TrustGuard: A Containment Architecture with Verified Output

Soumyadeep Ghosh

A Dissertation

Presented to the Faculty

of Princeton University

in Candidacy for the Degree

of Doctor of Philosophy

Recommended for Acceptance

by the Department of

Computer Science

Adviser: David I. August

January 2017

© Copyright by Soumyadeep Ghosh, 2016.

All Rights Reserved

Abstract

Computers today are so complex and opaque that a user cannot know everything occurring within the system. Most efforts toward computer security have focused on securing software. However, software security techniques implicitly assume correct execution by the underlying system, including the hardware. Securing these systems has been challenging due to their complexity and the proportionate attack surface they present during their design, manufacturing, deployment, and operation. Ultimately, the user's trust in the system depends on claims made by each party supplying the system's components.

This dissertation presents the Containment Architecture with Verified Output (CAVO) model in recognition of the reality that existing tools and techniques are insufficient to secure complex hardware components in modern computing systems. Rather than attempt to secure each complex hardware component individually, CAVO establishes trust in hardware using a single, simple, separately manufactured component, called the Sentry. The Sentry bridges a physical gap between the untrusted system and its external interfaces and *contains* the effects of malicious behavior by untrusted system components before the external manifestation of any such effects. Thus, only the Sentry and the physical gap must be secured in order to assure users of the containment of malicious behavior. The simplicity and pluggability of CAVO's Sentry enable suppliers and consumers to take additional measures to secure it, including formal verification, supervised manufacture, and supply chain diversification.

This dissertation also presents TrustGuard—the first prototype CAVO design—to demonstrate the feasibility of the CAVO model. TrustGuard achieves containment by only allowing the communication of correctly executed results of signed software. The Sentry in TrustGuard leverages execution information obtained from the untrusted processor to enable efficient checking of the untrusted system's work, even when the Sentry itself is simpler and much slower than the untrusted processor. Simulations show that TrustGuard can guarantee containment of malicious hardware components with a geomean of 8.5% decline

in the processor's performance, even when the Sentry operates at half the clock frequency of the complex, untrusted processor.

Acknowledgments

The process of writing my dissertation hasn't just helped me explain the story of my research; it has also reminded me of all the people whose words and experiences have influenced my graduate school work. I would first like to thank my advisor, Prof. David I. August, who has been the biggest of those influences. Throughout my years as a graduate student, I have drawn great inspiration from David's research vision, his willingness to pursue research problems that can create real impact in the world, and his faith in the ability of our research group to be able to solve those problems. His determination to never settle for second best has always spurred me to do better, making me a better student and researcher. I am thankful to David for his constant support and his faith in my abilities, even at times when I was grappling with self-doubt. I am grateful to count him as a mentor, a cheerleader, and guide.

The feedback from my Ph.D. committee has been central in shaping this dissertation. I am thankful to my readers, Prof. Andrew Appel and Prof. Aarti Gupta, for their painstaking efforts in reviewing the manuscript and for giving me many invaluable insights that have improved the quality of this dissertation. I would also like to thank Prof. Simha Sethumadhavan and Prof. David Wentzlaff for serving as my thesis committee members. Their feedback on my work and presentation during my preliminary FPO was extremely useful, especially in helping me contextualize my research.

Next, I would like to thank the many members of the Liberty Research Group who I have come to know over the years as colleagues and more importantly as friends—Arun Raman, Tom Jablin, Yun Zhang, Jialu Huang, Hanjun Kim, Prakash Prabhu, Nick Johnson, Feng Liu, Matt Zoufaly, Stephen Beard, Taewook Oh, Jordan Fix, Heejin Ahn, Hansen Zhang, Nayana Prasad Nagendra, Sergiy Popovych, Sotiris Apostolakis, Jae Lee, and Kevin Fan. My fondest memories of graduate school have inevitably included the members of the research group. I drew a lot of strength from the collaborative atmosphere in the group and the shared sense of purpose, whether it was in discussing ideas, writing

papers and code, preparing presentations, or discussing life, the universe, and everything else.

I joined the group at the same time as Taewook, Stephen, and Matt; my friendships with them will be one of my enduring memories of graduate school. I learned a lot from each of them—from Taewook's work ethic and focus, from Stephen's relentless enthusiasm for new ideas and his attention to detail, and from Matt's willingness to leave everything on the table when working on his research. I am grateful to all of them for their support and their friendship. I also thank Jordan for the years of collaboration and conversation. The times we spent in brainstorming ideas that eventually led to the design of TrustGuard were some of the most productive and satisfying times for my research.

I thank Stephen, Hansen, Sotiris, and Jordan for proof-reading various versions of this dissertation. I thank the "old-timers"—Arun, Yun, Tom, Prakash, Jialu, Hanjun, Nick, and Feng— who made me feel at home when I started working with the group. I shall always treasure the lessons learned from my first few paper deadlines, working on Commutative Set with Prakash and on RAFT with Yun. I will always remember advice from Tom, Yun, Prakash, Arun, and Nick on everything from my own research projects to the world of compilers and architecture to becoming a better graduate student. While every graduate student's path is different, their willingness to share their experiences made my journey a little easier and a lot more enjoyable. I also thank Hansen and Sotiris, with whom I have worked closely more recently. Their work ethic and enthusiasm for research have certainly my later years in graduate school much more enjoyable.

I also thank all the staff of the Computer Science department for their help with all manner of things during my stay at Princeton. I am especially grateful to Nicki Gotsis and Melissa Lawson, the graduate co-ordinators during my time at Princeton. I thank Nicki for her help in navigating the official requirements for my preFPO and FPO, and in general, for helping me out with bureaucratic minutiae whenever the need arose. I will always remember Melissa fondly, starting with the first email I received from her, informing me of

my acceptance to Princeton. I have always been touched by her concern for the graduate students around her and her warmth, both during her time at Princeton and in her retirement. I am also thankful to Bob Dondero, Maia Ginsburg, Donna Gabai, and Chris Moretti—lecturers in courses I have TA'ed—for giving me the best possible platform through which to experience teaching.

I would also like to acknowledge generous funding for my graduate school work, including: "SaTC: STARSS: An Architecture for Restoring Trust in Our Personal Computing Systems" (NSF Award #CNS-1441650); "SPARCHS: Symbiotic, Polymorphic, Autotomic, Resilient, Clean-slate, Host Security" (DARPA Award #FA8750-10-2-0253); "SI2-SSI: Accelerating the Pace of Research through Implicitly Parallel Programming" (NSF Award #OCI-1047879); and "CSR: Medium: Collaborative Research: Scaling the Implicitly Parallel Programming Model with Lifelong Thread Extraction and Dynamic Adaptation" (NSF Award #CNS-0964328).

During my summer internships, I had the opportunity to work with many great engineers and researchers. The lessons I learned from them helped me approach my research in a more focused manner. I especially thank Arun Raman for making my internship at Intel an enriching and stimulating experience. I am also grateful to Shankar Easwaran for showing me the ropes during my summer at Qualcomm and teaching me how to become a better engineer.

I have been lucky to have the support of a number of friends outside the Liberty Research Group during my time at Princeton, and I thank each one of them. Srinivas Narayana was a *comrade-in-arms*, a patient counselor, a fellow philosophy enthusiast, and a great room-mate through most of my time in graduate school. His cheerfulness and his ability to listen patiently are qualities I have always hoped to emulate. Stimit Shah showed me how level-headedness and a solid grounding could co-exist with a competitive spirit. His friendship has given me a lot of strength over the years. Prakash's role as a friend and guide helped me immensely in navigating the initial years of my graduate school career.

Josh Sanz-Robinson is one of the funniest, most outgoing people I've met; his brand of enthusiasm mixed with a tinge of cynicism, his awareness of the world outside the Orange Bubble, and his love of good food all made for great shared experiences and conversations. In Bharath Balasubramanian, I didn't just find a fellow cinema and cricket fan but also a cheerleader and guide. Sergiy, Hansen, Olivier, and Nayana made my later years in Princeton a lot of fun. I must also express my gratitude and affection towards Dr. Avinash Gupta, Dr. Geeta Gupta, Dr. Saumya Das, and Dr. Prabhat Das for their support as a family-away-from-home, right here in New Jersey. I have been touched by the way they welcomed me into their families and will always remember the time spent with them very fondly.

My friendships from my undergraduate studies in BITS Pilani have outlasted my time there and given me a great support network I can always rely on. Harshad Deshmukh has always been a great support and confidante. I am grateful for the friendships of Aditya Vijay and Ankur Gupta, who comprise the other half of the M2L2 quartet. Mayank Mohta has been a friend, a travel companion, a fellow dreamer, and a brother for the longest time. Akshay Sathe gave me plenty of inspiration with his straight, uncluttered thinking and also was an equally fanatic sounding board for all football (soccer) and Manchester United-related conversations. Sunanda Khosla was my agony aunt on more occasions than one, always reminding me that my personal happiness was key to my professional output. Rohit Varghese inspired me in moments where all I wanted is to rest, through his unending enthusiasm and energy for passionately pursuing a million exciting things under the sun.

My academic journey has been shaped by a number of teachers and mentors who have believed in me, motivated me, and spurred me to do better in my studies. As I reach the end of a major milestone in my academic life, I look back and thank all my teachers over the years for their unstinting efforts in helping me learn better. I am especially grateful to Prof. Sundar Balasubramanian, who first showed me the ropes of Computer Science research at BITS Pilani. I also thank Mr. Kisan Adsul and Mr. Pradeep Kumar Bajpai—my teachers

in school who have been immense well-wishers and motivators for the best part of two decades.

I am especially thankful to my time in Princeton for having met Roshni Srinath, who has grown to occupy such a special place in my heart and mind. Her zest for life and people always make my day brighter, even when I am in the middle of my strongest bout of *introvertedness*. I am thankful for her support and her calming influence as I have approached the final phases of my graduate school career. Roshni, I still remember the day when we had our first conversation and how that first conversation has continued unabated till this day. It is my most optimistic hope that this conversation never ends and that we write our story together far into the future.

Lastly, I come to my family—my biggest support in everything I have attempted in life. My big sister, Dr. Antara Ghosh, has been a co-conspirator, a formidable sparring partner, and an all-round grounding influence on my life for as long as I can remember. My brother-in-law, Sourish Rakshit, inspires me with his calmness and maturity.

I can never adequately express what my parents—Dr. Seema Ghosh and Dr. Atindra Krishna Ghosh—have meant to me, or thank them enough for making me the person I am today. I am eternally grateful to have them as the biggest influences in my life. The only constants I have known in my life have originated from them—their immense love, their unwavering encouragement, and their eternal faith in my abilities. Ma, you have been a role model every day with the joy you derive from both your work and your family. Baba, I have never been more appreciative of the way you built your life from so little and despite the limitations of small-town India, inspired me to dream of the world and beyond. I hope I can always embody the best of both of you, even as I receive your blessings and love in the years to come.

To

Ma and *Baba*

My first and most influential teachers

मातृपितृकृताभ्यासो गुणितामेति बालकः ।
न गर्भच्युतिमात्रेण पुत्रो भवति पण्डितः ॥

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computing devices have become all-pervasive in our lives. Cars, homes, emergency services, utilities, government services, defense systems, etc. are all computerized and connected to the Internet. However, these devices are often vulnerable to attack. Intrusions into computing devices may lead to financial losses [43, 45], damage to enterprise assets [54, 97], operational disruption [47, 46], industrial and military espionage [5, 69], or even physical harm to people and their environment [43, 60, 145]. The potentially adverse consequences of compromised computing devices have compelled system engineers to make security a primary concern, after decades of building computing devices with security as a secondary concern to performance.

Most efforts for securing computing devices focus on software security due to the proliferation of software vulnerabilities and the comparative ease with which attackers can exploit them. Software developers have traditionally approached security as a game of cat-and-mouse between attackers and defenders. Under this model, vulnerabilities often come to light after exploitation by attackers. Software developers then fix these vulnerabilities and issue patches to protect those systems in the future. Even when vulnerabilities are fixed, the time between the exploitation of a vulnerability, its disclosure, and its patching often results in huge losses [55].

A more principled approach based on formal methods provides users with much stronger guarantees about the security of their software. Significant effort has been directed at using formal methods to secure the software stack, from proving that compilers produce correct executables [12, 84], to ensuring a program's memory safety in order to prevent attacks such as buffer overflows [95], to verifying the functional correctness of critical software [13, 77]. However, secure software is only as secure as the underlying system.

A modern computing system is generally some combination of processors, volatile and nonvolatile storage of various types, interconnects, and I/O devices. Many of these components have firmware, microcode, or other means of field configuration that may provide a means for an attack [59]. Bugs in underlying hypervisors and virtual machines may circumvent the protections provided by trusted applications, allowing attackers to modify or steal data used by those applications [100, 141]. Trojans may be introduced during the design and manufacture of hardware components [22, 63, 132, 137]. Design errors and transient faults in hardware may be leveraged by an attacker [27]. Adversaries have been known to tamper with systems during delivery [146]. All of these attacks can undermine secure software. Moreover, hardware vulnerabilities are harder to patch as only a subset of them can be repaired through mechanisms such as microcode or firmware updates [14, 15].

The standard practice of building a secure computing device is to use a composed set of tools, techniques, and policies to secure every individual component of the system. These may include purchasing components only from trusted companies, tamper-proofing system components [52], formal verification of component designs [64, 70, 79], post-production testing [123], etc. However, the complexity of modern hardware and the intricacies of the hardware manufacturing process have posed a significant challenge for architects and manufacturers to prove that designs are both correct and have not been altered maliciously [33, 68, 72, 82].

Vulnerabilities may be introduced at any stage of the design, manufacturing, and distribution processes. These vulnerabilities may provide attackers access to sensitive or critical

user data, irrespective of whether the software running on the system is secure [22, 27, 32, 36, 63, 74, 75, 131, 132, 133, 137]. Manufacturers have long used simulation-based testing for functionally validating processor designs [25]. However, the test space may be prohibitively huge; for instance, it is infeasible to exhaustively test multipliers for the bug described above. To reduce the test space, designers often rely on pseudorandom test-case generation to cover the test space. Consequently, there is often a significant gap between the generated test space and the actual one [8, 30, 144]. Such testing may also fail to detect vulnerabilities that stay dormant during testing with random or functional stimuli [132, 123].

More recently, formal verification techniques have been integrated into the hardware design process for verifying the functional correctness of processors [33, 66, 72, 90, 116, 102, 106, 140, 105, 115, 81]. Most of these techniques do not verify the RTL design of the processor; they instead verify a high-level model of the processor microarchitecture against a processor specification. Significantly, Reid et al. reveal that verifying pre-RTL designs often misses many bugs, as most processor errors are introduced while translating the microarchitecture design into RTL and during subsequent optimization [106]. Moreover, for complex components such as processors, formal verification often requires skilled human support and requires considerably more time than designing the processor itself.

Even assuming the ability to verify complex designs, the logistical reality of creating modern systems—from the outsourcing and offshoring of design and fabrication to the incorporation of third-party components protected by intellectual property restrictions—compromises the ability to secure computing devices [71, 73]. Even though formal techniques may guarantee the correctness of hardware designs, they cannot ensure that those designs were subsequently manufactured faithfully. Post-production techniques for detecting malicious modifications to fabricated chips offer some assurance against certain misbehaviors or defects [63, 7, 68]. However, these techniques are typically not comprehensive. Some of these techniques may also have high runtime cost [132]; or they are statistical

and have both false negatives and false positives [7, 68]. Furthermore, the ability of these techniques to reliably detect Trojan circuits in large circuits remains an open question.

In general, work on hardware verification needs to mature significantly before providing end-to-end security guarantees for the complex hardware components present in modern computing devices. Thus, one must assume that any given complex component in a system can be compromised. Additionally, even relatively a simple component can be secured only via extreme care at each stage of its lifecycle—from design through manufacturing and transportation to its delivery to the end user. Thus, there is a limit to the number of secured components that can exist in a system due to cost considerations. In recognition of this reality, computing devices would ideally be built using a single, simple, separately manufactured component, whose sole purpose is to be the basis of security for the rest of the system.

Many proposed techniques recognize the importance of assuming that not all components in a computing device can be considered secure. One class of techniques (for e.g. AEGIS [120] and Bastion [37]) uses a specially designed processor to provide software with secure execution environments. Such techniques drop the goal of protection from all attacks, such as attacks on availability, to increase the security of sensitive data. The motivation behind these techniques is that securing the processor is easier than securing all of hardware.

However, this class of techniques relies on the ability to secure processors. As discussed earlier, the complexity of building modern processors makes it difficult to ensure that every step of the manufacturing process is secure. Verifying and validating complex designs of processors is a difficult, time-consuming, and sometimes intractable task with currently available tools [33, 68, 72, 82, 117, 133, 147, 148].

Another class of techniques [101, 11, 4, 52] introduces a simpler, easily verifiable hardware root of trust for sensitive operations (e.g. cryptographic functions, random number generation, etc.) and data (e.g. cryptographic key storage). Some of these "secure co-

processors" also offer attestation functionality to ensure that sensitive operations/data are performed/released in a trusted environment. Trusted Platform Module (TPM) [101] is a widely adopted implementation of this approach. Attestation by TPM chips involves verifying the identity of other components in the system. A verified identity is considered sufficient for trust regardless of whether those components actually work correctly.

However, secure coprocessors have their own set of limitations. The onus is on the programmer to correctly use the cryptographic primitives offered by these secure coprocessors to avoid the leakage of sensitive data. These techniques implicitly assume that the processor performs noncryptographic operations correctly. Verification and validation of the processor remains a challenge, as in the case of the secure processor-based techniques.

The limitations of these two existing classes of techniques motivate the need for a new approach. This new approach must also recognize that any computing device will contain untrusted and unverified components. At the same time, security in this approach must be founded on a simple, easily verifiable component. However, unlike current approaches, this component must also ensure that complex components such as the processor, which are responsible for computations in the device are working correctly. The approach should be insensitive to whether the remaining system components are compromised during design, fabrication, or deployment. This dissertation presents just such an approach.

1.1 Dissertation Contributions

This dissertation presents **CAVO** (Containment Architecture with Verified Output)—an approach that recognizes both the need for a secure system and the limitations of existing tools and techniques to secure complex hardware components. Rather than trying to secure each component in a complex system, CAVO focuses on isolation of the effects of malicious behavior by untrusted system components before the external manifestation of any such effects. The key to CAVO lies in a physical gap between the system and its external inter-

faces through which all external communication passes, thus enabling the *containment* of erroneous and malicious behavior by untrusted system components.

The Sentry in CAVO is the only bridge that spans the physical gap between the system and its external interfaces. The untrusted system must prove to the Sentry that any data sent externally by the device is the result of correct execution of trusted software. While untrusted components within the system could gain access to and manipulate user data, the Sentry guarantees that output resulting only from operations verified as correct can be communicated externally by the system. Thus, malicious effects of untrusted system components are contained within the untrusted system itself. Focusing on containment necessitates placing the Sentry on the I/O path, which allows it to be separately manufactured, independently verified, and installed in systems at the time of deployment. The Sentry's simplicity also makes it more tractable for suppliers and consumers to take additional measures to secure it using approaches such as formal verification, supervised manufacture, and supply chain diversification. An inexpensive and simple design may even allow the Sentry to be manufactured at a trusted fabrication plant, potentially using technology a few generations old.

To establish the feasibility of the CAVO model, this dissertation also presents the first prototype design of a CAVO system named TrustGuard (Figure 1.1). A TrustGuard system is comprised of the Sentry; a unicore, superscalar processor modified to interact with the Sentry; and the Sentry's interface to the rest the system. The Sentry verifies all attempts to send out information through the system's external interfaces by checking the following: execution of only those instructions that are a part of signed programs, correctness of execution of those instructions with respect to the specifications of the instruction set architecture (ISA), and external communication of data that originate only from the aforementioned correct execution. The challenge then is to ensure that the TrustGuard system has minimal performance decline, despite restrictions such as high communication latency and limited available bandwidth between the processor and the Sentry.

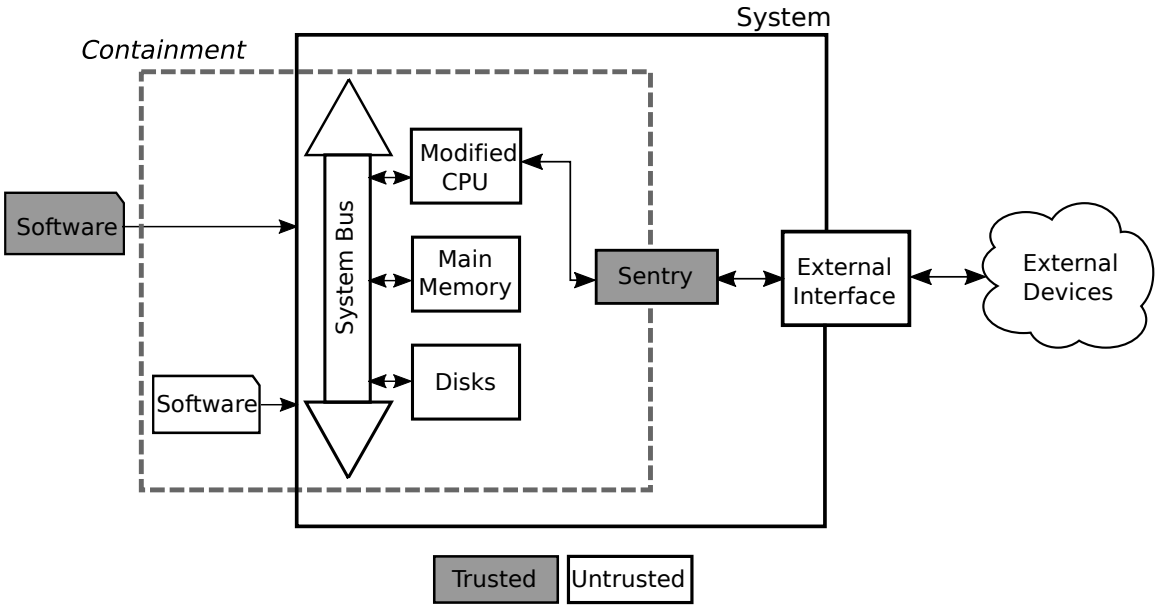


Figure 1.1: The TrustGuard architecture, where the trusted Sentry offers containment of untrusted system components by only allowing external communication of results of correct execution of signed software.

To address this challenge, the Sentry in TrustGuard leverages execution information sent by the untrusted processor to reduce the Sentry's complexity and performance requirements. In particular, it utilizes a speculative assumption that the processor executes instructions correctly and reports the corresponding execution information correctly. This speculative assumption allows the Sentry to efficiently check the correctness of instructions in parallel, regardless of the dependences between them. Furthermore, using the execution information allows the checking functionality of the Sentry to be decoupled from execution by the processor. Consequently, the processor can run nearly unhindered and experience minimal performance decline. In fact, the TrustGuard design even enables the Sentry to operate at clock frequencies much lower than the frequency of the untrusted processor.

In summary, the contributions of this dissertation are:

- The CAVO model:
  - An architecture where a simple, pluggable hardware element called the Sentry provides a foundation upon which to establish trust in the rest of the system;
  - A characterization of the security assurances provided by CAVO.

- TrustGuard, the first design of a CAVO, where:
  - The Sentry allows external communication that results from only the correct execution of signed software.
  - TrustGuard comprises of the Sentry, a modified conventional processor, and the Sentry's interface to the rest of the system. This design pushes much of the complexity required for verification of system output into the untrusted processor, thereby keeping the Sentry's design simple.
  - TrustGuard leverages execution information sent to the Sentry by the processor and decouples execution by the processor from checking by the Sentry. This minimizes the performance decline due to the introduction of the Sentry.
  - A basic FPGA prototype to validate TrustGuard functionality; and
  - A detailed simulation of TrustGuard, with a focus on its performance relative to an unprotected superscalar processor-based system.

1.2 Dissertation Organization

The rest of this dissertation is organized as follows: Chapter 2 discusses background information that motivates the Containment Architecture with Verified Output model presented in this dissertation. Chapter 3 establishes the threat model for CAVO and the general characteristics of the CAVO model. Chapter 4 presents TrustGuard—the first design and implementation of a CAVO system. Chapter 5 presents a security analysis for the resulting system. Chapter 6 demonstrates the performance of a TrustGuard system using simulation results, with a focus on the Sentry's effect on the performance of the untrusted processor. Chapter 7 discusses the simplicity of the Sentry relative to the design of an out-of-order processor. Chapter 8 discusses some other related work. Finally, Chapter 9 presents the conclusions and discusses directions for future work.

Chapter 2

Background and Motivation

Building trustworthy systems requires secure software that runs on secure hardware. Significant research has been done on securing the software stack [12, 13, 77, 84, 95]. However, secure software is only as trustworthy as the underlying hardware; compromised hardware may make the protections provided by software security mechanisms meaningless. A number of existing proposals attempt to establish the trustworthiness of complex computing systems. Each of these proposals contain some desirable properties for providing the basis of trust in a complex system. This section first discusses the various sources of vulnerabilities for building trustworthy systems, followed by a discussion of the limitations of existing proposals in accomplishing the same.

2.1 Trusted Hardware Elements

One approach for building trustworthy systems recognizes that not all hardware can be made secure. Instead, it relies on one or more hardware components that can act as the foundation of trust in the system. Table 2.1 compares some of the features of proposed techniques that fall into this category.

Many researchers have proposed designs for secure processors [37, 42, 86, 110, 119] to prevent hardware attacks. These solutions typically use memory encryption and authen-

	Secure Processors		Security Co-Processors			
	AEGIS [119]	Bastion [37]	Raksha [48]	LBA [41]	Secure Enclave [4]	TPM [101]
Trusted Hardware	Processor Chip	Processor Chip	All	All	Secure Enclave	TPM Chip
Independent Sourcing of Security Features	No	No	No	No	No	Yes
Protection Against Physical Attacks	Yes	Yes	No	No	Yes	Yes
Requires Programmer Intervention	Yes	No	No	No	Yes	Yes

Table 2.1: Comparison of existing proposals that use trusted hardware elements.

tication to protect data that leaves the processor chip. The two primary objectives for these proposals are to protect processors from physical attacks and to provide a secure execution environment for software running on the system. The AEGIS secure processor [119], in particular, presents techniques for control-flow protection and prevention of memory tampering. It also includes an optional secure OS for interfacing with the secure hardware. The cached hash tree based memory protection scheme proposed by AEGIS has formed the basis of many other architectural proposals that seek to protect the integrity of data in memory.

While secure processor designs protect against a broad class of physical and software attacks, their threat models do not acknowledge the difficulty of securing entire processor designs. While securing processors is indeed easier than securing all of hardware, modern processor designs are too complex to be reliably verified [33, 90, 82]. Secure processor proposals typically require extensive changes to existing processor designs, thus diminishing the potential for their adoption. Adding security features to processors increases the complexity of their designs and makes them even more difficult to verify. Additionally, the manufacturing process for the trusted processors is itself assumed to be secure—an assumption that can be violated if adversaries manage to mount various attacks on the integrated circuit (IC) supply chain, as described in Section 2.2.1. Thus, the system could be left reliant on untrustworthy computation and potentially corrupted security features.

A second approach for trusted hardware is the introduction of secure coprocessing elements, such as the Secure Enclave on Apple's mobile devices [4], IBM 4758 [52], and TPM chips [11, 101]. These coprocessing elements push security-critical features (e.g. random number generation, encryption/decryption, boot processor authentication, etc.) into a separate hardware module that can be more easily verified and implemented. In the case of TPM, the security-critical module is a separate chip, which may be manufactured independently of processor and the rest of the system. The simpler design and separate manufacturing allow these chips to be more easily protected against IC supply chain attacks.

However, these secure coprocessor approaches do not go far enough in terms of the protections they provide. For example, attestation of the platform by TPM chips involves verifying the identity of the other system components. A verified identity is considered sufficient for trust regardless of whether those components actually work correctly or not. The onus is on the programmer to correctly use the cryptographic primitives offered by these secure coprocessors to avoid the leakage of sensitive data. Additionally, noncryptographic computations are still performed by the processor; verification of the processor remains a challenge. Consequently, there needs to be a way to verify that other system components such as the processor, yield correct results of computations.

The category of secure coprocessor also includes instruction granularity monitoring coprocessors such as FlexCore [50], Raksha [48] and log-based lifeguard architectures (LBA) [39, 41, 40]. These techniques utilize additional hardware to monitor software execution and detect software vulnerabilities, resulting from events such as use of unallocated memory, use of uninitialized values, illegal memory overwrites, and data races.

However, these techniques all assume that the hardware is faithfully designed and manufactured. They also trust the processor to configure the monitoring hardware correctly or trust that the information flow through the hardware is correct. Consequently, these solutions are still susceptible to the supply chain attacks that compromise the processor's functionality (for example, the multiplier bug resulting in the leakage of RSA private keys [27]).

2.2 Chip Integrity Verification

The nature and complexity of the hardware manufacturing process poses a significant challenge to the security of modern computing systems, including those described in the previous section. Due to the complexity of hardware designs, architects and manufacturers have limited confidence in their verification processes to ensure systems have not been maliciously altered [33, 68, 72, 82]. Possible attacks on the IC supply chain include malicious modifications of ICs, copying of ICs to produce cloned devices, or thefts of intellectual property.

2.2.1 IC Supply Chain Attacks

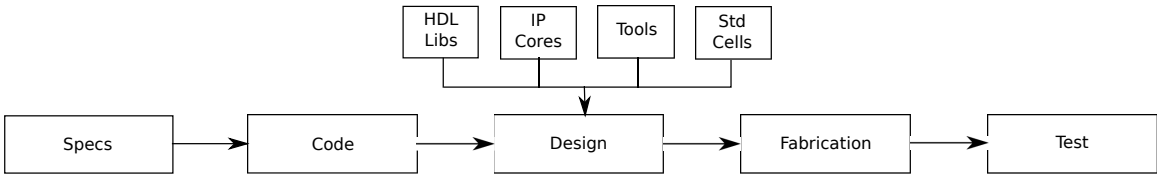


Figure 2.1: Various phases in the Integrated Circuit (IC) Supply Chain [49].

Figure 2.1 shows the various phases involved in the manufacturing of an IC [49]. For economic reasons, these phases may involve a number of different parties operating in different parts of the globe, each carrying different levels of trust.

The IC design phase comprises all code and inputs to tools to generate specifications for fabrication by the foundry. Some design processes can be closely observed and audited, thereby establishing trust in them. However, other elements of the design such as third-party intellectual property (IP) components are often opaque to the designers and consumers who use them. Due to this lack of verifiability and trust in various design components, this phase can be susceptible to attacks, including ones where *hardware Trojans* are inserted surreptitously into the chip design by a rogue designer.

A similar lack of verifiability and trust also plagues the fabrication phase of the IC supply chain. Closely controlling the fabrication process is impossible for most chip man-

ufacturers, who still design and sell chips but outsource fabrication to overseas foundries for economic reasons. As the chip designer and the foundry are separate entities, it gives rise to the possibility of the foundry mounting a hardware Trojan attack by incorporating malicious components into the chip [7, 26, 123].

During the testing phase, each IC is checked for manufacturing faults based on the specifications of known designs. However, it may be extremely hard to detect Trojans during validation as they may lie dormant during testing with random or functional stimuli [132, 123]. For instance, Biham et al. have demonstrated a devastating attack on the RSA cryptosystem that relies on a multiplier computing the wrong product for a single pair of 64-bit integers. If such a pair of numbers is known, it is possible to break *any key* used in *any RSA-based software* on that device using *a single chosen message* [27]. Obviously, it is impossible to exhaustively test multipliers for this bug.

Instead, some manufacturers use simulation-based tests to validate their chips. However, the test space for chips could be prohibitively huge, especially for complex chips such as processors. Consequently, designers often rely on pseudorandom test-case generation to cover the test space, thus leaving a big gap between the generated test space and the actual one [8, 30, 144].

