specified tainted variables. Without loss of generality, we discuss the following three cases regarding taint propagation for security analysis in TZSLICER.

- Assignment Statements, which is the most straightforward way of taint propagation, where the taint propagates from the variable(s) being read (i.e., the right side of the assignment
statement) to the those being written (i.e., the left side of the assignment statement). Based on our observation on the implemented prototype system, the Taint Analyzer is able to capture and taint all the variables being written, and the program slicer is thus able to place the code lines containing all those variables in the secure world (i.e., in the TZ-L case). Therefore, there are no security vulnerabilities that can be exploited due to the assignment statements.

- **Sub-function Calls**, which is also a common way of taint propagation. In this case, TZSLICER taints the arguments of the function that obtain assigned values from the user tainted variables, which ensures that the operations related to sensitive data are still protected (i.e., placed in the secure world) in the sub-functions.

- **Pointer Propagation**, where a tainted pointer instead of the value it points to is assigned to a new variable, via either assignment statements or sub-function calls. In this case, TZSLICER taints the new variable only if it is referencing the values of the corresponding pointer other than the pointer itself. In this way, the value of the sensitive variable will be protected, and even if the attacker has access to the pointer value, it is still impossible to access the value it points to.

In summary, we observe that TZSLICER is able to handle all the above channels where a sensitive variable has the potential of being leaked.

### 4.5.3 Quantitative Security Evaluation: TCB Size

We further evaluate the security of TZSLICER quantitatively by measuring the size of TCB, which is represented by the lines of code deployed in the secure world. Table 4.2 and Table 4.3 summarizes the results we obtained by running TZSLICER with the 7 test programs shown in Table 4.1. We first taint a random input variable in each program. Then, we evaluate all the proposed slicing methods, including TZ-M, TZ-B, TZ-L, and TZ-L+ (with various unrolling parameters) and compare them with the baseline approach.
Table 4.2: Resource usage evaluation results represented by the number of lines of code for TZ-M, TZ-B, and TZ-L. The percentage value in parenthesis represent the relative savings from the baseline, as described in Equation 4.1.

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>Original (Baseline)</th>
<th>TZ-M</th>
<th>TZ-B</th>
<th>TZ-L</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>83</td>
<td>83 (0%)</td>
<td>80 (-3.6%)</td>
<td>60 (-27.7%)</td>
</tr>
<tr>
<td>Sobel_Filter</td>
<td>121</td>
<td>121 (0%)</td>
<td>121 (0%)</td>
<td>106 (-12.4%)</td>
</tr>
<tr>
<td>Matrix_Multiplication</td>
<td>26</td>
<td>26 (0%)</td>
<td>17 (-34.6%)</td>
<td>19 (-26.9%)</td>
</tr>
<tr>
<td>AES_KeyExpansion</td>
<td>81</td>
<td>49 (-39.5%)</td>
<td>40 (-50.6%)</td>
<td>42 (-48.1%)</td>
</tr>
<tr>
<td>Linear_Regression</td>
<td>40</td>
<td>40 (0%)</td>
<td>27 (-32.5%)</td>
<td>24 (-40%)</td>
</tr>
<tr>
<td>Shift_Cipher</td>
<td>57</td>
<td>57 (0%)</td>
<td>15 (-73.7%)</td>
<td>15 (-73.7%)</td>
</tr>
<tr>
<td>DAXPY</td>
<td>33</td>
<td>33 (0%)</td>
<td>17 (-48.5%)</td>
<td>16 (-51.5%)</td>
</tr>
</tbody>
</table>

We observe from Table 4.2 that TZ-M achieves significant savings in the AES_KeyExpansion test case that involves non-tainted functions, while it does not contribute to the other cases where there are either too few functions or all of the functions are tainted. TZ-B reduces the TCB sizes for 6 out of the 7 programs. TZ-L achieves additional improvement by
cutting down the TCB size at the level of code lines, which results in TCB savings in all 7 examples (ranging from 12.4% to 73.7%) compared to the baseline. TZ-L+ increases the TCB sizes compared to TZ-L due to the code unrolling, which grows linearly with the unrolling parameter. Note that in programs where there are no loops or no world switching within the loops, including Matrix_Multiplication, AES_KeyExpansion, and Shift_Cipher, TZ-L+ do not apply and thus result in the same TCB sizes as compared to TZ-L.

4.5.4 Performance Evaluation: World Switches

The migration of the partial program into the secure world results in additional communications (i.e., world switches) between the two worlds in order to maintain the original functionality, which introduces additional timing overhead compared to the original program without HISA protection. Therefore, we quantitatively evaluate the communication overhead of the programs generated by TZSLICER. Table 4.4 shows our evaluation results represented by the number of world switches (i.e., SMC calls) using the 7 test programs. Since TZ-M and TZ-B do not require world switches below the function level, we set the results of TZ-L as the baseline for comparison. We observe that in 4 of the test programs, TZ-L+ is capable of reducing the communication overhead significantly (ranging from 7.8% to 63.2%) thanks to the benefit of unrolling. The other 3 examples, namely Matrix_Multiplication, AES_KeyExpansion, and Shift_Cipher, do not show improvements because they either do not have loops or do not contain world switches within the loops.

