TrustGuard: A Model for Practical Trust in Real Systems

Stephen R. Beard

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

June 2019

Abstract

All layers of today's computing systems, from hardware to software, are vulnerable to attack. Market and economic pressure push companies to focus on performance and features, leaving security and correctness as secondary concerns. As a consequence, systems must frequently be patched after deployment to fix security vulnerabilities. While this non-virtuous exploit-patch-exploit cycle is insecure, it is practical enough for companies to use.

Formal methods in both software and hardware can guarantee the security they provide. Ideally, modern systems would be comprised of formally verified and secure components. Unfortunately, such methods have not seen widespread adoption for a number of reasons, such as difficulty in scaling, lack of tools, and high skill requirements. Additionally, the economics involved in securing and replacing every component in all systems, both new and currently deployed, result in clean slate solutions being impractical. A practical solution should rely on a few, simple components and should be adoptable incrementally.

TrustGuard, the first implementation of the Containment Architecture with Verified Output (CAVO) model developed at Princeton, showed how a simple, trusted Sentry could protect against malicious or buggy hardware components to ensure integrity of external communications. This was accomplished by ensuring the correct execution of signed software, with support from a modified CPU and system architecture. However, TrustGuards practicality was limited due to its reliance on modified host hardware and its requirement to trust entire application stacks, including the operating system.

The work presented in this dissertation seeks to make the CAVO model a practical solution for ensuring the integrity of data communicated externally in an untrusted commodity system. This work extends CAVO in two ways. First, it makes the Sentry compatible with a wide range of devices without requiring hardware modifications to the host system, thus increasing its flexibility and ease of integration into existing environments. Second, it gives developers the option to use trusted code to verify the execution of untrusted code, thus reducing the size of the trusted code base. This is analogous to the small, trusted Sentry ensuring correctness of execution of a large amount of complex, untrusted hardware.

Acknowledgments

I would first like to thank my advisor, Professor David I. August, for his support and guidance of over the years. This dissertation would quite literally not exist, and I would not be the researcher I am today without him. David helped me grow from someone that loves to solve problems, to someone that looks for the broader context and motivation for what issues should be addressed and can to express to others why that issue matters. I have drawn great inspiration from his endless desire and enthusiasm to identify problems that genuinely impact the world, fix it, identify another problem, fix it, and repeat until its all fixed.

I sincerely appreciate the expedited effort my readers, Professor Andrew Appel and Dr. Jothy Rosenberg, have expended in providing feedback and guidance to this dissertation. Professor Appel has been especially critical to helping guide the formal sections of the dissertation. I would also like to thank Professor Aarti Gupta and Professor Arvind Narayanan for serving as examiners on my committee and providing early feedback to help strengthen my dissertation.

This dissertation would not have been possible, and my time in graduate school would have been much poorer overall, without the support and companionship of everyone in the Liberty Research Group. I thank all the members that welcomed and guided me in my early days, Arun Raman, Yun Zhang, Jialu Huang, Prakash Prabhu, Thomas B. Jablin, Hanjun Kim, Nick P. Johnson, and Feng Liu. I would especially like to thank Tom and Nick for their early guidance and mentorship. Surviving that first PLDI deadline while working with them on their papers was a true learn by doing moment to show me how to grad school. I thank my cohort of Taewook Oh, Soumyadeep Ghosh, and Matt Zoufaly for their companionship and support while surviving those early days together. The memories of late nights coding, writing papers, frantically trying to finish our theory homework, preparing for generals, and building energy drink pyramids are fond ones that I will always cherish. I also thank those that came after me, Jordan Fix, Heejin Ahn, Hansen Zhang, Nayana Prasad, and Sotiris Apostolakis, for listening to me drone on about this or that, and at least pretending that I was teaching them something. While I haven't gotten to know the newest members of the group, Ziyang Xu, Bhargav Reddy Godala, and Greg Chan, as well as I would have liked, I can tell that I am leaving the group in good hands.

I would especially like to thank Deep and Jordan for blazing the early trail for this project. Their work on the Sentry and TrustGuard laid the groundwork for my contributions. I also thank Hansen and Sotiris for their ongoing collaboration. Hansen's hardware expertise and Sotiris assistance in the compiler have made the project not only better, but their friendship has made it more enjoyable. I extend additional thanks to Deep, Hansen, and Sotiris for their efforts in proofreading early drafts of this dissertation. I also extend a great deal of gratitude to Cole Schlesinger and Santiago Cuellar for their aid in proofreading the formal sections of this dissertation.

I thank the staff of the Computer Science department for their help through the years. I especially thank Melissa Lawson and Nicki Gotsis for all the aid they have lent as Graduate Coordinator; navigating the paperwork and timelines for generals, PreFPO, and FPO would have been impossible without them. I also extend a sincere thank you to Bob Dondero and Brian Kernighan for their guidance, leadership, and mentorship during my time as their preceptor and teaching assistant. Bob's relentless dedication and enthusiasm for teaching are inspiring; I hope to someday be as beloved by my students as he is by the students of Princeton. I still find it hard to believe that I have developed a friendship with *the* Brian Kernighan. His patience, kindness, and insight seem to know no bounds, and I consider myself lucky to have had the privilege to get to know him.

I would also like to acknowledge the generous funding for the CAVO Project, including: "SaTC: STARSS: An Architecture for Restoring Trust in Our Personal Computing Systems" (NSF Award #CNS-1441650); and the Princeton IP Accelerator Fund.

I don't think there are words that can express how thankful I am to my parents, Corrina and Robert Beard, for all they have done for me. I can never repay all the patience, compassion, humor, and goodness they have imprinted into me through the years just by the strength of their own personalities. Their unwavering love and confidence in me have pushed me to be a better person. I consider myself truly lucky to have such amazing parents.

Finally, I thank my amazing, brilliant, beautiful partner, Kahye Song for her companionship and support during these last few years. Her enthusiasm for life and adventure have quite literally taken me to new places and shown me things I would have never encountered on my own. I know that if we can survive our final years in graduate school together, nothing can ever break our bond.

To my loving and supportive family.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Cars, homes, hospitals, banks, credit bureaus, utilities, government services, and defense systems are increasingly computerized and networked. However, computing devices in all of these areas are vulnerable to attacks that can lead to financial losses [4, 34], damage to enterprise assets [6, 45], operational disruption [1, 36], industrial and military espionage [2, 11], and even physical harm to people and their environment [3, 5, 34]. In most cases, designers, manufacturers, and developers do not possess an effective and practical way to secure their products. Thus, security and privacy breaches continue to be daily news.

System engineers and software developers have traditionally accepted security to be a never-ending game of cat-and-mouse between attackers and defenders. Under this model, vulnerabilities often first come to light when they are exploited by attackers. System designers and developers then fix these vulnerabilities and issue patches to protect those systems from that attack in the future. While patching software can be expensive, annoying, and disruptive (even when done correctly, which is often not the case), at least software is patchable. Even when vulnerabilities are fixed, the time between the exploitation of a vulnerability, its disclosure, and its patching can result in huge losses [47].

The story is different for hardware, where only a subset of vulnerabilities can be repaired through mechanisms such as microcode patches or firmware updates [8]. Hardware designers use a combination of formal verification and testing techniques to try to ensure correctness of hardware at various points in the design, manufacturing, and post-production phases [60, 64, 70, 109]. However, the complexity of modern hardware design has simply outpaced our ability to ensure their correctness, as demonstrated by the recent Spectre [69] and Meltdown [79] vulnerabilities.

An alternate approach to system design, called “clean slate design,” is gaining increasing traction in the research community. The idea of “clean slate” is to build computing systems using hardware and software components whose designs are formally proven secure. While formal methods have made progress over the years in proving security properties of software and hardware designs, they require a high level of specialized expertise and a significantly extended development period. Though there has been a great deal of progress in the formal verification of systems, especially RISC-V based systems [32], these methods currently do not scale to the large and complex designs of modern computing systems [65, 66, 68, 72, 84] However, even if this approach matures enough to handle the complexity of modern system designs, one cannot expect companies to throw away decades of established knowledge and infrastructure. Furthermore, economic and logistic issues make secure production and delivery of every component infeasible.

Acknowledging that modern computing hardware and software are too complex to be vulnerability-free, others have proposed a minimal Trusted Computing Base (TCB) approach. A TCB is composed of the minimal set of hardware, firmware, and/or software components that are critical to the security of a computing device [73, 94]. By using the small TCB to ensure important security properties, developers and designers can spend the necessary time and effort needed to properly secure them as ensuring the security of the TCB ensures the security of the entire system.

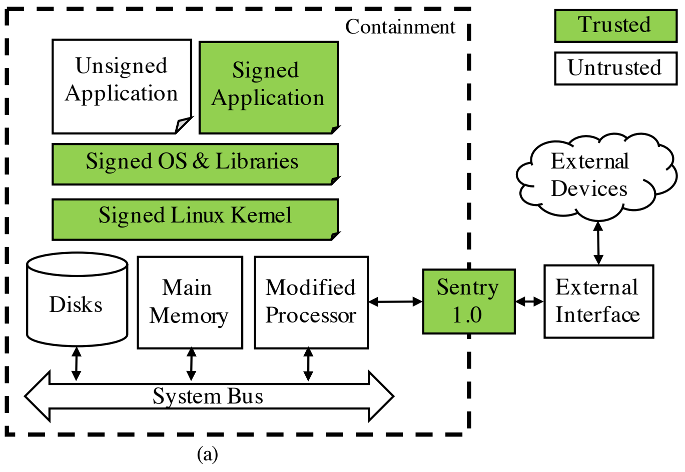
Recent attempts at creating real TCBs include processor based secure enclaves such as Intel’s SGX [61] and ARM’s TrustZone [16]. In these systems, a piece of secure code executes in an isolate environment inside the enclave. The enclave code is typically responsible for executing a security critical task (such as decrypting sensitive data, performing some operation on it, then re-encrypting it) while the hardware ensures the integrity and confidentiality of its execution. While secure enclaves can be an effective tool for some particular types of tasks, they still require placing trust in a complex processor (made even more complex by the enclave) that cannot yet be properly secured. Furthermore, the sensitive data cannot be used unencrypted outside the enclave without putting it at risk. Given that users ultimately need to interact with unencrypted data, they are left with little option to do so securely (§2.2).

1.1 Prior Work: TrustGuard

Rather than attempting to integrate the security components into the complex system as in processor-based secure enclaves, prior work promoted the approach of *containing* the effects of compromised hardware components within the system using a simple security component [54, 120]. Trusting a small, simple component for containment avoids the impractical task of securing complex hardware directly. The TrustGuard architecture, seen in Figure 1.1(a), was built using this approach.

Security and privacy assurances in TrustGuard are founded on a pluggable and simple hardware element, called the *Sentry*. The key to TrustGuard lies in a physical gap between the system and its external interfaces through which all external communications must pass. The gap is bridged by the Sentry, a hardware component designed to ensure that all output from the system is the result of correct execution of signed software. While containment by the Sentry does not provide availability guarantees (for instance, a processor may fail or halt as the result of an attack), it assures users that any output of the system is only the result of the correct execution of trusted software, not the result of errors or malice in hardware or interference by other software on the system.

TrustGuard utilizes a combination of instruction checking and cryptographic data in-



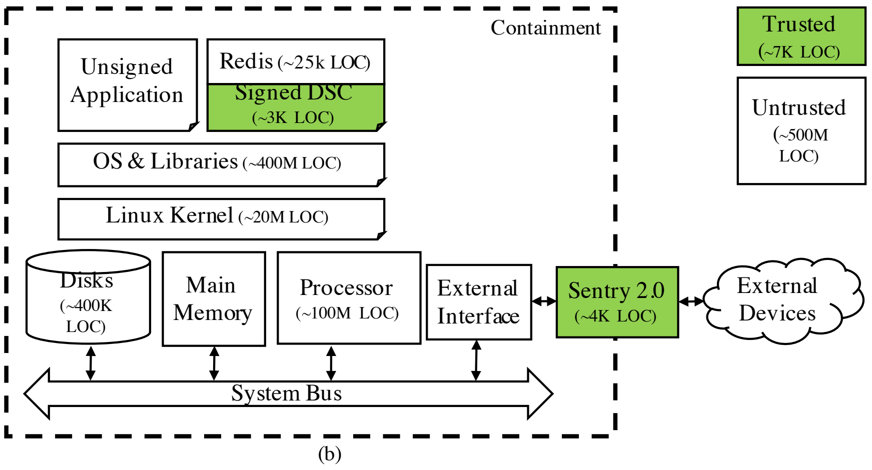


Figure 1.1: (a) Prior Work: TrustGuard. A Sentry, with help from a modified CPU, prevents erroneous (malicious or not) actions of untrusted hardware components from leaving the system through containment. (b) This Dissertation: CAVO. CAVO expands containment to the whole system, both hardware and software, and can protect existing systems. Proposed advances enable less than 10K LoC to protect more than 500M LoC.

tegrity assurance (via a Merkle Tree) to ensure correctness of output with respect to the specifications of the host instruction set architecture (ISA). TrustGuard consists of a conventional processor with modifications to communicate execution information to the Sentry, the Sentry itself, and the interface between these two chips. The goals of prior research were to investigate the feasibility of having a pluggable Sentry, whose design is optimized for simplicity and security, provide system containment for a modern complex system without impacting its performance. This goal was achieved by leveraging the idea that the untrusted system can perform much of the dynamic verification work for the Sentry without compromising security.

The Sentry's simplicity and pluggability make it tractable for suppliers and consumers to take additional measures to secure it using approaches such as formal verification, supervised manufacture, and supply chain diversification. Significantly, a simple Sentry running at a fraction of the clock frequency of the untrusted processor had only an 8% impact on system performance and energy consumption [54, 120]. These properties of the Sentry make the TrustGuard approach feasible for use in systems designed with Sentry support from the start (§2.3).

1.2 This Dissertation: CAVO

This dissertation generalizes TrustGuard's concept of containment to a broader, more practical, and effective *Containment Architecture with Verified Output (CAVO)* model. CAVO addresses two critical limitations of TrustGuard. First, CAVO is machine-architecture independent and thus is able to protect *existing* commodity systems. The CAVO model frees the Sentry from direct coupling to the customized CPU in TrustGuard, replacing it with efficient run-time support in software, thus removing the dependence on customized hardware. Second, CAVO generalizes the hardware protection from TrustGuard to provide protection for applications, thus enabling full system containment. In doing so, CAVO reduces the trusted code base from the entire application stack in TrustGuard to small per application *Dynamic Specification Checks* (DSCs). To encourage adoption, CAVO must be flexible and cause as little disruption as possible for all involved actors, including hardware designers and manufacturers, software developers, and end users. Several key ideas have improved the coverage and effectiveness of the CAVO model, as shown in Figure 1.1(b) and described below.

The CAVO model is designed to provide practical security benefits through containment for existing systems. For CAVO to be machine-architecture independent, the Sentry must not depend upon the custom processor and architecture in TrustGuard. To decouple the Sentry, this dissertation proposes moving the modified processor functionality that supports the Sentry into software. Removing the need for the modified processor allows the Sentry ISA to be independent of the host system, thus allowing the Sentry to have a simple RISC ISA but still be able to protect systems with complex performance and legacy focused ISAs, such as the ubiquitous x86-64 ISA.