2.2.2 Defenses against IC Supply Chain Attacks

Various defenses have been proposed against hardware Trojans, each targeting Trojans inserted at different stages of the design—specification, register-transfer level (RTL) design, IP integration, physical design, and fabrication. These defenses include post-fabrication detection [34, 142, 22, 68, 78, 139, 7], run-time monitoring [132], and design-time deterrence [112, 67, 35, 133, 148, 117]. Table 2.2 compares some of the characteristics of these defenses.

To a large extent, Trojan detection involves an arms race between attackers and chip designers—even while designers update their security measures to protect systems from

	UCI [63]	FANCI [133]	VeriTrust [148]	PCC [88]	BlueChip [63]	IC Finger-printing [7]	Path Delay Analysis [68]	Zebra [129]
Design Time HT Detection	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Fabrication Time HT Detection	No	No	No	No	No	Yes	Yes	Yes
False Positives	Yes	Yes	Yes	No	No	Yes	Yes	No
False Negatives	Yes	Yes	Yes	No	Yes	Yes	Yes	No
Runtime Overhead	None	None	None	None	Low	None	None	High
Requires Golden Designs/Chips	No	No	No	No	No	Yes	Yes	No
Sensitive to Process Variations	No	No	No	No	Yes	Yes	Yes	No
Requires Changes to IC Manufacturing	No	No	No	No	Yes	No	No	Yes

Table 2.2: Comparison of existing proposals that detect hardware backdoors.

known Trojans, attackers introduce more advanced Trojan designs to avoid the latest detection techniques. For instance, Hicks et al. formulated the hardware Trojan detection problem as one of unused circuit identification [63]. Soon after, researchers presented techniques to automatically construct hardware Trojans that evaded the UCI detection algorithm [117, 147].

Each of these techniques offers some assurance against certain misbehaviors or defects but are typically not comprehensive. For instance, post-fabrication detection techniques that rely on logic testing using likely Trojan triggers cannot detect backdoors designed to stay dormant during post-fabrication testing [123, 132]. Some of these techniques may also have high runtime cost [132]. Often, these techniques are statistical and have both false negatives and false positives.

Thus, techniques for ensuring that design and fabrication of chips faithfully implement the specifications are incomplete. Furthermore, the capabilities of post-fabrication techniques have only been on small circuits; their ability to detect Trojan circuits reliably in large chips remains a question. Even if it were possible to detect Trojan circuits reliably, it is expensive to test chips in large enough numbers for statistical significance. It is also expensive and difficult to produce "golden chips" against which manufactured chips can be tested.

An alternate approach detects hardware Trojans inserted in third-party intellectual property (IP) using a proof-carrying code framework [88]. In this method, a set of security-related properties is formulated, and a formal proof of these properties is created by the designer. The user of the IP carries out the validation of the security-related properties to ensure that no HDL code was modified.

A similar approach is taken by Zebra—a verifiable outsourcing scheme based on the CMT [124] and Allspice [128] interactive proof protocols. Zebra tries to verify correct execution of an untrusted hardware component using a trusted ASIC [129]. The focus for Zebra is on untrusted foundries that may introduce Trojans into the chip during fabrication.

This approach has several advantages, including the absence of false negatives and false positives that plague statistical Trojan detection schemes. However, the technique incurs high overheads compared to untrusted computations and has limited applicability. More generic protocols [31, 99, 24, 130] can handle a bigger class of applications. However, they cannot be easily designed on hardware and incur even more massive runtime overheads (10^5x–10^7x).

Considering the limitations of existing techniques in detecting backdoors inserted in complex hardware components, it is reasonable to assume the presence of untrusted hardware in the system. The challenge then is to guarantee a secure execution environment for the system and detect the effects of Trojan activation at runtime.

2.3 Trustworthy Systems based on Redundancy

The traditional approach for building trustworthy systems from untrustworthy components is based on redundant execution [21, 23]. In this approach, computations may be redundantly performed on several untrustworthy components and majority voting can be used to detect erroneous behavior. Design and manufacturing diversity of replicated components makes it less likely for a hardware bug or backdoor to escape detection. For instance, the system could use chips fabricated in two different foundries, thus reducing the probability that a *hardware Trojan* was inserted into the chip during fabrication. However, in general, the cost of replicating every single component of a system may be quite high, making it attractive only for high-assurance and high-security applications like airplanes and defense systems.

The redundant execution approach has also been used for hardware Trojan detection. SHADE [29] is a hardware-software approach that uses multiple ICs as guards in a single board to prevent data exfiltration and detect denial-of-service attacks. Due to the assumption that at least two ICs come from different foundries, malicious circuitry would not col-

lude between two or more ICs. The SHADE architecture is aimed at detecting the actions of hardware Trojans that are not detected through to the deployment of the system. Consequently, it is complementary and compatible with the design-time and fabrication-time techniques for hardware Trojan detection detected in Section 2.2.2.

The redundancy in SHADE lies in the two guards that are used for double encryption of off-chip data by the processor. While this protects against unauthorized data leakage, it does not address cases where wrong execution by the processor leads to compromise (as in the case of the multiplier bug leading to leakage of RSA private keys [27]). Furthermore, SHADE puts the dual encryption on the critical path to/from memory. This may result in significant slowdown in the processor's performance.

Other techniques use redundant execution on multiple processing elements to detect the presence of hardware Trojans. McIntyre et al. [91] spawn functionally equivalent but variant software processes on multiple identical processing elements, dynamically adjusting the trust in an individual processing element depending on compared outputs. However, the effectiveness of this technique relies on efficient generation of variants. The identical processing elements in this technique are all designed and fabricated together, so they may each have the same vulnerability. Another technique, SAFER PATH [23] uses a custom-designed processor, which replicates processing elements and uses redundant execution to detect the effects of hardware Trojans. However, this would require a complete redesign of the processor. Furthermore, both these techniques only protect the processor and ignore the trustworthiness of other system components such as memory.

The redundant execution approach has also been used to build systems resilient to transient faults [16, 17, 18, 19, 20, 38, 57, 92, 94, 98, 107, 108, 111, 113, 114, 121, 138, 134, 136, 149, 150]. One such system, DIVA [18] showed that it is possible to build a simple, redundant checker to detect errors in a processor's functional units and its communication channels with the register file and data cache. While the introduction of a simple checker presents a promising approach, DIVA was not designed for and is not trivially extended

to security. Architecturally, DIVA's checker is embedded in the processor's commit path and thus both the checker and microprocessor must be manufactured jointly using the same technology. From a security perspective, this makes the checker vulnerable to malicious changes during the processor's manufacturing. However, simply moving DIVA off-chip is not a straightforward process, as there are many issues to consider including separation from the commit path and a potentially infeasible off-chip bandwidth required between the processor and checker.

Additionally, DIVA does not provide any protections for memory and register files. It instead relies on ECC to detect any transient faults that may occur in these modules. This is obviously insufficient for security, as any malicious component, could change the contents of registers or memory, including both data and instructions. Finally, DIVA trusts the processor to correctly communicate trace information to the checker. Consequently, the checker cannot tell if the instruction execution stream it receives is maliciously modified, for example by insertion of new instructions, the modification of instructions, or deviance in control flow.

2.4 Motivating CAVO

The survey of prior work uncovers several key insights that motivate the Containment Architecture with Verified Output (CAVO) approach presented in this dissertation.

Secure processors show how to use cryptographic primitives to ensure secure execution in the presence of untrusted off-chip components such as memory. Many of these security primitives can be moved to a separate coprocessor, with a defined interface to the application processor. This allows for a clean design where security features can be abstracted out and more easily verified and validated. In the case of TPM chips, the security chips could be independently manufactured, using processes that are more closely monitored and controlled, leading to a higher level of trust in these chips. The insight that *a simple, separately*

manufactured component with an open design can be verified and form the basis of trust in a complex system is one of the primary motivators for the CAVO approach.

CAVO focuses on *containment* of a system by requiring that all communication from the system be approved by a simple, pluggable Sentry. The Sentry in CAVO is the only bridge that spans the physical gap between the system and its external interfaces. The Sentry's simplicity and pluggability makes it more tractable for suppliers and consumers to take additional measures to secure it using approaches such as formal verification, supervised manufacture at older but more secure foundries, and supply chain diversification. While containment by the Sentry does not provide some security guarantees (for e.g. availability), it assures users that any output of the system is the result of correct execution. Thus, a system protected by CAVO provides a root of trust, in that no data can escape the system unauthorized, even in the face of malicious or corrupted components within the system.

Functionally, the CAVO Sentry draws inspiration from redundancy-based security techniques that demonstrate how to detect the effects of maliciously modified hardware by leveraging diversity of system components. This is particularly necessary as an added defense because techniques to detect hardware Trojans prior to and after fabrication are either not complete and/or not applicable to complex circuit designs.

The CAVO Sentry reduces the cost of replicating every single component for redundant execution by leveraging the work done by untrusted system components. For instance, a modern out-of-order processor has a number of components such as branch predictors, dependence predictors, reorder buffers, multiple levels of caches, etc. to improve the performance of executed programs. The Sentry can utilize the information gleaned from many of these components to enable efficient redundant checking of the processor's execution, without having to rely on the correctness of the said information. The next chapter describes the CAVO approach in detail, including the threat model targeted by this work and the general characteristics of the CAVO Sentry.

Chapter 3

CAVO: Containment Architecture with Verified Output

The goal of CAVO is to serve as a foundation of trust upon which trustworthy systems can be built by assuring users that all external communication is verified before leaving the system. A manifestation of CAVO, such as in Figure 3.1, consists of a physical gap between the system and its external interfaces through which all external communication must pass. This physical gap allows *containment* of any information emanating from the untrusted system.

The physical gap is bridged by a Sentry that assures users of the *correctness of output* by verifying that all output from the system is the result of correct execution of a trusted program. The choice of different mechanisms for trusting programs depends on a particular implementation of CAVO. For instance, an implementation may choose to trust programs through signatures on the program binary from a trusted source (Chapter 4), similar to the App Store model [4]. An alternate implementation may require proof carrying codes to be furnished for the program binary [96].

An implementation of CAVO should define correctness of output with respect to some interface between the trusted program and the system, such as an instruction set architecture

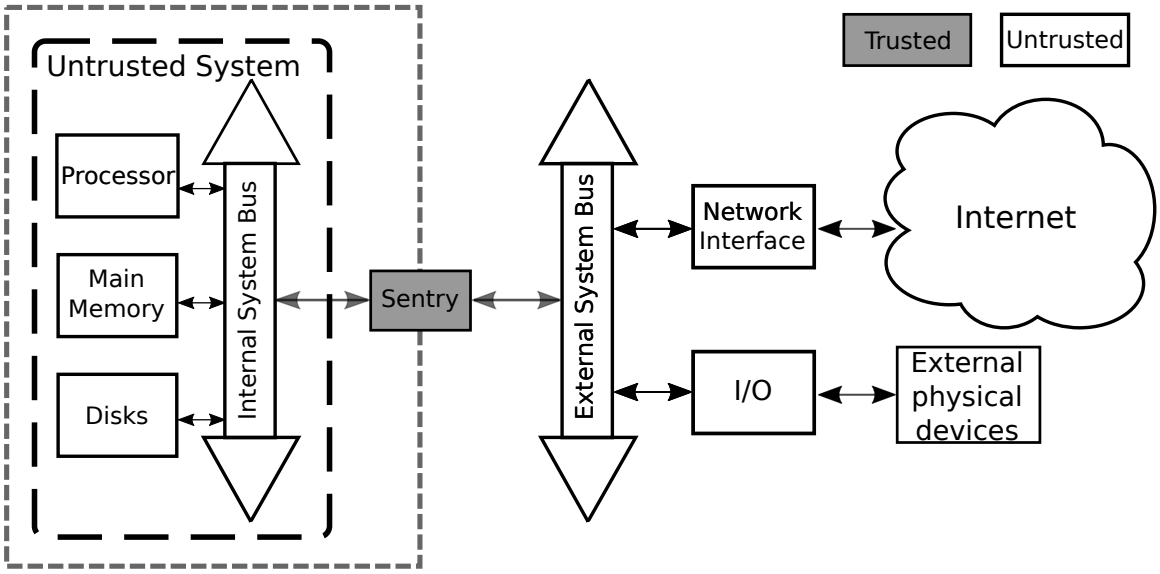


Figure 3.1: The CAVO Model

(ISA) or program specification. CAVO is similar to a whitelisting approach, where communication is only restricted to results of correct execution, as defined by this aforementioned chosen interface. This approach makes CAVO more robust in isolating the effects of malicious components, compared to techniques that rely on identifying specific malicious behaviors or use statistical analysis to identify malicious behaviors with high probability. Additionally, when the chosen interface is stable over time (for instance, in the case of an ISA), the CAVO approach can work well even in presence of new classes of backdoors. By preventing the external communication of incorrect or unapproved results, CAVO isolates the effects of faulty or malicious components to within the system.

The whitelisting approach used by CAVO also makes it more insensitive to where a vulnerability exists in the system components. As described in Section 2.2, backdoors could be inserted at any stage of the hardware manufacturing process. Different techniques detect the effects of backdoors in design, fabrication, and testing phases. However, the coverage for backdoors is typically not complete. Similar to the approach in SHADE [29], CAVO can detect the actions of Trojan ciruits that survive detection through to the deployment of the system. Consequently, it is complementary and compatible with the various

pre-deployment backdoor detection techniques discussed in Section 2.2.

The design of a CAVO Sentry should be *simple* to facilitate its trustworthiness through all stages of its creation, from design to manufacturing to deployment. Ideally, the design of the Sentry should be comparable to hardware designs that have been formally verified [90, 116]. Additionally, the Sentry should also be pluggable so that it can be sourced independently and manufactured at trusted, closely controlled fabrication plants, possibly using generations old technology.

To keep the Sentry simple and pluggable, additional functionality is added to the untrusted system. The additional functionality provides the Sentry with sufficient information to reduce the difficulty in performing checks regarding the correctness of output. For example, an implementation of CAVO may require processor modifications in order to send execution trace information, or utilize a custom software toolchain to insert instrumentation that communicates with the Sentry. Such information is validated before being relied upon[1] and thus, does not compromise the security of CAVO.

3.1 Threat Model

Hardware. CAVO considers all hardware components in the system other than the Sentry–including processor, memory, and peripherals–vulnerable to compromise. These vulnerabilities may arise as a result of unreliability, flaws in design, or malicious logic inserted during design or chip fabrication phases. CAVO requires that there are no communication channels out of the system other than through the Sentry.

Software. Untrusted programs are allowed to execute on the processor; however, the Sentry will prevent results from their execution, including malicious interference with trusted programs, from being communicated externally. The mechanism for how trust is established in software depends on a particular implementation of CAVO. For instance, in the

[1] but may be used speculatively while keeping security assurances intact

implementation presented in Chapter 4, it is assumed that trusted software is signed by a trusted authority.

Users. Adversaries are assumed to not have physical access to the Sentry nor the physical gap.

Covert Channels. CAVO addresses only explicit leakage via malicious or buggy hardware. It does not protect against information leaked via covert channels.

Adversaries. The adversary in the case of CAVO could be any entity that has compromised the working of the untrusted hardware components prior to the deployment of the system. This entity could be a rogue system designer, who has inserted backdoors into the RTL design. Or it could be a malicious foundry, which includes malware within the fabricated chip. The hardware threat model depends on two key assumptions holding true: (1) The system board is constructed in a trusted location to eliminate the possibility of channels on the board bypassing the physical gap. (2) The Sentry itself is manufactured at a trusted fabrication facility, separate from the processor and other system components.

3.2 Example Threats

CAVO protects against incorrect program output caused by the following example threats:

- Backdoors inserted in the untrusted processor during design or fabrication that write data to a peripheral without authorization;
- Malicious changes or inadvertent bugs in the processor that lead to incorrect execution of one of more instructions, weakening encryption or changing information sent to peripherals [27];
- Untrusted components such as memory and/or on-chip networks that manipulate data, weakening encryption or changing information sent to peripherals; and

- Insecure software that execute on the system and communicate sensitive data to the peripherals.

CAVO does not address threats such as:

- Information leaked via side channels (e.g. encoding of sensitive information in an energy usage pattern, long duration timing encodings, implicit information leaked by failures);
- Attacks on availability. For instance, it is possible for the adversary to mount an attack that causes communications from the system to cease.

Chapter 4

The TRUSTGUARD Architecture

To establish the feasibility of CAVO, this dissertation presents a prototype design named TrustGuard. In TrustGuard, the Sentry verifies the correctness of all attempts to send out information through the system's external interfaces by checking the following: execution of only those instructions that are a part of signed programs, correctness of execution of the aforementioned instructions, and external communication of data that originate only from such checked execution. TrustGuard checks the correctness of a program with respect to the specifications of the instruction set architecture (ISA). For simplicity, TrustGuard supports a system with a single-core processor and a trusted provider of signed software, including the OS. TrustGuard assumes that signed programs are nonmalicious, similar to the software threat model for AEGIS [119]. Additionally, TrustGuard requires that all I/O is executed through explicit I/O instructions.

Figure 4.1 shows the high-level design of the TrustGuard architecture, which allows output only from the correct execution of signed software. The processor sends an execution trace of its committed instructions to the Sentry. The Redundant Instruction Checking Unit (RICU) in the Sentry re-executes instructions sent by the processor using its own functional units to verify that the results produced by the processor were correct. This includes all arithmetic, control, and memory instructions (Sections 4.1 and 4.2). To check

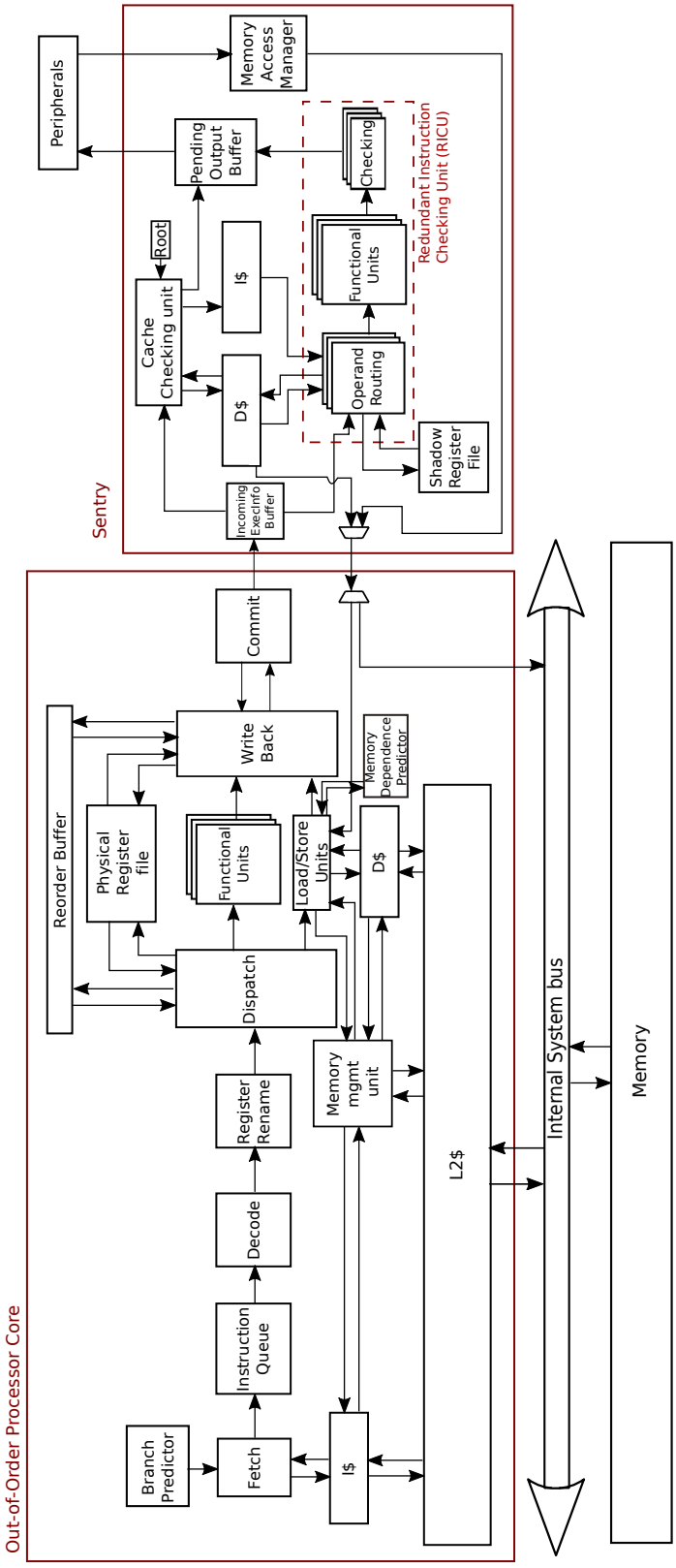


Figure 4.1: TrustGuard Architecture

the execution of memory instructions without having to duplicate memory, TrustGuard uses a cryptographic scheme based on message authentication codes (MACs) and a Bonsai Merkle tree (Section 4.3). Finally, output operations are only performed when the matching instruction for that output operation has been verified (Section 4.5). If an error is detected, the Sentry alerts the user and prevents any external communication by instructions that depend directly or transitively on the errant instruction.

This model of utilizing information from the processor in the Sentry presents both a security and an architectural challenge. The security challenge is to ensure that the Sentry catches any malicious communication of execution information by the processor. This is accomplished by keeping minimal architectural state in the Sentry to verify the correctness of both execution and communication by the processor.

Meanwhile, TrustGuard requires modifying the processor to communicate with the Sentry, as described in Section 4.6. The extra latency and the limited bandwidth of communication between the processor and the Sentry may have significant adverse impact on the performance of the system. The architectural challenge then is to introduce these changes to the processor and the additional Sentry while incurring minimal adverse effects on performance. TrustGuard addresses this challenge by enabling high throughput checking in the Sentry through dependence-free checking of instruction execution (Sections 4.1 and 4.2). Additionally, TrustGuard uses link compression to reduce the bandwidth requirements for the Sentry-processor communication (Section 4.4).

It is important to note that in this model, "redundant execution" by the Sentry is different from "execution" by the processor. As shown in Figure 4.1, execution by a processor involves multiple stages such as fetch, decode, "functional unit execution," memory access, and writeback. Additionally, for modern out-of-order processors, execution also includes some combination of dependence speculation, branch prediction, register renaming, reorder buffers, multiple levels of caches, instruction queues, instruction dispatch, load/store queues, inter-stage forwarding, and memory control (as shown in Figure 4.1.)

By contrast, the Sentry's RICU primarily does "enhanced functional unit execution", where it relies on the processor to direct its work and manage its state. Requiring the untrusted processor to send execution information to the Sentry adds extra logic to the processor, which may increase its complexity. However, this design decision reduces the complexity of the Sentry. This redundant execution model is similar to DIVA, which showed how to verify a processor's full execution using a simple, on-chip checker that only performs enhanced functional unit execution [18]. It is also important to note that this model of redundant execution does not impact the security guarantees provided by TrustGuard, as the Sentry does not trust execution information provided by the untrusted processor and detects effects that are the result of unreported or misreported instruction execution.

4.1 High-Throughput Checking of Instruction Execution

The untrusted processor in TrustGuard sends trace execution information to the Sentry, allowing for the Sentry to check that anything to be communicated externally resulted from correct execution of signed software. Instructions to be verified are sent by the untrusted processor, as part of the trace execution information, to the Sentry's instruction cache, using the mechanism described in Section 4.3. Correctness of instruction execution is checked by the Sentry's Redundant Instruction Checking Unit (RICU), as seen in Figure 4.1.

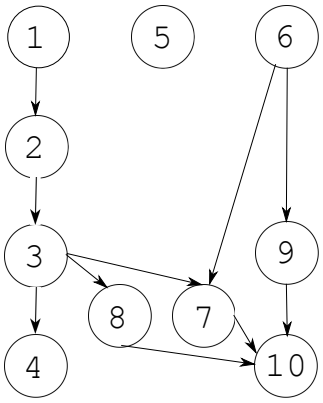
An RICU consists of four stages: *Instruction Read* (IR), *Operand Routing* (OR), *Value Generation* (VG), and *Checking* (CH). The IR stage retrieves the next instructions to be checked from the Sentry's instruction cache. The OR stage determines the operands to be used for redundant execution of these instructions. The VG stage redundantly executes these instructions using the Sentry's functional units. The CH stage compares the results generated by the Sentry against the results sent by the processor as part of the trace execution information. Thus, the Sentry can determine if the processor was reporting the correct value. Checking the processor's execution stream using redundant execution enables Trust-

Guard to detect errors (malicious or otherwise) in the processor's execution.

For instance, consider the trace snippet from `456.hmmer` shown in Figure 4.2(a). Figure 4.2(b) shows the dependences between the instructions in the trace. The most naïve design for the Sentry would check instructions one after another in program order. Assuming that each stage of the RICU takes one clock cycle to complete, this would require 4 clock cycles to check every instruction, resulting in a checking schedule lasting 40 clock cycles for the 10-instruction sequence in Figure 4.2(a).

```
1 : r1  = r1 & 0xFF
2 : r1  = r1 | (r1 << 0x8)
3 : r1  = r1 | (r1 << 0x10)
4 : r12 = r1 << 0x2
5 : r2  = r2 - 0x8
6 : r34 = r3 + 0x4
7 : Mem[r34]     = r1
8 : Mem[r32+0x4] = r1
9 : r3  = r3 + 0x8
10: r1  = Mem[r3]
```

(a)



(b)

Figure 4.2: Trace snippet from 456.hmmer. (a) Instructions in the trace. (b) Dependences between instructions in the trace.

One option to improve the throughput of checking is to pipeline the RICU stages and to check the correctness of instructions in program order. Such a design could utilize value forwarding, as is the case for many pipelined designs. The checking schedule for this Sentry is shown in Figure 4.3, where the 10-instruction cycle takes a total of 13 clock cycles. One advantage of this design is that it is similar in complexity to various inorder pipelined processor designs that have been formally verified previously [90, 116].

However, there are several disadvantages to using such a pipelined design for the RICU. Most modern processors have complex out-of-order engines designed to extract instruction-level parallelism from executed code in order to achieve superscalar performance (IPC or Instructions Per Cycle greater than 1). However, the pipelined design for the Sentry can only check a maximum of 1 instruction per cycle. Additionally, this pipelined design does

not take into account additional latencies introduced when memory instructions are re-executed. Thus, the real throughput of redundant execution by such a pipelined RICU will be much lower than 1 instruction per cycle.

One option to improve the throughput of checking is for the RICU in the Sentry to support checking of multiple instructions in parallel. Consider a Sentry that can check up to four instructions in parallel—such a Sentry has four instances of each RICU stage (for e.g. VG_A, VG_B, VG_C, and VG_D). The checking schedule for this Sentry is shown in Figure 4.4.

	Sentry Clock cycles												
	1	2	3	4	5	6	7	8	9	10	11	12	13
Inst 1	IR_A	OR_A	VG_A	CH_A									
Inst 2		IR_A	OR_A	VG_A	CH_A								
Inst 3			IR_A	OR_A	VG_A	CH_A							
Inst 4				IR_A	OR_A	VG_A	CH_A						
Inst 5					IR_A	OR_A	VG_A	CH_A					
Inst 6						IR_A	OR_A	VG_A	CH_A				
Inst 7							IR_A	OR_A	VG_A	CH_A			
Inst 8								IR_A	OR_A	VG_A	CH_A		
Inst 9									IR_A	OR_A	VG_A	CH_A	
Inst 10										IR_A	OR_A	VG_A	CH_A

Figure 4.3: Schedule for instruction checking when the Sentry RICU has a simple pipelined design with value forwarding.

	Sentry Clock cycles							
	1	2	3	4	5	6	7	8
Inst 1	IR_A	OR_A	VG_A	CH_A				
Inst 2		IR_A	OR_A	VG_A	CH_A			
Inst 3			IR_A	OR_A	VG_A	CH_A		
Inst 4				IR_A	OR_A	VG_A	CH_A	
Inst 5	IR_B	OR_B	VG_B	CH_B				
Inst 6	IR_C	OR_C	VG_C	CH_C				
Inst 7				IR_B	OR_B	VG_B	CH_B	
Inst 8				IR_C	OR_C	VG_C	CH_C	
Inst 9		IR_B	OR_B	VG_B	CH_B			
Inst 10					IR_A	OR_A	VG_A	CH_A

Figure 4.4: Checking schedule with 4-wide RICU for trace snippet in Figure 4.2(a). Pipelining and value forwarding augmented with out-of-order checking improves throughput compared to Figure 4.3, but dependences between instructions are still respected during checking.

As can be seen in the schedule, the Sentry must respect dependences between instructions during re-execution; this limits the throughput of checking. For example, instruction 2 in Figure 4.3 must wait for the result of instruction 1 to be available before starting its checking. (The operand `r1` of instruction 2 is dependent on the result `r1` of instruction 1.) Moreover, it may appear that the Sentry cannot start checking the dependent instruction until the instruction on which it depends is checked and found correct. For such a case, the checking of instructions in a dependence chain (instructions 1–4 in the example) would get serialized. Any parallelism to improve the utilization of the available resources in the RICU would have to come from looking at a larger instruction window and identifying instructions that do not have any dependences left to be satisfied. Instruction 5 and 6 in the example satisfy this requirement and thus, the Sentry may start checking them in parallel with instruction 1.

However, identifying instructions that can be checked in parallel in this way would require various components of out-of-order engines to be added to the Sentry . These components include a separate instruction queue, register dependence unit, register renaming unit, a large physical register file to store values generated during re-execution, reorder buffers, instruction dispatch unit, etc. The addition of these components would make the design of the Sentry comparable in complexity to that of the untrusted processor in the system. Even with these components, the throughput of checking is limited by the available instruction-level parallelism in the program being executed. As shown in Figure 4.4, this method of redundant execution takes a total of 8 clock cycles on the Sentry. Moreover, the checking throughput only peaks at three instructions in a cycle, leaving the fourth set of RICU components (IR_A, OR_A, VG_A, and CH_A) unutilized.

TrustGuard aims to further improve the throughput of checking by the Sentry, without requiring complex out-of-order components in the Sentry's design. To accomplish this, TrustGuard uses a form of speculation. Speculating that the *processor executed instructions and reported results in the sent trace correctly* allows the Sentry to utilize the sent trace

execution information during re-execution. Leveraging this sent information enables the Sentry RICU to re-execute instructions regardless of the dependences between them. This enables embarrassingly parallel checking of instructions, as demonstrated by the 6-cycle checking schedule shown in Figure 4.5.

	Sentry Clock cycles					
	1	2	3	4	5	6
Inst 1	IR_A	OR_A	VG_A	CH_A		
Inst 2	IR_B	OR_B	VG_B	CH_B		
Inst 3	IR_C	OR_C	VG_C	CH_C		
Inst 4	IR_D	OR_D	VG_D	CH_D		
Inst 5		IR_A	OR_A	VG_A	CH_A	
Inst 6		IR_B	OR_B	VG_B	CH_B	
Inst 7		IR_C	OR_C	VG_C	CH_C	
Inst 8		IR_D	OR_D	VG_D	CH_D	
Inst 9			IR_A	OR_A	VG_A	CH_A
Inst 10			IR_B	OR_B	VG_B	CH_B

Figure 4.5: Dependence-free parallel checking schedule for checking snippet from Figure 4.2(a), when results reported by the untrusted processor are used to break dependences during checking.