4.5.5 Discussion: Trade-off Between Security and Performance

Our evaluation results in Table 4.2, Table 4.3, and Table 4.4 indicate that there is a trade-off between the TCB size (i.e., security) and the number of world switches (i.e., performance). To better understand the trade-off, we plot the correlations between the normalized values of the two metrics in Figure 4.9 while varying the unrolling parameter in TZ-L+ from 2 to 4. The curves for the 4 test programs confirm the general trade-off between security and
Table 4.4: Evaluation of context switching overhead. The percentage value in parenthesis represent the relative savings from the baseline.

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>TZ-M</th>
<th>TZ-B (Baseline)</th>
<th>TZ-L (x=2)</th>
<th>TZ-L+ (x=3)</th>
<th>TZ-L+ (x=4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>0</td>
<td>0</td>
<td>77</td>
<td>68 (-11.7%)</td>
<td>71 (-7.8%)</td>
</tr>
<tr>
<td>Sobel_Filter</td>
<td>0</td>
<td>0</td>
<td>729</td>
<td>471 (-35.4%)</td>
<td>351 (-51.9%)</td>
</tr>
<tr>
<td>Matrix_Multiplication</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>AES_KeyExpansion</td>
<td>0</td>
<td>0</td>
<td>101</td>
<td>101 (0%)</td>
<td>101 (0%)</td>
</tr>
<tr>
<td>Linear_Regression</td>
<td>0</td>
<td>0</td>
<td>19</td>
<td>10 (-47.4%)</td>
<td>7 (-63.2%)</td>
</tr>
<tr>
<td>Shift_Cipher</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>DAXPY</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>14 (-30.0%)</td>
<td>8 (-60.0%)</td>
</tr>
</tbody>
</table>

performance, which provide reference to the developers to fine tune TZSLICER based on the application-specific security and performance requirements.

![Timing versus resource usage](image)

Figure 4.9: Timing versus resource usage (normalized) under TZ-L and different unrolling parameters (x = 2, 3, 4) in TZ-L+.

4.6 EVOISOLATOR: Evolving Program Slices for TZSLICER

Figure 4.10 shows our vision of EVOISOLATOR. It has two primary optimization steps. First, it determines and synthesizes the slice (TZSurgeon). Then, it optimizes that slice for performance (TZOptimizer). EVOISOLATOR starts with the original program and a set
of sensitive variables. It generates random initial secure and normal slices that pass a security test suite. It then transplants the sensitive computations into the secure world and the remaining computations into the normal world using genetic programming to find the best code configuration. In TZOptimizer it reorders the code to reduce communication overhead.

The EvoISOLATOR chromosome includes both secure and normal code blocks. The code blocks are split by the TrustZone SMC in HISA. A test suite used for fitness contains input-output pairs designed to detect information leakage and other traditional bugs. A second fitness function is added in TZOptimizer for reordering which counts the number of switches between the secure and normal worlds.

Fitness Function. We present a prototype fitness function for TZOptimizer:

\[
    f = \begin{cases} 
    -1000 & \text{if the program does not compile} \\
    w_1 \cdot P_{IO} - w_2 \cdot S & \text{if the program successfully compiles}
    \end{cases}
\]

\(P_{IO}\) indicates the number of the input-output pairs that pass the test suite, and \(S\) indicates the number of world switches. \(w_1\) and \(w_2\) indicate the weights for \(P_{IO}\) and \(S\), respectively. We leave normalization and tuning as future work.

To evaluate the feasibility of EvoISOLATOR, we implemented a version of the second phase, TZOptimizer, using TZSlicer as the input module. We leave TZSurgeon as future work. We demonstrate our approach with an example (Figure 4.11(a)).
This example contains the *add* and *sign* functions. The variable *a* in the *add* function is the secure (tainted variable). TZSlicer treats the lines 5, 8, 11, and 13 as sensitive computations (shown with the red text boxes). TZSlicer partitions the program as is seen in Figure 4.11(b), placing the lines 5, 11, and 13 from the original program into the secure world and removing the redundant/non-executed code (e.g., line 8). The arrows indicate the world switching flow.

Using the secure and normal slices generated by TZSlicer, *EvoISOLATOR* moves the slices to the secure and normal code blocks. To generate the fitness test suite, the code blocks are merged into a sliced program that is executable in a regular C environment (shown in Figure 4.12). The *SMC* lines in the sliced program count the number of world
switches in each loop iteration. In all, there are 31 switches in the initial sliced example.

TZOptimizer then tries to optimize solutions that pass all the test cases and achieves the minimum number of world switches. Figure 4.13 shows one of the crossover operations and one of the mutation operations for this phase. In Figure 4.13(a), assume that TZOptimizer randomly picks two secure slices from the two chromosomes. By randomly selecting a crossover point, it swaps the code blocks in the chromosome parents and generates the offspring. In Figure 4.13(b), assume that the mutation point is a line of the sensitive computation. TZOptimizer splits the target code block to two code blocks.

![Crossover and mutation examples](image)

Figure 4.13: Crossover and mutation examples.

We built a version of TZOptimizer as a genetic algorithm in Python. We first generate 50 test cases (input-output pairs) based on the original program. Then, we input the secure and normal slices generated by TZSlicer to the TZOptimizer part of EVOISOLATOR. We use a population size of 12 based on some initial experiments. We set $w_1 = 10$ and $w_2 = 1/50$ for the weights in the fitness function. After executing multiple runs, EVOISOLATOR outputs two solutions, which reduce the original 31 switches (from TZSlicer) to 21 switches as is shown in Figure 4.14. Solution 1 moves the line $e = a + 2$ backward and still keeps this line within the loop computation. In addition, EVOISOLATOR detects that it is unnecessary to place this line inside of the loop. Therefore, the second solution moves this line forward to the outside of the loop, which further reduces the resource usage during the computation and improves the efficiency of the program execution.