In the same way that the Sentry serves as a small, trusted verifier for a large amount of untrusted hardware, CAVO provides developers a way to create a small, trusted DSC to serve as the verifier for a large amount of application and system code. By combining the software protection provided by DSCs with the hardware protections provided by the Sentry, the CAVO model can provide an efficient and true bottom-up approach to device security, starting from correct instruction execution and going all the way up to overall system behavior.

DSCs dynamically validate that the results produced by untrusted code adhere to some specification or policy (§3.2.2). This allows developers to reduce the amount of trusted code from all software (including applications, libraries, OS, and kernel) to only the DSC, which makes formal verification of the security critical components much more tractable. This dissertation presents a programming model for CAVO that allows developers to easily create small DSCs for their systems. For example, the prototype key-value DSC presented in Chapter 6.1 was written in just $\sim$3K lines of code (LoC) using the CAVO programming model. When combined with the Sentry's $\sim$4K lines of RTL, CAVO protects a database system containing over 500 million LoC using only $\sim$7k trusted LoC. Additionally, Sentry verification is a one-time cost as its design is independent of applications, not just host processor type. DSCs are ideally designed to protect a general protocol or class of application, and thus could be reused for many different applications and implementations.

The CAVO model is composed of several flexible and exchangeable components. To allow for environments with different levels of trust in components. Recognizing that some users may not require the extra security offered by the hardware Sentry, the CAVO architecture allows for multiple configurations to suit the users' desired level of security. These configurations could range from very high security (e.g., a military application where only the Sentry is trusted) to lower security (e.g., a web server where a trusted machine performs validation). Furthermore, given that developers need the freedom to prioritize their resources to address security concerns, the programming model is flexible enough to support a wide range of DSC types (§3.2.2). Ideally, the DSC is a simple verifier that ensures that all communication from an application is correct, such as in the proof-of-concept DSC for Redis, a commercial grade key-value store (§6.1). For some applications, developers may be more concerned with enforcing certain security policies, such as preventing data loss or loss of cyber-physical control, with high assurance and low effort.

Finally, this dissertation presents a complete implementation of one prototype instantiation of the CAVO model. This implementation includes: a library for the C programming language that enables DSC creation; a modified toolchain that takes as input a Sentry C program and outputs a native binary instrumented to communicate with a Sentry; a Sentry Control runtime that manages the Sentry (analogous to the prior modified CPU functionality); and finally an evaluation of the system using a prototype implementation of the Sentry on an FPGA card, implemented by Hansen Zhang [119]. Together, these components enable a Sentry protected program to run on a commodity system, with the added Sentry FPGA card. This prototype implementation was designed to show the capabilities of CAVO rather than to test its performance limits. For example, the Sentry implementation is a simple, single pipeline version of the Sentry implemented on an FPGA running at 100 MHz, rather than the high performance, multi-instruction checking pipeline, fabricated chip running at 1 GHz as envisioned in TrustGuard [54, 120].

In summary, the contributions of this dissertation are:

- Design of the Containment Architecture with Verified Output (CAVO) model, which provides the flexibility to suit a desired effort-to-security trade-off.

- CAVO programming model that allows for the creation of Design Specification Checks (DSCs), which enable a small piece of trusted code to provide full system containment.
- A generalization of the Sentry design that enables it to protect existing commodity systems and support Design Specification Checks.
- The first complete design and implementation of an instantiation of the CAVO model, comprised of:
  - Sentry library and programming model for C to produce DSCs and Sentry protected programs.
  - Sentry software toolchain that takes a Sentry program and produces a native binary.
  - A software implementation of the Sentry Control.
  - Enhanced design of the Sentry to handle untrusted values and IO.
- A formal model of the untrusted and trusted components.
- A proof that the trusted Sentry will indeed accept results from a correctly implemented untrusted host.
- Validation of the CAVO Model through evaluation of a DSC designed to protect the Redis key-value store on a Prototype FPGA Sentry Implementation.

1.3 Building CAVO

This dissertation presents the first complete CAVO design and implementation. It builds heavily on prior work and current collaboration with other members of the Liberty Research Group. The Sentry from TrustGuard [54, 120] is the basis on which the entire CAVO model rests. A patent application for this work also exists [19]. I contributed to the original design of the Sentry, including being the primary designer of the Sentry's interactions with the system through program loading and peripheral interaction.

The prototype in this presentation relies on the prototype implementation of the Sentry on a NetSume FPGA card. The design of which was jointly developed with but fully implemented by Hansen Zhang [119], who also developed the parts of the Sentry Control responsible for managing the Merkle Tree and interfacing with the Sentry. Finally, Sotiris Apostolakis implemented the original Redis DSC, which I used as the basis for the Sentry Programming Model version in this thesis, and the pragma parsing front end for the Sentry Compiler. A patent application for the expanded CAVO model also exists [18].

1.4 Dissertation Organization

The rest of this dissertation is organized as follows: Chapter 2 discusses background information that motivates the need for the CAVO model. Chapter 3 presents the threat model and overall design of the CAVO model, including example DSC applications. Chapter 4 presents the details of the first full implementation of CAVO. Chapter 5 presents a simplified DSC language semantics. Chapter 6 evaluates the CAVO model using a DSC for the Redis key-value store. Chapter 7 discusses other related work. Finally, Chapter 8 concludes with a discussion of future research areas for the CAVO model.

Chapter 2

Background and Motivation

Modern computer systems are built upon a layered architecture, typically abstracted as software (including applications, libraries, operating systems, and hypervisors) comprising of the top layers and hardware comprising the bottom layers. Upper layers often depend on multiple layers below them for functionality. This means that the security of the upper layers also depends upon the security of the lower layers. Dyer et al. note: "Applications cannot be more secure than the kernel functions they call, and the operating system cannot be more secure than the hardware that executes its commands." [43] Thus, trust in a system must ultimately start from the hardware and be built upon in software.

Given the difficulties in securing hardware (§7), many have take the approach of using a minimal trusted hardware base to establish security in the system. The rest of this section discusses three hardware TCB approaches: attestation, enclaves, and containment.

2.1 Trust through Attestation

One of the earliest forms of a hardware root of trust came through attestation [51], with many modern systems implementing both local attestation, sometimes referred to as secured boot, and remote attestation [111], for establishing the identity of remote hosts in a network environment. The threat model for attestation assumes that there is a secure and trusted set of hardware and software that can be used to determine the level of trust in a remote system. The hardware module typically contains signatures of the trusted hardware, firmware, bios, bootloader, operating system, and applications. The hardware module can then watch the boot process and ensure that only legitimate versions of these components are used to establish the system.

While attestation can be useful, especially in relatively simple embedded and controlled environments, it is simply not sufficient for establishing security in today's complex commodity systems. Attestation can only prove that a particular piece of hardware or code is in use, it cannot prove anything about their security or runtime behavior. Thus, trust must be established through some other method.

Ideally, trust in these components would be based on formal guarantees of their correctness and security properties. Formal methods have made great strides and shown much promise in both hardware and software. For example, recent hardware efforts have proven simple in-order processors correct [81, 106]. Recent software efforts have formally verified a C compiler [75], a simple operating system kernel [58, 67, 84], a deterministic random bit generator [118], and an HMAC algorithm used for TLS [49]. Unfortunately, they are still not widely adopted due to a variety of reasons, many of which revolve around the size and complexity of modern code bases (§7). Note the change in perspective from 1989:

> The state of the art of computer security today is such that reasonably secure standalone operating systems can be built... [51]

to 2003:

> ...commodity operating systems are complex programs that often contain millions of lines of code, thus they inherently offer low assurance. Building simple, high-assurance applications on top of these operating systems is impossible... [50]

2.2 Trust through Enclaves

Recognizing that it is too difficult to ensure trust in the entire system, others use a trusted hardware module to build a secure enclave or trusted execution environment inside the system. The threat model for enclave based security assumes that the trusted enclave is secure, there is trusted code that will run within the enclave, and there is other potentially malicious hardware and software on the system that will try to extract information from the trusted code. Enclaves are typically used to secure some critical secret (such as cryptographic keys) or to ensure the integrity and privacy of sensitive data during operations on it in an untrusted environment (such as protecting medical records in a cloud environment). Enclave approaches can appear as either security functionality provided by the processor (such as Intel's SGX [61] and ARM's TrustZone [16]) or as secure co-processors (such as the IBM 4758 [43], Trusted Platform Modules [10, 15], and Apple's Secure Enclave [12]). The first approach necessitates trusting a modern complex processor, a risky proposition given their complexity. This risk has been demonstrated by the recent Spectre [69] and Meltdown [79] vulnerabilities. In fact, some researchers have already found ways to apply Spectre to attack SGX enclaves [37]. Co-processors typically come with a performance overhead, as they are usually much slower than the main processor and require off-chip communication, and fixed capabilities (such as encryption/decryption, remote attestation, or secure key management).

While the enclave approach is designed to ensure integrity and privacy *inside* the enclave, it cannot provide any assurances for anything that occurs outside the enclave. In current approaches, the trusted enclave is embedded within and surrounded by the untrusted components. Thus, the privacy and integrity of any sensitive data that leaves the enclave in an unencrypted state is at risk. Among other things, this means that the user cannot input or view sensitive data without it being exposed to the untrusted parts of the system. Thus, the enclave model is primarily useful for ensuring the privacy of data that never needs to directly interact with the user, such as encryption keys, and for ensuring the integrity and privacy of computation in a shared environment, such as in the cloud.

2.3 Trust through Containment

In contrast to the enclave approach, prior work presented TrustGuard [54, 120]. The threat model for TrustGuard assumes the following: there is a trusted hardware element, called the Sentry; trusted software; the only communication path off of the system passes through the Sentry; and that all other hardware and software in the system is potentially malicious. Thus, in TrustGuard, the Sentry *contains* the untrusted system by sitting between it and its external interfaces. Figure 2.1 shows the high-level design of the TrustGuard architecture. To gain access to the system's external interfaces, the untrusted system components must prove to the Sentry that any output results from the correct execution of signed software.

TrustGuard is a prototype CAVO system designed to test the feasibility of using a simple pluggable Sentry to verify the execution of an untrusted system. TrustGuard supports a uniprocessor system with trusted providers of signed software, including both the operating system and applications. The TrustGuard architecture includes a design of the only trusted hardware component in the system, called the *Sentry*, and supporting changes to the untrusted processor. TrustGuard demonstrated that a relatively simple and separately manufactured Sentry can validate the execution of a system with a fast, complex processor without a major impact to performance. Thus, TrustGuard proved the feasibility of the CAVO model during steady state operation.

In TrustGuard, the processor sends a trace of its committed instructions' results to the Sentry. The instruction checking unit in the Sentry re-executes the instructions using its own functional units to verify that the results produced were correct, as per the specifications for the instruction set architecture. Re-execution has the added benefit of protecting against design errors or transient faults in the untrusted system, thereby adding an extra layer of redundancy and reliability [20, 21, 22, 30, 56, 83, 98, 99, 102, 105, 115, 123].

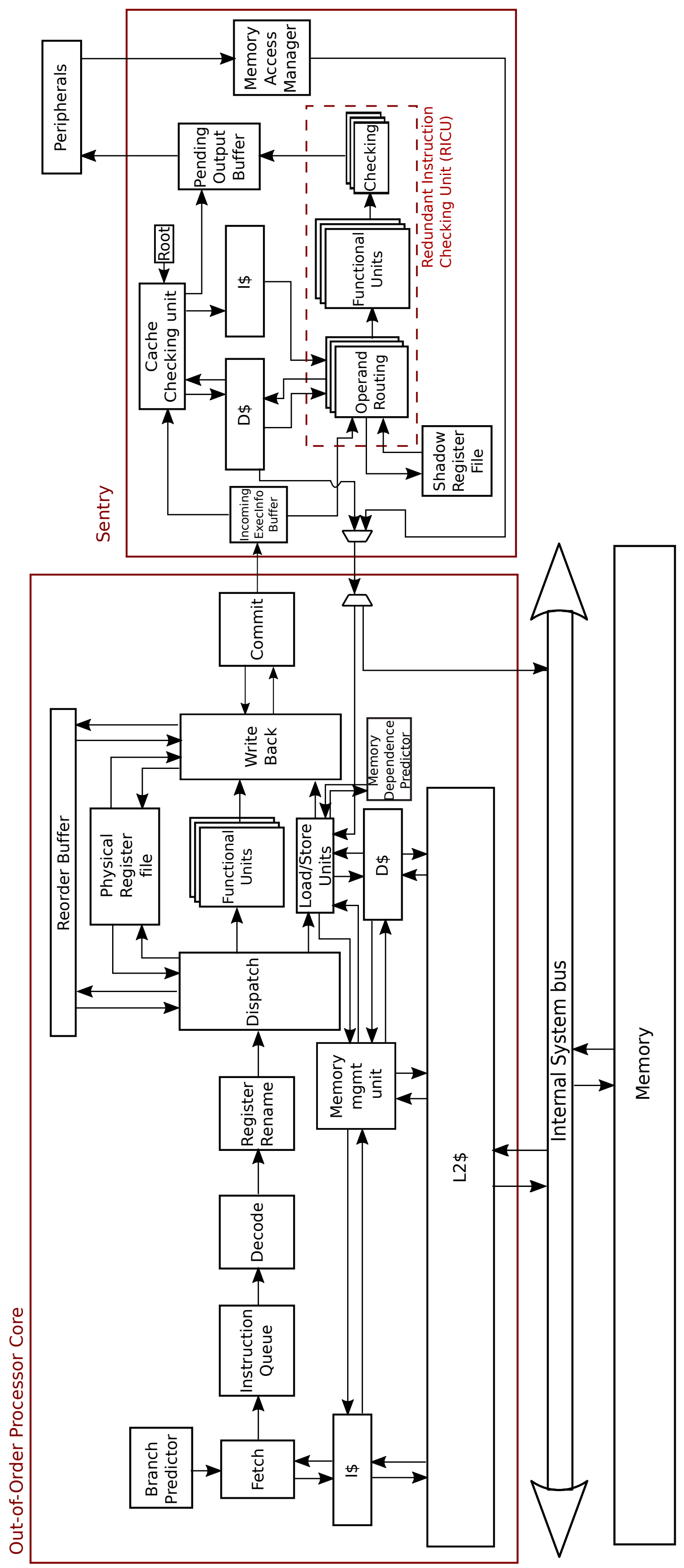


Figure 2.1: TrustGuard Architecture

Additionally, the Sentry uses a Bonsai Merkle Tree-based memory integrity scheme [101] to ensure that the data and instructions in memory that are sent by the processor are correct.

In TrustGuard, the untrusted processor is responsible both for sending execution trace information to the Sentry for verification (§2.3.1) and for managing the Sentry's instruction and data caches (§2.3.2). By pushing these responsibilities to the untrusted components, the complexity of the trusted Sentry's design is significantly reduced (§2.3.3). To ensure security, any values or control information sent by the untrusted system are validated before any external communication is allowed.

2.3.1 Parallel Redundant Instruction Checking

By speculatively assuming that the processor correctly executes and forwards results, the Sentry is able to break dependencies between instructions and validate multiple instructions in parallel. Thus, the Sentry can utilize older, extensively tested, but slower functional unit designs without materially impacting performance. This allows Sentry to be manufactured at trusted fabrication plants using several-generations-old technology.