As seen in the figure, the Sentry can begin checking correctness of instructions 2, 3, and 4 in the same clock cycle as instruction 1, despite the dependence chain between these instructions shown in Figure 4.2(b). This is because the Sentry can speculate that the result of instruction 1 reported by the processor was correct and use that information to check instruction 2. The Sentry can then check in the CH_A stage if the speculative assumption held true. Misspeculation in this case means detecting malicious behavior or otherwise incorrect execution by the processor, either in executing the instructions or sending its results.

The validity of this speculation depends on satisfying the following commit order requirement. Consider two instructions i_1 and i_2 that enter the RICU at clock cycles s_1 and s_2 and finish re-execution at clock cycles t_1 and t_2. If the untrusted processor commits the i_1 earlier than i_2, then the following condition must hold in the Sentry: $s_1 \leq s_2$ and $t_1 \leq t_2$. In the example from Figure 4.2(a), instruction 7 occurs after instruction 6 in the commit

order of the processor. Therefore, instruction 7 cannot enter the IR_C stage until the second clock cycle, which is when instruction 6 enters the IR_B stage. Also, instruction 7 cannot be deemed correct by CH_C until the fifth clock cycle, which is when CH_B deems the execution of instruction 6 correct.

The utilization of trace execution information to enable embarrassingly parallel checking of instructions leads to the question of what comprises the trace sent by the processor. The choice of what the trace consists of, affects the simplicity of the Sentry's design, the efficiency of checking (in turn, affecting the performance of the system), and off-chip bandwidth requirements. For instance, the processor could send the committed instructions, all their operands and the results of their execution to enable parallel checking. With this information, the Sentry could re-execute instructions with the reported operands. Thus, each instruction could be redundantly executed in parallel, regardless of the dependences between them. The CH stage would then assume responsibility for checking that each operand was correctly reported by the processor. However, this scheme requires prohibitively high bandwidth on the channel between the processor and the Sentry.

TrustGuard addresses this problem by recognizing that *the results of committed instructions* represent the minimal amount of information necessary for embarrassingly parallel checking of all executed nonmemory instructions, regardless of dependences between them. Relying on a trace of committed instructions presents an additional advantage. An out-of-order processor often does a lot of work, which does not get reflected in the processor's architectural state due to mispredictions, squashing, etc. By focusing only on committed instructions, the Sentry only needs to re-execute those operations that lead to a change in the processor's architectural state.

Finally, memory instructions pose an interesting challenge to the redundant execution model because of the additional latency incurred in redundant memory accesses. Instead of requiring the Sentry to interface with replicated memory, TrustGuard uses a separate MAC-based memory integrity scheme to check the execution of memory instructions; this

scheme is described in Section 4.3. Meanwhile, the next section explains how the RICU in the Sentry realizes the embarrassingly parallel checking schedule shown in Figure 4.5.

4.2 Redundant Instruction Checking Unit (RICU)

As mentioned earlier, the Redundant Instruction Checking Unit (RICU) consists of four stages: Instruction Read (IR), Operand Routing (OR), Value Generation (VG), and Checking (CH). Figure 4.6 shows the architectural design of a RICU that checks up to 4 instructions in parallel. The RICU utilizes a shadow register file that contains all register values corresponding to the instruction sequence, which has been verified correct. The Sentry also maintains a `NextInst` register to determine the instruction to be checked next. The instructions to be verified are sent by the untrusted processor to the Sentry's instruction cache, using the mechanism described in Section 4.3.

Table 4.1 lists the execution information sent across by the processor to the Sentry for different types of instructions. For all nonmemory instructions, the processor sends across the results of execution of those instructions to the Sentry. These are stored in the `ExecInfo` buffer. For control flow instructions (conditional and unconditional branches, calls, etc.), the result takes one of two forms: (1) If the jump is not taken, i.e. if the program continues to execute along the straight line path, then the processor sends a 1 as the Jump Status bit; and (2) If the jump is taken, the processor sends a 0 as the Jump Status bit, followed by the address of the next instruction to which the control flow is transferred.

Figure 4.7 shows the logic in the IR stage. In this stage, the RICU reads up to *n* instructions from the instruction cache using the value in the `NextInst` register, where *n* is the number of instructions that can be checked in parallel. Note that the `NextInst` register only stores the location for the earliest instruction to be checked in a given clock cycle. The IR stage determines if the `ExecInfo` buffer has trace information corresponding to the next *n* instructions to be checked by the Sentry. This is indicated by the i_n bit in Fig-

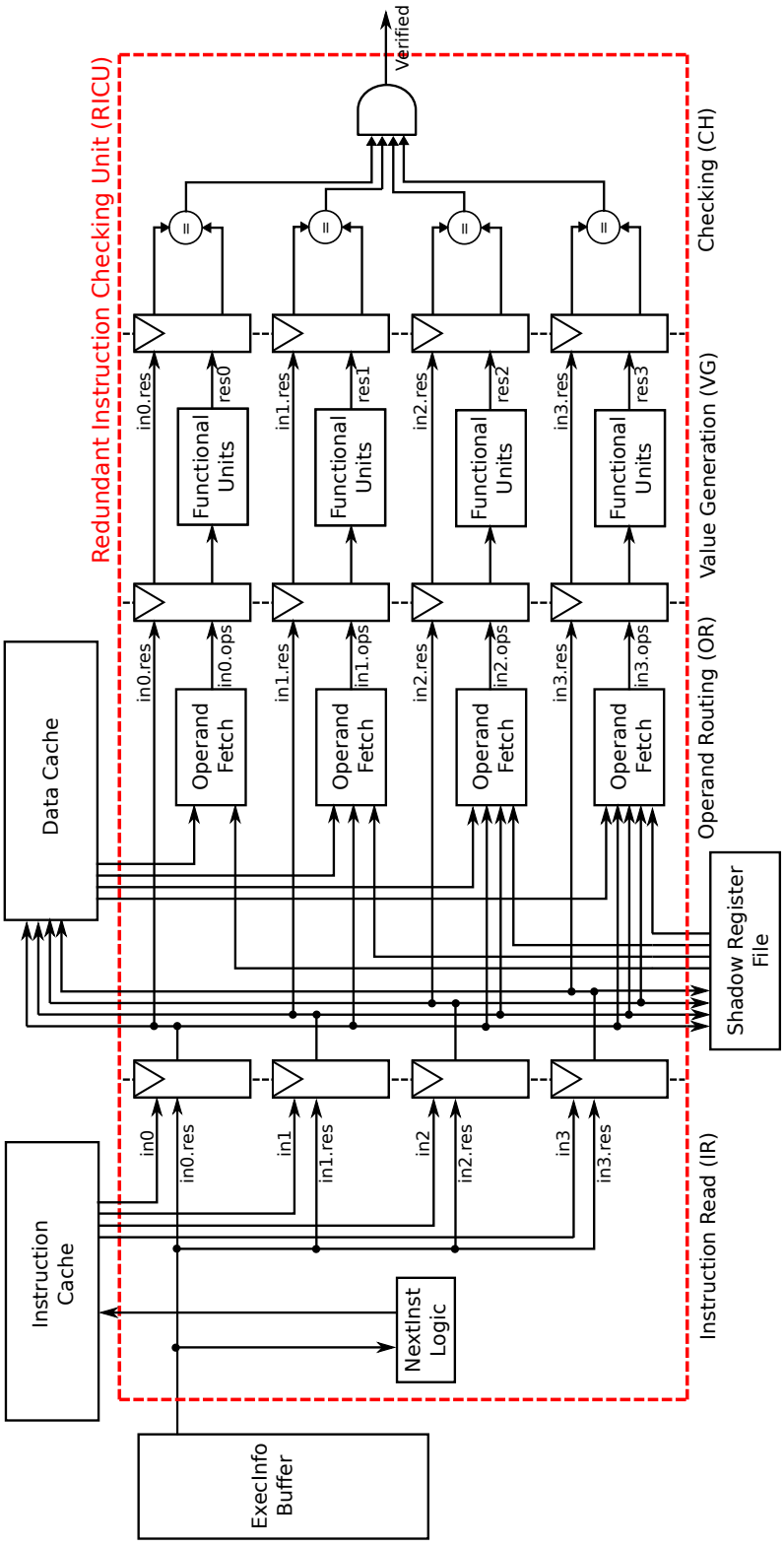


Figure 4.6: The Sentry's Redundant Instruction Checking Unit (RICU). This unit can check the correctness of up to 4 instructions in parallel.

Instruction Type	Packet Type	Data Sent
Memory Access (L1 cache hit)	Instructions	Address
	Data	Address
Memory Access (L1 cache miss)	Instructions	Instruction Cache Line, MAC
	Data	Data Cache Line, MAC
	Merkle Tree Nodes	Counter/Intermediate Nodes Cache Line
Control Flow	Jump Not Taken	Jump Status Bit
	Jump Taken	Jump Status bit, Jump Destination
Other Instructions		Results

Table 4.1: Information sent by the untrusted processor to the Sentry

ure 4.7. For example, if i_0 and i_1 are both 1, it indicates that trace information about the next two instructions is available in the buffer.

The IR stage also checks if any of the instructions available for checking in that clock cycle changed the control flow of execution. This is done by reading the `Jump Status bit` available in the `ExecInfo` buffer (inputs c_0, c_1, c_2, and c_3). The priority encoders and muxes in Figure 4.7 calculate the new value of `NextInst`. If none of the instructions to be checked have changed the processor's control flow, then the value of `NextInst` depends on the number of instructions picked up for re-execution in that cycle. For example, if i_0 and i_1 are both 1 while i_2 and i_3 are both 0 and all c bits are 0, then the value of `NextInst` must be incremented by 8 (the size of two instructions).

If on the other hand, one of the instructions read by the IR stage changes the control flow of execution, the `NextInst` register would need to reflect this change. For this purpose, the Sentry uses the addresses reported by the processor as the result of such control instructions. As an example, if three instructions are available to be checked (i_0, i_1, and i_2 are all 1) and only c_1 is 1, then the new value of `NextInst` is set to a_1. Due to limits on the number of times the `NextInst` register can be written per cycle, the IR stage only advances the instructions corresponding to i_0 and i_1 to the OR stage for checking in that cycle. The instruction corresponding to i_2 is delayed until the next cycle.

The OR stage determines the operands to be used for redundant execution by the Sentry. At this stage, the RICU is split into *n* pipelines (P_0, P_1, ..., P_n), each corresponding to

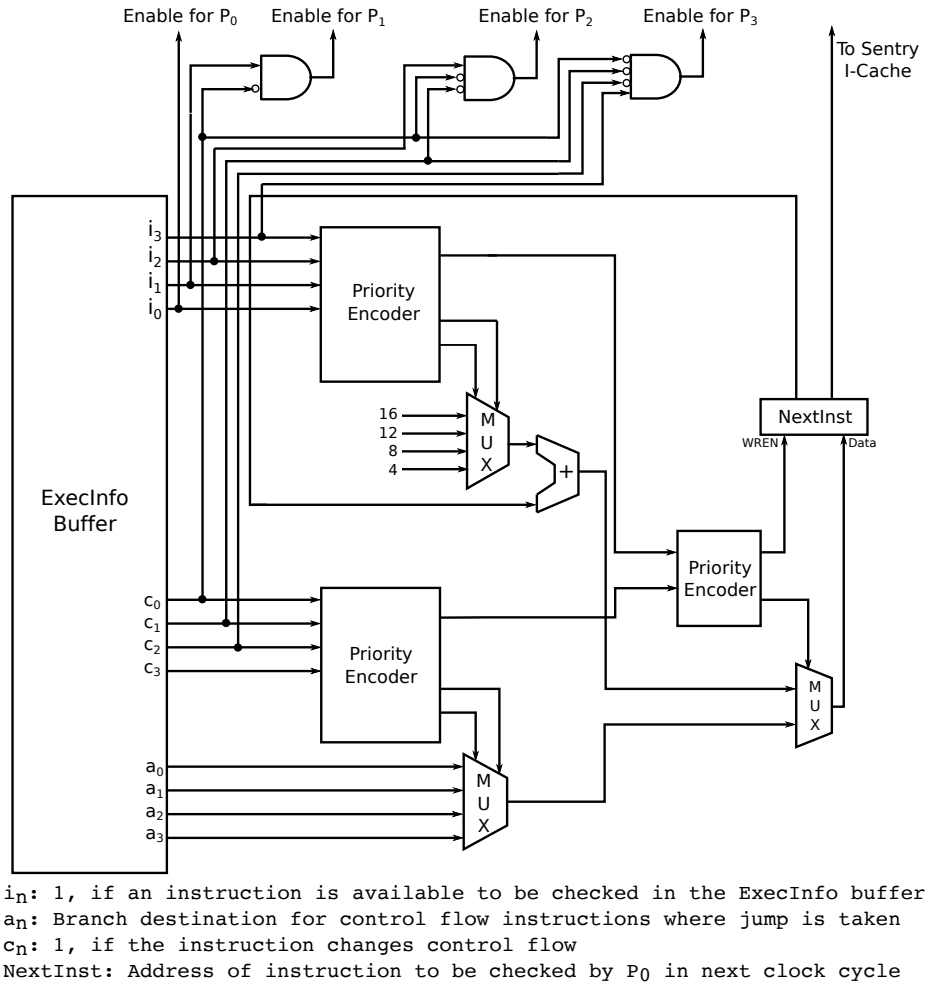


Figure 4.7: Logic to determine next instruction to be checked in the Instruction Read (IR) stage

an individual instruction to be verified. The RICU then reads the operands of these instructions and writes these values into pipeline registers. For nonmemory instructions, these operands could originate from either (a) the shadow register file; or (b) results of other instructions that enter the OR stage in the same cycle but occur earlier in the program order. For memory instructions, the operands would originate from the data cache instead of the shadow register file. The Operand Fetch unit in the OR stage disambiguates these cases and ensures that the correct values for the operands are passed to the VG stage.

Table 4.2 summarizes the rules used by the Operand Fetch unit to disambiguate between the various possible sources of register operands to be used for checking instructions. In

Pipeline	Possible Register Operands				
	Shadow Registers	in0.res	in1.res	in2.res	in3.res
P_0	✓	×	×	×	×
P_1	✓	✓	×	×	×
P_2	✓	✓	✓	×	×
P_3	✓	✓	✓	✓	×

Table 4.2: Operand Routing Rules for Disambiguating between Register Operands

every cycle, the earliest instruction in program order is always checked by the first pipeline, P_0. In this pipeline, the Operand Fetch unit reads the register operands from the shadow register file. Meanwhile, in pipeline P_3, the Operand Fetch unit checks if any of the results of the earliest instructions being checked in parallel (in0.res, in1.res, and in2.res as per the terminology in Figure 4.6) are used as an operand of the instruction to be checked by P_3. If such a dependence is detected, then the Operand Fetch unit reads the results from the corresponding pipeline and forwards it to the VG stage. If no dependences are detected, the Operand Fetch unit reads the register operands from the shadow register file.

The OR stage also speculatively writes the results of execution reported by the untrusted processor into the shadow register file for nonmemory instructions and into the data cache for memory instructions. Writing values speculatively into the shadow register file and the data cache allows the Sentry to avoid any inter-stage and intra-pipeline forwarding logic. It also ensures that any dependence checking is limited only to instructions that enter the OR stage in the same clock cycle.

If the instruction is a nonmemory instruction, the VG stage re-executes the instruction with the operands determined by the OR stage. The VG stage then passes this result to the CH stage. The CH stage checks that the result obtained from the VG stage is the same as the result retrieved from the `ExecInfo` buffer. Note that instructions that enter the VG stage together in parallel wait to proceed to the CH stage until the longest latency instruction has completed.

When the CH stage detects a mismatch, the Sentry flags an error and disables any output

resulting from that instruction using the mechanism described in Section 4.5. The Sentry can alert the user that an attack has been detected and prevent any external communication by instructions which directly or transitively depend on the errant instruction.

4.3 Memory Checking

For the Sentry to check instruction execution, it must have access to the data and instructions loaded by the processor. As data and instructions are stored in and delivered by untrusted memory, TrustGuard must protect their integrity. Prior works in memory integrity [109, 119] assume a secure processor that faithfully performs the cryptographic functions for ensuring integrity. In TrustGuard, however, the sensitive cryptographic functions must be performed by the Sentry, as it is the only trusted component. Thus, any memory integrity scheme must account for the latency of communication and the bandwidth between the processor and the Sentry.

4.3.1 Bonsai Merkle Tree

Past works have shown the strength of Merkle Tree-based schemes for protecting memory integrity [56, 103]. In a traditional Merkle tree, a single message authentication code (MAC) value is associated with every data block (typically a cache line). To protect against spoofing and splicing attacks, the MAC of the data block is a keyed cryptographic hash of the data itself and the address of the data block. A tree of MAC values is then built over the memory. The root of the tree is kept in a special on-chip register and this value never goes off-chip. When a memory block is fetched, its integrity can be checked by verifying the chain of MAC values starting from the MAC of the data block (leaf of the tree) up to the root MAC. As the root MAC present on-chip is built using information about every block in memory, the Merkle tree scheme ensures that an attacker cannot modify or replay any value in memory.

To minimize the overheads associated with maintaining memory integrity, TrustGuard adapts a variant of the Merkle Tree called the Bonsai Merkle Tree [109]. Unlike a traditional Merkle Tree, a Bonsai Merkle Tree also uses a counter value to calculate the MAC values associated with a data block. The counter represents the version of the data block, and is incremented any time a data block is evicted from the trusted cache. These counters are protected by MACs (MAC_C), which are keyed cryptographic hashes of the blocks containing the counters and the address of the counter blocks. The Bonsai Merkle tree is then built only over the counter block, unlike the traditional Merkle tree built on the entire data.

The choice of Bonsai Merkle trees over traditional Merkle tree implementations was dictated by the need to reduce the size of the shadow memory. As counters are much smaller than data blocks (1 cache line can hold counters for 32 cache lines holding data blocks), a Merkle tree over counters is much smaller and shallower than the Merkle tree over data. Moreover, using a Merkle tree over counter MACs offers the same integrity guarantees as a Merkle tree over data MACs [109].

Figure 4.8 shows the structure of the Bonsai Merkle tree used by TrustGuard. The root of the tree is stored in a special register in the Sentry. Note that this dissertation collectively refers to the metadata in memory needed to verify integrity, i.e. MACs, counters, and intermediate Merkle tree nodes, as *shadow memory*.

Instead of using a single counter per data cache line, TrustGuard uses the split counter proposed by Yan et al. [143]. The counter is split into two, with one smaller minor counter per-cache line, and a larger major counter that is shared by a group of cache lines. Cache lines are divided into 2KB groups in TrustGuard. Overflow of a minor counter requires incrementing the major counter and re-MACing of all the cache lines in the group. If the group counter overflows, the entire memory must be re-MACed with a different key. In TrustGuard, the minor counters are 14 bits long and the major counters are 64 bits long. This configuration achieves a balance between counter size and the number of re-MACing operations; it also means we can fit a group of 32 minor counters along with their major

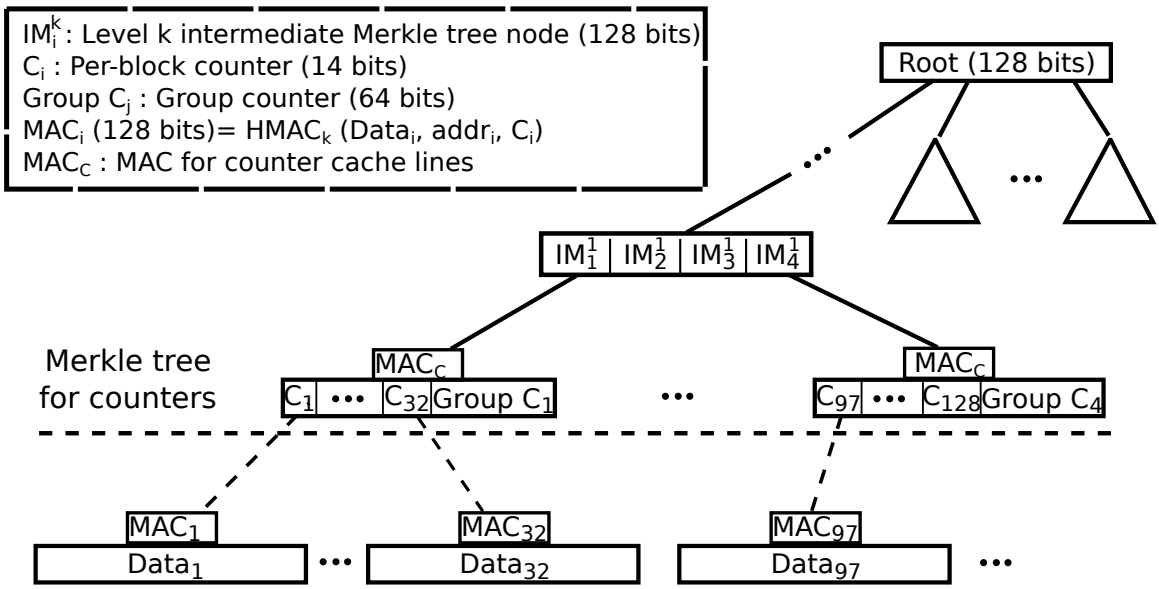


Figure 4.8: Bonsai Merkle Tree [109] used by TrustGuard to protect memory integrity

counter in a single 64 byte cache line.

The memory integrity scheme with a Bonsai Merkle tree works as follows: For every load instruction, the MAC (MAC_i) and the counter (C_i) corresponding to the accessed data block (C_i) are also read. The read MAC is compared against another MAC generated by using the address of the data block, the read data, and the counter. A mismatch in the MACs indicates that the integrity of either the data or the counter has been compromised. To ensure that the correct value of the counter was read from the memory, the MAC of the counter block is also verified. Finally, to ensure that old values of counters are not used to mount a replay attack, the chain of MAC values up to the root MAC is verified. For store instructions, a new MAC is generated for the modified data block with an incremented counter. Thus, the counter basically represents the version number of a cache line. New MACs are generated for the counter block (corresponding to the new value of the counter) and all the MAC values in the chain from the leaf node up to the root MAC are also re-computed and stored into memory.

The biggest issue with a Bonsai Merkle Tree implementation that works as explained above is the performance impact of all the shadow memory accesses. To avoid repeated

computation of Merkle tree nodes as blocks are read from and written to memory, prior work has proposed caching Merkle tree nodes in trusted on-chip caches [119, 109]. Using this optimization, the verification of a data block only needs to proceed up the tree until the first cached ancestor is found. Additionally, this optimization also enables the use of *evict counters* instead of *store counters*, i.e., counters only need to be incremented when a data block is evicted from the cache rather than on every store. TrustGuard leverages these optimizations proposed by prior work and introduces a cache on the Sentry to enable a more efficient implementation of the Bonsai Merkle Tree memory integrity scheme.

4.3.2 Cache Mirroring

Naïvely requiring the Sentry to access main memory to fetch shadow memory along with instructions and data would increase the complexity of the Sentry as it would need to interface with a memory controller. To keep the design of the Sentry relatively simple, the processor in TrustGuard performs all memory accesses (including shadow memory accesses) and sends this information to the Sentry. TrustGuard uses a *cache mirroring technique* to enable this scheme without incurring prohibitive performance overheads due to limited bandwidth between the processor and the Sentry.

In cache mirroring, the processor and the Sentry have L1 data and instruction caches of the same size[1]. For every cache line fill into the processor's L1 caches, the processor also sends this cache line to the Sentry so that the two sets of caches can maintain identical state. The Sentry and processor are set to use the same replacement policy, so for every eviction from the processor's L1 caches, the Sentry will evict the same line from its own caches. This cache mirroring scheme has several advantages. First, the processor's cache now acts as an oracle prefetcher, ensuring that any memory value required by the Sentry's RICUs will always be present in the Sentry's caches at the time of checking. Second, cache mirroring leverages fill, replacement, and timing logic already present on the processor's

[1]The Sentry only has single-level L1 caches, regardless of how many other cache levels in the processor.

caches, adding only extra communication from the processor to the Sentry.

In addition to data and instructions, the Sentry needs all of the necessary shadow memory needed to verify an incoming cache line's integrity. To achieve this, the processor performs all shadow memory accesses as if it was going to verify each cache line's integrity itself. Thus, every time the processor fetches a new cache line into its L1 cache, it also sends it across to the Sentry, and additionally sends any cache lines not already cached corresponding to the counter for the cache line and the intermediate nodes of the Bonsai Merkle Tree. Because any L1 misses and corresponding fills will be mirrored into the Sentry's caches from the processor, sending all the processor's cache fills provides the Sentry with enough information to verify integrity of the cache lines.

Because the caches on the Sentry are trusted, once a cache line is verified, no additional checking is necessary for future accesses to the cache line unless the cache line is evicted. Many of the counters and MACs in the Merkle tree nodes are likely already available in the processor's (and therefore also the Sentry's) L1 caches due to memory locality and adjacent placement of counters and MACs. Additionally, the processor only needs to send ancestor intermediate nodes up until a cached intermediate ancestor is found in its L1. These effects reduce bandwidth and improve performance.

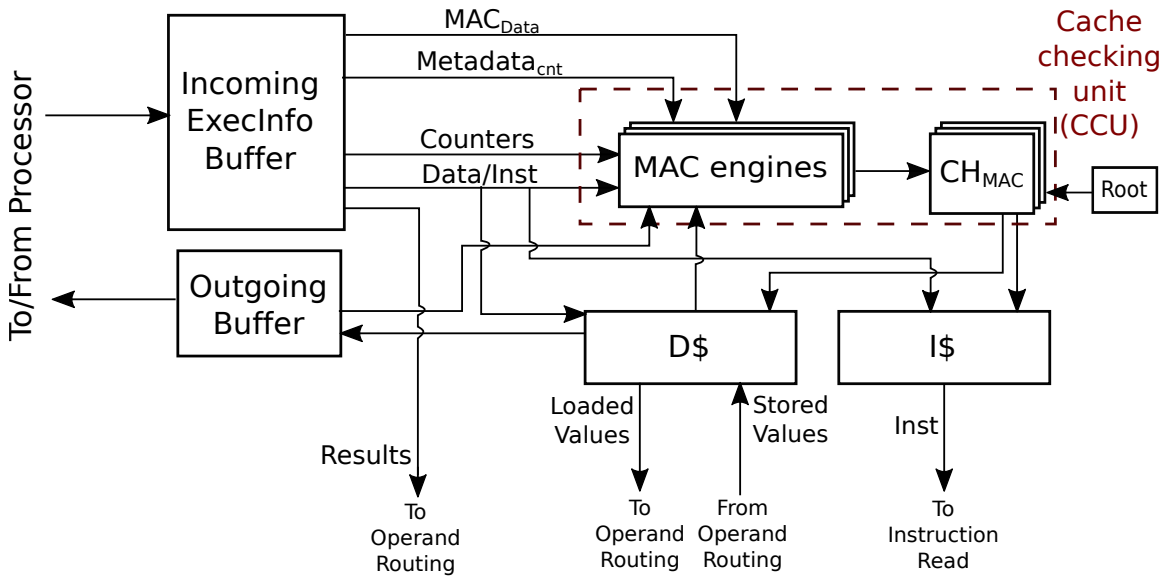


Figure 4.9: Design for the Sentry components that check memory integrity of cache lines

4.3.3 Cache Checking Unit

Figure 4.9 shows the structure of the cache checking unit (CCU) and its connected structures in the Sentry, which is responsible for determining if the integrity of the memory values communicated to it by the untrusted processor is maintained. On receiving new data or instruction cache lines from the processor, the Sentry speculatively stores the new cache line in the data or instruction cache, allowing for the RICU to proceed with instruction checking without waiting for integrity to be verified. All subsequent output instructions in the Pending Output Buffer (Section 4.5) are held until this speculation is confirmed correct.

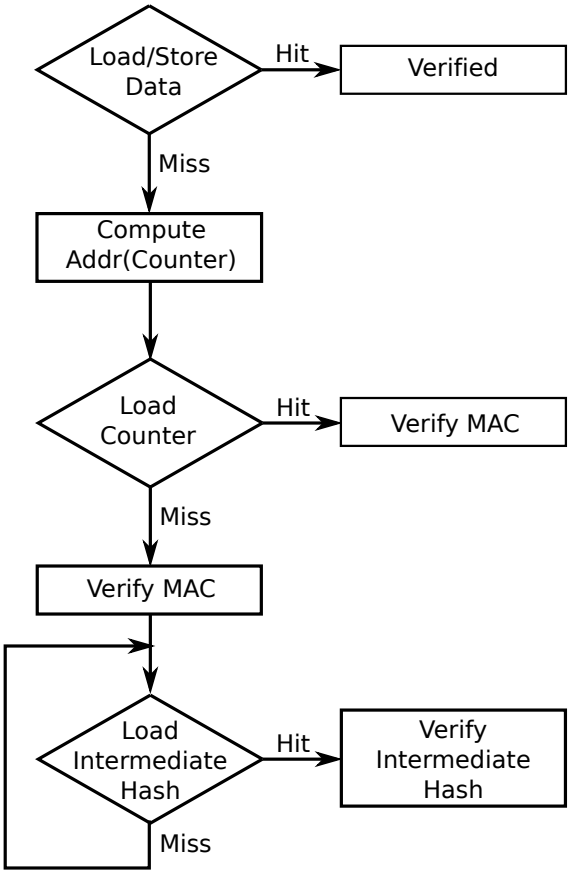


Figure 4.10: Flowchart illustrating CCU behavior for `load` and `store` instructions

For memory instructions (both loads and stores), the Sentry first checks if the data to be accessed was in the cache (hit) or it was sent by the processor (miss). For a cache hit on a load, the value from the cache is used for the redundant execution of the load instruction

and no further verification is done. Similarly, for a store, the Sentry performs the store to the cache line using the address reported by the processor.

In case of a miss, the Sentry first copies the cache line reported by the processor from the `ExecInfo` buffer. The Sentry then looks for the counter corresponding to the accessed cache line. The counter in TrustGuard represents the version of the accessed cache line, and is incremented every time the cache line in question is evicted from the Sentry's cache. If the counter is present in the Sentry's cache, then the Sentry uses the counter value to calculate the MAC for the data and compares it against the MAC reported by the processor in the CH_{MAC} unit. If the counter is not found in the cache, then the CCU finds the counter along with the corresponding MAC among the information sent to it by the processor. The CCU must check the MACs of the counters as well as all the intermediate Merkle tree nodes towards the root until a cached ancestor is found on the Sentry's cache. This process is shown in the flowchart in Figure 4.10.

If there are no cached ancestors, then the Sentry would verify each intermediate node of the Merkle tree on the path from the counter block to the root. It would finally finish with verifying the root value which is stored only in a special register on the Sentry. Once verification is complete, a confirmation signal is sent to the Pending Output Buffer (Section 4.5) to alert speculative output operations that they are no longer dependent on this instance of speculation.

Whenever a dirty data cache line is evicted from the Sentry's cache, the Sentry increments the corresponding counter and generates a new MAC using the data and the updated counter. If accessing the corresponding counter resulted in a cache miss on the processor, the Sentry must also verify the integrity of the counter block along with any intermediate Merkle tree nodes. Once the new MAC is calculated, it is sent back to the processor and stored back to memory (Figure 4.11(a)). Evicted dirty cache lines that correspond to counters are also sent to the processor (Figure 4.11(b)). However, for these cache lines, the parent nodes first have to be updated to reflect the new value for the counter line being

evicted. Only after these updates are propagated up the Merkle tree (until the first ancestor node that remains in the cache is encountered) is the eviction performed.