We ran the program 100 times to understand if it converges on a solution each time. We
found that the number of generations to find this solution was usually less than 3, and in all cases we found a better solution. While this is a simple example we believe this can scale to larger programs.

Figure 4.14: Optimized example generated by EvoISOLATOR.
CHAPTER 5
HYBRID TEE WITH PROTECTIVE PERTURBATION-BASED PRIVACY PROTECTION

After addressing Security-RQ and Programming-RQ for HISA, we aim to resolve Scalability-RQ in this chapter. HISA is a singular TEE platform and could be insufficient to execute large applications, such as deep learning-based image recognition due to the limited TEE resources on the local devices. For example, there are only a few megabytes available in the default OP-TEE secure world size on a Raspberry Pi [72]. However, the size of a DNN model could be at least dozens of megabytes [73]. To address Scalability-RQ, we extend our HISA work to a local-cloud HybridTEE framework as shown in Figure 5.1, which offloads intensive computations to the cloud for deep learning-based image recognition applications. In this framework, the computations could contain private data that needs to reside on the user end for privacy protection. Therefore, HybridTEE applies a privacy-preserving partitioning algorithm to divide the computations into a privacy-sensitive portion and a non-privacy-sensitive portion, which are executed by the local TEE and the cloud TEE, respectively. Specifically, the partitioning algorithm applies two methods to evaluate the privacy exposure and determine the partition point. The first method is applying an auxiliary DNN model to detect objects in the input image and the intermediate images at each layer. If the auxiliary model fails to detect objects in all the intermediate images at layer \( x \), it selects layer \( x \) as the partition point. The second method is applying scale invariant feature transform (SIFT) [74] to evaluate the matched key points between the input image and the intermediate images at each layer. If the matched key points at layer \( y \) are less than a pre-defined threshold, it selects layer \( y \) as the partition point.

Furthermore, to maintain the original functionality of the computations, HybridTEE builds a secure communication protocol between the local TEE and the cloud TEE. The local
TEE first conducts a remote attestation to authenticate the cloud TEE. Then, it exchanges cryptographic keys with the cloud TEE, so that the local and cloud TEEs could communicate with encrypted data to ensure confidentiality. After the attestation and key exchange, the user first provides the input image to the local TEE to execute the privacy-sensitive computations. Then, the local TEE encrypts and sends the results to the cloud TEE. The cloud TEE decrypts the results from the local TEE, executes the non-privacy-sensitive computations, and sends the results back to the local TEE after encryption. Finally, the local TEE generates the inference results for the user. More details about the HybridTEE framework can be found in [75].

![HybridTEE System Framework](image)

Figure 5.1: HybridTEE system framework.

However, in reality, the deep learning-based image recognition models are often propriety assets with intensive computations, which cannot be deployed, even partially, at the user end [76]. In other words, the entire model must stay at the service provider end to protect its intellectual property. Under this circumstance, we cannot apply the passive partitioning algorithms in HybridTEE to protect user privacy, as they require certain layers of the DNN to reside on the user end. To address this limitation, one category of approaches focuses on proactive image protection, such as homomorphic encryption [77, 78, 79], which encrypts the original image to eliminate information exposure and, more importantly, without requiring the service provider to decrypt the protected image for the required computations. Therefore, the privacy-sensitive information is never exposed to the untrusted parties for image recognition. However, such homomorphic encryption mechanisms consume huge
computational resources and incurs high performance overhead in both the training and inference phases [80, 81], which are hardly applicable to real-time image recognitions.

To balance between privacy protection and system efficiency, several recent works leverage differential privacy and image blurring/pixelation to conceal the sensitive features in the images [82, 83, 84, 85]. However, the effectiveness of these approaches relies on the accurate identification of all the sensitive features in the target image for hiding and blurring, which is technically difficult to achieve and hardly adaptable to different applications that consider different features as sensitive information.

To address the aforementioned limitations in HybridTEE and the existing state-of-the-art approaches, we develop a novel privacy protection approach for image recognition applications using protective perturbation. Our proposed protective perturbation is a proactive pixel mask being added to the privacy-sensitive input image, which achieves two objectives: (1) The target image is blurred to eliminate the exposure of sensitive information (i.e., invisible to human vision); and (2) The blurred image can still be directly adopted by the image recognition model for the original deep learning computations with no interruption of accuracy degradation (i.e., visible to machine vision). Such protective perturbations are generated by a real-time generative neural network model, which progressively optimizes for a loss function to minimize the similarity between the original and perturbed images (i.e., visibility to human vision) and maximize the accuracy of the image recognition model (i.e., visibility to machine vision). In a nutshell, our proposed protective perturbation approach aims to achieve both the goals of image blurring [82, 83, 84, 85] and computing on encrypted data (e.g., homomorphic encryption [77, 78, 79]) but without being subject to their technical difficulties and the limitations of the partitioning algorithms in HybridTEE. For example, the protective perturbation does not rely on the identification of individual features (i.e., required in traditional image blurring) and thus can be made universal to adapt to different applications with distinct sensitive features. Also, the perturbed images can be directly handled by the deep learning image recognition model, as if they were the original
images, without increasing the complexity and running time of the computation.