The Redundant Instruction Checking Unit (RICU) in the Sentry consists of multiple pipelines, each with four stages. The first stage, *Instruction Read* (IR), retrieves the next set of instructions to be checked from the Sentry's instruction cache. The second stage, *Operand Routing* (OR), determines and forwards the operands to be used for redundant execution to the appropriate checking pipeline. The third stage, *Value Generation* (VG), re-executes the instructions using Sentry's functional units. Finally, the fourth stage, *Checking* (CH), compares the result of re-execution to the value sent by the processor stored in the *Incoming ExecInfo Buffer* to determine if the processor had reported the correct value. The Sentry maintains a private shadow register file, which contains all the register values corresponding to the verified instruction sequence, in order to facilitate checking.

2.3.2 Memory Validation

In order to verify instructions, the Sentry needs to validate both the instructions and data supplied by the untrusted processor. Rather than requiring a trusted copy of the entire memory state of the program, TrustGuard uses a variant of the Bonsai Merkle Tree-based cryptographic memory integrity scheme [101]. In this scheme, each data block (typically a cache line) is protected by a Message Authentication Code (MAC) created by a keyed cryptographic hash of the block, the address of the block, and a counter. The counter represents the version of the block, and is incremented any time a new MAC is generated. A tree of MACs is built over the counters to protect the integrity of the entire memory space. To ensure security, the cryptographic key and root of the tree are kept only on the Sentry.

To validate memory integrity upon loading a data block and its MAC, the MAC for the block is recalculated (using the data value, address, and counter value) and compared against the loaded value. A mismatch indicates an integrity violation in either the data or the counter. To ensure integrity of the counter, the chain of MAC values to the root is validated.

To reduce the amount of data communicated and number of validations required to ensure memory integrity, prior work has proposed caching Merkle tree nodes in trusted caches [101, 107]. When performing a load, only Merkle tree nodes that are not in the trusted cache need to be validated. Further, MAC values and counters only need to be updated upon eviction from the cache, rather than upon store. Thus, the Sentry contains a mirror of the processor's L1 cache. To reduce validation overhead, the Sentry contains an independent cache checking unit. Upon receiving new cache values from the processor, the Sentry speculatively assumes they are correct and proceeds with instruction validation while the cache checking unit validates the memory integrity. The Sentry's cache contains an extra flag to indicate if the cache line has been validated. Results are released from the Pending Output Buffer only when the instruction that generated it has been validated by both the cache checking unit and RICU.

2.3.3 Simplicity of the Sentry Design

The Sentry in CAVO must be *simple* and *pluggable* to ensure its trustworthiness through all stages of its creation, from design to manufacturing to deployment. To keep the Sentry simple and pluggable, TrustGuard enhances the processor to provide the Sentry with sufficient information to reduce the difficulty in performing validation. This allows the Sentry to be manufactured at trusted fabrication plants using several generations old technology. The information sent by the processor is validated before being relied upon and thus does not compromise security and privacy.

The design of the TrustGuard Sentry is comparable in complexity to various simple in-order pipelined processor designs that have been formally verified previously [81, 106]. Additionally, as can be seen in Figure 2.1, the Sentry lacks a number of components that are typically present in an out-of-order superscalar processor, such as: branch predictor, register renaming unit, reorder buffers, L2 cache, instruction queue, dispatch unit, load/-store queues, memory dependence predictor, and inter-stage forwarding logic. Since the Sentry exclusively relies on information sent to it by the processor, including loaded cache lines which are cryptographically protected and verified, it also does not need a memory controller. Even some of the components on the Sentry that bear a similarity to components on the untrusted processor are much smaller and simpler. Furthermore, a slower Sentry can protect against incorrect program output in a system with a faster processor (§2.3.4). The functional units on the Sentry can utilize older, extensively tested but slower functional unit designs. Combined with the pluggability of the Sentry, this allows the Sentry to be manufactured in a separate, trusted supply chain at closely controlled, domestic fabrication plants.

2.3.4 TrustGuard Performance

TrustGuard [54, 120] was modeled in the gem5 simulator using an out-of-order (OoO) ARM-core based untrusted processor to perform the analysis. TrustGuard evaluated the Sentry's effect on the processors performance during steady state operation. Performance was evaluated against 8 SPEC INT2006 and three SPEC FP2006 workloads. Figures 2.2−2.5 reproduced with permission from Ghosh [54, 120].

TrustGuard reported performance degradation from three sources. First, from an increase in cache and memory pressure from the Merkle tree accesses. The untrusted processor performing the Merkle tree operations, without the Sentry, resulted in a geomean IPC decline of 5.8% as seen in Figure 2.2. This comes from an average 91.0% increase in the number of L1 cache misses and an average 55.7% more memory accesses.

The second source is from bandwidth stalls where the dedicated channel between the processor and Sentry becomes saturated and the Sentry must wait for execution information. When varying the bandwidth from 5, 10, and 15 GB/s TrustGuard reported a geomean IPC decline was 21.1%, 8.5%, and 7.5% respectively, as seen in Figure 2.3. The majority of the communication from the processor and Sentry comes from the Merkle tree operations. Given that the processor only needs to send information on cache misses, programs with good cache locality need significantly less bandwidth than those with poor locality. For example, 445.gobmk had an L1 data cache hit rate of 66.6% and had bandwidth stalls for 20.3% of execution cycles while 456.hmmer had a cache hit rate of 99.1% and had bandwidth stalls in only 0.0064% of execution cycles. All future experiments were performed at a bandwidth of 10GB/s.

The final source is from the Sentry itself creating stalls by failing to check instructions fast enough to keep up with the processor. TrustGuard reported two sets of experiments to evaluate the Sentry's performance. The first set of experiments varied the number of instruction checking pipelines (ICPs) on the Sentry, while clocking the Sentry at 500MHz ($^1/_4$th the clock frequency of the untrusted processor). As can be seen from Figure 2.4, as the number of ICPs on Sentry increased from 4 to 6 to 8, the untrusted processor experienced a geomean IPC decline of 36.81%, 18.99%, and 12.94% respectively on different SPEC benchmarks, compared to that on the untrusted processor without any TrustGuard modifications.

Finally, TrustGuard reported results from experiments that varied the Sentry's clock frequency, using 8 ICPs on the Sentry. The Sentry's throughput increased at higher frequencies, as shown in Figure 2.5. Compared to out-of-order baseline, geomean IPC reduction on different SPEC benchmarks was 40.01% at 250MHz, 12.94% at 500MHz, 10.46% at 750MHz and 8.77% at 1GHz. From this, TrustGuard showed that the Sentry can exist as a separate chip that runs significantly slower than the host CPU and still achieve security benefits with very little performance decline.

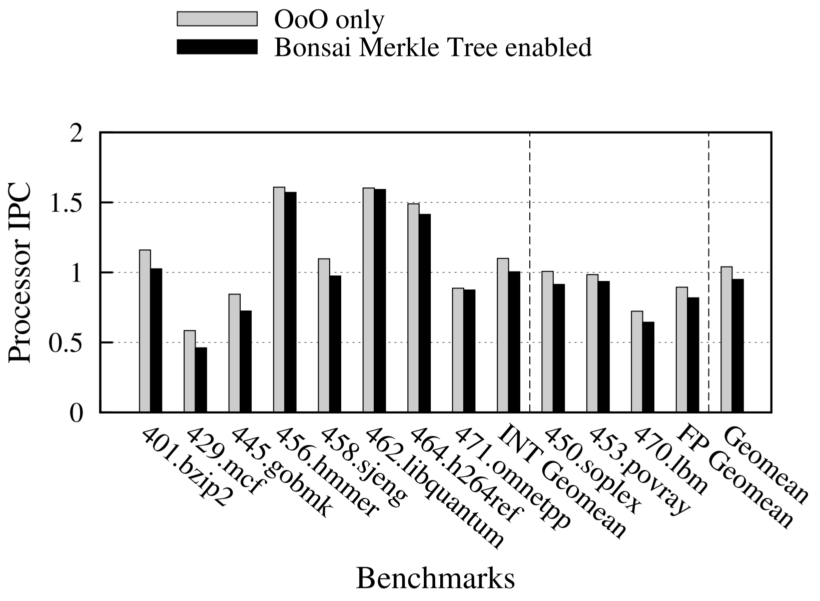


Figure 2.2: Effect of adding Merkle Tree accesses to untrusted processor

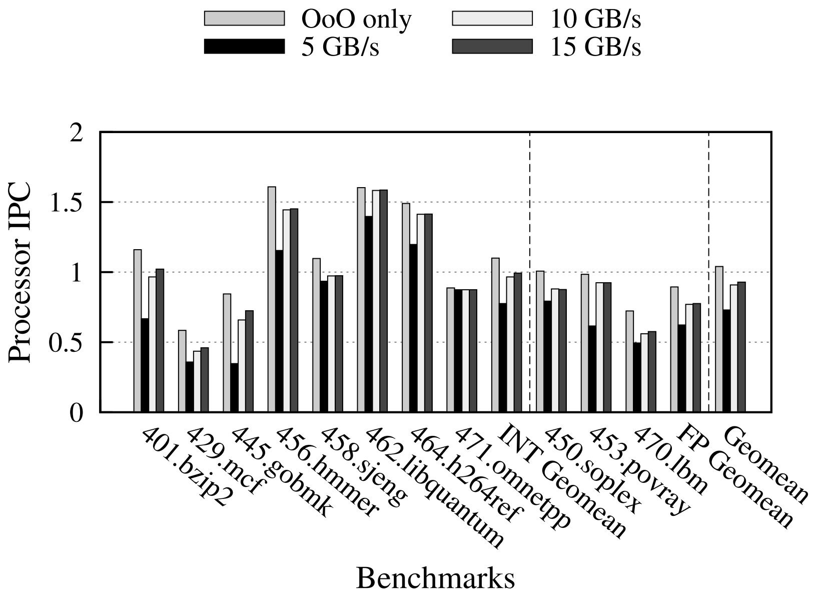


Figure 2.3: Parameter sweep through processor-Sentry bandwidth

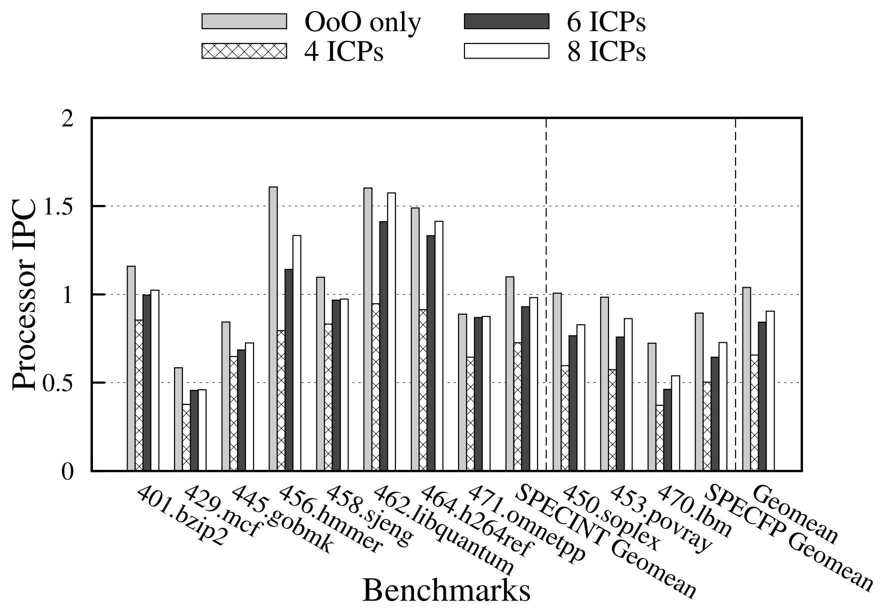


Figure 2.4: Parameter sweep through parallel checking pipelines

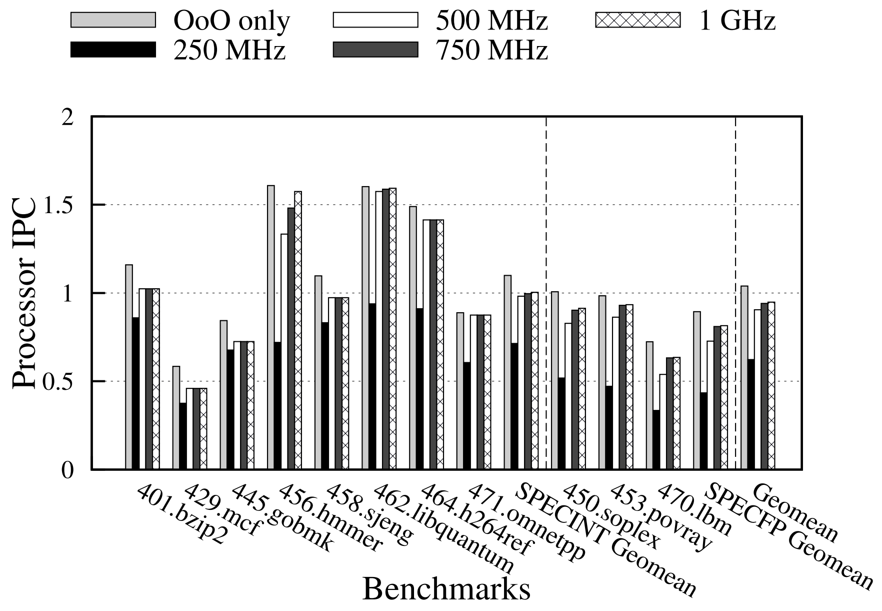


Figure 2.5: Parameter sweep through Sentry clock frequency

Chapter 3

Towards a Practical Containment Architecture

Security techniques must be practical to be adopted. To that end, this chapter presents an enhanced design for building Containment Architectures with Verified Output (CAVO) based systems. The design has been driven by the following guiding principles and key insights:

1. A trustworthy system can be created using untrustworthy hardware and software.
2. For important economic reasons, a practical solution *must* address existing systems by incorporating untrusted hardware and software components, representing decades of development and refinement.
3. Trust should only be placed in components that are simple enough to be verified.
4. Containment can protect against potentially damaging actions by allowing only trusted actions to have external effects.
5. Verifying the correctness of an application is often much easier than checking all of its computation.

The design has been made flexible and incrementally adoptable so that the containment model can ultimately be used in a variety of threat models, with different mixes of trusted and untrusted hardware and software components. For example, in some settings hardware or an OS provided by a trusted source may be secure enough to be trusted, while in some high security settings only components that can be formally verified are secure enough to be trusted. This thesis considers one of the most restrictive threat models, where only the Sentry and signed software are trusted, and notes where the design may be relaxed for less secure settings.

3.1 Threat Model

CAVO ensures that only correctly executed signed software is allowed to communicate externally. In this way, CAVO ensures that no errant or malicious process or hardware is able to communicate, nor affect the communication of signed software.

The only trusted hardware component in CAVO is the Sentry. All other hardware components (e.g. processor, memory, and disk) are considered untrusted. These untrusted components have the potential to produce incorrect results, either through flaws in their design or malicious tampering at any point during the design, manufacturing, or deployment of the components. While the Sentry ensures integrity of execution that leads to communication, it does not provide availability guarantees. CAVO requires that all communication channels out of the system pass through the Sentry.

Any software that is signed as trusted for verification by the Sentry is considered trusted and will be able to communicate out of the system. Establishing trust in the signed software would ideally be done through formal methods, but is ultimately the responsibility of the software developer. CAVO will ensure that the software is executed faithfully to its specification. Unsigned and untrusted software is able to run on the untrusted processor. However, any effects of unsigned software must be explicitly verified by trusted software before being allowed to leave the system.

Adversaries are assumed to not be able to physically bypass or tamper with the Sentry. CAVO does not address leakage through the Sentry via covert or side channels (e.g. encoding information in energy usage, long duration timings, availability failures).