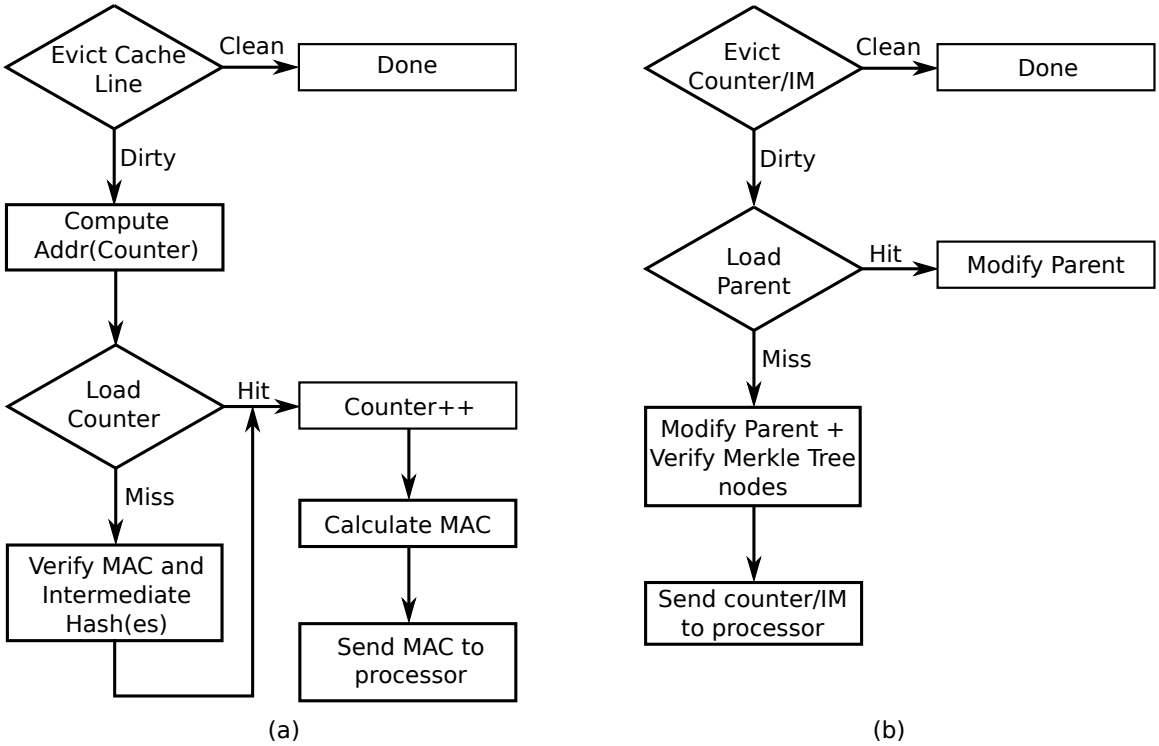


Figure 4.11: Flowchart illustrating cache checking unit behavior for (a) eviction of data from Sentry cache (b) eviction of counters/intermediate nodes from Sentry cache. IM corresponds to a node with intermediate hashes.

The added latency of communication between the untrusted processor and the Sentry raises the question of ensuring consistency of shadow memory values. For example, when a store is executed by the processor, it changes in the value stored in the addressed memory location. However, the MAC of the cache line stored in memory (and thereby accessed by the processor) still reflects the MAC corresponding to the old contents of the cache line until that line is next evicted from the Sentry's cache. It is possible that the processor refetches an evicted cache line (either from its L2 cache or from memory) before receiving the new MAC from the Sentry. In this case, the processor would report a stale MAC to the Sentry.

To ensure that the Sentry does not check memory integrity against stale MACs, the Sen-

try uses a small content addressable memory (CAM) structure called the *outgoing buffer*. The MAC and Merkle tree nodes being sent to the untrusted processor are all stored in the Sentry's outgoing buffer, as shown in Figure 4.9). When the processor receives these values and stores them to memory, it acknowledges the receipt of the shadow memory. Upon receiving the acknowledgement, the Sentry can remove the corresponding value from the outgoing buffer. Note that the outgoing buffer can be queried for outgoing cache lines by the Sentry, allowing it to continue with checking the integrity of cache lines without having to wait for the processor to complete the shadow accesses, thereby reducing the performance impact on the system.

4.3.4 Code Integrity

As stated earlier, the TrustGuard architecture only allows communication of correctly executed results of signed programs. It is assumed that all programs authorized to communicate externally are signed by a trusted authority. To ensure that only signed programs can communicate externally, the Sentry contains a special installation mode, similar to systems used in previous work on software integrity [51, 76]. The installation mode can be used to verify the signatures of trusted programs, and thus bootstrap the system. After a program's signature is verified, an HMAC is generated and stored along with the program.

TrustGuard requires a trusted loader to enable the secure loading of programs in memory. When a program is loaded, the program loader uses a special `load program` instruction to put the Sentry into load program mode. In this mode, the loader and the Sentry load the program and verify the HMAC that was computed during installation. Once the HMAC is verified, the Sentry generates the shadow memory values for the program instructions and stores these values into memory along with the program itself. Once the shadow memory values are all generated, the program leaves the loading mode and enters normal execution. The MAC-based memory integrity mechanism subsequently ensures that no instruction is subsequently modified after it is loaded into memory.

4.4 Link Compression

Limited available bandwidth between the untrusted processor and the Sentry requires a reduction in the amount of communication between them. TrustGuard uses a hybrid of two simple compression algorithms to achieve this reduction by compressing the data being sent on the link between the processor and the Sentry. The algorithms used are Significance-Width Compression (SWC) and Frequent Value Encoding (FVE) [125]. Since the compression and decompression logic is very simple, implementing this logic results in minimal added logic and complexity to the Sentry.

Original Data	Prefix Code	Compressed Data	Compressed Size (bits)
0x000000zz	00	0xzz	10
0x0000zzzz	01	0xzzzz	18
0x00zzzzzz	10	0xzzzzzz	26
0xzzzzzzzz	11	0xzzzzzzzz	34

Table 4.3: Significance Width Compression with 32-bit original data

SWC leverages the observation that the actual values of data used in programs are often small enough to be represented using fewer bits. It compresses data by replacing leading zeros with a prefix code, which indicates the compressed size of the value. Table 4.3 shows the SWC encoding used by TrustGuard. FVE uses a small value cache on each side of the link to achieve value reuse on the link. If a value to be sent on the link is found in cache, the processor sends the index instead of the data. For example, for 64-entry value caches, a cache hit would only require a 6-bit index to be sent.

For TrustGuard's link compression scheme, FVE is applied first using a 64-entry value cache. If a value is not found in the cache, the cache is updated using Least Recently Used (LRU) policy and the value sent is compressed using SWC.

4.5 Preventing Incorrect Output

As shown in Figure 4.1, the Sentry resides physically between the untrusted processor and the interconnect network connected to the peripherals. The Sentry prevents results of unverified instructions from communicating to the peripherals via the *Pending Output Buffer* (POB).

In TrustGuard, all I/O is performed through explicit I/O instructions. Checking an output instruction involves checking that all computations leading to the particular instruction are correctly performed. The correctness of these dependent instructions is ensured by the RICU in the Sentry. For output instructions, the Sentry queues up the values to be sent externally in the POB. Once an output operation makes it to the POB, it is assured that all computations leading up to that output have been checked by the RICU. However, these checked output operations may be dependent on a cache line that is in the process of having its integrity checked, as described in Section 4.3. This information is given to the POB by the CH_{MAC} component in the Sentry's cache checking unit. The POB uses this information to confirm that the cache checking unit has verified any speculative fills of cache lines into the Sentry's cache. Upon confirmation, the POB allows the output to proceed to the peripheral.

Peripheral driven direct memory access (DMA) is not supported in TrustGuard. Only trusted software can direct DMA transfers to and from untrusted peripherals. The Sentry includes a *Memory Access Manager* (shown in Figure 4.1) that creates MACs for the data read into memory. This impacts DMA perfomance because the data must be MACed on the way into memory.

4.6 Changes to Processor Design

TrustGuard defines an interface between the processor and the Sentry that requires some processor internals to be exposed to the designer of the Sentry . This results in the following

set of rules that the Sentry must conform to:

- As TrustGuard defines the correctness of instruction execution with respect to the specifications of the ISA, the Sentry must be designed to support the same ISA as the untrusted processor.
- The Sentry must have the same number of architectural registers as the untrusted processor.
- The cache mirroring scheme requires the Sentry's data and instruction caches to be of the same size as the processor's L1 data and instruction caches respectively. Additionally, the Sentry and the untrusted processor must use the same logic to access MACs, counters, and intermediate nodes of the Bonsai Merkle tree.
- The Sentry must use the same link compression and decompression techniques as the processor.

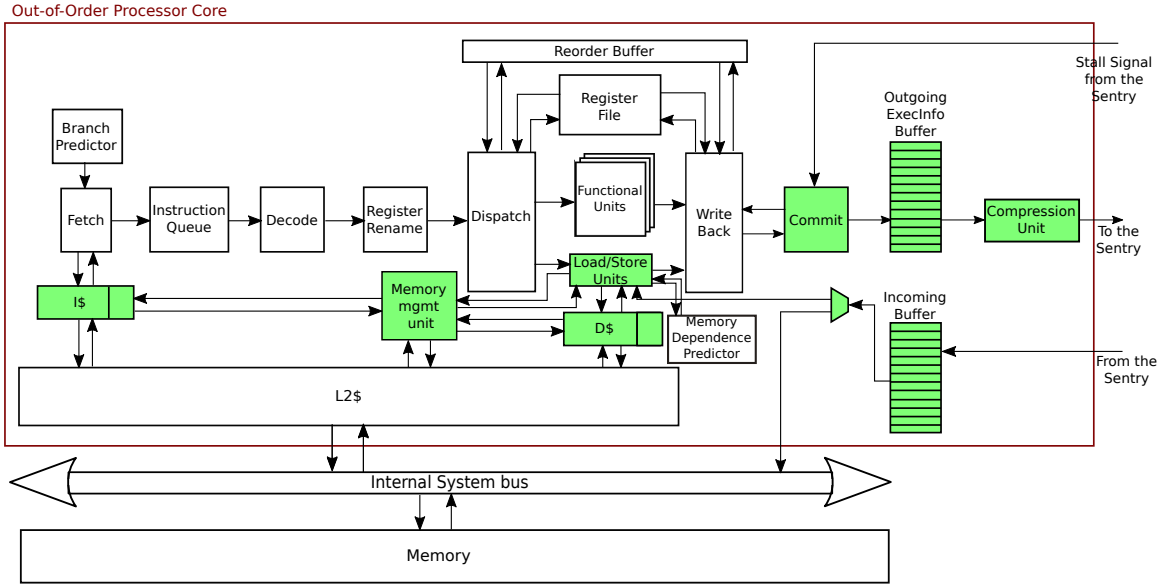


Figure 4.12: Summary of modifications to the processor in the TrustGuard architecture. Shaded/colored boxes indicate modified components.

The nature of interaction between the untrusted processor and the Sentry in TrustGuard requires several modifications to be made to the design of the untrusted processor. These modifications to the processor design are summarized in Figure 4.12.

The most substantial change in the processor design involves the realization of loads

and stores from/to memory. The untrusted processor must now support loading and storing or MACs from and to cache lines and memory. Additionally, the processor must include logic that also supports the loading and storing of counter values and intermediate Merkle tree nodes for memory accesses. This would entail a modification of the Memory Management Unit inside the processor. Note that these changes are similar to the changes suggested by prior work on memory integrity, such as AEGIS [119]. Additionally, the ISA must be augmented with a `load program` instruction that enables the Sentry to generate MACs for signed code being executed on the system.

However, unlike prior memory integrity techniques, the TrustGuard processor would have to work differently to support the updates of shadow memory values. This is because there is a time lag between the processor accessing a memory location and the Sentry sending the shadow memory values for that address back to the processor. The processor must now support the delayed insertion of these shadow memory values. For this, the processor needs a mechanism that allows insertion of these *shadow stores* into the load/store unit. Note that shadow stores in TrustGuard share the same load-store queue (LSQ) as other data. However, as the original memory and shadow memory accesses are disjoint, a secondary LSQ could be utilized for all the shadow memory accesses. This could further improve the performance of the processor due to the reduced pressure on the main LSQ.

The second set of changes in the processor involves enabling communication with the Sentry. As seen in Table 4.1, the processor must send information such as cache lines, results of instruction execution, shadow memory accesses, etc. to the Sentry. This communication is synchronized through the addition of two buffers to the processor's design: one for outgoing communication to the Sentry (`Outgoing ExecInfo Buffer`) and the other for information received from the Sentry (`Incoming Buffer`). A compression unit must also be added to support link compression of the values buffered in the `Outgoing ExecInfo Buffer`. Finally, the processor must support an input `stall` signal that originates from the Sentry. The Sentry would signal a stall whenever it falls

behind the processor's execution and the `ExecInfo` buffer on the Sentry is full. When this signal is found asserted, the processor stalls its pipeline at commit and waits for the Sentry to indicate that it is fine to proceed with execution.

Chapter 5

Detection of Malicious Behaviors

To evaluate its containment capabilities, we modeled the TrustGuard architecture in the gem5 simulator [28]. The performance analysis chapter (Chapter 6) describes the details of the simulation environment. We modified the behavior of various components in the modeled system to simulate different attack scenarios. Table 5.1 lists these attack scenarios, and the system components that were modified for the purpose.

	Malicious Behavior	**Processor Component(s) Modified**
1.	Incorrect arithmetic execution	ALU Multiplier [27]
2.	Changing loop condition to execute one more iteration	Branch Unit
3.	Modification of register values	Register File
4.	Jumping to an incorrect branch target	Branch Target Buffer Branch Unit
5.	Insertion of non-program instruction	Instruction Fetch Unit
6.	Skipping execution of program instructions	Dispatch Unit
7.	Reordering instructions in processor's instruction stream incorrectly	Dispatch Unit
8.	Modification of cached values	Cache controller
9.	Modification of uncached values (in memory)	Memory controller
10.	Relocation of data and MACs in memory	Memory controller
11.	Replay of data in memory	Memory controller
12.	Unsigned program communicating externally	-

Table 5.1: Attack scenarios related to incorrect instruction execution

This chapter describes the protection mechanisms offered by TrustGuard, which led to the detection of these malicious behaviors. The implemented malicious behaviors broadly fell into two categories: (1) incorrect instruction execution; and (2) manipulation of values flowing through memory.

5.1 Incorrect Instruction Execution

Attack scenarios 1–8 in Table 5.1 correspond to incorrect instruction execution because of malicious processor components. TrustGuard was able to detect and contain malicious behaviors in each of these cases. Attack scenarios 1 and 2 involve incorrect execution by the processor's functional units. Attack scenarios 4-7 involve subversion of the correct control flow of the program, either by the processor's fetch unit or the dispatch unit. Attack scenario 3 involves incorrect state maintained by the processor's register file, while attack scenario 8 entails malicious modifications of the processor's cache subsystem.

An example of incorrect execution by the processor's functional units could be the processor either discarding or changing the results of one or more instructions in the program. Figure 5.1 shows an example where a multiplier in the untrusted processor manipulates the result of `r1 = r0 × 3` to be `4000` instead of `3000`. Such a multiplier bug that computes the wrong product for a known, single pair of 64-bit integers can enable adversaries to leak *any private key* used in *any RSA-based software* on that device, possibly using a *single chosen message* [27].

As shown in Figure 5.1, the processor encounters a cache miss when it goes to the fetch the cache line containing the multiply instruction (instruction `I1`). The processor sends this cache line along with its associated MAC value to the Sentry. The Cache Checking Unit of the Sentry calculates the MAC for the cache line and checks it against the sent MAC value. Meanwhile, the Sentry's Redundant Instruction Checking Unit (RICU) detects the manipulation of the result by the processor when it re-executes the multiply instruction.

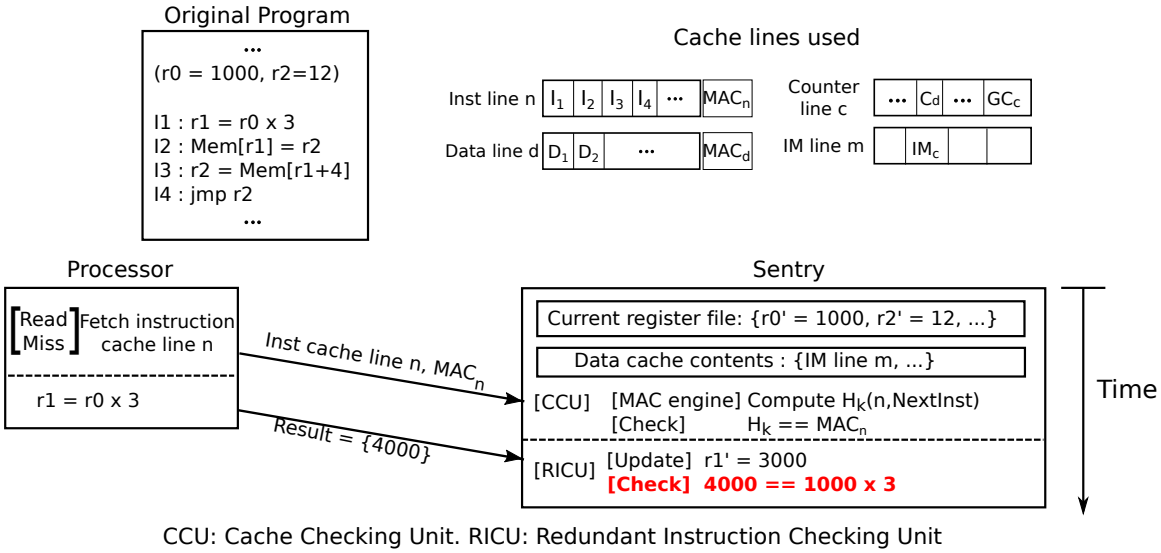


Figure 5.1: Example of how the Sentry detects the incorrect execution of an instruction by a processor's functional unit. `rx'`: Shadow register in the Sentry for the register `rx` in the untrusted processor. H_k: HMAC function with key k.

This situation corresponds to the attack scenario 1 mentioned in Table 5.1.

An alternate way for the processor to change the execution of the multiply instruction would be by changing the value of an operand. For instance, the register `r0` could be changed from `1000` to `1001`. Thus, the processor would send the result `3003` instead of the correct result `3000`. This situation corresponds to the attack scenario 3 mentioned in Table 5.1. The invalid change in the value of `r0` will not be seen when the Sentry RICU accesses its shadow register file. Therefore, the result of the instruction re-execution by the Sentry will differ from that reported by the processor and the manipulation by the processor will be detected.

Attack scenario 2 in Table 5.1 presents another interesting way to compromise a processor's functional unit. For this case, the branch predictor in the last iteration of a loop might indicate that the control flow would switch back to the beginning of the loop body, based on the branch history. Ordinarily, the branch unit in the processor would detect the branch misprediction and cause a squash of any instructions that start executing because of the misprediction. However, the branch unit itself might be compromised in a way that it ignores the correctly generated program counter value. Thus, for the corresponding branch

instruction, the processor would send over the incorrect result of the branch to the Sentry.

The Sentry uses this value in the `NextInst` logic (Figure 4.7) to determine the next instruction to be checked. Thus, instructions from the extra iteration of the loop would enter the RICU pipeline. However, the Sentry would use its own replicated state in the Value Generation (VG) stage to calculate the next address it should have used for checking instructions. A comparison of this generated address to the one reported by the processor will reveal the incorrect execution. As the Sentry checks instructions in program order, this incorrect behavior will be detected before any of the subsequent instructions executed by the processor are deemed to be correct. The same method also detects the behavior in attack scenario 4 (Table 5.1), when a modification in the branch target buffer and the branch unit result in a jump being taken to an incorrect branch target.

Insertion or Reordering of Instructions. Attack 5 involves a hardware Trojan in the untrusted processor inserting malicious instructions into its pipeline that did not come from a valid program. A similar Trojan was described by King et al. in the context of Illinois Malicious Processors (IMP) [75]. Through this backdoor, a malicious service allows any user to login to the system by sending an unsolicited network packet to the target system. The act of inspecting the packet triggers a hardware Trojan, which executes the packet contents invisibly as new firmware. When the malicious firmware detects a user trying to login into a particular application using a fake password, it modifies the return value of the password checking function to return true instead of false.

Consider the example shown in Figure 5.2. Assume that the processor inserts the instruction `r0 = r0 + 0x1` just before the instruction `I2` (`r1 = r0 + 0x8000`) to maliciously increment the value of `r0` as part of an attack. The processor can choose whether to send this instruction's result or not to the Sentry.

First assume the malicious instruction's result (`0x1001`) is sent to the Sentry, as shown in Figure 5.2. The Sentry will pop a result off the `ExecInfo` buffer and use that value to check against the result of re-executing the next instruction `I2`, `r1 = r0 + 0x8000`.

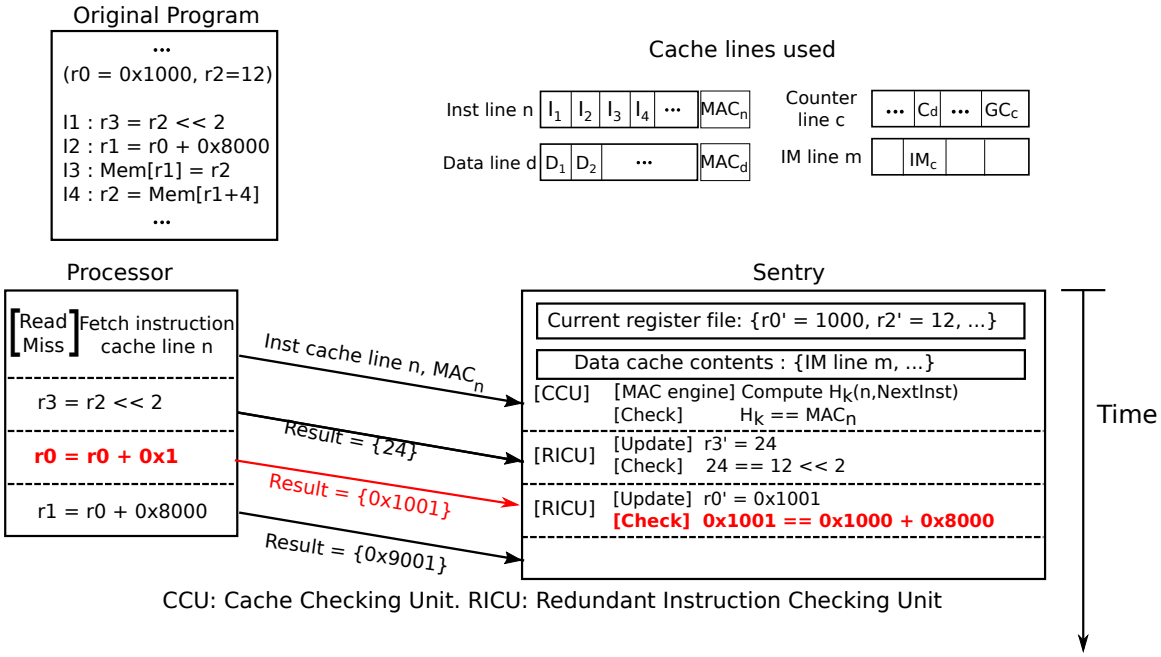


Figure 5.2: Example of how the Sentry detects insertion of malicious instructions. `rx'`: Shadow register in the Sentry for the register `rx` in the untrusted processor. H_k: HMAC function with key k.

The Sentry will calculate the correct value of `0x9000` and detect a mismatch when comparing against the malicious value of `0x1001`.

Alternatively, the processor could choose to not send the result of the maliciously inserted instruction. In this case, the untrusted processor will advance to the next instruction, and produce the result `0x9001` for instruction . However, the result of the execution of the next instruction (`0x9000`) will not match the result reported by the processor (`0x9001`), and this malicious behavior would be detected. This situation is illustrated in Figure 5.3.

5.2 Manipulation of Values Flowing Through Memory

In addition to malicious behaviors due to modifications of processor components, we also tested the TrustGuard implementation against malicious modifications to memory (Attack scenarios 9–12 in Table 5.1). We implemented the following attack scenarios for this purpose:

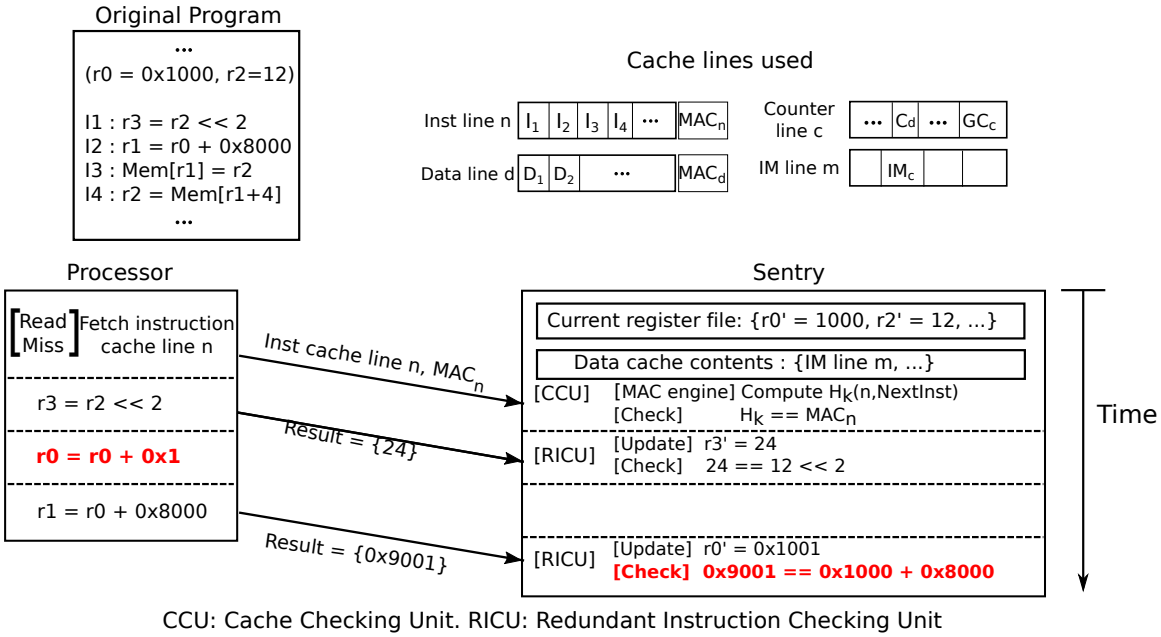


Figure 5.3: Example of the processor inserting an instruction and not reporting the results of that execution. `rx'`: Shadow register in the Sentry for the register `rx` in the untrusted processor. H_k: HMAC function with key k.

- Modification of data in memory
- Relocation of data/MACs in memory
- Replay of data in memory
- Unsigned program communicating externally

Consider the example from Figure 5.4. The untrusted processor could lie to the Sentry about faithfully executing `Mem[r1] = r2`. It could instead store some value other than `12`, such as `13`, to `0x9000`. As this store results in a write miss, the processor would bring the cache line and any other necessary cache lines corresponding to its shadow memory accesses (counters and Merkle tree nodes) into its own cache. It would also send those cache lines across to the Sentry.

The Sentry's cache checking unit (CCU) checks that the integrity of the data and that of its corresponding counter are not violated. Any external communication following the store in question is contingent upon this integrity check finding the cache line correct. Meanwhile, the store instruction goes to the RICU, which stores the correct value (`12`,

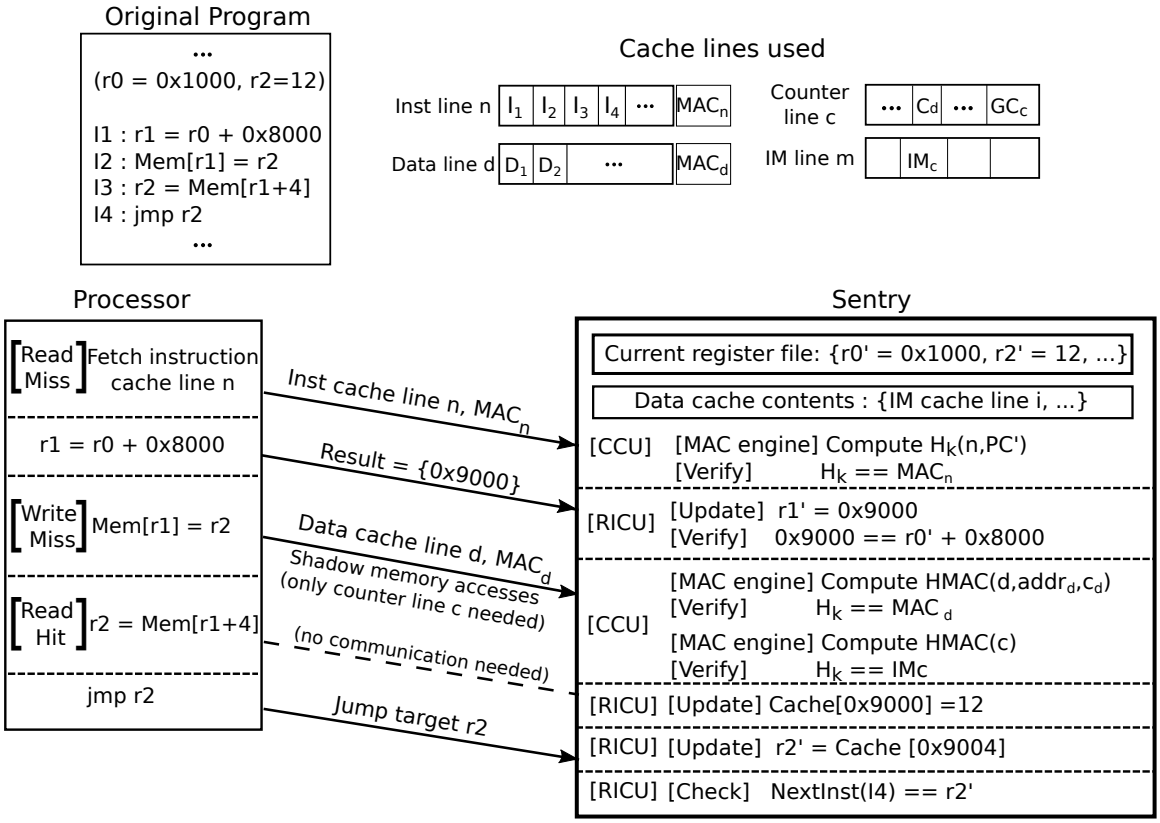


Figure 5.4: Example of the processor detecting illegal modification of values in memory. `rx'`: Shadow register in the Sentry for the register `rx` in the untrusted processor. H_k: HMAC function with key k.

using the shadow register file) into the Sentry cache at the correct address.

In this case, the processor may continue executing using the wrong value it stored to memory; however, the Sentry would perform the same calculations using the *correct* value determined by its internal state. Note that any output depending on the incorrectly executed store would actually use the correct value instead, as long as that cache line is present on the Sentry's cache. If this line were to be evicted, the Sentry would send a MAC of the cache line, assuming that the value stored was `12`. The next time the line is loaded back into memory, the Sentry's cache checking integrity would discover the maliciously executed store due to a mismatch in the MAC values.

Another possible attack could be attempted by replaying old data and MACs already seen at a location in memory. For example, assume that the memory at location `0x9000`

has value and MAC pair `(0xa,0xb)` prior to when instruction `Mem[r1] = r2` executes in Figure 5.4. Instead of executing `Mem[r1] = r2`, the processor could simply take no action and leave the data and MAC pair `(0xa,0xb)` in memory.

If the cache line containing the address `0x9000` remains in the Sentry's cache, the incorrect value will be seen the next time a load occurs at address `0x9000`. Thus, the attempted replay will be detected. Alternatively, if the line containing `0x9000` is evicted from the Sentry's cache, then the Sentry will increment the counter of the line and generate a new MAC (including the counter). It will then send all these generated values (MAC, counter, etc.) to the processor for storing to main memory. The next time the value is loaded, the MAC sent to the Sentry will not match the data stored there (as the new MAC is calculated using the new stored value). The processor may choose to not store the MAC it receives back for the dropped store. However, again the next time the value is loaded, the MAC loaded from the memory will not match, because the the loaded MAC was calculated using an old value of the counter.