To achieve the aforementioned design goals, our protective perturbation generator applies a generative neural network model, namely U-Net [86], to generate the protective perturbations. The loss function to train the generative model contains three components, including the accuracy loss of the original image recognition model (i.e., target model), an auxiliary model that mimics human vision and helps perturb the original image, and the similarity loss between the original image and the perturbed image.

The successfully protected images should maintain high prediction accuracy on the target model (i.e., high machine vision) without retraining, low prediction accuracy on the auxiliary model, and low similarity comparing with the original images (i.e., low human vision). In our empirical evaluations on 8 target models with 7 auxiliary models and the SSIM metric (i.e., structural similarity index measure) [87], the proposed method can generate the protective perturbations successfully in real-time (< 2.4ms per image) without hardware acceleration. To summarize, we have made the following key technical contributions in this chapter:

• We propose a novel protective perturbation-based approach to achieve privacy preserving image recognition, which addresses the limitation of the partitioning algorithms in HybridTEE;

• We implement the perturbation generator using a generative neural network model and deploy it in real systems; and

• We conduct comprehensive evaluations on the implemented perturbation generator using 8 popular image recognition models to justify the effectiveness and real-time performance of the proposed protective perturbation approach.
5.1 Background

5.1.1 Image Recognition System

Deep learning-based image recognition is an important and popular category of multimedia application that has been adopted in various domains. For example, eBay Image Search and Google Lens [88, 89] could identify objects in the input images and provide users with metadata of the detected objects (e.g., purchase links and customer reviews). In healthcare systems, image recognition has been adopted to assist medical diagnosis, such as tumor detection [90]. However, the input images in such applications could contain sensitive information (e.g., human faces, license plates, and medical records), which raises privacy concerns when the image recognition is handled by untrustworthy service providers (e.g., a public cloud). Also, the sensitive information is often subject to strict privacy regulations (e.g., HIPAA [91] and GDPR [92]). Therefore, from the perspective of multimedia system design, it is crucial to take privacy protection into consideration and minimize the exposure of sensitive information in the image recognition applications.

Figure 5.2 illustrates an image recognition system targeted by this chapter, which identifies the class of an input image by leveraging deep learning technology. First, the user provides the input image and sends the image to the service provider. Then, the service provider leverages a deep learning-based image recognition model to recognize the image. The model contains an input layer, hidden layers, and an output layer, which is trained by a training dataset and a validation dataset with parameter tuning. The input layer processes the input image and sends the intermediate results to the neurons (i.e., nodes in Figure 5.2) with weights (i.e., edges in Figure 5.2) in the hidden layers. After the hidden layers, the output layer outputs the recognition confidence for each class (e.g., 1% as a dog, and 91% as a car). Finally, the service provider sends the image recognition/inference result (i.e., the class with the highest recognition confidence) to the user.
5.1.2 Privacy Issues in Image Recognition

In such image recognition system, there exist threat models that could compromise user privacy with regard to the input image that may contain sensitive information. When the user sends the input image to the untrustworthy service provider, the adversary could intercept the image from the network channel or, alternatively, compromise the service provider to gain access to the image. Then, the adversary could breach the privacy of the victim user by recognizing the sensitive information in the image [81, 82, 83, 84, 85]. Even if the information, such as human faces and license plate numbers, is blurred for privacy protection, the image background and user’s outfit could still potentially disclose locality and demographic information [93, 94]. Note that this chapter assumes that the user is trustworthy and only focuses on preventing privacy leakage from the input images during the execution of image recognition. Other threat models regarding information leakage at other phases, such as when training the deep learning models or from the recognition results, are out of the scope for this chapter.
5.2 Related Work

5.2.1 Privacy Preserving Image Recognition

To address the aforementioned privacy issues, several solutions have been proposed in the community to protect the privacy in image recognition, including three categories:

**Model Partitioning.** Several research works [81, 93, 95] have shown that the input image and the intermediate results during the DNN execution could contain user's privacy-sensitive information. To maintain the private inference, partial layers that could leak user privacy must be deployed at the user end. However, the model could be trained by a confidential dataset, which is considered a propriety asset [76]. Under this scenario, all the layers in the model are required to execute at the service provider end. Therefore, further privacy-preserving solutions are required to resolve the challenge of protecting both the user privacy and the model intellectual property, which are discussed next.

**Homomorphic Encryption.** Another privacy protection approach is homomorphic encryption, which is different from the traditional encryption in that it does not require decryption to perform the computations. By leveraging homomorphic encryption, the service provider can perform the image recognition tasks (e.g., face recognition) on the encrypted images directly, and only the user can access the decrypted images [77, 78, 79]. However, these methods require huge computational resources in both the training and recognition processes, which compromises the efficiency of the image recognition system [80, 81].

**Image Blurring/Pixelation.** Image blurring/pixelation has been one of the popular solutions to protect the sensitive features in the image. For example, Fan and Sun et al. apply differential privacy to pixelate the sensitive features to protect user privacy [82, 83]. Also, the works of [84, 85, 94, 96] de-identify the recognized faces or license plates in the images/videos to avoid privacy exposure. However, these methods only protect specific sensitive features in the image, such as human faces and license plates. For the images
containing a variety of features, the method needs to be redesigned.