3.2 CAVO

The goal of CAVO is to provide users strong privacy and integrity guarantees for their system's communication. This guarantee relies only on a small hardware and software trusted computing base. This is accomplished by separating the system into trusted components (whose purpose is to ensure the security of the system) and untrusted components (whose purpose is performance and functionality). The work of the untrusted components is ultimately validated by trusted components before leaving the system. The trusted components are divided into hardware and software components. First, software *Dynamic Specification Checks* (DSCs) ensure that any communication that leaves the system adheres to the intention of the programmer. Next, the hardware Sentry ensures that DSCs are faithfully executed by the hardware.

While TrustGuard [54, 120] served as an excellent proof of concept for containment architectures, this thesis improves upon the design in two major ways. First, it presents a generalized framework that can be used to build CAVO systems without requiring substantial hardware modifications to the system under protection, such as the specialized processor and custom bus to interface with the Sentry required by TrustGuard. Second, it gives developers the option to use trusted code to verify the execution of untrusted software, thus enabling a reduction of the size of the trusted code base. This protection from untrusted software is analogous to the way that the small, trusted Sentry can ensure correctness of execution of a large amount of complex, untrusted hardware.

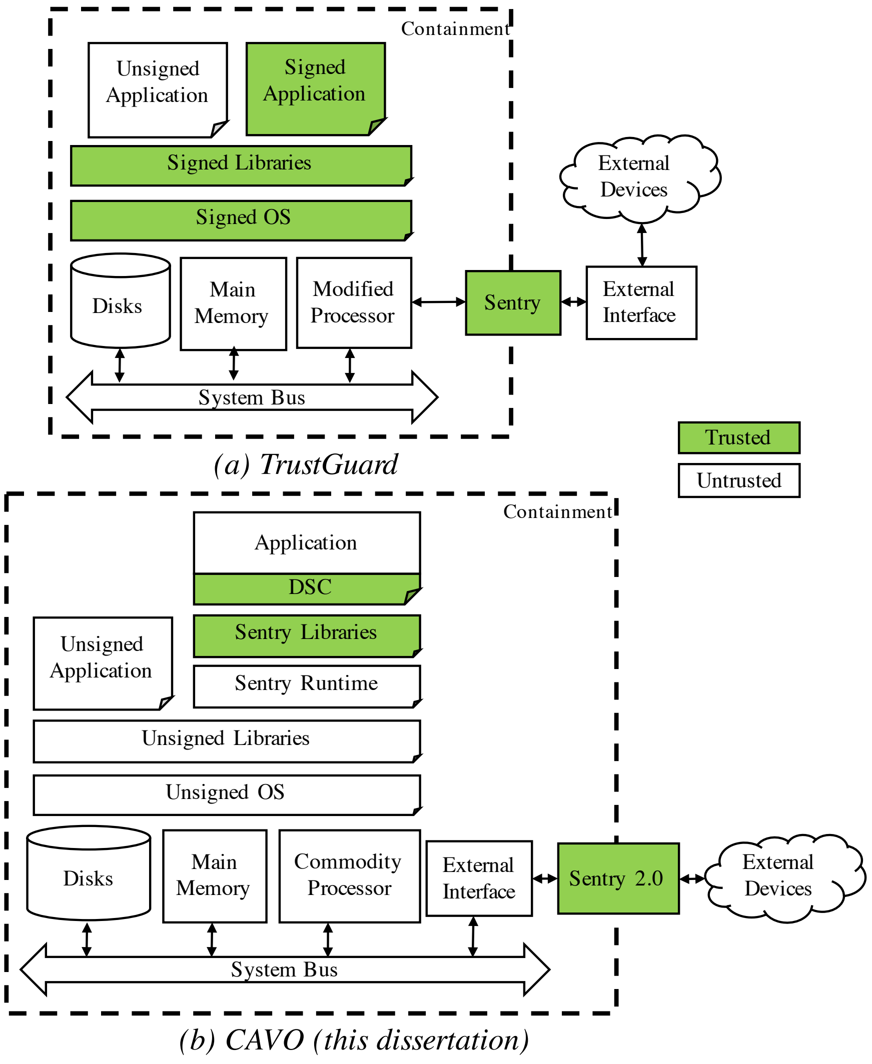


Figure 3.1: (a) TrustGuard, and (b) This Thesis: The CAVO model, containment made suitable for commodity systems and reducing the amount of trusted software in the system.

3.2.1 CAVO Infrastructure

One key idea behind CAVO is that the Sentry-enabling functionality in the modified processor from TrustGuard can instead be performed by logic implemented in software. This is shown in Figure 3.1, where the supporting functionality provided by *Modified Processor* from (a) is replaced by software components in the *DSC* and the *Sentry Runtime* in (b). Eliminating the need for a custom processor to support the Sentry allows the Sentry to become independent of the host processor ISA and to protect commodity systems.

In TrustGuard, the *Modified Processor* was responsible for sending execution results to the Sentry, managing the Sentry's instruction and data caches, and performing the Merkle Tree operations. While a Sentry Runtime can be used to manage the caches and Merkle Tree, there are many ways to extract and forward execution results. Some potential infrastructures considered were: an emulator / virtual machine approach, where DSCs compiled for the native Sentry ISA (SISA) could execute inside the emulator and the emulator would automatically forward results to the Sentry; an interpreter based approach, where a modified script interpreter could execute trusted scripts and forward results to the Sentry; and finally a compiler based approach, where Sentry assembly could be translated for native execution execution by the host and instrumented to forward results to the Sentry.

This dissertation presents an infrastructure using the compiler approach because it has several key advantages. First, a compiler-based approach should suffer from less performance penalties than the other approaches. Next, many of the technical challenges involved in the compiler-based approach are similar to the challenges faced by the other approaches. Thus, creating a compiler-based infrastructure should make it significantly easier to create an emulator- or interpreter-based approach in the future. Finally, a compiler-based approach opens a path for future research into using classical or developing new optimizations that improve performance and security further. For example, reverse program slicing (which calculates the backwards trace from an output operation) could be used to reduce the overhead of the Sentry's validation by only checking instructions that affect output. Sections 4.2 and 4.3 discuss details of the software toolchain and runtime, respectively.

Finally, there is the integration of the Sentry with the commodity system, rather than the custom integrated Sentry from TrustGuard. The Sentry could exist in a number of potential form factors, such as a PCIe network card, USB "bump in the wire," or in rack/datacenter appliance. However, to appeal to lower security use cases, the verification element could be simply a trusted system or a virtual machine / hypervisor. The programming model and software toolchain should support any physical manifestation of the verification element. This dissertation makes use of a prototype Sentry on an FPGA PCIe network card [119] due to its ease of testing both the network card and appliance type manifestations.

3.2.2 Dynamic Specification Checks

The Sentry can ensure that only correctly executed signed software is allowed to communicate. However, the Sentry cannot verify that the signed software itself is correct, namely free of vulnerabilities and bugs. Thus, CAVO allows for custom checking functionality, either through libraries or custom written Design Specification Checks (DSCs), thereby reducing the trusted code base to just the DSC and Sentry libraries, rather than the OS, libraries, and whole applications as in TrustGuard.

Ideally, DSCs are small pieces of verification code that ensure a dynamic specification for an entire untrusted system. In such cases, the DSC dynamically validates all values generated by the untrusted system that are to be communicated externally. For example, a database DSC can ensure that all database requests are serviced with only correct responses by using authenticated data structures to check the integrity of all database operations [108].

In some cases, it may be more desirable for a DSC to enforce a particular security policy for the application rather than ensuring correctness. Since simple and flexible DSC creation is the key to its adoption, CAVO seeks to give developers flexibility in implementing DSCs to suit their needs. Policy enforcement also serves as a powerful tool to allow developers to gain customized security assurances, with relatively low effort, that is more flexible than the correctness specification of their programs.

DSCs facilitate formal verification by drastically reducing the size of the trusted code base, analogous to the way that the Sentry greatly reduced the amount of trusted hardware. Additionally, separating the trusted validation code from the untrusted, performance critical, application code allows the validation code to be written in a language that facilitates formal verification while the performance critical code can be written in language that focuses on performance.

Furthermore, DSCs allow for decreased runtime verification overhead. For example, it is much easier to verify that a sorting algorithm has executed correctly than it is to actually perform the sort. By verifying rather than re-executing, CAVO reduces the amount of validation work the Sentry must perform and also benefits from the additional assurances to program correctness through implementation diversity. Additionally, developers can write their programs such that the untrusted code performs additional work to reduce the amount of work done by the DSC to perform validation, analogous to the way that the untrusted processor in TrustGuard performed additional work to facilitate validation performed by the Sentry.

DSCs are flexible enough to enforce a wide range of specifications and policies. However, all DSCs must meet the following requirements:

- The correctness of the DSC in implementing its specification or policy must be ensured as it is critical to the security of the system. Thus, DSCs should be simple enough to be thoroughly tested or formally validated.
- A DSC must ensure that all values returned from untrusted execution are validated before being trusted by other trusted code that uses those values.
- Upon detecting a violation, a DSC must trigger an alert or take remedial action.

Example Applications

Table 3.1 summarizes various types of DSCs each with different trade-offs between levels of protection and developer effort. At the base level, a developer that simply wishes to treat their application as trusted can do so and still gain the security properties guaranteed by the Sentry, namely protection from malicious hardware and interference from untrusted applications. Similarly, programs that have already been formally verified, such as CompCert [75], Milawa [38], or FSCQ [31], can be signed and executed with the security guarantees offered by their verification and those of the Sentry. Below are some example areas where DSCs can be used to ensure correctness in programs or enforce security properties.

	Protection Type	Effort	Protection	Examples
1.	Sentry Only	None	Malicious HW; and Interference from malicious programs	Any Program
2.	Algorithmic Verification	Low	Protections from (1); and Ensure correct output of algorithm through verification	Sort; SAT; Gradient Descent; Euler's Method; Newton's Method
3.	Security Policy Verification	Low	Protections from (1); and Ensure security policy	Data Loss Prevention; Login Restrictions; Mechanical Protections for Physical Systems
4.	Specification Verification	Medium	Protections from (1); and Ensure correct output according to specification	Database Operations; Compiler Transformations; Circuit Equivalence Checking
5.	Full Program Verification	High	Protections from (1); and Ensure correct execution of entire program	CompCert; Milawa; FSCQ

Table 3.1: Various levels of protection offered by the Sentry depending upon the level of developer effort in DSR creation and program verification.

Algorithmic Verification There are many algorithms that are easier to validate than to execute. Developers may wish to protect such algorithms with a small DSC to ensure proper execution of a critical algorithm in their code. Such examples not only include straightforward examples such as sorting algorithms, but also for many mathematical algorithms. For example, it would be easy for a validator to execute a few extra rounds of an iterative algorithm, such as gradient descent or Newton's Method, to prove convergence.

Some algorithms may require some additional work on the part of the solver to facilitate efficient verification by a validator. Verified SAT solvers may be the most well known such algorithm [17, 121]. In verified SAT solvers, witnesses returned from an untrusted solver allow a trusted core to verify solutions. Thus, a developer may have the untrusted SAT solver code produce a witness ("unSAT core") to reduce the amount of work performed by the small DSC SAT validator. Verifiable computing [114], where an untrusted machine computes a result and a proof that the result is correct, is another example of this class of algorithm.

Policy Enforcement In some cases enforcing a security policy may be more desirable or feasible than enforcing a particular program specification, as noted by work on reference monitors [44]. For example, many companies are highly concerned with data loss prevention [80, 91, 116] and preventing conflicts of interest [29]. Thus, ensuring correct operation of a database may not be sufficient. For example, companies are highly concerned that their database server doesn't leak private data (such as passwords and credit card information) even if such an operation is correct with respect to the database specification. Thus, a policy based DSC for such a database might completely restrict the external release of certain rows or only allow their release under certain conditions.

Similarly, many physical devices are directed by computerized control systems. Such systems usually take in input from physical sensors, perform some computation, and output control signals to manipulate robotic devices. In such cases, it may be more desirable to have a DSC prevent certain failure conditions, such as spinning a centrifuge too quickly or accelerating a car to unsafe speeds, rather than validating computation on potentially imperfect input information.

Specification Verification Many server-client type applications have a clear specification for the operations of the server. For example, databases have a clear specification relating their input to their output. The first specification DSC that integrates with the Sentry validates execution from a commercial grade database server, Redis [97]. The DSC code validates database operations, similar to prior work in outsourced databases [42, 76, 77, 85, 86, 90, 117, 124]. A more detailed discussion of the Redis DSC is found in Chapter 6.1.

Additionally, there are other application types that have a clear and easily verifiable specification for the relationship between their input and output. One classic example is in translation validation, where the input and output should be functionally equivalent. For such transformations, it is typically easier to verify the equivalence of the input and output rather than try to determine the correctness of the transformation directly. Classic examples include compiler optimizations and register allocation [87, 100, 103]; circuit equivalence checking [26, 71]; and High-Level Synthesis tools, such as C-to-RTL translation [13].

3.3 First CAVO Prototype

For this thesis, I chose to create a CAVO prototype utilizing a compiler-based toolchain capable of producing programs consisting of untrusted native code and protected Sentry code. These programs are capable of communicating to an FPGA-based Sentry [119]. This system seemed to give maximum flexibility in not only showing the capability of the system, but also gave opportunity for future research into improving and optimizing the system.

Chapter 4

CAVO Prototype Implementation

This work introduces the first prototype implementation of a full CAVO system and relies upon an FPGA Sentry implementation developed by Hansen Zhang [119]. Figure 4.1 shows a simplified overview of the system. There are three main phases that will be discussed in more detail in this chapter. First is writing a program in the CAVO programming model (§4.1), which divides an application into trusted and untrusted code. In this model, an application starts in trusted code and can then invoke untrusted code to perform work on its behalf. Upon receiving a result from the untrusted code, the trusted code will verify the received results to ensure that what the untrusted code has produced is acceptable. Additionally, the model contains a library that allows the programmer to direct the Sentry to perform I/O operations and use select Sentry functionality.

The second component is the software toolchain (§4.2). The software toolchain takes as input both the trusted and untrusted code. The trusted code is sent through the CAVO Compiler, where it is compiled down to the Sentry Instruction Set Architecture (SISA). This is used to generate a trusted Sentry Binary, with matching signature, and an equivalent C program with instrumentation to forward the results of the untrusted execution of trusted code to the Sentry. These are then passed to the native Host Compiler, along with the untrusted code, to create an executable binary for the host system.