Finally, the Sentry prevents all external communication originating from unsigned programs. The untrusted processor can run an unsigned program that reads sensitive data and sends it out through a peripheral. While this program could read the sensitive data, it would never be able to leak the information read. In this attack, the processor could choose whether or not to send the instructions of this unsigned program to the Sentry. If the instructions are sent, they will not pass the instruction integrity check on the Sentry because they will not be signed. This means no output operation can be completed. Alternatively if the instructions are not sent, no output operation will ever make it to the Sentry for checking. Thus, the output can never make it past TrustGuard's physical gap and again no external communication will occur.

5.3 Limitations of TrustGuard Security Assurances

As stated earlier, TrustGuard guarantees to the user that any external communication from a system originates from the correct execution of signed software. However, signatures associated with software do not guarantee that the software is actually secure. Thus, TrustGuard does not offer any security if the signed software itself is insecure or vulnerable. Similarly, the threat model for TrustGuard only focuses on explicit data leakage by untrusted hardware components. It does not handle information leaked via side channels, such as through encoding of sensitive information in energy usage patterns, long duration timing encodings, and implicit information leaked by failures. Finally, while TrustGuard guarantees the correctness of communicated data, it does not give any availability guarantees. As a result, the malicious hardware could simply affect the availability of the device by executing instructions incorrectly, thereby disabling any communication from the device.

Chapter 6

Performance Analysis

6.1 Methodology

As mentioned in Chapter 5, we modeled TrustGuard in the gem5 simulator [28] using an out-of-order (OoO) ARM-based untrusted processor. The focus of this performance evaluation was on measuring the effect of checking by the Sentry on the performance of the untrusted processor. In particular, the design space of parameters such as the parallelism in the Sentry, the frequency of the Sentry, and the Sentry-processor bandwidth was explored.

The architectural parameters used are specified in Table 6.1. The bandwidth between the processor and the Sentry was set by default at 10GB/s (easily achievable using an interconnect such as 16-lane PCIe [10, 9]). Each lane of the PCIe interconnect has one

Feature	**Parameter**
Architecture	ARMv7 32-bit
Processor Frequency	2 GHz
Processor Commit Width	8 instructions/cycle
L1 I-Cache	4-way set associative, 64KB, 64B cache line
L1 D-Cache	4-way set associative, 64KB, 64B cache line
L2 Cache	16-way set associative, 2MB, 12 cycle hit latency
Off-chip Latency	100 CPU cycles
MAC Function	HMAC with MD5

Table 6.1: Architectural parameters for simulation

pair of signal wires in each direction, so a 16-lane PCIe interconnect has 64 wires.

We simulated the execution of all 8 SPEC INT2006 workloads that work with gem5 [1] as well as three additional SPECFP benchmarks: 450.soplex, 453.povray, and 470.lbm. Whole program simulations were completed for three benchmarks—445.gobmk, 450.soplex, and 462.libquantum—as they had relatively short execution times. For all other benchmarks, five random checkpoints were chosen for simulation. From each checkpoint, benchmarks were simulated for 25 million instructions to warm up the microarchitectural state (*warm-up phase*). Subsequently, the benchmarks were simulated cycle-accurately for 200 million instructions to collect performance data (*timing phase*). For each experiment, the baseline is the out-of-order,superscalar processor-based system without any TrustGuard modifications (OoO only).

6.2 Additional Overheads of TrustGuard

To demonstrate the performance implications of TrustGuard, we evaluated the effects of various design parameters, such as the number of instructions checked in parallel by the Sentry RICU (Section 6.3), frequency of the Sentry (Section 6.4), and the bandwidth between the processor and the Sentry (Section 6.5), on the IPC of the untrusted processor. These simulations identified three major sources of runtime overhead introduced by the TrustGuard modifications to the system: (1) shadow memory accesses performed by the untrusted processor; (2) lag between the processor and the Sentry due to latency of communication and a mismatch between the processor's execution speed and the Sentry's checking speed; and (3) bandwidth constraints on the channel between the processor and the Sentry.

The first source of overhead is the increased cache and memory pressure from the shadow memory accesses performed by the processor. These accesses include MACs, counters, and intermediate hashes stored in non-leaf Merkle tree nodes. Figure 6.1 shows the effect of performing these shadow memory accesses on the IPC of the untrusted pro-

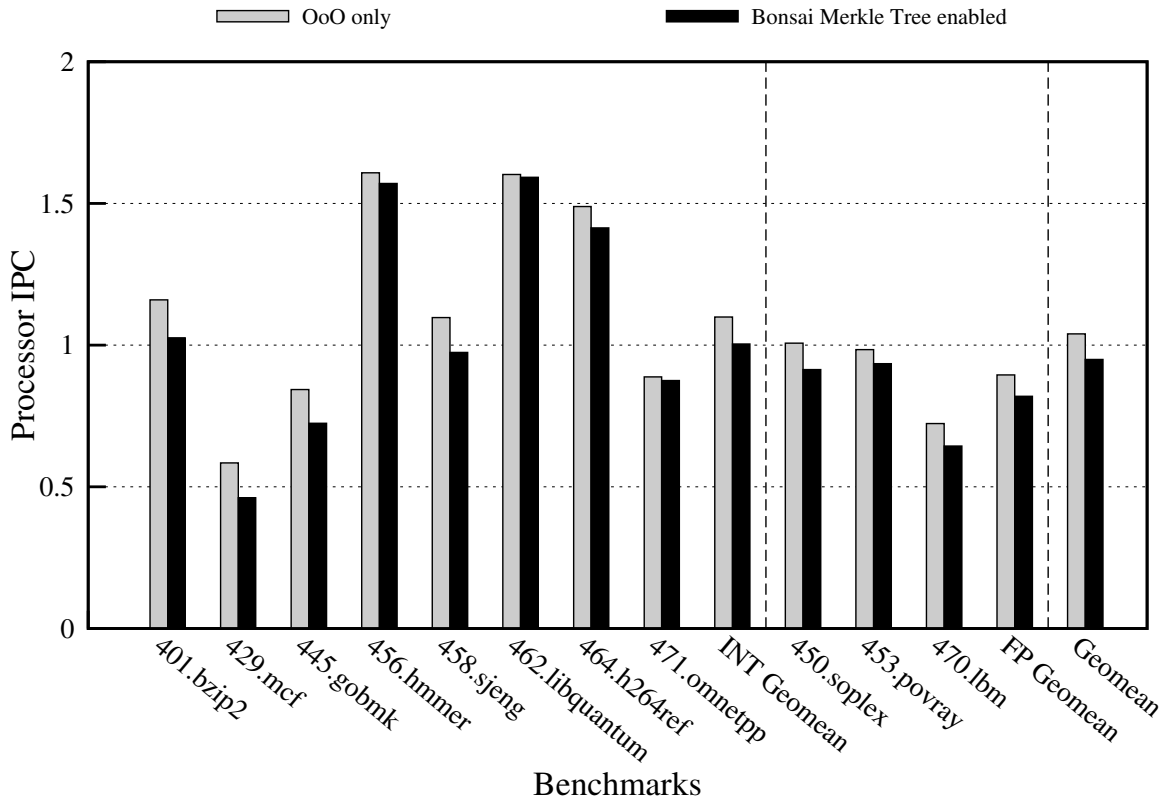


Figure 6.1: Effect on untrusted processor IPC of introducing shadow memory (SMACs, counters, and Merkle Tree) accesses

cessor.

Performing the shadow memory accesses resulted in a geomean IPC decline of 5.8%. The performance decline was higher than 10% for five benchmarks—401.bzip2 (11.5%), 429.mcf (21.0%), 445.gobmk (14.2%), 458.sjeng (11.3%), and 470.lbm (11.0%). The performance decline is explained by observing the number of L2 cache misses, which indicates the additional memory pressure introduced by shadow memory accesses. The number of L2 cache misses increased by a geomean of 55.7% across the eleven benchmarks. Additionally, the number of L1 cache misses increased by a geomean of 91.0%. This too adds some extra latency to memory operations as accesses to L1 caches are faster than accesses to the L2 cache.

The above-10% IPC decline for the aforementioned five benchmarks was mainly caused by an increase in the absolute number of data cache misses and L2 cache misses (these

Increase in	401.bzip2	429.mcf	445.gobmk	458.sjeng	470.lbm
D-Cache Misses	9,357,624	255,020,393	4,654,657	7,359,527	6,337,592
L2-Cache Misses	104,619	8,503,603	994,270	412,664	1,084,647

Table 6.2: Increase in Absolute Number of data cache misses and L2-cache misses across all the timing phase simulations of benchmarks for which IPC decline due to shadow memory accesses was more than 10%.

metrics are shown in Table 6.2). The other six benchmarks all had less than 3 million more data cache misses and less than 14,000 more L2 cache misses. By contrast, the five benchmarks showing more than 10% IPC decline all had at least 6 million more data cache misses and at least 100,000 more L2 cache misses. The latency added by these misses contributed to the higher performance decline for the five aforementioned benchmarks.

In addition to shadow memory accesses, additional overheads can be introduced if execution by the Sentry lags behind execution by the processor. Specifically, the introduction of the Sentry may lead to two new kinds of processor stalls.

The first of these stalls are *slow Sentry stalls* that are the result of the Sentry's inability to check instructions as fast as the processor executes them. Due to the lag in the Sentry's checking speed, the `ExecInfo` buffer on the Sentry fills up, thereby requiring the processor to stall its operations. The number of these stalls is dependent on the throughput of the Sentry, which in turn depends on the number of instructions that can be checked in parallel and the frequency at which the Sentry operates. Sections 6.3 and 6.4 quantify the number of slow Sentry stalls experienced by the processor, when the throughput of the Sentry is varied by varying the checking parallelism and the Sentry frequency.

The second type of stalls are *bandwidth stalls*, caused by bandwidth limitations on the channel between the processor and the Sentry. Consequently, the `ExecInfo` buffer in the processor fills up, thus necessitating stalls until the information in the `ExecInfo` buffer is communicated to the Sentry. The number of bandwidth stalls is dependent on the amount of communication between the processor and the Sentry. One way for TrustGuard to reduce the number of bandwidth stalls is to use the cached Bonsai Merkle tree scheme explained

in Section 4.3. In fact, our experiments showed that the processor spent almost all its time in bandwidth stalls, if the Sentry did not cache the Bonsai Merkle tree nodes. Additionally, the amount of communication can also be reduced by using link compression, as explained in Section 4.4. Section 6.6 quantifies the effect of link compression on the number of bandwidth stalls experienced by the processor. Meanwhile, Section 6.5 quantifies the number of bandwidth stalls when the bandwidth between the processor and the Sentry is varied.

6.3 Varying Checking Parallelism in the Sentry

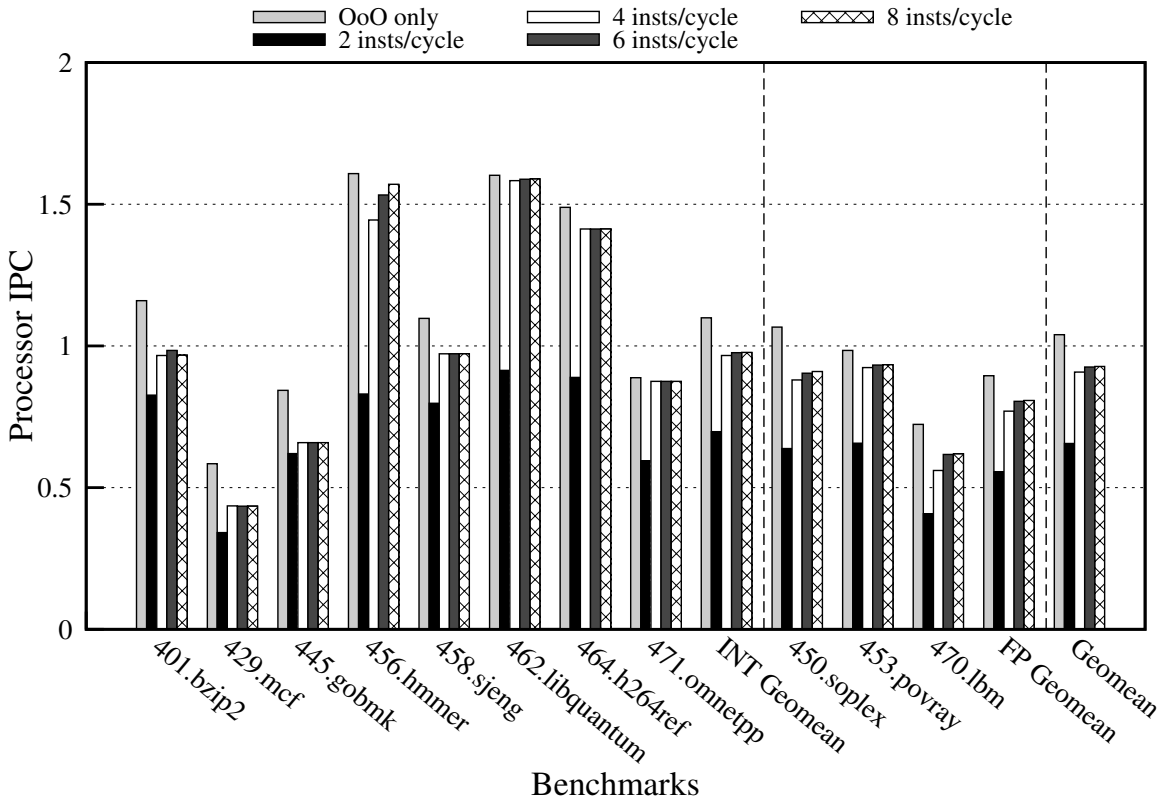


Figure 6.2: Reduction in IPC when varying the number of instructions that can be checked in parallel by the Sentry RICU. Sentry frequency: 1 GHz. Sentry-Processor bandwidth: 10 GB/s. Untrusted processor frequency: 2 GHz.

Parallel instruction checking enabled by the Sentry RICU allows the slower Sentry (lacking in various performance-related components) to keep up with the faster untrusted processor. As part of exploring the design space for the Sentry, we varied the number of

instructions that can be checked in parallel by the Sentry's RICU (we call this parameter the *RICU width*) and measured its impact on the IPC of the untrusted processor. Figure 6.2 shows the results of these simulations. For each configuration of RICU width, the Sentry-processor bandwidth was set to 10 GB/s and the frequency of the Sentry was set to 1 GHz (i.e. half the frequency of the untrusted processor).

As seen in Figure 6.2, the geomean decline in processor IPC with respect to the out-of-order only baseline was found to be 35.9%, 8.5%, 7.1%, and 6.1% respectively for RICU widths of 2, 4, 6, and 8 instructions per cycle. Configurations with higher RICU width showed higher processor IPCs. The improvement in processor IPC was directly correlated to the increase in the Sentry's checking throughput. Table 6.3 shows the checking throughput for various benchmarks (in terms of instructions checked per Sentry cycle), when the RICU width is varied. To understand the overheads that affected performance, we measured the number of slow Sentry and bandwidth stalls experienced by the aforementioned Sentry configurations. Figure 6.3 shows the percentage of slow Sentry and bandwidth stalls encountered during the timing phase as the RICU width was varied.

Benchmark	**Baseline Processor IPC**	**Sentry Throughput** (Insts/Sentry cycle)			
		2	4	6	8
401.bzip2	1.16	1.65	1.93	1.97	1.87
429.mcf	0.59	0.68	0.87	0.87	0.87
445.gobmk	0.84	1.24	1.32	1.32	1.32
450.soplex	1.07	1.28	1.76	1.81	1.82
453.povray	0.98	1.31	1.85	1.87	1.87
456.hmmer	1.61	1.66	2.89	3.07	3.14
458.sjeng	1.10	1.60	1.95	1.95	1.95
462.libquantum	1.60	1.83	3.17	3.18	3.18
464.h264ref	1.49	1.78	2.83	2.83	2.83
470.lbm	0.72	0.82	1.12	1.24	1.24
471.omnetpp	0.89	1.28	1.75	1.75	1.75
Geomean	1.04	1.32	1.82	1.85	1.86

Table 6.3: Checking throughput of the Sentry, in instructions per Sentry cycle, when the RICU width is varied

For RICU width of 2, the untrusted processor spent an average of 27.46% cycles in slow

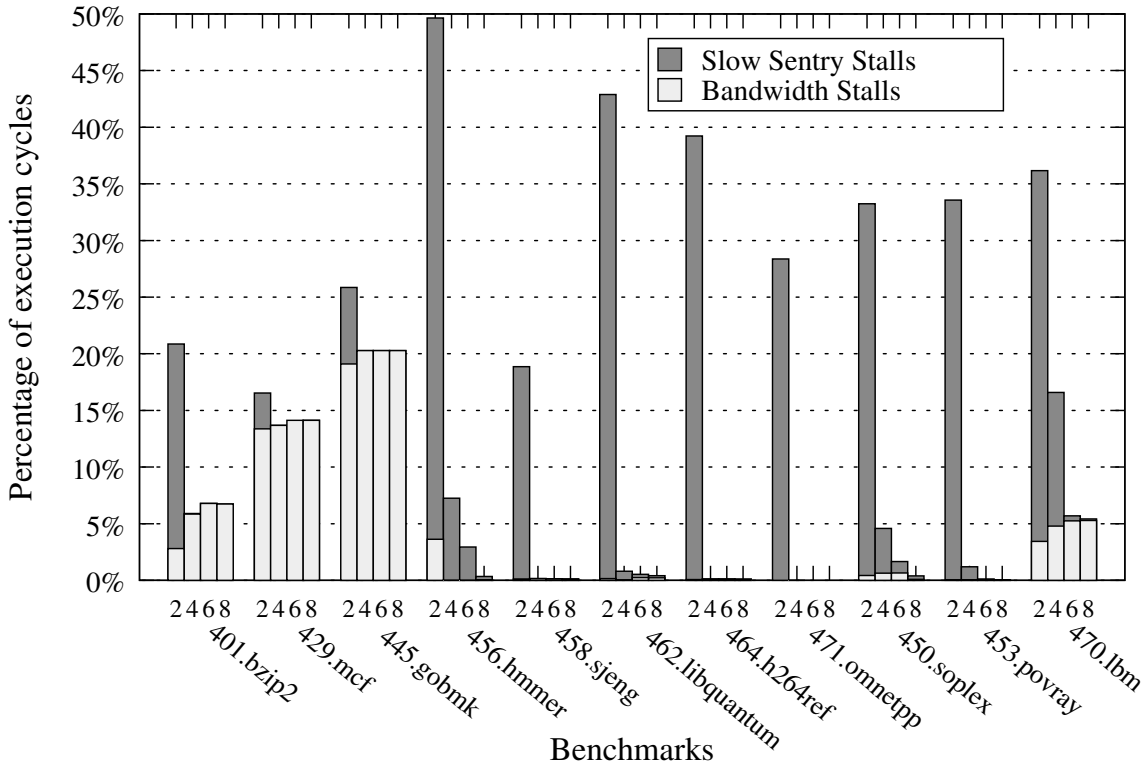


Figure 6.3: Stalls induced by the Sentry when varying the number of instructions that may be checked in parallel by the Sentry RICU (corresponding to Figure 6.2). Sentry frequency: 1 GHz. Sentry-Processor bandwidth: 10 GB/s. Untrusted processor frequency: 2 GHz.

Sentry stalls. Meanwhile, the average percentage of cycles spent in bandwidth stalls was 3.93%. In general, the percentage of slow Sentry stalls was higher for benchmarks with higher baseline IPCs. For the three benchmarks with the highest IPCs, this percentage was 46.03% (456.hmmer), 42.74% (462.libquantum), and 39.16% (464.h264ref). The higher percentage of slow Sentry stalls was a direct result of the low checking throughput of the Sentry (a geomean of 1.32 instructions per Sentry cycle, as shown by Table 6.3).

Increasing the RICU width led to increased checking throughput and reduced the percentage of slow Sentry stalls, thereby resulting in improved performance. At a RICU width of 4, the Sentry checked a geomean of 1.82 instructions per Sentry cycle (Table 6.3); consequently, the average percentage of slow Sentry stalls fell to 2.27%. The only benchmarks for which the processor showed a significant percentage of slow Sentry stalls were 456.hmmer (7.24%), 450.soplex (3.96%), and 470.lbm (11.79%). 456.hmmer is a bursty bench-

mark; it had the largest percentage of timing cycles where 5 or more instructions were committed in a single cycle by the untrusted processor. During these bursty sequences, the Sentry was overwhelmed by the number of instructions sent for checking, resulting in the Sentry sending the `stall` signal to the processor. On the other hand, 450.soplex and 470.lbm experienced a large number of slow Sentry stalls because they have a higher proportion of floating point instructions, which get re-executed by high-latency floating point units on the Sentry.

When the RICU width was increased to 6, the checking throughput increased further as the Sentry now checked a geomean of 1.85 instructions per Sentry cycle (Table 6.3). Most of this gain was from the three benchmarks that experienced the most slow Sentry stalls for a RICU width of 4. The percentage of slow Sentry stalls decreased to 2.94% for 456.hmmer, 3.96% for 450.soplex, and 0.44% for 470.lbm. The average percentage of slow Sentry stalls across the 11 benchmarks was found to be 0.44%.

Finally, at RICU width 8, the Sentry checked a geomean of 1.86 instructions per Sentry cycle. The average percentage of slow Sentry stalls went down further to 0.09%. In fact, for every single benchmark, the processor experienced slow Sentry stalls in less than 0.4% cycles.

As can be seen from both Figures 6.2 and 6.3, at 4 RICUs and above, the overhead of TrustGuard is dominated by the overhead of Merkle tree accesses and the bandwidth stalls. This is especially visible for the three benchmarks that showed an IPC decline above 15%—401.bzip2, 429.mcf, and 445.gobmk. For the four configurations, the percentage of bandwidth stalls ranged between 11.8%–13.7% across these three benchmarks (average calculated for these benchmarks separately), as against an average range of 3.9%–4.3% across all the eleven benchmarks.

6.4 Varying Sentry Frequency

One of the main insights behind TrustGuard is that a Sentry running at a lower clock frequency may verify the execution of instructions by the untrusted processor without impacting its performance too adversely. Along with reducing the energy consumption of the Sentry, the lower frequency could also enable the fabrication of the Sentry at older, trusted fabrication plants using technology that is a few generations old. Figure 6.4 shows the effect of varying the Sentry clock frequency on the IPC of the untrusted processor. Each different configuration of the Sentry has a RICU width of 4 instructions per cycle and a Sentry-Processor bandwidth of 10 GB/s.

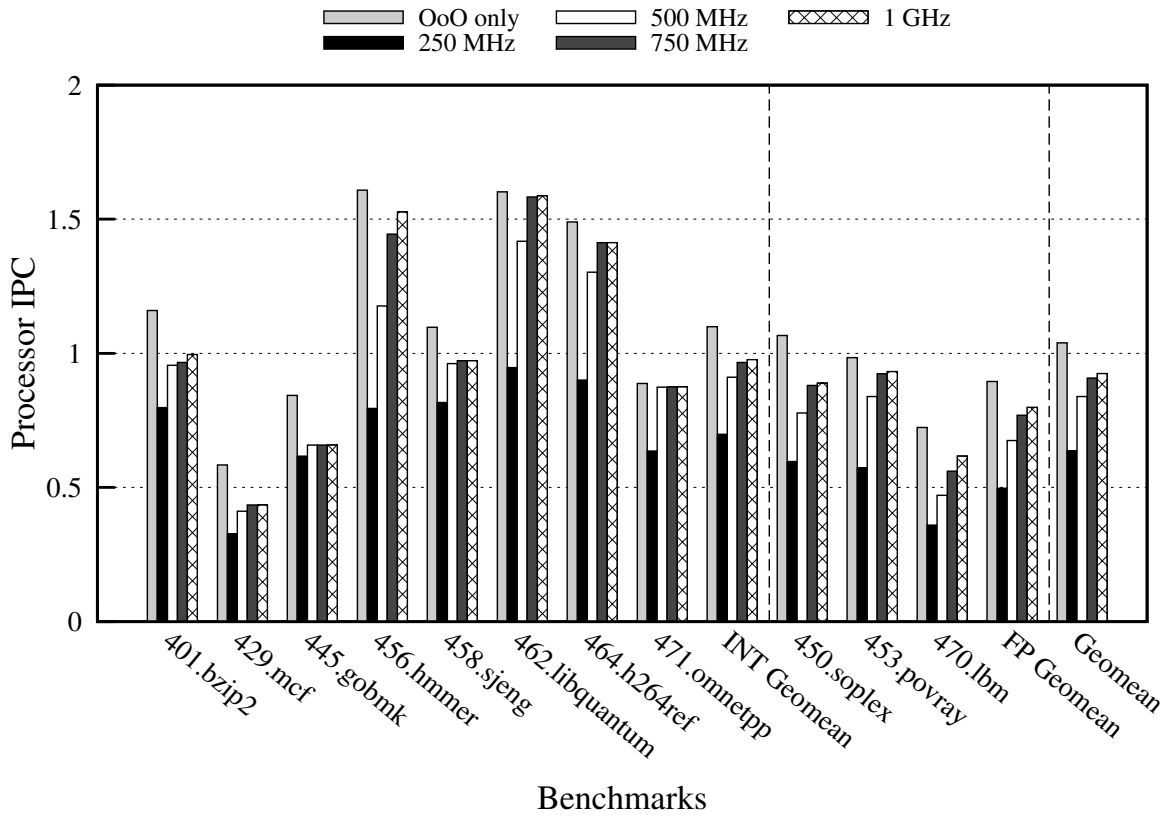


Figure 6.4: Effect on untrusted processor IPC of varying the Sentry's frequency. RICU width: 4 instructions/cycle. Sentry-Processor bandwidth: 10 GB/s. Untrusted processor frequency: 2 GHz.

The challenge with running the a Sentry at a lower clock frequency is to achieve a high enough checking throughput to keep up with execution by the untrusted processor. Com-

pared to the out-of-order processor baseline, the eleven benchmarks showed a geomean IPC reduction of 37.2% at 500 MHz, 15.3% at 750 MHz, 8.5% at 1 GHz, 7.2% at 1.25 GHz, and 6.5% at 1.5 GHz. As expected, higher Sentry frequencies led to higher checking throughput, which in turn, improved the performance of the untrusted processor.

The average number of instructions checked by the Sentry per cycle presented an interesting metric to show how the RICU in the Sentry was overwhelmed by the number of instructions that needed to be verified. At 500 MHz, the Sentry checked a geomean of 2.55 instructions per Sentry cycle as against a maximum possible 4 instructions per cycle. This was by far the highest value of this metric seen in these experiments, including for the case when an 8-wide RICU was used at 1 GHz Sentry frequency. This is because the processor was sending instructions to the Sentry at a faster rate than the Sentry could check. So, for a large number of cycles, many instructions were always queued up in the `ExecInfo` buffer, which allowed the Sentry to utilize the full width of its RICU.

The performance impact of varying the Sentry frequency can also be gleaned from the Sentry's checking throughput. However, comparing checking throughput per Sentry cycle is meaningless as each configuration has a different clock period. Therefore, we measure the Normalized Sentry Throughput (NSTP), defined as the average number of instructions checked by the Sentry RICU per nanosecond. The NSTP can be calculated as follows:

$$\text{NSTP} = (\text{Average checking throughput}) \times (\text{Sentry frequency in GHz})$$

Table 6.4 shows the NSTP achieved by the Sentryduring the execution of each benchmark. The geomean NSTP was 1.27 instructions/ns at 500 MHz, 1.68 instructions/ns at 750 MHz, 1.82 instructions/ns at 1 GHz, 1.85 instructions/ns at 1.25 GHz, and 1.86 instructions/ns at 1.5 GHz. Thus, the checking throughput of the Sentry increased at higher frequencies, which explains the improved processor IPC.

To assess the overheads impacting performance while varying Sentry frequency, we once again measured slow Sentry and bandwidth stalls experienced by the untrusted processor. Figure 6.5 shows the percentage of these stalls.

Benchmark	Baseline Processor IPC	Normalized Sentry Throughput (Insts/ns)				
		500MHz	750MHz	1GHz	1.25GHz	1.5GHz
401.bzip2	1.16	1.60	1.91	1.93	1.99	1.99
429.mcf	0.59	0.66	0.82	0.87	0.87	0.87
445.gobmk	0.84	1.24	1.32	1.32	1.32	1.32
450.soplex	1.07	1.19	1.56	1.76	1.78	1.79
453.povray	0.98	1.15	1.68	1.85	1.86	1.87
456.hmmer	1.61	1.57	2.34	2.89	3.05	3.15
458.sjeng	1.10	1.63	1.93	1.95	1.95	1.95
462.libquantum	1.60	1.89	2.84	3.17	3.18	3.18
464.h264ref	1.49	1.80	2.61	2.83	2.83	2.83
470.lbm	0.72	0.72	0.94	1.12	1.23	1.24
471.omnetpp	0.89	1.27	1.75	1.75	1.75	1.75
Geomean	1.04	1.27	1.68	1.82	1.85	1.86

Table 6.4: Checking throughput of the Sentry, in instructions per nanosecond, when the Sentry frequency is varied

At 500 MHz, the performance impact is dominated by the Sentry's checking throughput lagging behind execution by the processor. This is reflected in the average percentage of slow Sentry and bandwidth stalls experienced by the processor. The processor experienced slow Sentry stalls for a geomean of 30.40% of timing cycles in this configuration. The average percentage of bandwidth stalls for this configuration was 3.74%. The effect of bandwidth stalls was more dominant than slow Sentry stalls for 401.bzip2, 429.mcf, and 445.gobmk. These benchmarks are comparatively more memory intensive and incur a large number of shadow memory accesses. These accesses result in significant information being sent across from the processor to the Sentry—a process slowed down by the bandwidth constraints on the link between the two components.

For all the other benchmarks, which are more computationally intensive, the processor suffered a large decline in its IPC. The largest declines were for benchmarks with the highest baseline IPCs. For 456.hmmer, the processor was stalled for more than half the timing cycles, waiting for the Sentry to check instruction execution. Once again, the burstiness of the benchmark aggravated the effect of the low checking throughput of the Sentry.