5.2.2 Adversarial Attacks

Adversarial attacks, such as PGD [97] and C&W [98], have been studied in the deep learning and security communities recently. In adversarial attacks, very small perturbations (i.e., almost invisible to the human visual perception) added to the input image could significantly alter the DNN inference results (i.e., misleading the machine vision), since the human visual perception is different from the machine vision [99, 100]. Inspired by the adversarial attacks, there have been a small number of research works that leverage the adversarial perturbations as a defense mechanism to prevent private data leakage [101, 102, 103]. These approaches proactively add adversarial perturbation to the objects (e.g., video and audio) under protection, which would mislead machine learning-based threat models (e.g., face authentication attack) but maintain the original functionality of the application intended for human vision. Different from such benign use of the adversarial perturbations, our proposed protective perturbation approach target an opposite optimization direction, which maximizes the perturbations to interfere the human vision for privacy protection but minimizes the impact to machine vision (i.e., DNN-based image recognition).

Figure 5.3: Privacy-preserving image recognition system workflow with the protective perturbation generator.
5.3 Protective Perturbation

5.3.1 System Overview

To address the privacy issues in image recognition applications, we develop a protective perturbation generator at the user end to proactively add human-visible but machine-invisible masks to the input images. Such protective perturbations would help hide the privacy-sensitive information from being exposed to adversaries but still maintain the original functionality of the image recognition application. Figure 5.3 shows the overall workflow of the proposed privacy-preserving image recognition system involving three parties with various system components and operations: (1) At the user end, the proposed perturbation generator processes the original input images and injects the protective perturbations before offloading it to the service provider for image recognition; (2) At the service provider end, since the injected protective perturbation transparent to the target deep learning-based image recognition model, the image recognition computations would proceed as normal without impacting the accuracy and speed of the inference results; and (3) The adversary attempts to gain access and breach the privacy-sensitive information in the perturbed images; however, such efforts are expected to fail given the protective perturbations.

In the end-to-end workflow, the perturbation generator plays a central role in adding privacy-preserving protective perturbations, which are expected to be non-intrusive to both the user and the service provider of the original image recognition application, but intrusive to the adversaries:

- The non-intrusive objective for the user and the service provider are two-fold: (1) From the perspective of model deployment, the protective perturbation-based approach should eliminate the need to retrain or alter the original image recognition model in any form, presenting an non-intrusive privacy-preserving solution that can be immediately deployed in the commodity systems today; and (2) From the perspective of runtime execution, the protective perturbation should maintain the original accuracy of the image recognition
and, more importantly, the additional delay incurred by the perturbation generation should not be noticeable by the user or jeopardise the real-time requirement of the original application.

- The intrusive objective for the adversary requires that the input images obfuscated by the protective perturbation are not perceivable to human vision, defeating the adversary’s intent to obtain any privacy-sensitive information.

The above non-intrusive and intrusive objectives appear to be opposite and contradicting to each other. However, there exists a clear distinction between the contexts of the two objectives in that the non-intrusive objective targets machine learning models/algorithms, while the intrusive objective targets human vision. Even though the goal of machine vision has always been to approach and emulate human vision, there is still a big enough gap between them that can be leveraged to achieve distinct effects on the two vision systems with the same set of inputs. As an evidence, the recent developments in the AI community on adversarial attacks [97, 98, 104, 105] could successfully generate perturbations to the input data that lead to erroneous inference results by the machine but remain unnoticeable by human. This clearly justifies the feasibility of leveraging the gap between machine and human visions to achieve the distinctive objectives and thus forms the foundation of our proposed protective perturbation generation method, as described next.

### 5.3.2 Protective Perturbation Generation

Figure 5.4 shows our proposed protective perturbation generation process. We first pre-process the original image $x_i$ (e.g., by random cropping, random horizontal flipping, and normalization). Then, we feed $x_i$ to a generative neural network model, namely U-Net [86], which has been adopted for generating traditional adversarial perturbations [104, 106]. We define the U-Net model as $U(\cdot)$, and it generates the perturbation $\delta$ that aims to hide image features. With $\text{clamp}(\cdot)$ that clips $x_i + \delta$ into a valid pixel range, we have the protected image $x'_i = \text{clamp}(x_i + \delta)$ that protects user privacy. Using only one forward propagation, U-Net
avoids time-consuming iterations to produce outputs, which makes real-time generation possible.

![Figure 5.4: The process of generating a protected image by the U-Net model.](image)

The major challenge in applying U-Net to our protective perturbation generation lies in the aforementioned intrusive and non-intrusive objectives that are opposite to the traditional adversarial attacks. In particular, our protective perturbation aims to maximize the perturbation to human vision and minimize the impact to the image recognition model, while the U-Net model for traditional adversarial attack aims to minimize the former and maximize the latter. To summarize, we have two objectives in the design of the protective perturbation generator, consistent with the discussions in subsection 5.3.1:

- Non-intrusive objective - Maintain the machine vision, which can be formulated as $\text{Accuracy}(F_{\text{target}}(x'_i)) \rightarrow \text{Accuracy}(F_{\text{target}}(x_i))$, assuming the original image recognition model is the target model $F_{\text{target}}(\cdot)$; and
• Intrusive objective - Maximize the human-perceivable perturbation, which can be formulated as $\text{Accuracy}(F_{\text{adv}}(x')) \rightarrow 0$, assuming $F_{\text{adv}}(\cdot)$ mimics the adversarial recognition.