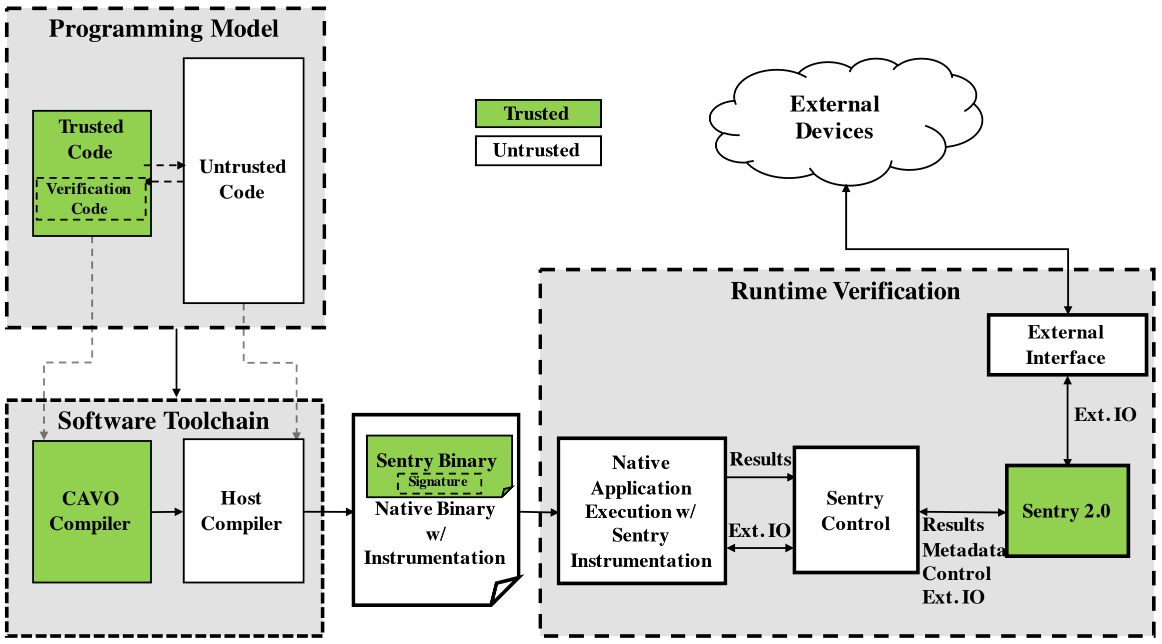


Figure 4.1: Simplified view of the first prototype implementation of a CAVO based system.

Runtime verification (§4.3) is the final major phase. At runtime, a Sentry Loader first loads the Sentry Binary into the Sentry's memory space and verifies that the binary correctly matches the signature. The program may then begin execution. During execution, the instrumentation for trusted code streams results to the Sentry Control as well as receiving the results of any external communication. The Sentry Control is responsible for managing all of the functionality previously handled by the untrusted processor in TrustGuard, namely managing the Sentry's cache, Merkle Tree data (including the MACs, block counters, and the tree of MACs §2.3.2), and streaming results to the Sentry. The Sentry Control can either be a daemon running on the untrusted machine, which is the current implementation, or as an untrusted hardware component. Finally, the Sentry itself validates instructions and communicates with the external interface.

In this model, only the programmer's trusted code, the CAVO compiler, and the Sentry are trusted. The CAVO compiler takes the programmer's trusted code and produces a Sentry Binary protected by a signature. This signature is checked by the Sentry during program loading (§4.3.1). Thus, any malicious modifications during either compile or runtime by any of the other components, including the native compiler or processor, will be detected, either through a signature mismatch on the Sentry Binary or as a runtime verification failure detected by the Sentry.

4.1 Programming Model

The goals of the CAVO Programming Model are to ease the creation of DSCs and enable trusted communication through the Sentry. For the purposes of this dissertation, the presented model uses C for both the trusted and untrusted languages, though any language is extendable in this way. Future implementations of CAVO may use a language whose programs can be formally verified more easily.

4.1.1 Supporting DSCs

As discussed in Section 3.2.2, one of the core ideas of CAVO is to allow a small, trusted DSC to validate the execution of a large amount of untrusted code. Thus, DSCs require the following mechanisms:

- demarcate trusted and untrusted code;
- call into untrusted code from trusted code;
- receive untrusted values from untrusted execution; and
- trigger an alert if validation fails.

The first step in the process of writing a CAVO-protected program is to determine what sections of code will be trusted and what sections will be untrusted. Program control must start in trusted code to give the program a trusted basis from which to call into untrusted code. Additionally, all external communication must originate in trusted code so that it may be validated by the Sentry. Untrusted code sections could be used to perform operations that do not lead to output, such as logging, and operations whose results can be validated. Since the execution of untrusted code is not directly validated by the Sentry, untrusted code

```
1  typedef struct sqrtVals{
2    double val;
3    double res;
4  } sqrtVals_t;
5  const size_t sz = sizeof(double);
6
7  int main(int argc, char *argv[]){
8    uint64_t sockfd;
9    double x, res;
10
11   TG_init_server(LST_PORT);
12   while(1){
13     TG_recvAny(&sockfd, &x, sz);
14     res = sqrt(x);
15     TG_send(sockfd, res, sz);
16   }
17   return 0;
18 }

20 #pragma TrustGuard(UNT_EXEC)
21 void sqrtUnt(void *arg);
22
23 void sqrtChk(sqrtVals_t *arg){
24   double x = arg->val;
25   TG_getUntVal(&(arg->res), sz);
26   if(fabs(x*x - *arg->res) > EPS){
27     TG_alert(-1);
28   }
29 }
30
31 double sqrt(double x){
32   sqrtVals_t arg;
33   arg.val = x;
34   sqrtUnt(&arg);
35   sqrtChk(&arg);
36   return arg.res;
37 }
```

Figure 4.2: Example of a square root server written with the CAVO programming model. The function `sqrtChk` is a DSC that validates that the value returned from untrusted computation is the square root of the argument by ensuring that the square of the argument less the returned value is within some epsilon value, rather than recomputing the square root calculation.

cannot directly communicate externally. DSCs must be written to validate the results from untrusted code before the results are used to generate external communication.

These concepts are demonstrated with an example Square Root Server, seen in Figure 4.2. In this example, the actual square root functionality is considered untrusted and performed by the host system, since square root is both a complicated and costly procedure. The untrusted square root operation is validated by a relatively simple sequence of multiplication, subtraction, and comparison. The results from the untrusted square root are validated by the DSC before being used by the rest of the program. Untrusted computations and DSCs for real programs will typically be more complex, but these simple examples serve as instructive proxies.

The Square Root Server in Figure 4.2 accepts incoming connections (line 11), receives a value (line 13), passes the value to untrusted code to compute its square root (line 34), uses a DSC to validate the result (line 35), signals an alert if the DSC fails (line 27), and sends the result back to the client (line 15). The networking components will be discussed further in the next section.

As the untrusted code is not executed by the Sentry, the programmer needs to indicate which functions call into untrusted code so that the toolchain will not generate such calls in the trusted code. This is accomplished by the `TrustGuard(UNT_EXEC)` pragma shown on line 20, which is used to indicate that the `sqrtUnt` function is untrusted. The pragma ensures that the untrusted function is called only on the host and is not executed by the Sentry, which is accomplished by emitting any calls to such functions with the `call.unt` pseudoinstruction.

The signature for untrusted calls, a void pointer argument and no return value, is shown on line 21. The argument to the function should be a user-defined structure, shown in this example on lines 1-4, that contains all arguments and return values. Packing the arguments and returns into a single structure simplifies the code emitted by the toolchain. Additionally, structuring the untrusted calls in this way easily allows for future extensions to execute the untrusted calls in a parallel thread. This signature was modeled after the signature used by `pthread_create`.

After returning from the untrusted call to `sqrtUnt`, the result from the untrusted code is stored in arg$\rightarrow$res. This value is currently untrusted, since it was generated by untrusted code, and not registered with the Sentry, since the Sentry has not yet seen this value. Thus, its first use in trusted code must be to register them with the Sentry using a DSC. Registering the values with the Sentry allows it to protect them from any future malicious or erroneous interference by the untrusted system, in the same way that the Sentry protects values generated by trusted code.

Lines 23-39 show the `sqrtChk` DSC example. Note that the signature of the DSC is not required to be the same as that of the untrusted calls, but untrusted values should only be passed by reference to the DSC to avoid a use of the value before registration with the Sentry.

Value registration is accomplished by the `TG_getUntVal` function. `TG_getUntVal` takes the address and length of an untrusted value in order to emit special instructions (`r.unt`) to send these values to the Sentry for registration. It is important to note that after this step, the values from untrusted execution are now part of the Sentry's integrity structure but have not yet been validated. Thus, the programmer should take great care to ensure that any such values do not escape from the DSC without proper validation.

After loading the untrusted values into the Sentry, the DSC should validate the values returned from the untrusted code in some way. The `sqrtChk` DSC ensures that the received result is indeed the square root of the argument `x` by ensuring that the result of subtracting the value provided by the untrusted code from the square of `x` is less than some epsilon threshold. If the result passes this check, it is considered correct and returned to the program. If it does not, then the special `TG_alert` function is called. `TG_alert` (which converts to the SISA instruction `alrt`) raises an alert in the same way that the Sentry alerts if program execution validation fails.

Currently, an alert will simply halt the program. Future versions of the Sentry may make this a user defined parameter, with potential actions including halting the program, continuing in a special error handler, or halting after some number of failures. Other DSCs may be able to take some level of remediation to correct or repair the result returned by the untrusted code. The semantics of the programming model are discussed further in Section 5.

4.1.2 Network Communication

A custom networking library is used to communicate through the Sentry due to its modified networking behavior. To simplify the Sentry, all communication occurs via explicit `get` and `put` communication instructions. To remove the nondeterminism inherent in communication and reduce the amount of communication flowing through the Sentry, the TCP/IP stack exists on the other side of the Sentry from the host machine. These design decisions are discussed in further detail in Section 4.3.3.

The library includes standard networking calls such as `connect`, `select`, `send`, `recv`, and `close`. Additionally, as all DSCs written thus far have been for server type programs, specialized functions have been written to make such server programs easier to write and more efficient. These include the calls seen in the example in Figure 4.2. `TG_init_server` initializes the server to listen for any incoming connections on the specified port. `TG_recvAny` accepts communication from any incoming communication and stores the descriptor in the first argument. Additionally, there is a `TG_recvRdy()` function that checks to see if any incoming data has been received and is waiting for processing.

4.2 Compiler Based Software Toolchain

The toolchain is responsible for converting the trusted and untrusted source code of an application into a native binary and the signature-protected Sentry binary. The native binary is augmented with instrumentation to send results from the untrusted execution of trusted code to the Sentry. Figure 4.3 shows the basic flow of the toolchain.

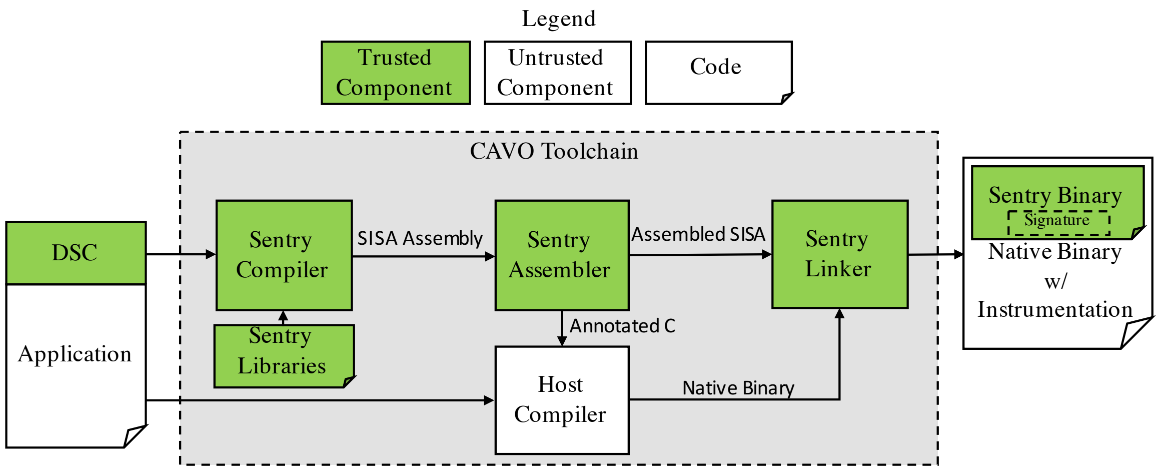


Figure 4.3: CAVO Software Toolchain that takes as input the trusted and untrusted program source and generates the instrumented native executable along with the signature-protected Sentry binary.

The developer first passes the trusted and untrusted source files into the toolchain. Then any trusted source files, including the DSCs and Sentry Libraries, are passed into the Sentry Compiler, which is a modified version of the RISCV LLVM compiler [9]. The trusted code is then compiled down to the native SISA assembly. SISA is a simplified version of RISCV, consisting of only the basic memory, arithmetic, logical, branch, and jump instructions. RISCV was chosen as the basis for the Sentry ISA due to its open design and ongoing work for a formal specification and verification. It is additionally augmented with the Sentry instructions `get`, `put`, `r.unt`, and `alrt` and the pseudoinstruction `call.unt` discussed previously.

The resulting assembly is then passed into the Sentry Assembler, an augmented version of RISCV GAS. The assembler produces two results. First, a functionally equivalent version of the program in C (further discussed in §4.2.1). This type of emulation is similar to the binary translation used in other projects, such as the PowerPC to x86 translator and the Itanium Execution Layer [24]. This C code is additionally augmented with instrumentation to send execution results to the Sentry and is passed to the system's native compiler to produce a native binary. The assembler also passes an assembled version of the program to the Sentry Linker.

Finally, the host compiler passes the native binary to the Sentry Linker. The linker then performs final linking on the assembled SISA to ensures that the addresses of any objects that are visible to both the trusted and untrusted pieces of the program are aligned. The linker then signs the resulting Sentry Binary with its private key to ensure that it is not tampered with and produces the final combined native and Sentry binaries. Note that the Host Compiler does not need to be trusted in this setup as all operations it performs are validated through the Sentry's re-execution of the trusted and protected Sentry Binary.

```
sqrt:
  ...
  call.unt sqrtUnt
  ...

sqrtChk:
  ...
  r.unt     ft1, 0(a1)
  fsd       ft1, 0(a1)
  ...
  fld       fa0, 0(a0)
  fmul      ft0, fa0, fa0
  ...
  blt       ft0, ft1, l1
l0:
  ...
```

(a)

```
sqrt:
  ...
  sqrtUnt(a0);
  ...

sqrtChk:
  ...
  ft1 = *(0+a1);
    sendToSentry(ft1);
  *(0+a1) = ft1;
    sendToSentry(ft1);
  ...
  fa0 = *(double *)(0+a0);
    sendToSentry(fa0);
  ft0 = fa0 * fa0;
    sendtoSentry(ft0);
  ...
  if(ft0 < ft1) {
    sendtoSentry(0);
    sendtoSentry(Sentry_l1_addr);
    goto l1;
  } else {
    sendtoSentry(1);
  }
l0:
  ...
```

(b)

Figure 4.4: Example showing the conversion of parts of the DSC from Figure 4.2 into SISA (a) and the equivalent C with instrumentation (b). Note that results are sent in (b) to break depenencies and enable parallel checking within the Sentry. SISA assembly instructions:
call.unt= call untrusted function;
r.unt= receive untrusted value;
fsd= floating point store double;
fld: floating point load double;
fmul= floating point multiply;
blt= branch less than

4.2.1 Sentry Emulation and Program Instrumentation

To enable the protection of commodity systems, the Sentry Assembler converts trusted Sentry code to an instrumented C version that can be compiled for native execution as well as an assembled SISA version. This involves three primary functions. First, inserting instrumentation to send results to the Sentry via the helper function `sendToSentry`. Second, converting SISA instructions into their equivalent C codes, so that the C code can then be compiled to a native binary. Third, untrusted calls are properly executed in the untrusted code but not executed by the Sentry. Figure 4.4 shows a snippet of the DSC from Figure 4.2 in SISA (a) and instrumented C (b).