As the Sentry frequency was increased, the number of slow Sentry stalls dropped, re-

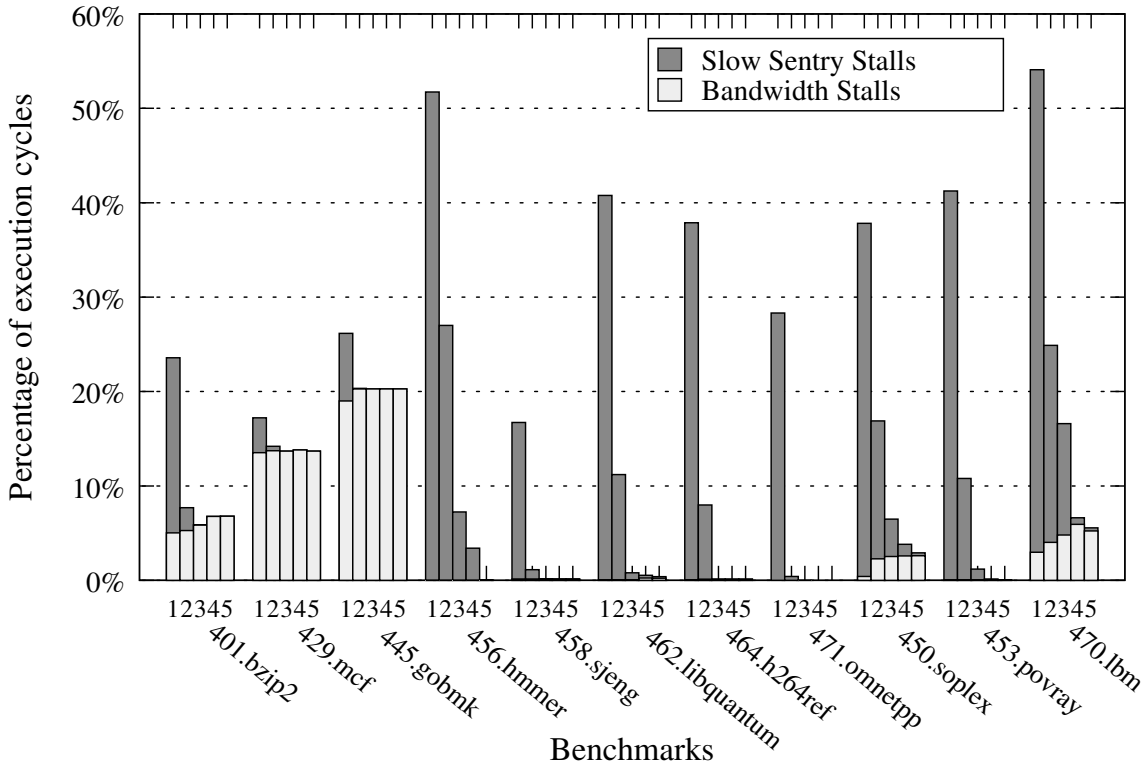


Figure 6.5: Stalls induced by the Sentry when varying the Sentry frequency (corresponding to Figure 6.4). RICU width: 4 instructions/cycle. Sentry-Processor bandwidth: 10 GB/s. Untrusted processor frequency: 2 GHz. 1: 500 MHz, 2: 750 MHz, 3: 1 GHz, 4: 1.25 GHz, 5: 1.5GHz.

sulting in improved processor IPC. The average percentage of slow Sentry stalls was 8.79% at 750 MHz, 2.27% at 1 GHz, 0.52% at 1.25 GHz, and 0.08% at 1.5 GHz. By contrast, the average percentage of bandwidth stalls was relatively constant: 4.17% at 750 MHz, 4.14% at 1 GHz, 4.23% at 1.25 GHz, and 4.16% at 1.5 GHz. This intuitively makes sense because the absolute amount of communication occurring between the processor and the Sentry is constant for each of these configurations.

An interesting result related to the occurrence of slow Sentry and bandwidth stalls can be seen from 401.bzip2 in Figure 6.5. At 500 MHz, there are a lot more slow Sentry stalls because of lower checking throughput. These stalls actually help hide the effect of bandwidth limitations for a number of clock cycles. As the Sentry's checking throughput increases due to higher Sentry frequency, most of these slow Sentry stalls disappear and instead, the bandwidth of communication becomes the new bottleneck. This accounts for

the increase in the percentage of bandwidth stalls at higher frequencies: 5.03% at 500 MHz, 5.28% at 750 MHz, 5.85% at 1 GHz, 6.78% at 1.25 GHz, and 6.80% at 1.5 GHz. A similar trend is also visible for 450.soplex and 470.lbm. This trend does not manifest for other benchmarks because slow Sentry stalls, not bandwidth stalls, are the bottleneck for them.

6.5 Varying Sentry-Processor Bandwidth

The average percentage of bandwidth stalls in Figures 6.3 and Figures 6.5 remained relatively stable because the communication between the processor and the Sentry depends on program characteristics (number of cache lines accessed by memory, distribution of instructions, L1 cache miss rates, L1 cache eviction rates, etc.) With cache mirroring, the processor need not send data corresponding to L1 cache hits to the Sentry. Therefore, programs with greater cache locality will save on communication. Similarly, for branch instructions where control flow does not jump to a new location, the new program counter is not sent across to the Sentry, saving on some more communication.

The overhead of shadow memory accesses was by far the biggest contributor to communication between the processor and the Sentry . For example, when the Bonsai Merkle tree was implemented in the simulator without the presence of a cache on the Sentry, the processor spent most of its time in bandwidth stalls and its IPC dropped to near-zero levels. This was because the processor was having to send across MACs, counters, and all Merkle tree nodes leading up to the root to the Sentry for verification. Adding an instruction and a data cache on the Sentry was the first optimization that enabled this system to work without prohibitive overheads.

Figure 6.6 shows the effect of varying the Sentry-Processor bandwidth on the IPC of the untrusted processor. For each configuration of bandwidth, the RICU width was set to 4 instructions and the frequency of the Sentry was set at 1 GHz, i.e. half the frequency of the untrusted processor.

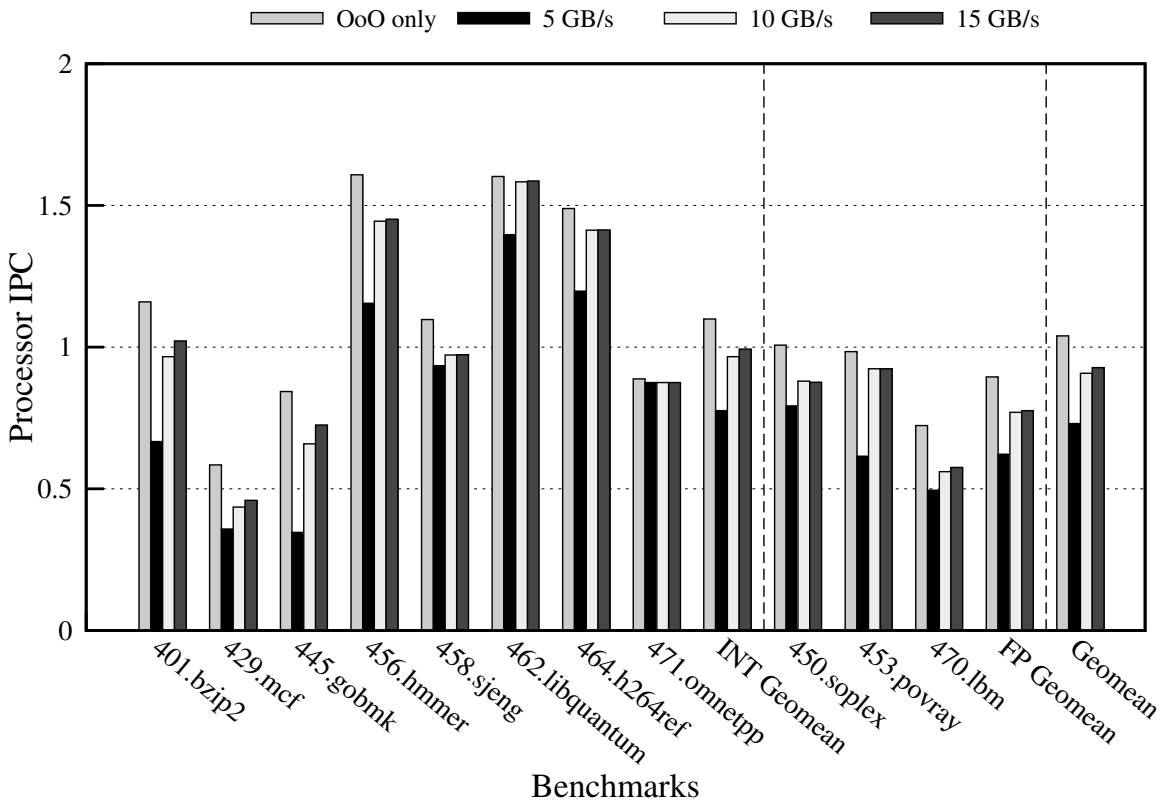


Figure 6.6: Effect on untrusted processor IPC of varying the bandwidth between the processor and the Sentry. RICU width: 4 instructions/cycle. Sentry frequency: 1 GHz. Untrusted processor frequency: 2 GHz.

As seen in Figure 6.6, configurations with higher bandwidth achieved higher processor IPCs. The geomean decline in processor IPC with respect to the out-of-order only baseline was found to be 21.1% at 5 GB/s, 8.5% at 10 GB/s, and 7.5% at 15 GB/s. There was a significant increase in processor IPCs when increasing bandwidth from 5 GB/s to 10 GB/s. However, when the bandwidth increased from 10 GB/s to 15 GB/s, there was an improvement in processor IPC for only three benchmarks—401.bzip2, 429.mcf, and 445.gobmk. For the other eight benchmarks, there was no significant improvement in the processor IPC (less than 2 percentage points). Once again, we measured the percentage of slow Sentry and bandwidth stalls experienced by the untrusted processor in order to assess the overheads affecting performance. This data is shown in Figure 6.7.

As Figure 6.7 shows, the average percentage of bandwidth stalls was found to be 25.34% at 5 GB/s, 4.14% at 10 GB/s, and 0.24% at 15 GB/s. Most of the gains while

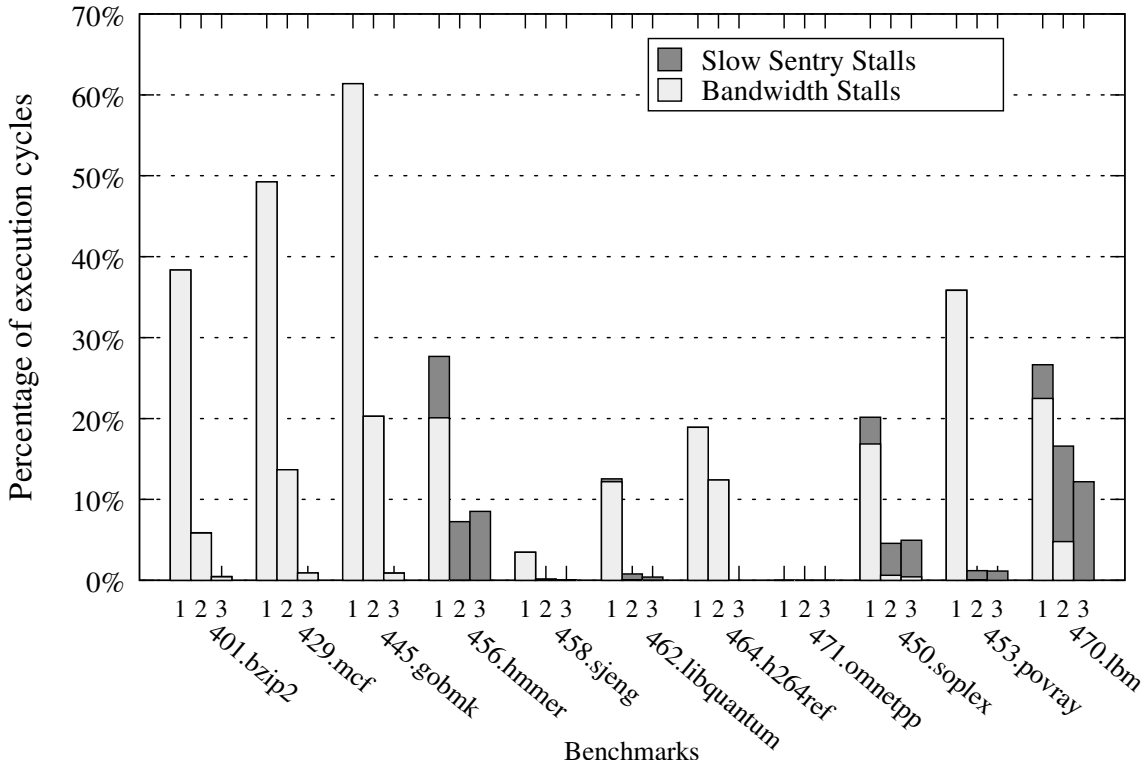


Figure 6.7: Stalls induced by the Sentry when varying the processor to Sentry channel bandwidth (corresponding to Figure 6.4). RICU width: 4 instructions/cycle. Sentry frequency: 1 GHz. Untrusted processor frequency: 2 GHz. 1: 5 GB/s, 2: 10 GB/s, 3: 15 GB/s.

increasing the bandwidth from 10 GB/s to 15 GB/s came from the above mentioned three benchmarks. For 401.bzip2, the percentage of bandwidth stalls was 38.34% at 5 GB/s, 5.85% at 10 GB/s, and 0.44% at 15 GB/s. For 429.mcf, the same number was 49.24% at 5 GB/s, 13.68% at 10 GB/s, and 0.90% at 15 GB/s. For 445.gobmk, this metric was 61.38% at 5 GB/s, 20.27% at 10 GB/s, and 0.89% at 15 GB/s.

429.mcf and 445.gobmk showed a high number of bandwidth stalls because they experienced relatively high miss rates in the data cache, which in turn, increased the amount of information to be communicated to the Sentry. The data cache miss rate was approximately 27% for 429.mcf and approximately 15% for 445.gobmk, compared to an average of 3% across the other nine benchmarks. 401.bzip2 too had an above average data cache miss rate of approximately 5%. The effects of these misses was exacerbated by the high number of memory operations performed in this benchmark. By contrast, 456.hmmer with 99.1%

data cache hit rate only experienced bandwidth stalls for 0.0064% of timing cycles.

As the percentage of bandwidth stalls declines due to higher bandwidth, it sometimes results in an increase in the percentage of slow Sentry stalls, as can be observed most noticeably for 456.hmmer in Figure 6.7. As bandwidth increases, execution information is delivered at a higher rate to the Sentry . Thus, the Sentry RICU may have to achieve higher throughput to match this higher rate of delivery. However, the RICU in 456.hmmer already has a high throughput and cannot increase its pace of checking without an increase in the RICU width. Consequently, we see an uptick in the percentage of slow Sentry stalls for 456.hmmer.

Finally, it must be noted that at 15 GB/s, the processor experiences bandwidth stalls in less than 1% of timing cycles for every single benchmark evaluated. At this bandwidth, eight of the eleven benchmarks do not experience any significant number of slow Sentry stalls. For these benchmarks, the overhead of shadow memory accesses becomes the bottleneck. Thus, the processor IPCs observed are very similar to the corresponding processor IPCs reported in Figure 6.1.

The effect of varying bandwidth on performance can also be gauged by measuring the bandwidth requirements when the processor executes these benchmarks. Table 6.5 shows the mean bandwidth usage, when the maximum Sentry-Processor link bandwidth was 10 GB/s and also when the maximum bandwidth was unrestricted. As seen from the case where the maximum bandwidth was unrestricted, two benchmarks had very high bandwidth requirements—429.mcf and 445.gobmk. These were also the benchmarks which showed the maximum percentage of bandwidth stalls in Figures 6.3, 6.5, and 6.7. The geomean of the average bandwidth usage across the eleven benchmarks with unrestricted bandwidth on the link was 6.95 GB/s; the value of the same metric was 6.16 GB/s, when the bandwidth was limited to 10 GB/s.

Benchmark	Mean Bandwidth Usage (GB/s)	
	Max 10GB/s B/W	Unlimited B/W
401.bzip2	7.33	7.82
429.mcf	8.15	10.06
445.gobmk	8.72	15.32
450.soplex	5.33	5.56
453.povray	7.63	7.88
456.hmmer	6.14	6.64
458.sjeng	4.67	4.95
462.libquantum	5.56	5.56
464.h264ref	5.54	6.47
470.lbm	6.30	6.56
471.omnetpp	4.15	4.44
Geomean	6.16	6.95

Table 6.5: Mean Bandwith Usage for Sentry-Processor link for: (1) Maximum Bandwidth 10 GB/s, and (2) Unrestricted Bandwidth

6.6 Effect of Link Compression

Besides caching on the Sentry, the other mechanism used by TrustGuard to reduce the amount of communication between the processor and the Sentry was link compression. The previous graphs reported processor IPC numbers for configurations where link compression was always enabled. This section quantifies the effect of link compression on the performance of the untrusted processor. To this end, we ran simulations for two additional configurations: (1) No link compression at bandwidth 10 GB/s; and (2) No link compression at bandwidth 15 GB/s. Figure 6.8 shows the processor IPC for these two configurations as well as for the out-of-order only baseline and the configuration with link compression enabled at 10 GB/s. For the four configurations shown in the figure, the RICU width was set to 4 instructions per cycle and the Sentry was clocked at 1 GHz.

As shown in Figure 6.8, link compression enabled a significant reduction in bandwidth requirements. Without link compression, the untrusted processor experienced a geomean IPC decline of 10.9% at 15 GB/s bandwidth and 18.7% at 10 GB/s bandwidth. By contrast, using link compression at 10 GB/s bandwidth incurred much lower overhead than either configuration without link compression (a geomean IPC decline of 8.5%).

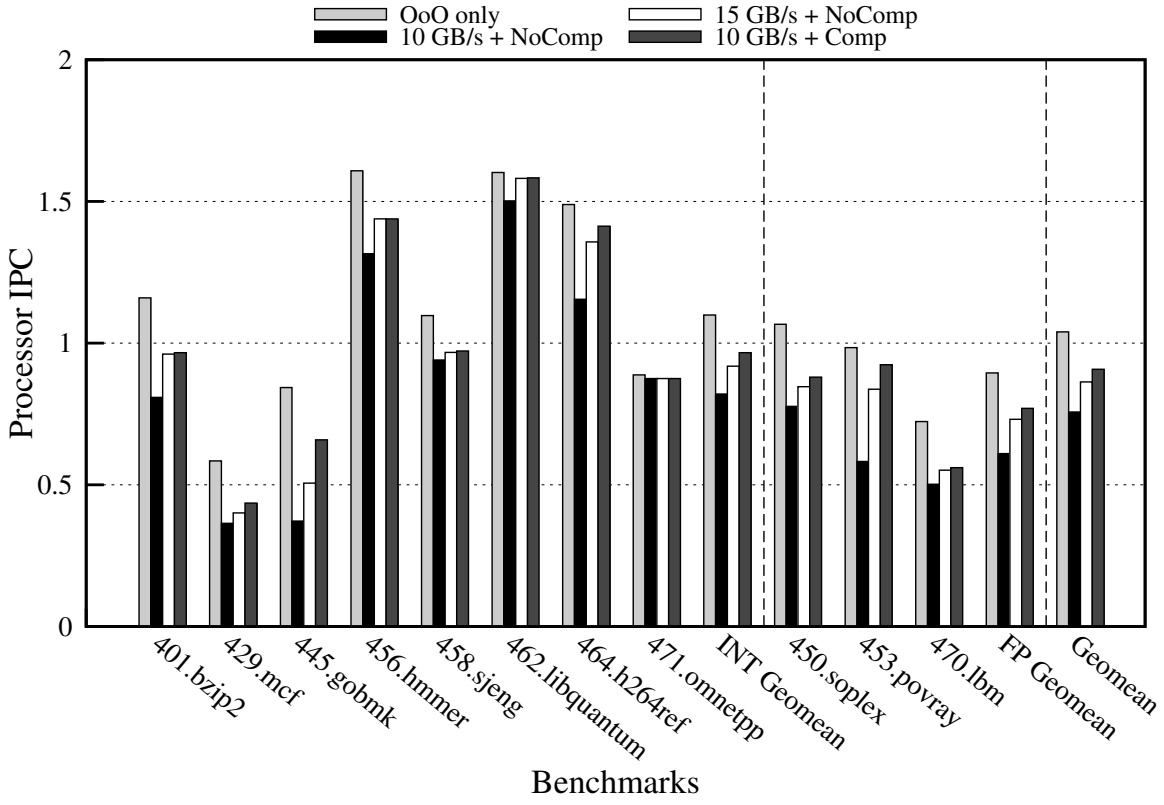


Figure 6.8: Effect of link compression on untrusted processor IPC. NoComp: Link compression not used. Comp: Link compression used. RICU width: 4 instructions/cycle. Sentry frequency: 1 GHz. Untrusted processor frequency: 2 GHz.

Figure 6.9 shows the percentage of bandwidth stalls experienced by the processor for each of the above configurations. While all benchmarks experienced a massive reduction in the percentage of bandwidth stalls due to link compression, the effect was seen most noticeably for 429.mcf, 445.gobmk, and 453.povray. The average percentage of bandwidth stalls was found to be 21.79% when no link compression was used at 10 GB/s, 9.68% when no link compression was used at 15 GB/s, and 4.14% when link compression was used at 10 GB/s. The highest number of bandwidth stalls were seen in 445.gobmk, which according to Table 6.5 requires a mean bandwidth usage of 15.32 GB/s (when link compression is enabled).

An important consideration when it comes to link compression is the simplicity of the compression algorithm used. As explained in Section 4.4, TrustGuard uses a combination of Significance-Width Compression (SWC) and Frequent Value Encoding (FVE). More

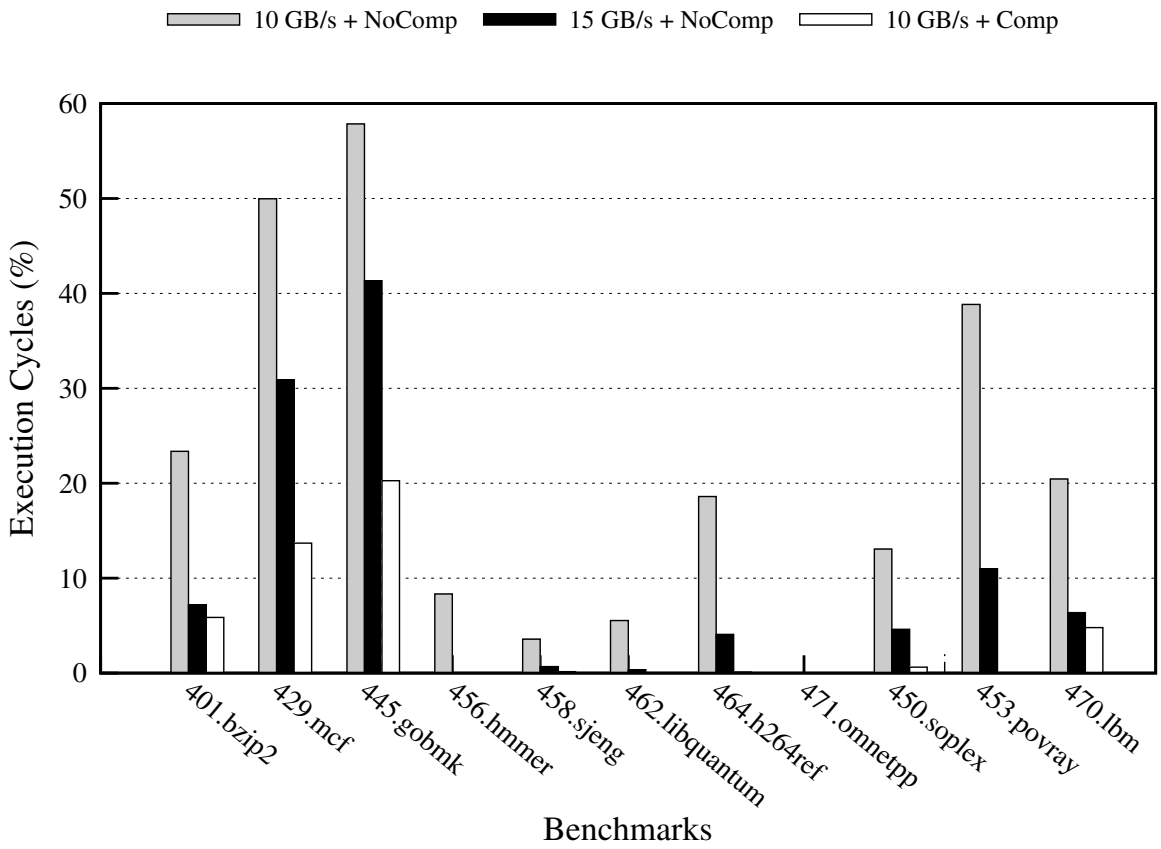


Figure 6.9: Effect of link compression on percentage of bandwidth stalls experienced by the processor. NoComp: Link compression not used. Comp: Link compression used. RICU width: 4 instructions/cycle. Sentry frequency: 1 GHz. Untrusted processor frequency: 2 GHz.

complex compression algorithms could be used to further reduce the bandwidth requirements. However, these algorithms might require more complex implementations, which might also increase the latency of communication between the processor and the Sentry.

6.7 Link Utilization

Mean bandwidth usage (shown in Table 6.5) is only one metric that characterizes the communication between the Sentry and the untrusted processor. Another important metric for the system is the utilization of the link itself. Figure 6.10 quantifies this utilization by showing the cumulative distribution of instantaneous bandwidth usage (bandwidth used in a particular cycle) over the number of execution cycles for the benchmarks. The Sentry

configuration for Figure 6.10 has RICU width 4, maximum Sentry-Processor bandwidth of 10 GB/s, and a Sentry frequency of 1 GHz.

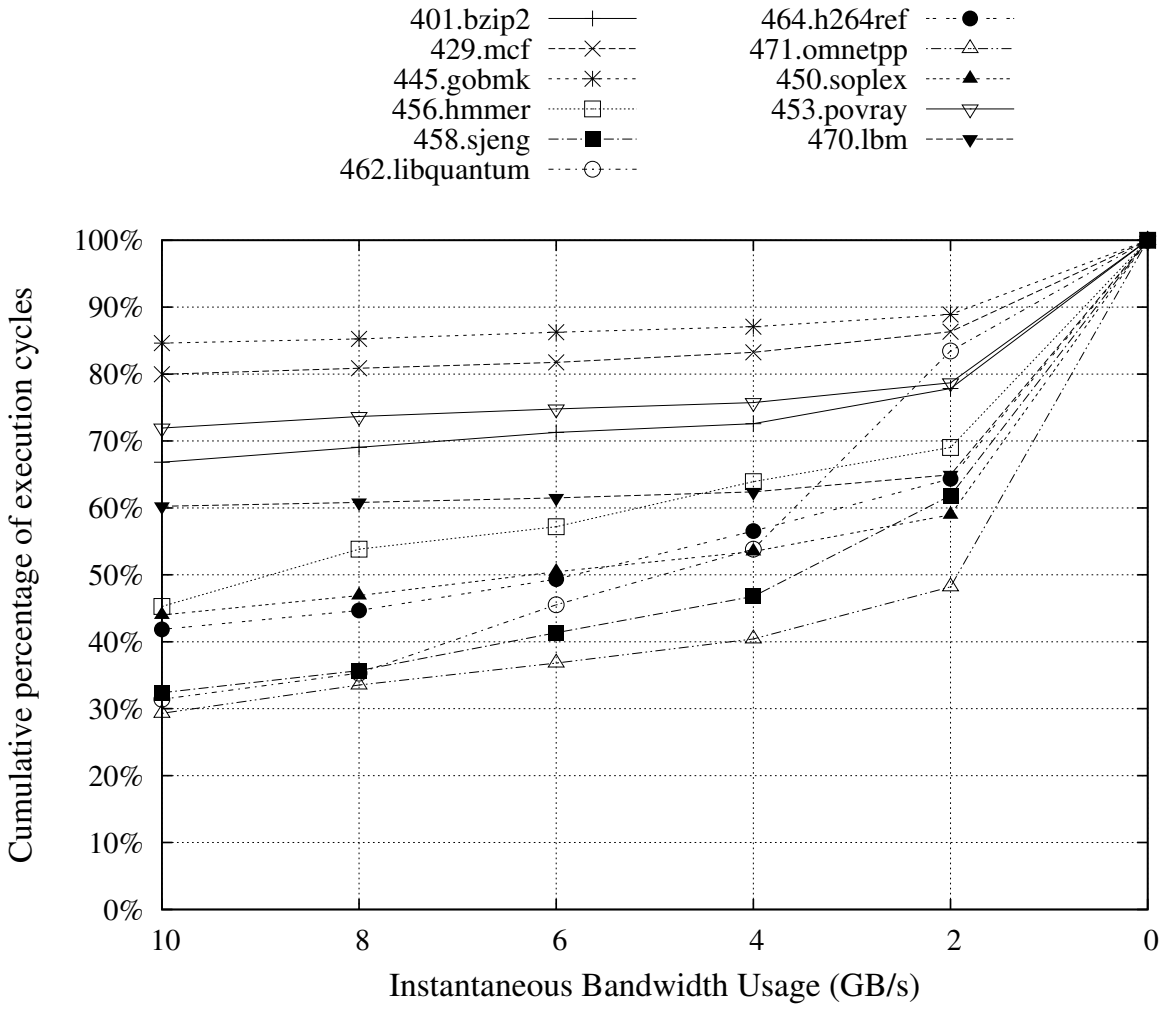


Figure 6.10: Cumulative distribution of instantaneous bandwidth usage over percentage of execution cycles, showing utilization of the link between the processor and the Sentry. RICU: 4 instructions/cycle. Sentry frequency: 1 GHz. Sentry-Processor bandwidth: 10 GB/s. Untrusted processor frequency: 2 GHz.

As shown in Figure 6.10, benchmarks with relatively high communication between the processor and the Sentry have a large number of execution cycles with large instantaneous bandwidths. For instance, for 445.gobmk, the instantaneous bandwidth usage is less than 8GB/s for only 14.9% of execution cycles. By contrast, for benchmarks like 471.omnetpp and 458.sjeng that have lesser communication, more than 50% of execution cycles have instantaneous bandwidth usage less than 4GB/s. There is also a correlation between

higher mean bandwidth usage and the percentage of execution cycles where the instantaneous bandwidth usage is equal to the maximum possible bandwidth. For instance, 429.mcf (mean bandwidth usage 10.06 GB/s when bandwidth is unrestricted) and 445.gobmk (mean bandwidth usage 15.32 GB/s with unrestricted bandwidth) use up the full 10 GB/s bandwidth for 79.98% and 84.62% respectively of execution cycles. By contrast, 471.omnetpp (with the minimum mean bandwidth usage of 4.44 GB/s) uses up the full 10 GB/s bandwidth for only 29.30% of execution cycles.

6.8 Average Latency of Instruction Verification

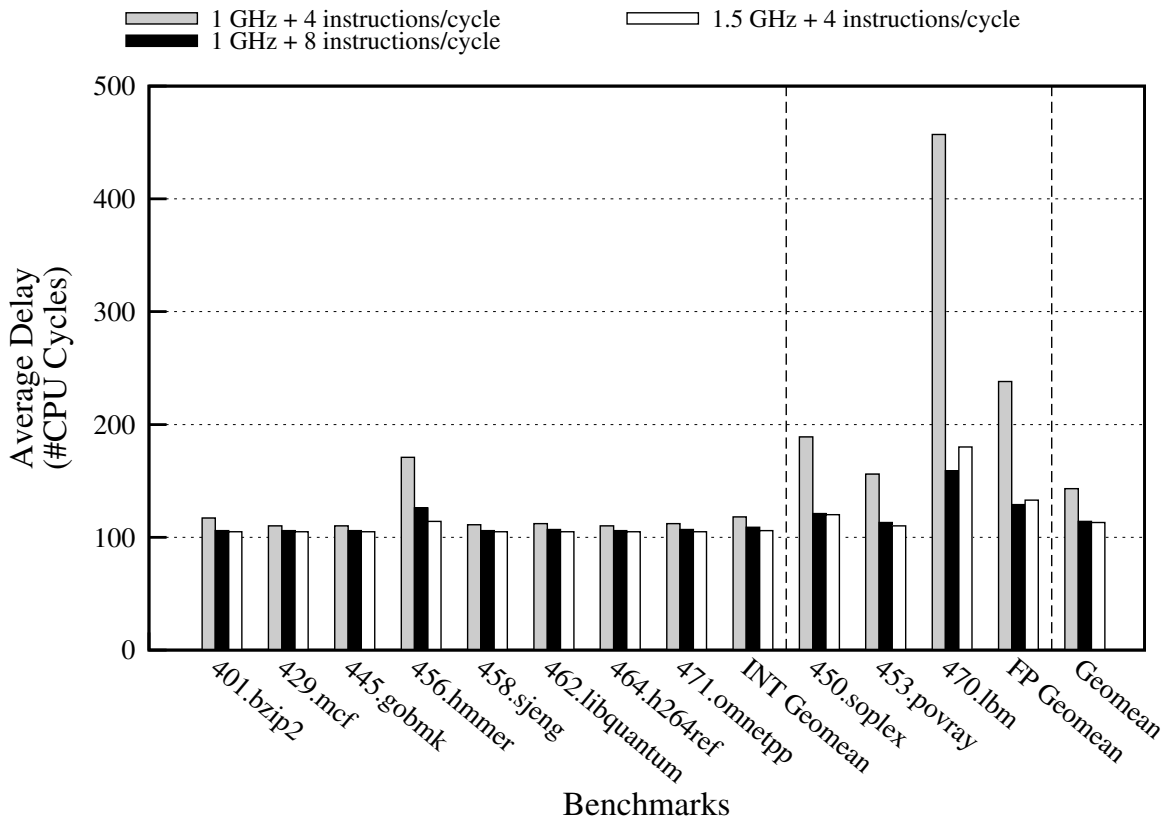


Figure 6.11: Average latency, in terms of number of processor cycles, between committing of an instruction in the untrusted processor and its checking by the Sentry. Sentry-Processor bandwidth: 10 GB/s. Untrusted processor frequency: 2 GHz.