The following subsection elaborates on how to achieve these two goals simultaneously and train an effective and efficient protective perturbation generation model $U(\cdot)$.

5.3.3 Training the Perturbation Generator

Figure 5.5 shows the training process of our proposed protective perturbation generator. We first provide the original training images to an initial $U(\cdot)$ to generate the initial protected images. Then, we feed the original images and the protected images into the target image recognition model. The target model provides the true labels of the original images and predicted labels of the protected images to the $\text{CrossEntropy}$ loss function, which helps achieve the aforementioned non-intrusive objective. Let a true label be $y_i$ and the corresponding label predicted by the target model be $y_{\text{pred\_target}}$. We have the loss calculation as follows:

$$\text{Loss}_{\text{target}} = \text{CrossEntropy}(y_{\text{pred\_target}}, y_i)$$

(5.1)

We leverage two modules to achieve the aforementioned intrusive goal. First, we apply the original images and the protected images to an auxiliary model, which is an image classification model to evaluate the classification results seen by the adversary. The auxiliary model provides the true labels of the original images and the predicted labels of the protected images. The $\text{CrossEntropy}$ loss function calculates the loss value for the auxiliary model, which is applied a negative sign to maximize the loss between the two sets of the labels and achieve the aforementioned intrusive goal. The loss function represented by the auxiliary model is defined as:

$$\text{Loss}_{\text{aux}} = -\text{CrossEntropy}(y_{\text{pred\_aux}}, y_i)$$

(5.2)

Second, we apply a similarity function to determine the human-perceivable difference
between the original images and the protected images to ensure that the two images are visually different and prevent privacy exposure. We select structural similarity index measure (SSIM) metric as our similarity function, since it is one of the most popular metrics to evaluate the similarity between two images and is similar to human visual perception [87]. The loss function represented by the SSIM metric is defined as:

\[ \text{Loss}_{SI} = \text{SSIM}(x_i, x'_i) \]  

Finally, the optimizer obtains the total loss value and updates the weights on the U-Net model. The final loss function is formulated as follows:

\[ \text{Loss} = \omega_{\text{target}} \times \text{Loss}_{\text{target}} + \omega_{\text{aux}} \times \text{Loss}_{\text{aux}} + \omega_{SI} \times \text{Loss}_{SI}, \]  

where \( \omega_{\text{target}}, \omega_{\text{aux}}, \) and \( \omega_{SI} \) are the pre-set coefficients to control the effect of each loss component. Algorithm 1 shows the detailed pseudocode of the presented model training process. The training keeps executing and updating \( U(\cdot) \) until it reaches \( \text{max\_epochs} \). Finally, a successfully protected image \( x'_i \) generated by \( U(\cdot) \) requires to fulfill the following requirements at the same time:

1. \( F_{\text{target}}(x'_i) = y_i \), where the predicted label of the target model is the same as the true label \( y_i \);
2. \( F_{\text{aux}}(x'_i) \neq y_i \), where the predicted label of the auxiliary model is different from the true label; and
3. Minimal \( \text{Loss}_{SI} \), where the similarity level between \( x_i \) and \( x'_i \) should be as low as possible.
Algorithm 1: Protective Perturbation Generator Training Algorithm

Input: Training dataset $X = \{x_1, \ldots, x_m\}$, true labels $Y = \{y_1, \ldots, y_m\}$, target model $F_{\text{target}}(\cdot)$ and its loss weight $\omega_{\text{target}}$, auxiliary model $F_{\text{aux}}(\cdot)$ and its loss weight $\omega_{\text{aux}}$, weight of the similarity loss $\omega_{SI}$, and the maximal epochs $\text{max\_epochs}$.

Result: Trained protective perturbation generator $U(\cdot)$.

Randomly initialize $U(\cdot)$.

for $j \leftarrow 1$ to $\text{max\_epochs}$ do

foreach $x_i, y_i$ in $X, Y$ do

$\delta \leftarrow U(x_i)$;

$x_i' \leftarrow \text{clamp}(x_i + \delta)$;

$y_{\text{pred\_target}} \leftarrow F_{\text{target}}(x_i')$;

$y_{\text{pred\_aux}} \leftarrow F_{\text{aux}}(x_i')$;

$\text{Loss} \leftarrow \omega_{\text{target}} \cdot \text{CrossEntropy}(y_{\text{pred\_target}}, y_i) - \omega_{\text{aux}} \cdot \text{CrossEntropy}(y_{\text{pred\_aux}}, y_i) + \omega_{SI} \cdot SI(x_i, x_i')$;

minimize $\text{Loss}$ to update $U(\cdot)$;

end

end

5.4 Experimental Evaluation

5.4.1 Experimental Setup

Dataset. We train and test our protective perturbation generator based on the CIFAR-10 dataset [107]. We use the original CIFAR-10 training dataset (50k images) and split the CIFAR-10 test dataset (10k images) into two halves as our validation dataset and test dataset. The dataset is provided by the torchvision library [108]. We preprocess the images in the dataset following [109].