Instruction conversion falls into two main categories for standard instructions: control flow changing instructions and all other instructions. To maintain its simplicity, the Sentry has very simple logic to gather instructions, with control flow changing instructions being handled specially. Control flow instructions send a special status bit to indicate if the jump was taken (0) or not taken (1). If the jump was taken, the jump target address is additionally sent [54, 120]. This can be seen in Figure 4.4, where the `blt` in (a) is converted into the `if/else` block in lines 18-24 of (b). All other normal instructions are converted in the logical manner to maintain their functionality. For example, the `fmul` instruction in line 12 (a) is converted into the multiply expression in line 15 (b).

The special pseudoinstruction `call.unt` indicates to the Sentry Assembler that the function call should be executed only by the untrusted host and not by the Sentry. Thus, the instruction is converted into a standard function call in the C version and is totally omitted from the assembled SISA version. This can be seen in Figure 4.4, where the `call.unt sqrtUnt` call in line 3 (a) becomes the simple function call `sqrtUnt(a0)` in line 3 (b). This also demonstrates how the requirement for all untrusted calls to take a single argument simplifies the C code generation of the Sentry Assembler, the function call generated by `call.unt` always has the single `a0` argument.

Of final note is the `r.unt` instruction, which is converted frmo the `TG_getUntVal` function. This instruction causes the untrusted host to load the value from the specified address into the specified register. However, as this value originated from untrusted execution, the Sentry simply accepts the incoming value from the *Incoming ExecInfo Buffer* and stores this value into the specified register. This value should then be validated by the DSC to ensure that an acceptable value has been sent by the untrusted host.

4.3 Runtime Verification

The final phase of CAVO is the execution and runtime verification of a protected program. Figure 4.5 shows the current implementation of the CAVO Runtime Verification system. In this system, both the application under protection and the Sentry Control run as separate processes in the commodity host system. The Sentry is instantiated on an FPGA on a PCIe network card[119]. A Linux system is also instantiated on the FPGA using a softcore system, i.e. the full system is implemented in FPGA logic. The softcore is used to control the external Ethernet interface and run the TCP/IP server that the Sentry communicates through.

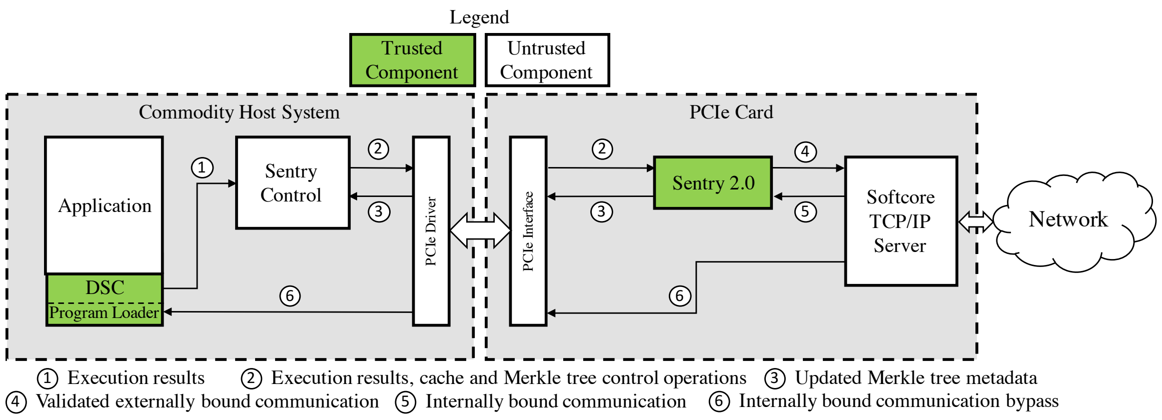

Legend
Trusted Component
Untrusted Component
Commodity Host System
PCIe Card
Application
DSC
Program Loader
Sentry Control
PCIe Driver
PCIe Interface
Sentry 2.0
Softcore TCP/IP Server
Network
① Execution results
② Execution results, cache and Merkle tree control operations
③ Updated Merkle tree metadata
④ Validated externally bound communication
⑤ Internally bound communication
⑥ Internally bound communication bypass

Figure 4.5: CAVO Runtime verification for a system with the Sentry on an FPGA PCIe network card.

When a program is launched, the Sentry Program Loader, discussed in more detail in the next section, first sends the Sentry Binary to the Sentry Control. Next, it loads the Sentry Binary into the Sentry's memory and Merkle tree and validates its signature. After successfully loading, the program then executes. The instrumentation inserted by the toolchain ensures that the result from every trusted instruction is sent to the Sentry Control process through the IPC channel ①.

The Sentry Control is responsible for delivering results to the Sentry and managing the Sentry's cache system. The Sentry Control delivers these to the Sentry through the PCIe driver ②. The Sentry Control also receives all results of cache evictions from the Sentry through the PCIe Driver ③. The Sentry itself validates all results received through its functional unit re-execution.

The Sentry sends the results of all successfully validated `put` instructions to the softcore through an internal FPGA buffer ④. The softcore accepts or creates TCP/IP connections according to the commands received through the Network Communication library discussed in Section 4.1.2. All inbound communication is sent via an internal FPGA buffer to the Sentry ⑤, where it waits until a `get` instruction captures it. Additionally, inbound communication is sent along the bypass channel ⑥ so that the application can execute `get` instructions locally without waiting for the Sentry. The results of `get` instructions are later validated against the value received by the Sentry, thus ensuring that the untrusted system cannot manipulate received values that it receives through the bypass.

4.3.1 Program Loader

The program loader is responsible for loading the trusted binary into the Sentry's memory and Merkle Tree, and validating the binary's signature. To load the binary into the Sentry's memory, the program loader itself must execute as trusted thus creating a bootstrapping problem. One possible solution is for the manufacturer of the Sentry to generate the Merkle Tree metadata for the program loader upon creating the Sentry's private key. The root of this tree will then be stored in a private static register on the Sentry. Upon initialization, the Sentry will load this value into the Merkle Tree root register. Additionally, the Merkle Tree

16 byte hash	8 byte addr	8 byte len	text section	8 byte count	8 byte addr	8 byte len	data	...

Figure 4.6: Encoding for Sentry Binary format

node values will be stored as part of the Sentry Control so that every program will start with the program loader already loaded into memory. The loader additionally contains the public key for the trusted software source.

The program loader will then read the Sentry Binary as untrusted data and store it into memory. The Sentry Binary format, shown in Figure 4.6, is designed to easily be looped over and read by the program loader. While loading the binary, the program loader also generates the hash of the binary. Upon fully loading the binary, the loader decrypts the 16-byte hash using the trusted source's public key and compares it with the generated hash value. If the values match, the loader jumps to the start address and begins program execution. If the values do not match, an alert is triggered. Once the program is validated and loaded into memory, it will be protected from erroneous or malicious modification by the Sentry's memory protection provided by the Merkle tree.

4.3.2 Sentry Control

The Sentry Control (SC) handles all communication with and control of the Sentry, functionality previously handled by the modified CPU in TrustGuard [54, 120]. The SC currently runs as a daemon process on the host machine and communicates to the application process via a Unix socket. Algorithm 1 shows the basic control algorithm for the Sentry Control.

The SC starts by sending a reset command to the Sentry that causes it to load the default root value into the Merkle Tree Root Register and to flush its cache. After receiving the program binary from the application, the SC goes into the main loop and waits to receive results. Upon receiving a result, the SC determines what instruction this result should have

Algorithm 1: Sentry Control algorithm

```
1  resetSentry()
2  prog = receiveProgram()
3  PC = 0
4  while not exited do
5      res = receiveResult()
6      inst = prog(PC)
7      if notCached(inst) then
8          update instruction cache line
9          update Merkle Tree data
10     if isLoad(inst) or isStore(inst) then
11         if notCached(res.addr) then
12             update res.addr cache line
13             update Merkle Tree data
14         else
15             send res.addr
16         PC++
17     else if isControl(inst) then
18         send res.jumpStat
19         if jumpTaken(res.jumpStat) then
20             send res.jumpDst
21             PC = res.jumpDst
22         else
23             PC++
24     else
25         send res.val
26         PC++
```

originated from. Next, it determines if this instruction is already in the Sentry's cache. If not, it sends the cache line for the instruction and the appropriate Merkle Tree data as well as the appropriate evict messages to ensure these lines are available.

If the instruction is a memory instruction, i.e. a load or a store, the SC determines if the accessed address is already in the cache. If so, it sends accessed address to the Sentry. If the address would be a cache miss, it sends the appropriate cache line for the address and associated Merkle Tree data as well as the evict messages to ensure these lines are cleared. All dirty cache lines must additionally have their Merkle Tree nodes updated. The

SC can direct the Sentry to perform these updates, but must wait for the Sentry to return the updated lines as the SC is untrusted and does not have access to the Sentry's private key. In both cases, the SC increments its PC index to the next instruction.

If the instruction is a control flow changing instruction, the SC first sends the jump status, taken or not taken. If the jump was taken, the destination is sent and the SC updates its PC to the new value. If the jump was not taken, the SC takes no special action. For all other instructions, the SC simply sends the result and increments its PC to the next instruction.

If the program exits normally, the SC exits out of the loop and waits for the next program to start. To prevent the SC from getting stuck in the case a program crashes, all applications send a special reset packet to the SC to force it to exit from the main loop, reset the Sentry, and receive the new Sentry Binary.

4.3.3 Sentry 2.0

Several enhancements and four new instructions have been added to the Sentry to enable it to protect commodity systems and support DSCs. Figure 4.7 shows an updated architectural diagram for the Sentry.

First, a special saved root (S. Root) static register has been added. This register holds the root of the Merkle Tree when the program loader is the only thing in memory. Upon powering on or receiving a reset signal, the S. Root register's contents are loaded into the working root register (W. Root) so that the program loader may load the application binary into the Sentry's memory and Merkle Tree.

`get rd` The get instruction is used to receive a value from the external softcore system. Communication from the softcore is held in the Sentry's Input Buffer. When a `get` instruction is executed, the next value in the Input Buffer is loaded into the destination register. The received value is compared against the value reported by the Host.

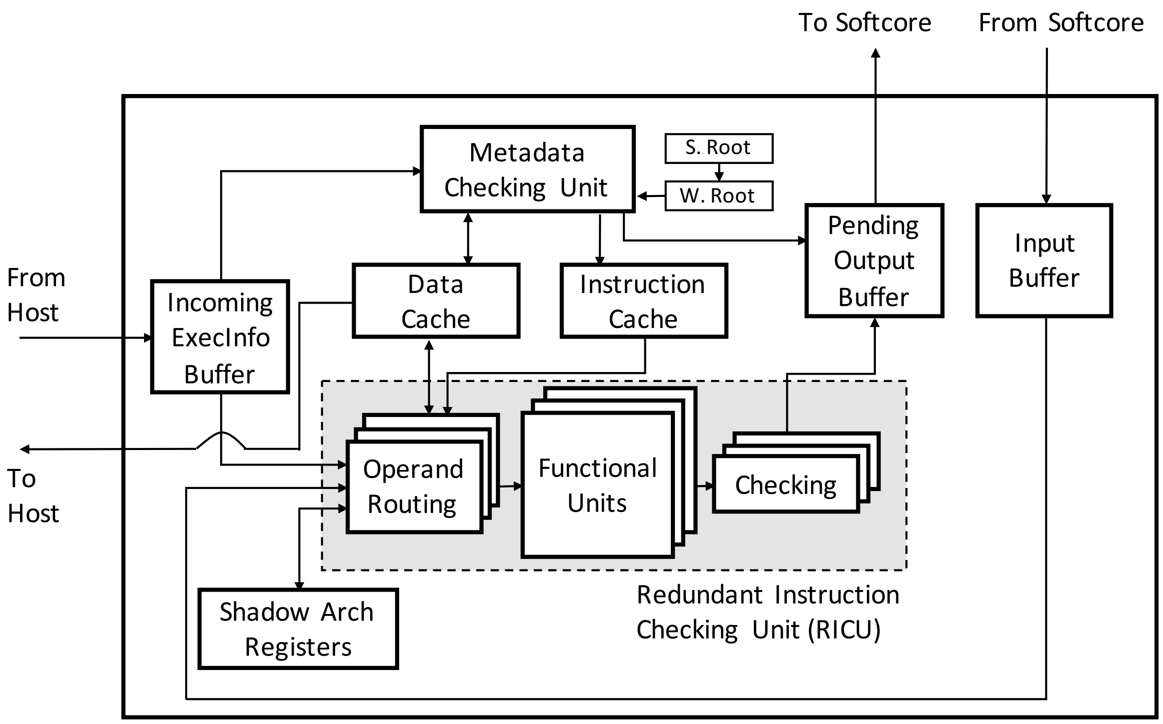


Figure 4.7: Architectural diagram of Sentry 2.0.

`put rs1` The put instruction is used to send a value to the external softcore system. The value is compared against the shadow register value. If the values match, the output value is stored in the Pending Output Buffer. Recall that instruction checking in the Sentry is performed in a speculative manner, assuming that all associated memory validation will succeed[54, 120]. Thus, the potential output value will wait in the buffer until all associated memory checking has been completed by the Metadata Checking Unit.

`r.unt rd` The receive untrusted instruction is used to load a value from untrusted execution into the Sentry without causing a checking violation. The `r.unt` instruction simply takes the result from from the Incoming ExecInfo Buffer and stores it into the specified register, thus no check is performed on this instruction. Instead, the programmer should be sure that their DSC properly validates the untrusted value.

`alrt` The alert instruction is used to indicate that the DSC validation check has failed. This instruction simply causes a failure in the checking unit in the same way that a normal instruction would if its check fails.

4.4 Conclusion

This chapter has presented the first prototype implementation of a CAVO system that runs on a commodity system protected by a prototype Sentry implementation running on an FGPA PCIe network card. Many of the design decisions for this initial prototype were focused towards generating a stable and easily modifiable prototype, rather than focusing on performance. The next chapter presents the DSC Semantics while Chapter 6 describes two initial case study DSCs.

Chapter 5

DSC Semantics

This chapter presents a simplified DSC semantics at the programmer, host assembly, and Sentry assembly levels. It concludes by presenting the effective semantics when the host and Sentry are combined and treated as a single entity.

5.1 Programmer-Level Semantics

The programmer-level semantics is a simplified model that the programmer can use to understand the behavior of trusted Sentry code and DSCs. It is a modified version of the Dynamic Security Labels semantics [126], which is a security-typed lambda calculus that supports first class dynamic labels. In the programmer level semantics, the labels are used to represent the status of the value stored at a particular memory address, either trusted and protected by the Sentry or untrusted. Thus, the `TG_getUntVal` function from Chapter 4.1.1 is simplified to an untrusted load operation. Additionally, variables are stored separately from memory and are considered unmodifiable by untrusted code.

naturals	$n \in$	$\mathbb{Z}$
locals	$x \in$	**Var**
local storage	$R \in$	**Var** $\rightarrow \mathbb{Z}$
memory	$M \in$	$\mathbb{Z} \rightarrow \mathbb{Z}$
labels	$\mathcal{L} \in$	$U \mid T$
label storage	$\Gamma \in$	$\mathbb{Z} \rightarrow \mathcal{L}$
get queue	$\mathcal{Q}_g \in$	$[n_0, \ldots, n_n]$
put queue	$\mathcal{Q}_p \in$	$[n_0, \ldots, n_n]$
arithmatic expr	$a\ (\in AExp) ::=$	$n \mid x \mid (a_0 \oplus a_1)$
bool	$t\ (\in \mathbb{T}) ::=$	**true** $\mid$ **false**
bool expr	$b\ (\in BExp) ::=$	$t \mid (a_0 \odot a_1) \mid (b_0 \oslash b_1) \mid (\neg b)$
commands	$c\ (\in Com) ::=$	$x := a \mid x := [a]_T \mid x := [a]_U \mid [a]_T := a \mid c_0 \ ; \ c_1$
		$\mid$ **get** x $\mid$ **put** x $\mid$ **if** b **then** c_1 **else** c_2
		$\mid$ **while** b **do** c $\mid$ **alert** $\mid$ **invoke** a $\mid$ **skip**
arithmatic operators	$\oplus ::=$	$+ \mid * \mid -$
comparitors	$\odot ::=$	$\leq \mid =$
logical operators	$\oslash ::=$	$\vee \mid \wedge$

Figure 5.1: Syntax for programmer level semantics.