Output operations in TrustGuard cannot proceed until they are verified by the Sentry. Figure 6.11 shows the average latency for each instruction from when it is committed in the

untrusted processor to when it is verified by the Sentry. This metric gives a sense of how long output operations would be delayed. With improvement in frequency, the throughput of checking increases, which results in a decline in the average latency. The geomean average latency for each instruction is 143 CPU cycles (0.072μs) at 1GHz and RICU width 4 instructions/cycle, 114 CPU cycles (0.057μs) at 1GHz and RICU width 8 instructions/cycle, and 113 CPU cycles (0.057μs) at 1.5GHz and RICU width 4 instructions/cycle. Note that every instruction incurs a latency of at least 100 cycles, which is assumed to be the latency of off-chip communication by the processor.

There is a clear difference between the SPECINT and SPECFP benchmarks. At 1GHz and RICU width 4 instructions/cycle, SPECINT has a geomean latency of 118 CPU cycles (0.059μs), while SPECFP has a geomean latency of 238 CPU cycles (0.119μs). The reason for this difference is that floating point operations have higher latencies than integer operations; this latency is magnified because the Sentry runs at half the frequency of the untrusted processor. At higher frequency (1.5GHz) or with greater parallelism in the Sentry (RICU width 8 instructions/cycle), the throughput of the Sentry increases, leading to significant decline in the latency of committed instructions. The average latency of SPECFP benchmarks drops to 129 CPU cycles (0.065μs) at 1GHz and 8RICUs and to 133 CPU cycles (0.067μs) at 1.5GHz and RICU width 4 instructions/cycle.

6.9 Energy

We used McPAT v1.2 [85] to model the energy of the TrustGuard processor and Sentry using execution statistics from our performance simulations. Power for the MAC engines was estimated using an HMAC-MD5 accelerator [135], adapted to our design using technology scaling. The processor-Sentry link was modeled as a PCIe interconnect [2].

Figure 6.12 shows the energy consumption of TrustGuard, normalized to the energy consumption of the baseline untrusted processor. The Sentry configuration used had RICU

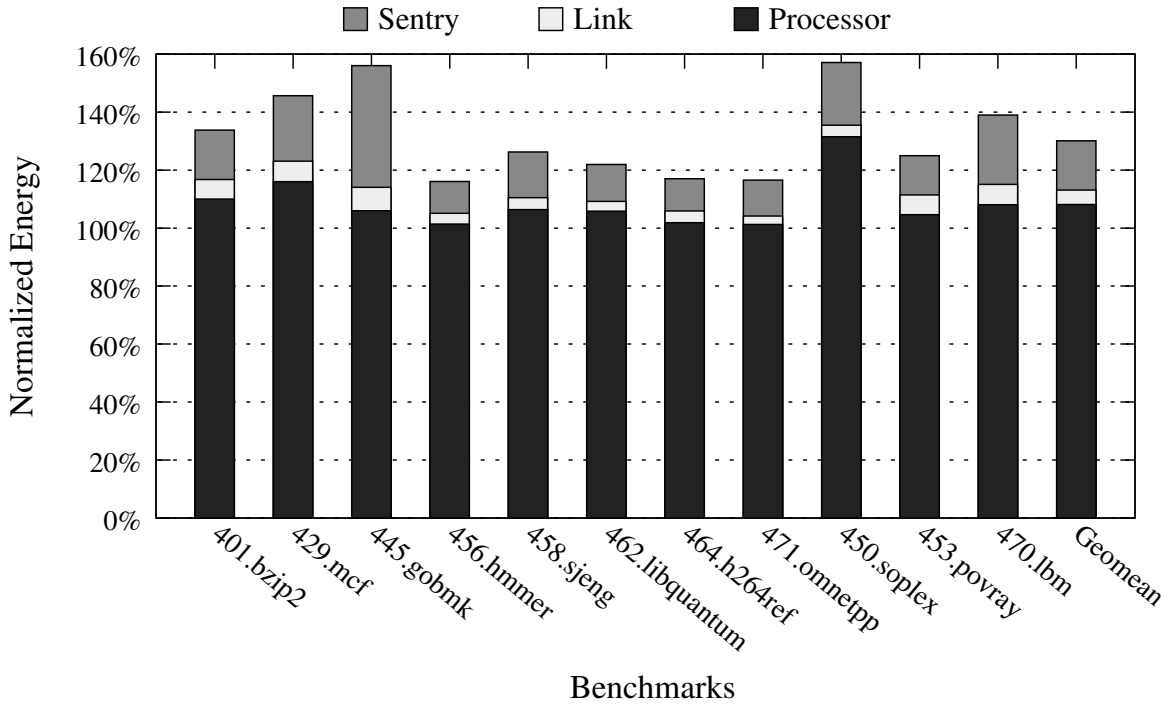


Figure 6.12: TrustGuard's energy usage. RICU width: 4 instructions/cycle. Sentry frequency: 1 GHz. Sentry-Processor bandwidth: 10 GB/s. Untrusted processor frequency: 2 GHz.

width 4 instructions/Sentry cycle and was clocked at 1 GHz. The maximum bandwidth of the Sentry-processor interconnect was set at 10 GB/s. The geomean energy consumption of TrustGuard was 28.7% greater than the baseline. The untrusted processor in TrustGuard showed a geomean 8.14% higher energy consumption than the baseline processor. The geomean energy consumption of the Sentry itself is 17% of the energy consumption of the baseline processor, which is significantly lower than the 100+% increase that would have come from a second redundant processor. Furthermore, the dynamic runtime power for the TrustGuard-enabled system only increased by a geomean of 9.6% over the baseline.

The main reasons for the lower energy consumption of the Sentry compared to the untrusted processor are the lower frequency of the Sentry, the absence of an L2 cache, and the absence of the out-of-order support structures. Finally, the link consumes a geomean 5.0% of the energy of the untrusted processor.

Chapter 7

The Simplicity of the Sentry

Chapter 2 discussed the shortcomings of existing systems to secure complex hardware components such as modern out-of-order processors. One of these shortcomings is that most verification techniques do not easily scale to complex hardware designs. Consequently, it is desirable for the Sentry to have a relatively simple design to make it more amenable to verification. Moreover, a simple design combined with the Sentry's limited impact on performance even when operating at lower clock frequencies, would enable the manufacture of the Sentry at closely-controlled fabrication plants using generations old technology. Thus, a necessary condition for the viability of the TrustGuard architecture is that the Sentry should be much less complex than the untrusted processor.

This section compares the complexity of the Sentry design against the complexity of an out-of-order processor. Processor execution involves fetch, decode, "functional unit execution," write-back, and in modern processors, also includes dependence speculation, branch prediction, register renaming, reorder buffers, L2 cache, instruction queue, dispatch, load/store queues, inter-stage forwarding logic, and memory control. Additionally, these components are directed by extremely complex control logic. Note that "functional unit execution" is just one small step in processor execution. The Sentry can only "functional unit execute" instructions at the direction of the untrusted processor. The Sentry does

perform more work than the functional units in a processor (e.g. result comparison), but the Sentry's work, complexity, and control are still very much closer to "functional unit execution" than to "full processor execution."

This difference between the complexities of the processor and the Sentry becomes clear, once we compare the complexities of each different stage in the processor and the Sentry respectively. The following sections present this comparison.

7.1 Fetch vs Instruction Read

The fetch unit on the processor is responsible for fetching the next instruction(s) to be executed from memory. Most modern multiple-issue superscalar processors utilize an integrated instruction fetch unit that performs several functions: branch prediction, instruction prefetch, and instruction memory access and buffering. Table 7.1 shows the components present in the processor's fetch unit and the Sentry's RICU.

Instruction Fetch Unit (Processor)	**Instruction Read (IR) Unit** (Sentry)
Instruction Cache Access	Instruction Cache Access
Next Address Logic	`NextInst` Logic
Branch Predictor	
Branch Target Buffer	
Return Address Predictor	
Instruction Prefetcher	
Memory fetch hardware	

Table 7.1: Components of the processor's instruction fetch unit and the Sentry's Instruction Read (IR) unit

Both the processor's fetch unit and the Sentry's IR unit perform instruction cache accesses to initiate instruction execution and checking respectively. To determine the next instruction to be checked, the processor's fetch unit must have a `Next Address` logic that calculates the next program counter value. This logic determines the next address by choosing from among the calculated PC value and the address resulting from a synthesis of

the branch predictor, branch target buffer (BTB), and return address predictor outputs. The next address to be fetched from may sometimes be the result of a branch misprediction, in which case the processor must also implement logic to signal a squash and enable recovery to a nonspeculative state.

By contrast, the Sentry's `NextInst` logic (Figure 4.7) utilizes information about control flow sent by the processor to determine the next instruction to be checked. There is no misprediction of the next instruction address unless the processor executed a control flow instruction incorrectly or reported the said address incorrectly. As per the Sentry's functionality (Section 4.2), the RICU would detect either of these cases and prevent output emanating from those instructions.

The presence of the branch predictor adds considerable complexity to the processor fetch unit, compared to the IR-stage of the Sentry. In addition to a circuit to predict if a branch is taken or not (the predictor itself), a branch prediction unit typically also utilizes a branch target buffer (BTB) to determine the exact address of the next instruction and a return address predictor to handle returns from function calls. In a modern processor, this integrated branch prediction unit can be both large and complex. For instance, a typical ARM A8 processor has a two-level branch predictor unit, a 512-entry BTB, a 4K sized global history, and an 8-entry return stack (to enable return address prediction) [61].

Additionally, many processors also have a hardware prefetching unit to reduce the access latency for instruction fetch [61]. Typically, instruction prefetch involves the processor fetching multiple blocks on a cache miss. The requested block is placed in the instruction cache when it returns, while the prefetched block(s) are placed into a separate instruction stream buffer. If the requested block is present in the instruction stream buffer, the original cache request is canceled and the block is read from the instruction stream buffer. Thus, this additional circuitry for instruction prefetch adds further to the complexity of the fetch unit. By contrast, the Sentry does not need a hardware prefetching unit for instructions (or for data) as the processor must deliver all the accessed lines to the Sentry. In that sense, the

processor acts as the prefetcher for the Sentry's memory accesses. It must also be noted that the cache organization on the processor is more complex than the cache organization on the Sentry, as discussed later in this Section 7.5.

7.2 Decode/Register Renaming/Dispatch vs Operand Routing

The next RICU stage is Operand Routing that determines the operands for instructions to be checked. The Operand Routing (OR) stage also does instruction scheduling—a straightforward process because instructions for checking are both fetched and scheduled in program order on the Sentry. However, the out-of-order processor needs multiple stages to schedule instructions and to determine their operands. These include the decode, register renaming, and dispatch stages. Table 7.2 shows the components used by these processor stages, compared to the components used by the OR stage.

Decode/Register Renaming/ Dispatch Units (Processor)	**Operand Routing (OR)** (Sentry)
Instruction Decoder	Instruction Decoder
Architectural Registers	Architectural Registers
Physical Registers	
	Operand Fetch Unit
Register Renaming Unit	
– Register Map Table	
– Rename Buffer	
– Free Register List	
Instruction Dispatch Unit	
– Issue Queue	
– Instruction Wakeup Logic	
– Selection Logic	
Memory Issue Logic	Data Cache Access
– Memory Dependence Predictor	
– Memory Dependence Matrix	

Table 7.2: Comparison of the processor's components in the decode, register renaming, dispatch, and issue stages and the Sentry's Operand Routing (OR) stage

Out-of-order (OoO) processors execute instructions in the order in which operands of instructions and the resources to execute those instructions become available. However, fairly complex circuitry is needed to convert from program order to the OoO processor's execution order, and to maintain the program order of the output [58]. Further, a number of complex structures are also introduced to reduce the impact of various data hazards that may slow down execution. By contrast, the Sentry checks instructions in program order, thereby precluding the need for complex structures for out-of-order execution. Similarly, using execution information from the processor helps the Sentry avoid data hazards that may plague the processor's execution. These are the two main reasons for the Sentry being much simpler in design than the untrusted processor.

Both the processor and the Sentry decode machine instructions; therefore, the complexity of the decoding unit is comparable. It is important to note that for RISC processor (like the ARM-based processor used in Chapter 6), the instruction decoder is a relatively simple structure. Once an instruction is decoded by the processor, it is dispatched to an instruction queue, where it waits until its operands become available. The processor needs to maintain state corresponding to the instructions in the instruction queue (the typical instruction window size is around a hundred instructions.) By contrast, the maximum number of in-flight instructions in the Sentry is much lesser (16 for an RICU width of 4), thus necessitating much less state to be kept in the Sentry .

In OoO processors, register renaming follows decode and removes false dependences among in-flight instructions caused by the reuse of architectural registers. For this purpose, a number of structures such as rename buffer, register map table, and free register list are used [61, 58]. The size of the physical register file is greater than the size of the architectural register file. When an instruction is renamed, the rename map table is checked to determine its source operands. In addition, if the instruction produces a register result, a free physical register is allocated to that instruction from the free list. If no free registers are available, the instruction must be stalled until an earlier instruction commits and releases a

register. The destination operand for the instruction is renamed to this allocated free register, with the register map table reflecting this change. By contrast, the Sentry only deals with the architectural state of the processor and hence, only requires the smaller architectural register file. The OR stage can directly interface with the register file to perform the register reads and writes, without requiring access to the additional state related to register renaming.

Figure 7.1: Issue queue entry for a single instruction. R: Ready bit. V: Valid bit. In addition to this information, the dispatch stage also has CAM structures to look up source data and select logic to dispatch instructions to functional units [58].

After renaming, an instruction enters the dispatch stage, which reserves some of the resources used by the instruction in the future. These entries include entries in structures such as the issue queue, the reorder buffer, and the load-store queue. Figure 7.1 shows the amount of information stored for a single instruction in the issue queue. This information includes control information for functional units, register operands, and status bits to indicate when those register operands are available. In addition to a queue with multiple entries (typically issue queues are tens of instructions in size), the dispatch unit also has Content Addressable Memory (CAM) lookup tables for indexing into instructions in the queue. Finally, the dispatch unit also contains wakeup logic to notify that source operands for instructions have been produced.

The issue logic for instructions in the OR stage of the Sentry is also much simpler than the dispatch unit of the processor. In the Sentry, the operands for instructions are always available by the end of the OR stage. Consequently, once the decoding is complete, the instruction can be dispatched for checking to the Value Generation (VG) stage. No complex issue queue or issue logic is necessary. The Operand Fetch unit uses the decoded instruction to route the correct source operands values to the VG stage. The logic for this unit comprises of comparisons among the decoded source and destination operands followed by

multiplexing of possible operands, as shown in Figure 4.6. This logic is inherently smaller and less complex than that deployed by the OoO data dependence logic, as the number of instructions in-flight in an OoO processor is typically much higher than the maximum number of instructions considered by the Operand Fetch unit (RICU width).

Finally, the dispatch unit also implements the memory disambiguation policy for the processor, which in turn, affects the complexity of the dispatch unit. Most OoO processors reorder loads and stores with respect to each other to hide the latencies of memory operations. Some processors also perform memory dependence prediction that allows speculative execution of loads and stores. Finally, the memory dispatch must also contain separate wakeup logic to determine when an instruction is dispatched to the execution units. Some processors even implement a speculative wakeup of loads to improve performance. The memory dispatch unit is often implemented as another pipeline (specific implementations dictate whether the processor has a unified pipeline or separate pipelines for loads and stores), separate from the nonmemory instruction dispatch unit [58].

By contrast, the Sentry does not have to perform memory reordering, speculation, or even memory accesses. The data needed by the Sentry is always available in the data cache due to the cache mirroring scheme described in Section 4.3.2. The Sentry checks loads and stores in total program order using the addresses reported by the processor. Consequently, the Sentry need not wait for load and store addresses to be generated before performing cache accesses.

7.3 Execute vs Value Generation

The Execute Stage of the processor generates the actual results of instruction execution. In this stage, an instruction's source operands are sent to the processor's functional units along with the operation defined by the instruction. The functional units of the processor operate on the source operands to produce the results of computations. Similar computations are

also performed by the Sentry's Value Generation (VG) stage using the source operands determined by the OR stage. The following table (Table 7.3) shows the various components of the execution and writeback stages in the processor and the Sentry's value generation stage.

Execute/Writeback Stages (Processor)	**Value Generation (VG) Unit** (Sentry)
Functional Units Data Cache Access Load/Store Queue Bypass Network	Functional Units

Table 7.3: Components of the processor's execute and writeback stages and the Sentry's Value Generation (VG) unit

One aspect of checking by the Sentry is that it does not need to replicate all the functional units in the processor. For example, the OoO processor typically contains both an integer multiply and an integer divide unit to support these respective operations. Consider a divide operation $(Q, R) = A/B$, where Q and R are the quotient and remainder respectively of dividing A by B. The Sentry has access to both the operands as well as the results (Q, R) reported by the processor. Therefore, the Sentry could instead check the division operation using only a multiplier and an adder by checking if $A = QB + R$. Thus, the Sentry need not implement a divider—particularly useful because the latency of an integer divider is typically much higher than the latency of an integer multiplier.

Moreover, the functional units in the Sentry need not have the same designs as the functional units in the processor. The Sentry could use slower, more reliable functional units to check the execution performed by much faster, state-of-the-art functional units. For example, the VFP9-S floating point unit used in many ARM processors has a gate count of 100-130K logic gates and highly optimized functional unit latencies (for e.g. 4 clock cycles for a multiply operations [6]). On the other hand, the Sentry could use a less optimized FPU, such as the DIVA or MONARCH FPU [80], which have a gate count of around 10K logic gates. The functional unit latencies are slightly higher (5 clock cycles

instead of 4 for a multiply operation) but the design complexity becomes much lower. Moreover, the Sentry can compensate for the higher latency of re-execution in such cases with the higher throughput enabled by parallel checking.

In addition to the functional units, the execute stage in the processor must also interface with the data cache. Additionally, for memory instructions, the processor must also contain a load-store queue and the related logic for out-of-order memory instruction execution. Finally, the processor must contain a bypass network to improve performance by speculatively making the results of computations available to later instructions after the write-back stage (and before commit).

The bypass network implements a new data path in the processor. This data path is dominated by wires and multiplexors. The design of bypass networks in modern processors can be extremely complex [58]. Modern processors typically have multilevel bypass networks, which may affect the cycle time of the processor. Clustering may be used to reduce the complexity of the bypass network implementation, but it may require replication of components such as register files as well as distributed issue queues. Relative to these complexities that must be handled in the processor, the Sentry's VG stage simply accepts source operands from the OR stage and forwards the generated results to the Checking (CH) stage.

7.4 Writeback/Commit vs Checking (CH)

The commit stage in an OoO processor helps maintain the program order. Until the commit stage, the processor operates with two separate states: the architectural state and the speculative state. The speculative state related to a commit instruction is merged into the architectural state of the processor. In the commit stage, the execution resources allocated during an instruction's execution such as the ROB entries, memory order buffer entries, physical registers, etc. are also reclaimed. In addition to committing the speculative state,

the commit stage is also responsible for recovering from misspeculation, including branch misspeculation and memory speculation.

Compared to all these functions of the commit unit, the checking stage in the Sentry compares the results of its own re-execution with the results reported by the processor. For output operations, the results calculated by the Sentry are sent to the Pending Output Buffer (a FIFO structure) for release to peripherals. If the checking unit finds any part of the execution to be incorrect, no data is sent to the Pending Output Buffer and the detection of incorrect execution is flagged to the user.

7.5 Processor's Memory/Cache Access vs Sentry's Cache Access

Another aspect of comparison between an OoO processor and the Sentry is their respective caches. The processor contains multiple levels of cache, including separate L1 data and instruction caches and a unified L2 cache. Thus, processors implement extra logic to ensure that a memory request is forwarded to higher levels of the memory hierarchy, whenever the request misses in the first level cache.

Most processors also implement nonblocking caches, where the processor continues executing instructions while a cache miss is serviced. Nonblocking caches also allow the processor to issue new load/store instructions even in the presence of pending cache misses. These caches require a number of additional components such as miss status/information holding registers (MSHRs) to hold information about pending misses and a fill buffer to hold fetched data until they are written to the cache array.

By contrast, the Sentry only contains the L1 data and instruction caches, mirroring the characteristics of the processor's L1 caches, as described in Section 4.3.2. Furthermore, the cache controller on the Sentry can be much simpler than the processor because the processor is responsible for the fill, replacement, timing, and coordination with the rest of

the memory subsystem. The caches themselves can be blocking caches as all values are supplied ahead of time by the processor.

The one source of added complexity in the Sentry's cache comes from the introduction of HMAC engines and the outgoing buffer on the Sentry. In Chapter 6, the HMAC-MD5 implementation presented in [135] with a gate count of 29.2K was used. TrustGuard could even use an alternate HMAC implementation using HMAC-SHA256, with similar HMAC engine complexity. For instance, various HMAC-SHA256 engine designs are available with a gate count less than 24.6K gates [122]. The size of these HMAC engines is much less than many functional units used in a conventional processor. For example, as stated earlier, the VFP9-S floating point unit used in many ARM processors is between 100-130K gates in size [3]. Even basic FPU units used in embedded processors have a gate count over 10K [80]. The HMAC engines on the Sentry may be considered as functional units that are used independently of the RICU checking functionality, operating on cache lines as they are communicated to and from the Sentry's cache. The outgoing buffer is a CAM structure, but its size is limited, which in turn, limits its complexity. The calculation of Merkle Tree addresses on the Sentry can be performed using simple operations such as bit masking and shifting. Therefore, the additional complexity introduced by the cache checking unit of the Sentry is much less significant than the complexity of the processor-only components.

7.6 Summary

The five previous sections break down both execution by the processor and checking by the Sentry into their corresponding stages and compare the complexities of the most closely related stages with each other. It is clear from these sections that in each stage, the number of components in the processor and the amount of work done by those components during execution exceeds the work done by the Sentry's components during checking. The absence of many of these processor components in the Sentry also decreases the amount and

complexity of control logic required.

The key observation is that an untrusted processor can do almost all of the work and hold almost all of the state, including acting as control for the Sentry, without compromising the Sentry's containment guarantees. This stems from the fact that the Sentry only needs to check the architectural state of the processor, thereby obviating the need to check much of the speculative state utilized by some of the processor-only components described earlier. Consequently, we achieve a Sentry design, which is much less complex than out-of-order processors commonly used today and is of comparable complexity to designs that have been verified formally [116, 90].

Chapter 8

Other Related Work

Chapter 2 motivated the CAVO approach, with respect to prior work in building trustworthy hardware. This chapter discusses some other related work, some of which is complementary to CAVO.

Instruction Granularity Monitoring. Prior proposals such as FlexCore [50], Raksha [48] and log-based architectures [39, 41, 40] have focused on utilizing additional hardware to detect software vulnerabilities. These techniques either trust the processor to configure the monitoring hardware correctly or trust that the information flow is correct. Thus, these techniques are all vulnerable to the effects of any malicious change introduced in the hardware during the design or fabrication process. By contrast, the boundary of trust in TrustGuard is restricted to the separately manufactured Sentry. TrustGuard verifies the data sent by the processor independently using its own instruction cache and shadow register file; this allows TrustGuard to detect and isolate the effects of maliciously behaving hardware.

System Monitoring in Hardware. Vigilare [93] and S-Mon [62] propose snooping-based architectures that use a *Snooper* to monitor the system bus and collect the contents of real-time bus traffic. A separate *Verifier* then examines the snooped data to look for sequences of processor execution that violate the integrity of the host system's software.

These techniques are similar to TrustGuard in the sense that the snooper and the verifier can be separately designed from the other components of the system, just like the Sentry in TrustGuard. However, these techniques are aimed at protecting against attacks on kernel integrity (such as kernel-level rootkits) by identifying immutable regions of the kernel and detecting illicit attempts to modify them. Both these techniques trust that the underlying hardware itself is secure. Thus, a backdoor inserted in the processor or memory could circumvent the protections provided by Vigilare and S-Mon. By contrast, TrustGuard assumes the existence of untrusted commodity hardware and isolates any effects of malicious behavior by those components. TrustGuard could be augmented with the idea behind Vigilare and S-Mon to provide added kernel integrity assurance to users.

3-D Integration for Security. Recent research has investigated the implications of 3-D circuit-level integration for secure computer hardware [65, 67]. This approach can provide security through diversity as different layers of the IC stack can be manufactured in separate foundries. Imeson et al. [67] leverage 3-D IC integration to obfuscate circuit designs and protect chips from malicious modifications during fabrication. However, this technique does not protect against malicious modifications during the design phase. Furthermore, their technique leads to a considerable increase in the power dissipation (almost double the original circuit) and circuit delay (upto 115% increase).

Valamehr et al. propose a general methodology to integrate system monitors to a host computation plane by attaching an optional control plane using 3-D integration [126]. The advantages of this design are the availability of high bandwidth and low delays between the host and the monitoring chip. 3-D integration provides an alternate way of integrating the Sentry with a commodity system and would satisfy TrustGuard's bandwidth requirements. However, such a design would not be pluggable and swappable as the security chip is overlaid as a foundry-level configuration option.

Processor Verification. Processor designers rely on functional verification techniques to detect hardware Trojans and ensure correctness of the system. Some designers use simulation-based tests to validate the processor. As the test space for modern-day complex processors is prohibitively huge, designers rely on pseudorandom test-case generation to cover the test space. Consequently, there is often a big gap between the generated test space and the actual one [8, 30, 144].

An alternate approach uses formal verification techniques guaranteed to be complete [33, 66, 72, 90, 102, 105, 116, 127, 140]. However, formal verification may take much longer than simulation-based methods and often requires skiled human support for complex processor designs. Hardware designers have also used formal equivalence checking to formally prove that the RTL code and the netlist synthesized from it have exactly the same behavior [83].

In general, two significant factors affect the use of formal verification techniques in processor development: (1) scaling issues related to the size of the processor specifications, the size and complexity of processor designs, and the size of the design and verification teams; and (2) return on investment issues including the need to catch bugs early in and through the development process and the need to reuse verification IP, tools, and techniques across a wide range of designs [106]. Recently, Reid et al. presented ISA-Formal as a broadly applicable formal verification technique for verifying processor pipeline control [106]. However, even though formal verification can ensure the correctness of hardware designs, they cannot ensure that those designs were faithfully fabricated during the later stages of the manufacturing process.

Side Channel Attacks. A lot of research has been devoted to protecting systems against side channel attacks. Phantom [89] and Raccoon [104] use obfuscation to provide confidentiality guarantees. Sebastian et al. proposed hardware transformation techniques to construct leakage resistant circuits [53]. All these works are complementary to TrustGuard.

Used in conjunction with TrustGuard, they could be used to strengthen the overall trustworthiness of the system, in particular to counter techniques that might bypass the physical gap bridged by the Sentry.

Memory Integrity Assurance. The XOM architecture [87] uses an encryption key to preserve privacy. For stores, the stored data is appended with a hash of itself, and for loads, XOM can verify that the fetched data were indeed stored by the program previously. Similar to our MAC-based memory integrity, XOM's integrity mechanism is vulnerable to replay attacks by which the memory returns stale data previously stored at the same address during the same execution. To solve this issue, Gassend et al. utilize a hash tree structure to verify the integrity of large memory and also optimize the number of hash reads [56]. To further reduce runtime overhead, Suh et al. propose a new encryption mechanism that can hide the encryption latency by decoupling computations for decryption from off-chip data accesses and keep a log to enable a separate integrity check operation when necessary [118, 44]. While many of these ideas are similar to SMACs, they are all based on the assumption that the processor can be trusted. TrustGuard ensures memory integrity even when the processor itself is untrusted.

Chapter 9

Conclusion and Future Directions

This dissertation has proposed the Containment Architecture with Verified Output (CAVO) model to enable the containment of malicious effects of untrusted hardware. It proposed and evaluated the TrustGuard architecture to show the feasibility of the CAVO model.

9.1 Conclusion

Current hardware verification tools are not mature enough to secure complex hardware components through the entire process of manufacturing. To address the problem of establishing trust in the system in the presence of untrusted hardware components, this dissertation proposed the concept of a *Containment Architecture with Verified Output* (*CAVO*). CAVO is based on the insight that the effects of malicious behavior by untrusted hardware can be *contained* as long as malicious components are not allowed to communicate externally. Containment is achieved by requiring that all communication from the system originate from a small, separately manufactured component, called the Sentry. The Sentry acts as a gatekeeper whose sole purpose is to be the foundation of security for the rest of the system and is simple enough to be reliably secured and verified.

This dissertation demonstrated the feasibility of CAVO with the presentation and evaluation of the first prototype design, named *TrustGuard*, along with the security guarantees it

provides. In TrustGuard, the untrusted processor and other system components must send execution information to the Sentry in order to be able to communicate externally. The Sentry can use this execution information to check that any attempted communication results from the correct execution of signed software. Using the execution information also allows the Sentry to minimize the performance impact to the system.

Simulations on 11 SPEC CPU benchmarks demonstrated that the TrustGuard Sentry can provide containment guarantees for the system, while causing a geomean decline of 8.50% (and an arithmetic mean decline of 12%) in the untrusted processor's IPC, when the Sentry operates at half the clock frequency of the processor. Containment by TrustGuard also requires the addition of one new chip (the Sentry) to the system and a geomean energy overhead of 28.7%. The average performance of TrustGuard is comparable to the performance decline reported by various hardware security proposals such as AEGIS (average 11% slowdown for secure processing), Raksha (average 37% for dynamic information flow tracking), FlexCore (average 8% for runtime monitoring), and traditional Merkle tree implementations for memory integrity (average 13% IPC decline). Moreover, these techniques implicitly trust the complex processor to function correctly, while TrustGuard can provide additional security by containing malicious behavior by the processor itself.

The use of the Sentry for containment allows the separation of the manufacturers supply chain into two different paths—one that emphasizes performance and another that emphasizes security and trust. The performance-centric supply chain builds complex, difficult-to-verify components such as the processor using the state-of-the-art semiconductor and microarchitectural technology. The security-centric supply chain builds the Sentry in domestic, closely-controlled, trusted fabrication plants. This separation provides an attractive middle path between the two extremes of either building security-conscious components that may discard the latest microarchitectural advances or ignoring security requirements in the pursuit of better performance and efficiency.

9.2 Future Research Directions

We propose some avenues for future research into the CAVO model, based on relaxing some of the requirements of the TrustGuard design.

Software Trust Model. TrustGuard assures that communications from the system originate only from signed software executing correctly. However, signing is not a guarantee of the software actually being secure. The generic concept of CAVO may enable supporting stronger guarantees about the security of the software. For example, the system could require all software to carry a proof of correctness, which could be efficiently verified at runtime with the cooperation of the Sentry.