Target and Auxiliary Models. We evaluate 8 different neural network models pre-trained by [109] as our target image recognition models, which include VGG13$_{\text{bn}}$, VGG16$_{\text{bn}}$ [110], ResNet18, ResNet34 [111], DenseNet121 [112], MobileNet_v2 [113], GoogLeNet [114], and Inception_v3 [115]. For each target model, we apply the 7 other models as the auxiliary models.

Training and Validation Processes. First, we define loss parameters based on Equation 5.4 in subsection 5.3.3. In the loss function, $\omega_{\text{target}}$, $\omega_{\text{aux}}$, and $\omega_{SI}$ are 1.0, 1.0, 0.5,
respectively. In the training process, we use AdamW [116] as our optimizer, where the learning rate is 0.005 and the weight decay is 0.0005 with other parameters remaining as default. We define the batch size as 256, maximum epoch number as 100, number of workers as 8, and precision bit as 16. After the training process, the validation process selects the best trained generator with the lowest loss value among all the 100 epochs. Pytorch [117] and Pytorch Lightning [118] are adopted to train and test our proposed method.

**Hardware Setup.** In the training and validation processes, we use a graphics workstation with 256GB RAM, a NVIDIA TITAN RTX 24GB GPU, and an 18C36T 2.3 GHz Intel Xeon W-2195 CPU. The trained generator runs on a CPU-only workstation with a 4C8T 3.4GHz Intel Core i7-6700 CPU.

![Figure 5.6: Prediction accuracy results of the target and auxiliary models on the protected images. The x-axis indicates the target models. The y-axis indicates the accuracy values (%). The legend indicates the auxiliary models and the baseline (i.e., on the original images).](image)

5.4.2 Effectiveness Evaluation

We apply the following two metrics to evaluate the effectiveness of the protective perturbations:

- Prediction accuracy: As shown in Equation 5.5, the accuracy calculates a matching rate of true labels and predicted labels for the protected images. The baseline results
indicate the matching rates of true labels and predicted labels for the original images. The implementation of the accuracy calculation is based on [119].

- **SSIM [87]**: A lower SSIM value indicates less similarity between a protected image and an original image. Considering that a negative SSIM value means an inverted structure of the image [120], the SSIM values we present in the evaluations are absolute values. The implementation of SSIM is based on [121].

\[
Accuracy = \frac{1}{N} \times 100\%
\]

Figure 5.6 (a) and (b) indicate the prediction accuracy of the target models and the auxiliary models, respectively. There are 56 pairwise experiments with the 8 target models and 7 auxiliary models each. In particular, there are 43 experiments achieving high accuracy of the target models, which only compromises less than 5% accuracy compared with the baseline case, justifying the non-intrusive objective to machine vision. For the accuracy of the auxiliary models, the results are between 1.30% and 10.44%, which indicates that the auxiliary models fail to recognize more than 89.56% protected images, justifying the intrusive objective to human vision.

Furthermore, we compare the accuracy results of the target models with and without auxiliary models to train the generators, as shown in Table 5.1. In the experiments with the auxiliary models, most of the target model accuracy results are lower than the no auxiliary models, as the auxiliary models are intended to strengthen the perturbations to meet the intrusive objective for human vision. We note that the difference between the two groups of accuracy is not statistically significant, which justifies the effectiveness of our generator training process with the proposed loss function. Furthermore, the corresponding SSIM values are much higher than most of the results in the auxiliary model cases, as shown in Figure 5.7. With the auxiliary models, the SSIM values are from 0.0011 to 0.2660. However, without the auxiliary models, the SSIM values are from 0.1729 to 0.7270.
Table 5.1: Target-model accuracy of the protective perturbation generator with and without auxiliary models.

<table>
<thead>
<tr>
<th>Target Model</th>
<th>With Auxiliary Model</th>
<th>No Auxiliary Model</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Minimal</td>
<td>Median</td>
</tr>
<tr>
<td>VGG13_bn</td>
<td>80.39%</td>
<td>91.02%</td>
</tr>
<tr>
<td>VGG16_bn</td>
<td>80.92%</td>
<td>91.18%</td>
</tr>
<tr>
<td>ResNet18</td>
<td>83.74%</td>
<td>89.12%</td>
</tr>
<tr>
<td>ResNet34</td>
<td>86.33%</td>
<td>90.44%</td>
</tr>
<tr>
<td>DenseNet121</td>
<td>82.01%</td>
<td>90.58%</td>
</tr>
<tr>
<td>MobileNet_v2</td>
<td>81.78%</td>
<td>89.88%</td>
</tr>
<tr>
<td>GoogLeNet</td>
<td>82.63%</td>
<td>88.94%</td>
</tr>
<tr>
<td>Inception_v3</td>
<td>82.98%</td>
<td>89.56%</td>
</tr>
</tbody>
</table>

Figure 5.7: SSIM results on the protected images. The x-axis indicates the target models. The y-axis indicates the SSIM values from 0 to 0.8. The legend indicates the auxiliary models and the case without the auxiliary model (i.e., black bars).
image samples in Figure 5.8, where (a)-(h) show the images from the generators without (left) and with (right) the auxiliary models. The auxiliary models at the right sub-set images are selected by the lowest SSIM values. We observe that the right sub-set images in Figure 5.8 (a)-(h) contain much more successful protective perturbations than the left sub-set images. Specifically, most of the right sub-set images meet the requirements of high accuracy on the target models, low accuracy on the auxiliary models, and low SSIM values.