5.1.1 Syntax

The syntax of the programmer level semantics is shown in Figure 5.1, where *R* represents the storage space of variables and *M* represents the memory storage. The labels, represented by $\mathcal{L}$, U and T are stored in the label space Γ, which can be thought of as a shadow storage to the main storage *M*. Finally, $\mathcal{Q}_g$ and $\mathcal{Q}_p$ represent infinite storage queues through which the DSC can communicate with the network. The big-step evaluation of constants, variables, arithmetic and boolean operations take the form of the judgment $\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, e\rangle \Downarrow v$, where $e \in \mathbb{Z}$, **Var**, $AExp, \mathbb{T}, BExp$. The small step evaluation of commands is defined as the judgment $\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, c\rangle \rightarrow \left\langle R', M', \Gamma', \mathcal{Q}'_g, \mathcal{Q}'_p, c'\right\rangle$.

5.1.2 Operational Semantics

The operational semantics for the programmer level language extends standard operational semantics with the security-enhanced memory operations, the ability to invoke external code, and the network **get** and **put** operations.

Arithmetic Expressions

$$\frac{}{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, n\rangle \Downarrow n}\ \text{constants}$$

$$\frac{}{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, x\rangle \Downarrow R(x)}\ \text{regLoad}$$

$$\frac{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, a_0\rangle \Downarrow n_0 \quad \langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, a_1\rangle \Downarrow n_1 \quad n_2 = n_0 \oplus n_1}{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, a_0 \ \oplus \ a_1\rangle \Downarrow n_2}\ \text{AExp}$$

Boolean Expressions

$$\frac{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, a_0\rangle \Downarrow n_0 \quad \langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, a_1\rangle \Downarrow n_1 \quad t = n_0 \odot n_1}{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, a_0 \ \odot \ a_1\rangle \Downarrow t}\ \text{comp}$$

$$\frac{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, b_0\rangle \Downarrow t_0 \quad \langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, b_1\rangle \Downarrow t_1 \quad t_2 = t_0 \oslash t_1}{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, b_0 \ \oslash \ b_1\rangle \Downarrow t_2}\ \text{boolOp}$$

$$\frac{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, b \Downarrow t_0\rangle \quad t_1 = \neg t_0}{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, \neg b\rangle \Downarrow t_1}\ \text{neg}$$

Commands

$\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, \textbf{skip}\rangle$ and $\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, \textbf{alert}\rangle$ are final configurations, with **alert** indicating that a violation has occurred.

$$\frac{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, a\rangle \Downarrow n}{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, x := a\rangle \rightarrow \langle R[x := n], M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, \textbf{skip}\rangle}\ \text{asgn}$$

$$\frac{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, a\rangle \Downarrow n_0 \quad M(n_0) = n_1 \quad \Gamma(n_0) = T}{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, x := [a]_T\rangle \rightarrow \langle R[x := n_1], M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, \textbf{skip}\rangle}\ \text{load}$$

$$\frac{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, a\rangle \Downarrow n \quad \Gamma(n) = U}{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, x := [a]_T\rangle \rightarrow \langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, \textbf{alert}\rangle}\ \text{load.fail}$$

A trusted load that returns a T label operates in the same way as a normal load operation. A trusted load that returns a U label triggers an **alert**.

$$\frac{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, a\rangle \Downarrow n_0 \quad M(n_0) \rightarrow n_1}{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, x := [a]_U\rangle \rightarrow \langle R[x := n_1], M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, \textbf{skip}\rangle}\ \mathit{load.unt}$$

$$\frac{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, a_0\rangle \Downarrow n_0 \quad \langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, a_1\rangle \Downarrow n_1}{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, [a_0]_T := a_1\rangle \rightarrow \langle R, M[n_0 := n_1], \Gamma[n_0 := T], \mathcal{Q}_g, \mathcal{Q}_p, \textbf{skip}\rangle}\ \text{store}$$

$$\frac{\mathcal{Q}'_g = n :: \mathcal{Q}_g}{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, \textbf{get}\ x\rangle \rightarrow \langle R[x := n], M, \Gamma, \mathcal{Q}'_g, \mathcal{Q}_p, \textbf{skip}\rangle}\ \text{get}$$

$$\frac{\langle R, M, \Gamma, x\rangle \Downarrow n \quad \mathcal{Q}_g = n :: \mathcal{Q}'_g}{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, \textbf{put}\ x\rangle \rightarrow \langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}'_p, \textbf{skip}\rangle}\ \text{put}$$

$$\frac{\forall n\, \langle M(n) \neq M'(n) \implies \Gamma(n) = U \wedge M(n) = M'(n) \implies \Gamma(n) = \Gamma'(n)\rangle}{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, \textbf{invoke}\ a\rangle \rightarrow \langle R, M', \Gamma', \mathcal{Q}_g, \mathcal{Q}_p, \textbf{skip}\rangle}\ \text{invoke}$$

Invoke allows untrusted code to execute and store values to memory. Untrusted code may update multiple elements of M, but any update will change the corresponding label in Γ to U.

$$\frac{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, c_0\rangle \rightarrow \langle R', M', \Gamma', \mathcal{Q}'_g, \mathcal{Q}'_p, c'_0\rangle}{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, c_0; c_1\rangle \rightarrow \langle R', M', \Gamma', \mathcal{Q}'_g, \mathcal{Q}'_p, c'_0; c_1\rangle}\ \text{seq}$$

$$\frac{}{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, \textbf{skip}; c\rangle \rightarrow \langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, c\rangle}\ \text{sseq}$$

$$\frac{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, b\rangle \Downarrow \textbf{true}}{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, \textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1\rangle \rightarrow \langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, c_0\rangle}\ \text{if.true}$$

$$\frac{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, b\rangle \Downarrow \textbf{false}}{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, \textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1\rangle \rightarrow \langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, c_1\rangle}\ \text{if.false}$$

$$\frac{}{\langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, \textbf{while}\ b\ \textbf{do}\ c\rangle \rightarrow \langle R, M, \Gamma, \mathcal{Q}_g, \mathcal{Q}_p, \textbf{if}\ b\ \textbf{then}\ (c; \textbf{while}\ b\ \textbf{do}\ c)\ \textbf{else skip}\rangle}\ \text{while}$$

5.2 Host Assembly Level Semantics

The host level assembly semantics is a simplified model that represents the execution of code on the untrusted host machine. It is an extension of the WHILE3ADDR language [57]. The extensions include memory and the Sentry specific features described below. In the host level semantics, execution can either be in trusted or untrusted mode. In trusted mode, execution information is sent to the Sentry depending upon the instruction type.

5.2.1 Syntax

The syntax of the host assembly semantics is shown in Figure 5.2, where *pc* represents the program counter, r_k represents the kth register, *R* represents the register file, *M* represents memory, *P* represents the program, $\mathcal{Q}_g$ represents the incoming queue from the network, and *S* represents the current trusted state as either untrusted (0) or trusted (1). There are no labels in the host side semantics as the host does not update labels nor perform memory validation. Additionally, there is no put queue as the host does not have access to the put queue and must send all outgoing traffic through the Sentry. The small-step judgment for the host assembly semantics have the following form, $P \mid_{\overline{host}} \langle R, M, S, \mathcal{Q}_g, pc\rangle \xrightarrow{\tau} \left\langle R', M', S', \mathcal{Q}_g', pc'\right\rangle$, where $\xrightarrow{\tau}$ represents the transfer of trace values from the execution of the instruction from the host to the Sentry.

5.2.2 Operational Semantics

The host assembly level semantics represents a simplified system where all data from an instruction is sent after each operation, including the instruction, its operands, and its result. The host communicates these values, represented by τ, to the Sentry. Note that τ is used to represent the transfer of trace values and ϵ is used to denote an empty transfer. The **alert** and **return** instructions represent the end of the program.

naturals	$n \in$	$\mathbb{Z}$
program counter	$pc \in$	$\mathbb{Z}$
register	$r \in$	${r_k}^{k \in \{0 \ldots 31\}}$
register file	$R \in$	$r \to \mathbb{Z}$
memory	$M \in$	$\mathbb{Z} \to \mathbb{Z}$
program	$P \in$	$\mathbb{Z} \to c$
get queue	$\mathcal{Q}_g \in$	$[n_0, \ldots, n_n]$
trusted mode	$S ::=$	$0 \mid 1$
commands	$c ::=$	add r_d, r_s, r_t $\mid$ addi r_d, r_s, n $\mid$ sub r_d, r_s, r_t $\mid$ j n
		$\mid$ bgtz r_s, n $\mid$ sw $[r_d], r_s$ $\mid$ lw.U $r_d, [r_s]$ $\mid$ lw.T $r_d, [r_s]$
		$\mid$ **alert** $\mid$ **return** $\mid$ **get** r_d $\mid$ **put** r_s
		$\mid$ **trustedMode** $\mid$ **untrustedMode**

Figure 5.2: Syntax for the host side assembly semantics.

Arithmetic

$$\frac{\begin{array}{c}P[pc] = \text{add}\ r_d, r_s, r_t \quad R(r_s) = n_0 \quad R(r_t) = n_1 \quad n_2 = n_0 + n_1 \\ \text{if } S = 1 \text{ then } (\tau = P[pc], n_0, n_1, n_2) \text{ else } (\tau = \epsilon)\end{array}}{P \vdash_{host} \langle R, M, S, \mathcal{Q}_g, pc\rangle \xrightarrow{\tau} \langle R[r_d := n_2], M, S, \mathcal{Q}_g, pc+1\rangle}\ \text{add}$$

$$\frac{\begin{array}{c}P[pc] = \text{addi}\ r_d, r_s, n_1 \quad R(r_s) = n_0 \quad n_2 = n_0 + n_1 \\ \text{if } S = 1 \text{ then } (\tau = P[pc], n_0, n_2) \text{ else } (\tau = \epsilon)\end{array}}{P \vdash_{host} \langle R, M, S, \mathcal{Q}_g, pc\rangle \xrightarrow{\tau} \langle R[r_d := n_2], M, S, \mathcal{Q}_g, pc+1\rangle}\ \text{addi}$$

$$\frac{\begin{array}{c}P[pc] = \text{sub}\ r_d, r_s, r_t \quad R(r_s) = n_0 \quad R(r_t) = n_1 \quad n_2 = n_0 - n_1 \\ \text{if } S = 1 \text{ then } (\tau = P[pc], n_0, n_1, n_2) \text{ else } (\tau = \epsilon)\end{array}}{P \vdash_{host} \langle R, M, S, \mathcal{Q}_g, pc\rangle \xrightarrow{\tau} \langle R[r_d := n_2], M, S, \mathcal{Q}_g, pc+1\rangle}\ \text{sub}$$

Control Flow

$$\frac{P[pc] = \mathsf{j}\ n \quad \text{if } S = 1 \text{ then } (\tau = P[pc], n) \text{ else } (\tau = \epsilon)}{P \vdash_{host} \langle R, M, S, \mathcal{Q}_g, pc\rangle \xrightarrow{\tau} \langle R, M, S, \mathcal{Q}_g, n\rangle}\ \text{jump}$$

$$\frac{\begin{array}{c}P[pc] = \mathsf{bgtz}\ r_s, n_0 \quad R(r_s) = n_1 \quad n_1 > 0 \\ \text{if } S = 1 \text{ then } (\tau = P[pc], n_1, n_0) \text{ else } (\tau = \epsilon)\end{array}}{P \vdash_{host} \langle R, M, S, \mathcal{Q}_g, pc\rangle \xrightarrow{\tau} \langle R, M, S, \mathcal{Q}_g, n_0\rangle}\ \text{bgtz.g}$$

$$\frac{\begin{array}{c}P[pc] = \mathsf{bgtz}\ r_s, n_0 \quad R(r_s) = n_1 \quad n_1 \leq 0 \\ \text{if } S = 1 \text{ then } (\tau = P[pc], n_1, pc + 1) \text{ else } (\tau = \epsilon)\end{array}}{P \vdash_{host} \langle R, M, S, \mathcal{Q}_g, pc\rangle \xrightarrow{\tau} \langle R, M, S, \mathcal{Q}_g, pc + 1\rangle}\ \text{bgtz.l}$$

Memory

$$\frac{\begin{array}{c}P[pc] = \mathsf{lw.U}\ r_d, [r_s] \quad R(r_s) = n_0 \quad M(n_0) = n_1 \\ \text{if } S = 1 \text{ then } (\tau = P[pc], n_1) \text{ else } (\tau = \epsilon)\end{array}}{P \vdash_{host} \langle R, M, S, \mathcal{Q}_g, pc\rangle \xrightarrow{\tau} \langle R[r_d := n_1], M, S, \mathcal{Q}_g, pc + 1\rangle}\ \text{load.unt}$$

$$\frac{\begin{array}{c}P[pc] = \mathsf{lw.T}\ r_d, [r_s] \quad R(r_s) = n_0 \quad M(n_0) = n_1 \\ \text{if } S = 1 \text{ then } (\tau = P[pc], n_0, n_1) \text{ else } (\tau = \epsilon)\end{array}}{P \vdash_{host} \langle R, M, S, \mathcal{Q}_g, pc\rangle \xrightarrow{\tau} \langle R[r_d := n_1], M, S, \mathcal{Q}_g, pc + 1\rangle}\ \text{load}$$

$$\frac{\begin{array}{c}P[pc] = \mathsf{sw}\ [r_d], r_s \quad R(r_d) = n_0 \quad R(r_s) = n_1 \\ \text{if } S = 1 \text{ then } (\tau = P[pc], n_0, n_1) \text{ else } (\tau = \epsilon)\end{array}}{P \vdash_{host} \langle R, M, S, \mathcal{Q}_g, pc\rangle \xrightarrow{\tau} \langle R, M[n_0 := n_1], S, \mathcal{Q}_g, pc + 1\rangle}\ \text{store}$$

Input/Output

$$\frac{P[pc] = \mathbf{get}\ r \quad \mathcal{Q}_g' = n :: \mathcal{Q}_g \quad \text{if } S = 1 \text{ then } (\tau = P[pc], n) \text{ else } (\tau = \epsilon)}{P \vdash_{host} \langle R, M, S, \mathcal{Q}_g, pc\rangle \xrightarrow{\tau} \left\langle R[r := n], M, S, \mathcal{Q}_g', pc + 1\right\rangle}\ \text{get}$$

$$\frac{P[pc] = \mathbf{put}\ r \quad R(r) = n \quad \text{if } S = 1 \text{ then } (\tau = P[pc], n) \text{ else } (\tau = \epsilon)}{P \vdash_{host} \langle R, M, S, \mathcal{Q}_g, pc\rangle \xrightarrow{\tau} \langle R, M, S, \mathcal{Q}_g, pc + 1\rangle}\ \text{put}$$

Checking Enable/Disable

$$\frac{P[pc] = \textbf{trustedMode}}{P \vdash_{host} \langle R, M, S, \mathcal{Q}_g, pc\rangle \xrightarrow{\epsilon} \langle R, M, 1, \mathcal{Q}_g, pc + 1\rangle}\ \text{enable}$$

$$\frac{P[pc] = \textbf{untrustedMode}}{P \vdash_{host} \langle R, M, S, \mathcal{Q}_g, pc\rangle \xrightarrow{\epsilon} \langle R, M, 0, \mathcal{Q}_g, pc + 1\rangle}\ \text{disable}$$

Final States

$$\frac{P[pc] = \textbf{alert} \quad \text{if } S = 1 \text{ then } (\tau = P[pc]) \text{ else } (\tau = \epsilon)}{P \vdash_{host} \langle R, M, S, \mathcal{Q}_g, pc\rangle \xrightarrow{\tau} \langle R, M, S, \mathcal{Q}_g\rangle}\ \text{alert}$$

$$\frac{P[pc] = \textbf{return} \quad \text{if } S = 1 \text{ then } (\tau = P[pc]) \text{ else } (\tau = \epsilon)}{P \vdash_{host} \langle R, M, S, \mathcal{Q}_g, pc\rangle \xrightarrow{\tau} \langle R, M, S, \mathcal{Q}_g\rangle}\ \text{return}$$

5.3 Sentry Assembly Level Semantics

The Sentry level assembly semantics is a simplified model that represents the re-execution of code by the Sentry. It is an extension of the WHILE3ADDR language [57]. The Sentry semantics considers only the execution of trusted code. As the Sentry does not have trusted memory, it must ensure memory integrity through a Merkle tree.