Formal Verification of the Sentry. As stated in Chapter 7, the Sentry in TrustGuard is simpler than the untrusted processor, which makes it more amenable to formal verification by existing tools and techniques. Formally proving the correctness of the Sentry and the security guarantees offered by TrustGuard can be an avenue for future research.

Addressing Architectural Restrictions of TrustGuard. TrustGuard provides containment guarantees for a system with a single-core out-of-order processor. However, extending TrustGuard to today's multicore processor-based systems is essential to make a case for its widespread adoption. Extending the TrustGuard design to multicore processors would involve answering research questions such as: (1) increasing the throughput of the Sentry in the face of the higher bandwidth requirements for communication between the multicore processor and the Sentry ; and (2) handling the nondeterminism introduced by parallel execution contexts in a multicore processor.

Furthermore, TrustGuard requires the design of the Sentry to be coupled with the design of the untrusted processor (for example, using the same ISA, requiring minor processor modifications, etc.) The applicability and effectiveness of TrustGuard can be improved

by relaxing these requirements. To enable this, future research could focus on the question of whether the functionality provided by the modified processor could be efficiently performed by logic implemented in software.

Bibliography

[1] SPEC CPU2006 benchmarks – gem5. Website:http://www.gem5.org/SPEC_CPU2006_benchmarks.

[2] Synopsys DesignWare IP for PCI Express (PCIe) solution. Website:http://www.synopsys.com/IP/InterfaceIP/PCIExpress/Pages/default.aspx.

[3] Floating point. Website:https://www.arm.com/products/processors/technologies/vector-floating-point.php, November 2016.

[4] iOS Security: iOS 9.3 or later (White Paper). May 2016. Website:https://www.apple.com/business/docs/iOS_Security_Guide.pdf.

[5] Spencer Ackerman and Sam Thielman. US Intelligence Chief: We Might Use the Internet of Things to Spy on You. https://www.theguardian.com/technology/2016/feb/09/internet-of-things-smart-home-devices-government-surveillance-james-clapper, February 2016.

[6] Advanced RISC Machines. VFP9-S Floating-Point Coprocessor Technical Reference Manual. Version r0p2. Technical report, Advanced RISC Machines, 2010.

[7] Dakshi Agrawal, Selcuk Baktir, Deniz Karakoyunlu, Pankaj Rohatgi, and Berk Sunar. Trojan Detection Using IC Fingerprinting. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 296–310, Washington, DC, USA, 2007. IEEE Computer Society.

[8] Aharon Aharon, Dave Goodman, Moshe Levinger, Yossi Lichtenstein, Yossi Malka, Charlotte Metzger, Moshe Molcho, and Gil Shurek. Test Program Generation for Functional Verification of PowerPC Processors in IBM. In *Proceedings of the 32nd Design Automation Conference*, pages 279–285, 1995.

[9] Jasmin Ajanovic. PCI Express* (PCIe*) 3.0 Accelerator Features, 2008.

[10] Jasmin Ajanovic. PCI Express 3.0 Overview. In *Proceedings of Hot Chips: A Symposium on High Performance Chips*, August 2009.

[11] R. Anderson, M. Bond, J. Clulow, and S. Skorobogatov. Cryptographic Processors-A Survey. *Proceedings of the IEEE*, 94(2):357–369, Feb 2006.

[12] Andrew W. Appel. Verified Software Toolchain. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ESOP'11/ETAPS'11, pages 1–17, Berlin, Heidelberg, 2011. Springer-Verlag.

[13] Andrew W. Appel. Verification of a Cryptographic Primitive: SHA-256. *ACM Trans. Program. Lang. Syst.*, 37(2):7:1–7:31, April 2015.

[14] Desire Athow. Intel's Haswell Processors Hit by TSX Bug. `http://www.techradar.com/news/computing-components/processors/intel-s-haswell-processors-hit-by-tsx-bug-1261578`, August 2014.

[15] Desire Athow. Pentium FDIV: The Processor Bug that Shook the World. http://www.techradar.com/news/computing-components/processors/pentium-fdiv-the-processor-bug-that-shook-the-world-1270773, October 2014.

[16] Todd Austin and Valeria Bertacco. Deployment of Better than Worst-Case Design: Solutions and Needs. In *Proceedings of IEEE International Conference on Computer Design*, pages 550–555. IEEE, 2005.

[17] Todd Austin, Valeria Bertacco, David Blaauw, and Trevor Mudge. Opportunities and Challenges for Better than Worst-Case Design. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pages 2–7. ACM, 2005.

[18] Todd M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 32, pages 196–207, Washington, DC, USA, 1999. IEEE Computer Society.

[19] Todd M Austin. DIVA: A Dynamic Approach to Microprocessor Verification. *Journal of Instruction-Level Parallelism*, 2:1–6, 2000.

[20] Todd M Austin. Designing Robust Microarchitectures. In *Proceedings of 41st Design Automation Conference*, pages 78–78. IEEE, 2004.

[21] A. Avizienis. The methodology of N-version programming. Chapter 2 of Software Fault Tolerance, M. R. Lyu (ed.), Wiley, 23-46, 1995., 1995.

[22] Mainak Banga and Michael S. Hsiao. A Region Based Approach for the Identification of Hardware Trojans. In *Proceedings of the 2008 IEEE International Workshop on Hardware-Oriented Security and Trust*, HST '08, pages 40–47, Washington, DC, USA, 2008. IEEE Computer Society.

[23] Mark Beaumont, Bradley Hopkins, and Tristan Newby. SAFER PATH: Security Architecture Using Fragmented Execution and Replication for Protection Against Trojaned Hardware. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, pages 1000–1005, San Jose, CA, USA, 2012. EDA Consortium.

[24] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct Non-interactive Zero Knowledge for a Von Neumann Architecture. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 781–796, Berkeley, CA, USA, 2014. USENIX Association.

[25] Janick Bergeron. *Writing Testbenches: Functional Verification of HDL Models, Second Edition*. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[26] S. Bhunia, M. S. Hsiao, M. Banga, and S. Narasimhan. Hardware Trojan Attacks: Threat Analysis and Countermeasures. *Proceedings of the IEEE*, 102(8):1229–1247, Aug 2014.

[27] Eli Biham, Yaniv Carmeli, and Adi Shamir. Bug attacks. In David Wagner, editor, *Advances in Cryptology–CRYPTO 2008*, volume 5157 of *Lecture Notes in Computer Science*, pages 221–240. Springer Berlin Heidelberg, 2008.

[28] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 Simulator. *SIGARCH Comput. Archit. News*, 39(2), August 2011.

[29] Gedare Bloom, Bhagirath Narahari, Rahul Simha, and Joseph Zambreno. Providing Secure Execution Environments with a Last Line of Defense Against Trojan Circuit Attacks. *Comput. Secur.*, 28(7):660–669, October 2009.

[30] P. Bose, T.M. Conte, and T.M. Austin. Challenges in processor modeling and validation [Guest Editors' introduction]. *Micro, IEEE*, 19(3):9–14, May 1999.

[31] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Verifying Computations with State. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 341–357, New York, NY, USA, 2013. ACM.

[32] Jonathan Brossard. Hardware Backdooring is Practical. In *Blackhat*, 2012.

[33] J. R. Burch and D. L. Dill. Automatic Verification of Pipelined Microprocessors Control. In *Proceedings of the Sixth International Conference on Computer-Aided Verification (CAV)*, volume 818, pages 68–80, Stanford, California, United States, 1994. Springer-Verlag.

[34] Michael Bushnell and Vishwani D Agrawal. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*, volume 17. Springer Science & Business Media, 2000.

[35] Rajat Subhra Chakraborty and Swarup Bhunia. HARPOON: An Obfuscation-based SoC Design Methodology for Hardware Protection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(10):1493–1502, 2009.

[36] Rajat Subhra Chakraborty, Somnath Paul, and Swarup Bhunia. On-demand Transparency for Improving Hardware Trojan Detectability. In *Proceedings of the 2008 IEEE International Workshop on Hardware-Oriented Security and Trust*, HST '08, pages 48–50, Washington, DC, USA, 2008. IEEE Computer Society.

[37] David Champagne and Ruby B. Lee. Scalable Architectural Support for Trusted Software. In *Proceedings of the Sixteenth International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–12, January 2010.

[38] Saugata Chatterjee, Chris Weaver, and Todd Austin. Efficient Checker Processor Design. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 87–97. ACM, 2000.

[39] Shimin Chen, Babak Falsafi, Phillip B. Gibbons, Michael Kozuch, Todd C. Mowry, Radu Teodorescu, Anastassia Ailamaki, Limor Fix, Gregory R. Ganger, Bin Lin, and Steven W. Schlosser. Log-based Architectures for General-purpose Monitoring of Deployed Code. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, pages 63–65, New York, NY, USA, 2006. ACM.

[40] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, and Todd C. Mowry. Log-based Architectures: Using Multicore to Help Software Behave Correctly. *SIGOPS Oper. Syst. Rev.*, 45(1):84–91, February 2011.

[41] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. Flexible Hardware Acceleration for Instruction-Grain Program Monitoring. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 377–388, Washington, DC, USA, 2008. IEEE Computer Society.

[42] Siddhartha Chhabra, Yan Solihin, Reshma Lal, and Matthew Hoekstra. Transactions on Computational Science VII. chapter An Analysis of Secure Processor Architectures. Springer-Verlag, 2010.

[43] Cisco. IoT Threat Environment White Paper. Technical report, 2015.

[44] Dwaine Clarke, Srinivas Devadas, Marten van Dijk, Blaise Gassend, and G. Edward Suh. Incremental Multiset Hash Functions and Their Application to Memory In-

tegrity Checking. In *Advances in Cryptology–CRYPTO 2003 Proceedings, volume 2894 of Lecture Notes in Computer Science*, pages 188–207. Springer-Verlag, 2003.

[45] Stephen Cobb. Cybersecurity and Manufacturers: What the Costly Chrysler Jeep Hack Reveals. http://www.welivesecurity.com/2015/07/29/cybersecurity-manufacturing-chrysler-jeep-hack/, July 2015.

[46] Lucian Constantin. Armies of Hacked IoT Devices Launch Unprecedented DDoS Attacks. http://www.computerworld.com/article/3124345/security/armies-of-hacked-iot-devices-launch-unprecedented-ddos-attacks.html, September 2016.

[47] Michael J Covington and Rush Carskadden. Threat Implications of the Internet of Things. In *Cyber Conflict (CyCon), 2013 5th International Conference on*, pages 1–12. IEEE, 2013.

[48] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 482–493, New York, NY, USA, 2007. ACM.

[49] Sajal K. Das, Krishna Kant, and Nan Zhang. *Handbook on Securing Cyber-Physical Critical Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[50] Daniel Y. Deng, Daniel Lo, Greg Malysa, Skyler Schneider, and G. Edward Suh. Flexible and Efficient Instruction-Grained Run-Time Monitoring Using On-Chip Reconfigurable Fabric. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 137–148, Washington, DC, USA, 2010. IEEE Computer Society.

[51] Milenko Drinic and Darko Kirovski. A Hardware-Software Platform for Intrusion Prevention. In *Proceedings of the 37th Annual IEEE/ACM International Symposium*

on Microarchitecture, MICRO 37, pages 233–242, Washington, DC, USA, 2004. IEEE Computer Society.

[52] J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, and S. W. Smith. Building the IBM 4758 Secure Coprocessor,. *Computer*, 34(10):57–66, Oct 2001.

[53] Sebastian Faust, Tal Rabin, Leonid Reyzin, Eran Tromer, and Vinod Vaikuntanathan. Protecting Circuits from Leakage: the Computationally-Bounded and Noisy Cases. In *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings*, pages 135–156, 2010.

[54] E. Fernandes, J. Jung, and A. Prakash. Security Analysis of Emerging Smart Home Applications. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 636–654, May 2016.

[55] Stefan Frei, Bernhard Tellenbach, and Bernhard Plattner. 0-day Patch - Exposing Vendors (in)Security Performance. In *Blackhat Europe*, 2008.

[56] Blaise Gassend, G. Edward Suh, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Caches and Hash Trees for Efficient Memory Integrity Verification. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA '03, pages 295–, Washington, DC, USA, 2003. IEEE Computer Society.

[57] Mohamed Gomaa, Chad Scarbrough, T. N. Vijaykumar, and Irith Pomeranz. Transient-Fault Recovery for Chip Multiprocessors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 98–109. ACM Press, 2003.

[58] Antonio Gonzalez, Fernando Latorre, and Grigorios Magklis. *Processor Microarchitecture: An Implementation Perspective*. Morgan & Claypool Publishers, 2010.

[59] Mikhail Gorobets, Oleksandr Bazhaniuk, Alex Matrosov, Andrew Furtak, and Yuriy Bulygin. Attacking Hypervisors via Firmware and Hardware. In *Blackhat*, 2015.

[60] Andy Greenberg. Hackers Remotely Kill a Jeep on the Highway. https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/, July 2015.

[61] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[62] Ingoo Heo, Daehee Jang, Hyungon Moon, Hansu Cho, Seungwook Lee, Brent Byunghoon Kang, and Yunheung Paek. Efficient Kernel Integrity Monitor Design for Commodity Mobile Application Processors. *Journal of Semiconductor Technology and Science*, 15(1):48–59, 2015.

[63] Matthew Hicks, Murph Finnicum, Samuel T. King, Milo M. K. Martin, and Jonathan M. Smith. Overcoming an Untrusted Computing Base: Detecting and Removing Malicious Hardware Automatically. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 159–172, Washington, DC, USA, 2010. IEEE Computer Society.

[64] A. J. Hu. Formal Hardware Verification with BDDs: An Introduction. In *Communications, Computers and Signal Processing, 1997. 10 Years PACRIM 1987-1997 - Networking the Pacific Rim. 1997 IEEE Pacific Rim Conference on*, volume 2, pages 677–682 vol.2, Aug 1997.

[65] Ted Huffmire, Timothy Levin, Michael Bilzor, Cynthia E. Irvine, Jonathan Valamehr, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. Hardware Trust Implications of 3-D Integration. In *Proceedings of the 5th Workshop on Embedded Systems Security*, WESS '10, pages 1:1–1:10, New York, NY, USA, 2010. ACM.

[66] Warren A. Hunt, Jr. Microprocessor Design Verification. *J. Autom. Reason.*, 5(4):429–460, November 1989.

[67] Frank Imeson, Ariq Emtenan, Siddharth Garg, and Mahesh Tripunitara. Securing Computer Hardware Using 3D Integrated Circuit (IC) Technology and Split Manufacturing for Obfuscation. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 495–510, Washington, D.C., 2013. USENIX.

[68] Yier Jin and Yiorgos Makris. Hardware Trojan Detection Using Path Delay Fingerprint. In *Proceedings of the 2008 IEEE International Workshop on Hardware-Oriented Security and Trust*, HST '08, pages 51–57, Washington, DC, USA, 2008. IEEE Computer Society.

[69] Moritz Jodeit. Hacking Video Conferencing Systems. In *Blackhat Europe*, 2013.

[70] R. Kaivola and N. Narasimhan. Formal Verification of the Pentium®4 Floating-Point Multiplier. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '02, pages 20–, Washington, DC, USA, 2002. IEEE Computer Society.

[71] Henrique Kawakami, Roberto Gallo, Ricardo Dahab, and Erick Nascimento. Hardware Security Evaluation Using Assurance Case Models. In *Proceedings of the 2015 10th International Conference on Availability, Reliability and Security*, ARES '15, pages 193–198, Washington, DC, USA, 2015. IEEE Computer Society.

[72] Christoph Kern and Mark R. Greenstreet. Formal Verification in Hardware Design: A Survey. *ACM Trans. Des. Autom. Electron. Syst.*, 4(2):123–193, April 1999.

[73] H. Khattri, N. K. V. Mangipudi, and S. Mandujano. HSDL: A Security Development Lifecycle for Hardware Technologies. In *2012 IEEE International Symposium on Hardware-Oriented Security and Trust*, pages 116–121, June 2012.

[74] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *Proceedings of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 361–372, Piscataway, NJ, USA, 2014. IEEE Press.

[75] Samuel T. King, Joseph Tucek, Anthony Cozzie, Chris Grier, Weihang Jiang, and Yuanyuan Zhou. Designing and Implementing Malicious Hardware. In *Proceedings of the 1st USENIX Workshop on Large-Scale Exploits and Emergent Threats*, LEET'08, pages 5:1–5:8, Berkeley, CA, USA, 2008. USENIX Association.

[76] Darko Kirovski, Milenko Drinić, and Miodrag Potkonjak. Enabling Trusted Software Integrity. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 108–120, New York, NY, USA, 2002. ACM.

[77] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[78] Farinaz Koushanfar and Azalia Mirhoseini. A Unified Framework for Multimodal Submodular Integrated Circuits Trojan Detection. *IEEE Transactions on Information Forensics and Security*, 6(1):162–174, 2011.

[79] Thomas Kropf. *Introduction to Formal Hardware Verification: Methods and Tools for Designing Correct Circuits and Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1999.

[80] Taek-Jun Kwon, J. Sondeen, and J. Draper. Design Trade-Offs in Floating-Point Unit Implementation for Embedded and Processing-in-Memory Systems. In *2005 IEEE International Symposium on Circuits and Systems*, pages 3331–3334 Vol. 4, May 2005.

[81] Shuvendu K. Lahiri and Randal E. Bryant. Deductive Verification of Advanced Out-of-Order Microprocessors. In Warren A. Hunt and Fabio Somenzi, editors, *Computer Aided Verification: 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003. Proceedings*, pages 341–354, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[82] William K. Lam. *Hardware Design Verification: Simulation and Formal Method-Based Approaches (Prentice Hall Modern Semiconductor Design Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[83] Luciano Lavagno, Grant Martin, and Louis Scheffer. *Electronic Design Automation for Integrated Circuits Handbook - 2 Volume Set*. CRC Press, Inc., Boca Raton, FL, USA, 2006.

[84] Xavier Leroy. Formal Verification of a Realistic Compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[85] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 469–480, New York, NY, USA, 2009. ACM.

[86] David Lie, John Mitchell, Chandramohan A. Thekkath, and Mark Horowitz. Specifying and verifying hardware for tamper-resistant software. In *Proceedings of the*

2003 IEEE Symposium on Security and Privacy, SP '03, pages 166–, Washington, DC, USA, 2003. IEEE Computer Society.

[87] David Lie, Chandramohan A. Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John C. Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. In *Architectural Support for Programming Languages and Operating Systems*, pages 168–177, 2000.

[88] E. Love, Y. Jin, and Y. Makris. Proof-Carrying Hardware Intellectual Property: A Pathway to Trusted Module Acquisition. *IEEE Transactions on Information Forensics and Security*, 7(1):25–40, Feb 2012.

[89] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. PHANTOM: Practical Oblivious Computation in A Secure Processor. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 311–324, 2013.

[90] Panagiotis Manolios and Sudarshan K. Srinivasan. Automatic Verification of Safety and Liveness for XScale-Like Processor Models Using srinivasanWEB Refinements. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '04, pages 10168–, Washington, DC, USA, 2004. IEEE Computer Society.

[91] D. McIntyre, F. Wolff, C. Papachristou, S. Bhunia, and D. Weyer. Dynamic Evaluation of Hardware Trust. In *Proceedings of the 2009 IEEE International Workshop on Hardware-Oriented Security and Trust*, HST '09, pages 108–111, Washington, DC, USA, 2009. IEEE Computer Society.

[92] Maher Mneimneh, Fadi Aloul, Chris Weaver, Saugata Chatterjee, Karem Sakallah, and Todd Austin. Scalable Hybrid Verification of Complex Microprocessors. In

Proceedings of the 38th Annual Design Automation Conference, DAC '01, pages 41–46, New York, NY, USA, 2001. ACM.

[93] Hyungon Moon, Hojoon Lee, Jihoon Lee, Kihwan Kim, Yunheung Paek, and Brent ByungHoon Kang. Vigilare: Toward Snoop-Based Kernel Integrity Monitor. In *ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 28–37, 2012.

[94] Shubhendu S. Mukherjee, Michael Kontz, and Steven K. Reinhardt. Detailed Design and Evaluation of Redundant Multithreading Alternatives. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, pages 99–110, Washington, DC, USA, 2002. IEEE Computer Society.

[95] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: highly compatible and complete spatial memory safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[96] George C. Necula. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 106–119, New York, NY, USA, 1997. ACM.

[97] H.M. O'Brien. The Internet Of Things: Liability Risks For Tech Cos. http://www.law360.com/articles/680256/the-internet-of-things-liability-risks-for-tech-cos.

[98] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Error Detection by Duplicated Instructions in Super-Scalar Processors. In *IEEE Transactions on Reliability*, volume 51, pages 63–75, March 2002.

[99] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly Practical Verifiable Computation. In *Proceedings of the 2013 IEEE Symposium on*

Security and Privacy, SP '13, pages 238–252, Washington, DC, USA, 2013. IEEE Computer Society.

[100] Diego Perez-Botero, Jakub Szefer, and Ruby B. Lee. Characterizing Hypervisor Vulnerabilities in Cloud Computing Servers. In *Proceedings of the 2013 International Workshop on Security in Cloud Computing*, 2013.

[101] Graeme Proudler, Liqun Chen, and Chris Dalton. *Trusted Computing Platforms - TPM2.0 in Context*. Springer, 2014.

[102] S.P. Rajan, N. Shankar, and M.K. Srivas. Industrial Strength Formal Verification Techniques for Hardware Designs. In *VLSI Design, 1997. Proceedings., Tenth International Conference on*, pages 208–212, Jan 1997.

[103] Ralph C. Merkle. Protocols for Public Key Cryptosystems. In *IEEE Symposium on Security and Privacy*, pages 122–122, April 1980.

[104] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 431–446, 2015.

[105] M. Rangarajan, P. Alexander, and N.B. Abu-Ghazaleh. Using Automatable Proof Obligations for Component-Based Design Checking. In *Engineering of Computer-Based Systems, 1999. Proceedings. ECBS '99. IEEE Conference and Workshop on*, pages 304–310, Mar 1999.

[106] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. End-to-End Verification of Processors with ISA-Formal. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification: 28th International Conference, CAV*

2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II, pages 42–58, Cham, 2016. Springer International Publishing.

[107] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 25–36. ACM Press, 2000.

[108] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software Implemented Fault Tolerance. In *Proceedings of the 3rd International Symposium on Code Generation and Optimization*, March 2005.

[109] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 183–196, Washington, DC, USA, 2007. IEEE Computer Society.

[110] Brian Rogers, Milos Prvulovic, and Yan Solihin. Efficient data protection for distributed shared memory multiprocessors. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, PACT '06, pages 84–94, New York, NY, USA, 2006. ACM.

[111] Eric Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, page 84. IEEE Computer Society, 1999.

[112] Jarrod A Roy, Farinaz Koushanfar, and Igor L Markov. EPIC: Ending Piracy of Integrated Circuits. In *Proceedings of the conference on Design, automation and test in Europe*, pages 1069–1074. ACM, 2008.

[113] Alex Shye, Tipp Moseley, Vijay Janapa Reddi, Joseph Blomstedt, and Daniel A. Connors. Using Process-Level Redundancy to Exploit Multiple Cores for Transient

Fault Tolerance. *International Conference on Dependable Systems and Networks*, 0:297–306, 2007.

[114] Timothy J. Slegel, Robert M. Averill III, Mark A. Check, Bruce C. Giamei, Barry W. Krumm, Christopher A. Krygowski, Wen H. Li, John S. Liptay, John D. MacDougall, Thomas J. McPherson, Jennifer A. Navarro, Eric M. Schwarz, Kevin Shum, and Charles F. Webb. IBM's S/390 G5 Microprocessor design. In *IEEE Micro*, volume 19, pages 12–23, March 1999.

[115] S. K. Srinivasan. Automatic Refinement Checking of Pipelines with Out-of-Order Execution. *IEEE Transactions on Computers*, 59(8):1138–1144, Aug 2010.

[116] Sudarshan K. Srinivasan and Miroslav N. Velev. Formal Verification of an Intel XScale Processor Model with Scoreboarding, Specialized Execution Pipelines, and Impress Data-Memory Exceptions. In *Proceedings of the First ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, MEMOCODE '03, pages 65–, Washington, DC, USA, 2003. IEEE Computer Society.

[117] Cynthia Sturton, Matthew Hicks, David Wagner, and Samuel T. King. Defeating UCI: Building Stealthy and Malicious Hardware. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 64–77, Washington, DC, USA, 2011. IEEE Computer Society.

[118] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Efficient Memory Integrity Verification and Encryption for Secure Processors. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 339–, Washington, DC, USA, 2003. IEEE Computer Society.

[119] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing.

In *Proceedings of the 17th Annual International Conference on Supercomputing*, ICS '03, pages 160–171, New York, NY, USA, 2003. ACM.

[120] G. Edward Suh, Charles W. O'Donnell, and Srinivas Devadas. AEGIS: A Single-Chip Secure Processor. *IEEE Des. Test*, 24(6):570–580, November 2007.

[121] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenburg. Slipstream Processors: Improving Both Performance and Fault Tolerance. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–268. ACM Press, 2000.

[122] Helion Technology. Full Datasheet–Fast Hash Core Family for ASIC. `http://www.heliontech.com/downloads/fast_hash_asic_datasheet.pdf`.

[123] M Tehranipoor and F Koushanfar. A Survey of Hardware Trojan Taxonomy and Detection. *Design Test of Computers, IEEE*, 27(1):10–25, Jan 2010.

[124] Justin Thaler, Mike Roberts, Michael Mitzenmacher, and Hanspeter Pfister. Verifiable Computation with Massively Parallel Interactive Proofs. In *4th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'12, Boston, MA, USA, June 12-13, 2012*, 2012.

[125] Martin Thuresson, Lawrence Spracklen, and Per Stenstrom. Memory-Link Compression Schemes: A Value Locality Perspective. *IEEE Trans. Comput.*, 57(7), 2008.

[126] Jonathan Valamehr, Mohit Tiwari, Timothy Sherwood, Ryan Kastner, Ted Huffmire, Cynthia E. Irvine, and Timothy E. Levin. Hardware Assistance for Trustworthy Systems through 3-D Integration. In *Twenty-Sixth Annual Computer Security Applications Conference, ACSAC 2010, Austin, Texas, USA, 6-10 December 2010*, pages 199–210, 2010.

[127] Muralidaran Vijayaraghavan, Adam Chlipala, Arvind, and Nirav Dave. Modular Deductive Verification of Multiprocessor Hardware Designs. In *Computer Aided Verification - 27th International Conference, CAV*.

[128] Victor Vu, Srinath Setty, Andrew J Blumberg, and Michael Walfish. A Hybrid Architecture for Interactive Verifiable Computation. In *IEEE Symposium on Security and Privacy (SP)*, pages 223–237. IEEE, 2013.

[129] Riad S. Wahby, Max Howald, Siddharth Garg, abhi shelat, and Michael Walfish. Verifiable ASICs. In *IEEE Symposium on Security and Privacy (Oakland) 2016*, 2016.

[130] Riad S. Wahby, Srinath T. V. Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and Control Flow in Verifiable Outsourced Computation. In *22nd Annual Network and Distributed System Security Symposium (NDSS)*, 2015.

[131] Adam Waksman and Simha Sethumadhavan. Tamper Evident Microprocessors. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 173–188, Washington, DC, USA, 2010. IEEE Computer Society.

[132] Adam Waksman and Simha Sethumadhavan. Silencing Hardware Backdoors. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 49–63, Washington, DC, USA, 2011. IEEE Computer Society.

[133] Adam Waksman, Matthew Suozzo, and Simha Sethumadhavan. FANCI: Identification of Stealthy Malicious Logic Using Boolean Functional Analysis. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 697–708, New York, NY, USA, 2013. ACM.

[134] Cheng Wang, Ho-Seop Kim, Youfeng Wu, and Victor Ying. Compiler-Managed Software-based Redundant Multi-Threading for Transient Fault Detection. In *Pro-*

ceedings of the International Symposium on Code Generation and Optimization, pages 244–258, Washington, DC, USA, 2007. IEEE Computer Society.

[135] Mao-Yin Wang, Chih-Pin Su, Chih-Tsun Huang, and Cheng-Wen Wu. An HMAC Processor with Integrated SHA-1 and MD5 Algorithms. In *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, ASP-DAC '04, pages 456–458, Piscataway, NJ, USA, 2004. IEEE Press.

[136] Nicholas J Wang and Sanjay J Patel. ReStore: Symptom-based soft error detection in microprocessors. *IEEE Transactions on Dependable and Secure Computing*, 3(3):188–201, 2006.

[137] Xiaoxiao Wang, Mohammad Tehranipoor, and Jim Plusquellic. Detecting Malicious Inclusions in Secure Hardware: Challenges and Solutions. In *Proceedings of the 2008 IEEE International Workshop on Hardware-Oriented Security and Trust*, HST '08, pages 15–19, Washington, DC, USA, 2008. IEEE Computer Society.

[138] Chris Weaver and Todd M. Austin. A Fault Tolerant Approach to Microprocessor Design. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (Formerly: FTCS)*, DSN '01, pages 411–420, Washington, DC, USA, 2001. IEEE Computer Society.

[139] Sheng Wei, Saro Meguerdichian, and Miodrag Potkonjak. Malicious Circuitry Detection Using Thermal Conditioning. *IEEE Transactions on Information Forensics and Security*, 6(3):1136–1145, 2011.

[140] P.J. Windley. Formal Modeling and Verification of Microprocessors. *Computers, IEEE Transactions on*, 44(1):54–72, Jan 1995.

[141] Rafal Wojtczuk. Poacher Turned Gatekeeper: Lessons Learned from Eight Years of Breaking Hypervisors. In *Blackhat*, 2014.

[142] Francis Wolff, Chris Papachristou, Swarup Bhunia, and Rajat S. Chakraborty. Towards Trojan-free Trusted ICs: Problem Analysis and Detection Scheme. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, pages 1362–1365, New York, NY, USA, 2008. ACM.

[143] Chenyu Yan, Daniel Englender, Milos Prvulovic, Brian Rogers, and Yan Solihin. Improving Cost, Performance, and Security of Memory Encryption and Authentication. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, pages 179–190, Washington, DC, USA, 2006. IEEE Computer Society.

[144] Yingbiao Yao, Jianwu Zhang, Bin Wang, and Qingdong Yao. A Pseudo-Random Program Generator for Processor Functional Verification. In *International Symposium on Integrated Circuits, 2007*, pages 441–444, Sept 2007.

[145] Kim Zetter. A Cyberattack Has Caused Confirmed Physical Damage for the Second Time Ever. https://www.wired.com/2015/01/german-steel-mill-hack-destruction/, August 2015.

[146] Zetter, Kim. NSA laughs at PCs, prefers hacking routers and switches. `http://www.wired.com/2013/09/nsa-router-hacking/`, September 2013.

[147] J. Zhang and Q. Xu. On Hardware Trojan Design and Implementation at Register-Transfer Level. In *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 107–112, June 2013.

[148] Jie Zhang, Feng Yuan, Lingxiao Wei, Zelong Sun, and Qiang Xu. VeriTrust: Verification for Hardware Trust. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 61:1–61:8, New York, NY, USA, 2013. ACM.

[149] Yun Zhang, Soumyadeep Ghosh, Jialu Huang, Jae W. Lee, Scott A. Mahlke, and David I. August. Runtime Asynchronous Fault Tolerance via Speculation. In *Pro-*

ceedings of the Tenth International Symposium on Code Generation and Optimization, CGO '12, pages 145–154, New York, NY, USA, 2012. ACM.

[150] Yun Zhang, Jae W. Lee, Nick P. Johnson, and David I. August. DAFT: Decoupled Acyclic Fault Tolerance. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 87–98, New York, NY, USA, 2010. ACM.