Figure 5.8: Protected image samples from our test dataset. For the text under each set of images, the left sub-set indicates the images from the generator without the auxiliary models, and the right sub-set indicates the images from the generator with the auxiliary models. Also, the first row indicates the target and auxiliary model information. The second row indicates the accuracy results of the target model and the auxiliary model (if the auxiliary model exists) for the test dataset. The third row indicates the SSIM values for the test dataset.

In addition, we compare the effectiveness of each auxiliary model. Figure 5.9 presents the average target-model accuracy results for each auxiliary model. We observe that
ResNet18 is the best auxiliary model to maintain the high accuracy for the target models, while ResNet34 compromises the target-model accuracy more than other auxiliary models. Figure 5.10 presents the average SSIM results for each auxiliary model. The results indicate that VGG13_bn is the best auxiliary model to minimize the similarity between the original images and the protected images. Also, MobileNet_v2 performs the worst on achieving low SSIM values for the protected images.

![Average SSIM Results for Auxiliary Models](image)

**Figure 5.9:** Average target-model accuracy results on the protected images produced by the generators with the auxiliary models. The x-axis indicates auxiliary models. The y-axis indicates the average accuracy results (%) of all the 8 target models.

5.4.3 Efficiency Evaluation

Furthermore, we measure the timing cost of the proposed perturbation generation method. Table 5.2 shows the time to generate one protected image running in the CPU-only machine. We also execute the generators on the GPU-accelerated machine that we use in the training process to show a potential performance upgrade from the user end.

Based on the results, it is worth noting that our proposed method generates the protected images at real-time in both the CPU-only (2.217 ms per image on average) and the GPU-accelerated machines (0.039 ms per image on average). These results indicate that
Figure 5.10: Average SSIM results on the protected images produced by the generators with the auxiliary models. The x-axis indicates auxiliary models. The y-axis indicates the average SSIM results.

our proposed protective perturbation approach has a potential to be leveraged in privacy-preserving real-time video streaming scenarios as well, in addition to individual image recognition.

Table 5.2: Timing evaluation (ms) of generating one protected image from the generators, which execute in a CPU-only machine and a GPU-accelerated machine.

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>2.141</td>
<td>0.036</td>
</tr>
<tr>
<td>First Quartile</td>
<td>2.188</td>
<td>0.038</td>
</tr>
<tr>
<td>Median</td>
<td>2.210</td>
<td>0.039</td>
</tr>
<tr>
<td>Third Quartile</td>
<td>2.243</td>
<td>0.040</td>
</tr>
<tr>
<td>Maximal</td>
<td>2.317</td>
<td>0.045</td>
</tr>
<tr>
<td>Average</td>
<td>2.217</td>
<td>0.039</td>
</tr>
</tbody>
</table>
5.5 Limitation and Discussion

Although our protective perturbation generator could produce protected images with high effectiveness and efficiency, there are some limitations in the current method, which we plan to address in the future work.

Auxiliary Model Selection. As shown in section 5.4, the auxiliary models play an important role in the protective perturbations generation. We will evaluate the correlation between the target models and the auxiliary models and further research on an automatic selection mechanism for a given target model.

Parameter Tuning. In our experimental evaluation, we fix the parameters for all the generators with different target and auxiliary models. In the future, we will apply customized tuned parameters for each target model during the training process to further improve the effectiveness.

Similarity Calculation and Privacy Evaluation. In our current evaluation, we apply SSIM to calculate the similarity between the protected images and the original images. There has been study showing that the SSIM metric does not always match human visual perception [122, 123]. We will apply different similarity functions in the future experiments, such as Singular Value Decomposition (SVD) [82, 124]. Also, we will conduct a human study to further evaluate the effectiveness of the protective perturbations.

Datasets. We apply CIFAR-10 in the current study. We will apply our proposed method to more datasets with different classes and image size (e.g., CIFAR-100 [107] and ImageNet [125]) to further evaluate the effectiveness and efficiency of the proposed approach.
CHAPTER 6
CONCLUSION

In this dissertation, we have developed HISA, a hardware isolation-based secure architectural extension to mitigate the new threats in emerging CPU-FPGA heterogeneous systems. HISA provides an isolated secure execution environment for both the CPU and FPGA cores in the system, which enforces access control and output verification policies to ensure the security and integrity of the heterogeneous system. To address the security challenge when offloading the computations from CPU to FPGA, we have developed ApproVer, an approximate computing based hardware security verification mechanism by leveraging spatial and temporal approximation strategies to verify the FPGA 3PIP security in HISA. To address the programming challenge, we have developed TZSlicer, a security-aware dynamic program partitioning framework. TZSlicer automatically partitions the target program into a secure slice and a normal slice to work with the CPU part of HISA. To address the scalability challenge, we have extended the heterogeneity of HISA to a new dimension of local-cloud framework for deep-learning applications with security and privacy protection.
ACKNOWLEDGMENT OF PREVIOUS PUBLICATIONS


REFERENCES


t=3s.


[56] J. Sykora, *Motion detection in VGA video (640x480) on ZC702 (Zynq FPGA)*, www.youtube.com/watch?v=LT2mGCnVoYA.


[60] “Yahoo says 1 billion user accounts were hacked,” https://www.nytimes.com/2016/12/14/technology/yahoo-hack.html.


mobile platforms,” in *ACM SIGSAC conference on computer and communications security (CCS)*, 2016, pp. 1675–1689.