In this simplified semantics, the Merkle tree operations are performed only at the leaves. This simplification reduces the complexity of the example by eliminating the additional tree walk and corresponding additional hash operations required for a true Merkle tree. Prior work has formally shown that the use of Merkle trees is secure under the standard assumption of a collision-resistant hash function [82, 89]. The model presented here could be extended using these methods.

Additionally, to further simplify the semantics presented, the Sentry has a private memory that it uses to store all hash values. This allows for a unidirectional communication channel from the host to the Sentry that the host uses to send trace values to the Sentry. While this is a valid CAVO architecture, the implementation presented in Chapter 4 uses a two way communication channel between the host and Sentry, where the host sends both trace and Merkle tree values to the Sentry and the Sentry performs any required updates to the Merkle tree before sending it back to the host for storage. Chapter 5.4.1 discusses this further.

5.3.1 Syntax

The syntax of the Sentry assembly semantics is shown in Figure 5.3, where *pc* represents the program counter, r_k represents the kth register, *R* represents the register file, *P* represents the program, $\mathcal{Q}_g$ represents the incoming queue from the network, $\mathcal{Q}_p$ represents the outgoing queue to the network, τ represents the incoming trace values from the untrusted host. $P[0]$ represents the program exiting in an alert state, and $P[1]$ represents the program exiting successfully.

hash represents an internal Sentry function that uses a keyed cryptographic hash (e.g. HMAC). Only the Sentry has access to the key, thus the untrusted components cannot compute the hash. The hash values are used to validate memory operations and are stored in γ.

5.3.2 Operational Semantics

The Sentry attempts to validate successful execution of instructions using the value τ sent by the untrusted host and to validate loaded memory values using γ.

naturals	$n \in$	$\mathbb{Z}$
program counter	$pc \in$	$\mathbb{Z}$
register	$r \in$	$r_k{}^{k \in \{0 \ldots 31\}}$
register file	$R \in$	$r \to \mathbb{Z}$
	hash	Keyed cryptographic hash function $\mathbb{Z} \to \mathbb{Z}$
hash storage	$\gamma \in$	$\mathbb{Z} \to \mathbb{Z}$
program	$P \in$	$\mathbb{Z} \to c$
get queue	$\mathcal{Q}_g \in$	$[n_0, \ldots, n_n]$
put queue	$\mathcal{Q}_p \in$	$[n_0, \ldots, n_n]$
trace values	$\tau \in$	$c, [n_0, \ldots, n_n]$
commands	$c ::=$	add r_d, r_s, r_t | addi r_d, r_s, n | sub r_d, r_s, r_t | j n | bgtz r_s, n
		| sw $[r_d], r_s$ | lw.U $r_d, [r_s]$ | lw.T $r_d, [r_s]$
		| **alert** | **return** | **get** r_d | **put** r_s

Figure 5.3: Syntax for the Sentry side assembly semantics.

Arithmetic

$$\frac{\tau = (\tau_c, n_0, n_1, n_2) \quad \tau_c = P[pc] = \text{add}\ r_d, r_s, r_t \quad R(r_s) = n_0 \quad R(r_t) = n_1 \quad n_2 = n_0 + n_1}{P \vdash_{sentry} \langle R, \gamma, \mathcal{Q}_g, \mathcal{Q}_p, pc\rangle \xrightarrow{\tau} \langle R[r_d := n_2], \gamma, \mathcal{Q}_g, \mathcal{Q}_p, pc + 1\rangle}\ \text{add}$$

$$\frac{\tau = (\tau_c, n_0, n_2) \quad \tau_c = P[pc] = \text{addi}\ r_d, r_s, n_1 \quad R(r_s) = n_0 \quad n_2 = n_0 + n_1}{P \vdash_{sentry} \langle R, \gamma, \mathcal{Q}_g, \mathcal{Q}_p, pc\rangle \xrightarrow{\tau} \langle R[r_d := n_2], \gamma, \mathcal{Q}_g, \mathcal{Q}_p, pc + 1\rangle}\ \text{addi}$$

$$\frac{\tau = (\tau_c, n_0, n_1, n_2) \quad \tau_c = P[pc] = \text{sub}\ r_d, r_s, r_t \quad R(r_s) = n_0 \quad R(r_t) = n_1 \quad n_2 = n_0 - n_1}{P \vdash_{sentry} \langle R, \gamma, \mathcal{Q}_g, \mathcal{Q}_p, pc\rangle \xrightarrow{\tau} \langle R[r_d := n_2], \gamma, \mathcal{Q}_g, \mathcal{Q}_p, pc + 1\rangle}\ \text{sub}$$

Control Flow

$$\frac{\tau = (\tau_c, n) \quad \tau_c = P[pc] = \text{j}\ n}{P \vdash_{sentry} \langle R, \gamma, \mathcal{Q}_g, \mathcal{Q}_p, pc\rangle \xrightarrow{\tau} \langle R, \gamma, \mathcal{Q}_g, \mathcal{Q}_p, n\rangle}\ \text{jump}$$

$$\frac{\tau = (\tau_c, n_1, n_0) \quad \tau_c = P[pc] = \text{bgtz}\ r_s, n_0 \quad R(r_s) = n_1 \quad n_1 > 0}{P \vdash_{sentry} \langle R, \gamma, \mathcal{Q}_g, \mathcal{Q}_p, pc\rangle \xrightarrow{\tau} \langle R, \gamma, \mathcal{Q}_g, \mathcal{Q}_p, n_0\rangle}\ \text{bgtz.g}$$

$$\frac{\tau = (\tau_c, n_1, pc+1) \quad \tau_c = P[pc] = \text{bgtz}\ r_s, n_0 \quad R(r_s) = n_1 \quad n_1 \leq 0}{P \vdash_{sentry} \langle R, \gamma, \mathcal{Q}_g, \mathcal{Q}_p, pc\rangle \xrightarrow{\tau} \langle R, \gamma, \mathcal{Q}_g, \mathcal{Q}_p, pc+1\rangle}\ \text{bgtz.l}$$

Memory

$$\frac{\tau = (\tau_c, n_0) \quad \tau_c = P[pc] = \text{lw.U}\ r_d, [r_s]}{P \vdash_{sentry} \langle R, \gamma, \mathcal{Q}_g, \mathcal{Q}_p, pc\rangle \xrightarrow{\tau} \langle R[r_d := n_0], \gamma, \mathcal{Q}_g, \mathcal{Q}_p, pc+1\rangle}\ \text{l.unt}$$

As the Sentry does not have access to memory, the memory load $[r_s]$ is ignored by the Sentry. Note that no validation is performed on the untrusted load instruction, the value of τ_0 is simply used to update the shadow register file. This is the equivilent of the `r.unt` instruction discussed in Chapter 4.3.3. It is the responsibility of the programmer to ensure their DSC properly validates this value.

$$\frac{\tau = (\tau_c, n_0, n_1) \quad \tau_c = P[pc] = \text{lw}\ r_d, [r_s] \quad R(r_s) = n_0 \quad \gamma(n_0) = n_2 \quad \textbf{\textit{hash}}(n_1) = n_2}{P \vdash_{sentry} \langle R, \gamma, \mathcal{Q}_g, \mathcal{Q}_p, pc\rangle \xrightarrow{\tau} \langle R[r_d := n_1], \gamma, \mathcal{Q}_g, \mathcal{Q}_p, pc+1\rangle}\ \text{load}$$

$$\frac{\tau = (\tau_c, n_0, n_1) \quad \tau_c = P[pc] = \text{sw}\ [r_d], r_s \quad R(r_d) = n_0 \quad R(r_s) = n_1 \quad \textbf{\textit{hash}}(n_1) = n_2}{P \vdash_{sentry} \langle R, \gamma, \mathcal{Q}_g, \mathcal{Q}_p, pc\rangle \xrightarrow{\tau} \langle R, \gamma[n_0 := n_2], \mathcal{Q}_g, \mathcal{Q}_p, pc+1\rangle}\ \text{store}$$

Input/Output

$$\frac{\tau = (\tau_c, n) \quad \tau_c = P[pc] = \textbf{get}\ r \quad \mathcal{Q}_g = n :: \mathcal{Q}_g'}{P \vdash_{sentry} \langle R, \gamma, \mathcal{Q}_g, \mathcal{Q}_p, pc\rangle \xrightarrow{\tau} \left\langle R[r := n], \gamma, \mathcal{Q}_g', \mathcal{Q}_p, pc+1\right\rangle}\ \text{get}$$

$$\frac{\tau = (\tau_c, n) \quad \tau_c = P[pc] = \textbf{put}\ r \quad R(r) = n \quad \mathcal{Q}_p' = \mathcal{Q}_p :: n}{P \vdash_{sentry} \langle R, \gamma, \mathcal{Q}_g, \mathcal{Q}_p, pc\rangle \xrightarrow{\tau} \left\langle R, \gamma, \mathcal{Q}_g, \mathcal{Q}_p', pc+1\right\rangle}\ \text{put}$$

Final States

$$\frac{\tau = (\tau_c) \quad \tau_c = P[pc] = \textbf{alert}}{P \models_{sentry} \langle R, \gamma, \mathcal{Q}_g, \mathcal{Q}_p, pc \rangle \xrightarrow{\tau} \langle R, \gamma, \mathcal{Q}_g, \mathcal{Q}_p, 0 \rangle} \text{ alert}$$

$$\frac{\tau = (\tau_c) \quad \tau_c = P[pc] = \textbf{return}}{P \models_{sentry} \langle R, \gamma, \mathcal{Q}_g, \mathcal{Q}_p, pc \rangle \xrightarrow{\tau} \langle R, \gamma, \mathcal{Q}_g, \mathcal{Q}_p \rangle} \text{ return}$$

5.3.3 Sentry Accept and Reject

If the Sentry is able to complete a step using the above rules, it accepts the result and thus has validated the successful execution of the instruction by the host system. If it cannot, either because of a mismatch between a value in τ sent by the untrusted system or a failure to properly validate a loaded memory value's hash in γ, the Sentry will trigger an alert. It is easily decidable to detect when the Sentry is able to take a step. Thus the negative step for *reject* is easily constructable. If the host sends an empty τ, the Sentry will consume it and its state will remain unchanged.

$$\frac{\langle R, \gamma, \tau, \mathcal{Q}_g, \mathcal{Q}_p, pc \rangle \xrightarrow{\tau} \langle R', \gamma', \tau', \mathcal{Q}'_g, \mathcal{Q}'_p, pc' \rangle}{P \vdash_{sentry} \langle R, \gamma, \tau, \mathcal{Q}_g, \mathcal{Q}_p, pc \rangle \xrightarrow{\tau} \langle R', \gamma', \tau', \mathcal{Q}'_g, \mathcal{Q}'_p, pc' \rangle} \text{ accept}$$

$$\frac{\langle R, \gamma, \tau, \mathcal{Q}_g, \mathcal{Q}_p, pc \rangle \overset{\tau}{\nrightarrow} \langle R', \gamma', \tau', \mathcal{Q}'_g, \mathcal{Q}'_p, pc' \rangle}{P \vdash_{sentry} \langle R, \gamma, \tau, \mathcal{Q}_g, \mathcal{Q}_p, pc \rangle \xrightarrow{\tau} \langle R, \gamma, \tau, \mathcal{Q}_g, \mathcal{Q}_p, 0 \rangle} \text{ reject}$$

$$\frac{}{P \vdash_{sentry} \langle R, \gamma, \tau, \mathcal{Q}_g, \mathcal{Q}_p, pc \rangle \xrightarrow{\epsilon} \langle R, \gamma, \tau, \mathcal{Q}_g, \mathcal{Q}_p, pc \rangle} \text{ empty}$$

5.4 Combining Host and Sentry

The Sentry semantics show (trivially, by definition) that the Sentry maintains correctness (only the results of the correct execution of programs are allowed to communicate externally) in the presence of any given processor. However, this is true even for a processor that produces no output. This section shows that a host processor *can* prove its execution correct to the Sentry and be allowed to communicate externally. More concretely, if a processor properly executes according to the semantics in Chapter 5.2, then the Sentry semantics in Chapter 5.3 will accept its results and allow external communication.

Theorem 1. *If a host executes a program according to the host semantics, i.e. by stepping through the program until reaching* **return***, and the user's DSC accepts the results (i.e. does not execute the* **alert** *command), then the Sentry will step through the program according to the Sentry semantics, forwarding any external communication generated by the host, until reaching* **return***.*

We prove Theorem 1 by showing that the sentry simulates the host: For each step the host takes, the Sentry takes a similar step leading to an equivalent state. Toward that end, we first relate Sentry and host states when the host is in trusted mode.

Definition 1 (State simulation)**.**

$$H \sim \sigma \triangleq$$

$$\begin{array}{rcll} pc_H & = & pc_\sigma & \wedge \\ R_H & = & R_\sigma & \wedge \\ \forall\, n \in M, \textsf{hash}(M[n]) & = & \gamma[n] & \wedge \\ Q_{g,H} & = & Q_{g,\sigma} & \end{array}$$

Informally, a sentry state simulates a host state when its program counter pc_σ matches the host's program counter pc_H, the Sentry's register file R_σ matches the host's register file R_H, the Sentry's input queue $Q_{g,\sigma}$ matches the host's input queue $Q_{g,H}$, and for all locations in the host's memory, the Sentry stores a hash of the contents.

We write $P \vdash_{host} H \xrightarrow{\tau} H'$ to denote the host processor taking a step and updating its state H to H'. We write $P \vdash_{sentry} \sigma \xrightarrow{\tau} \sigma'$ to denote the Sentry taking a step and updating its state σ to σ'.

Lemma 1 (Host-Sentry Simulation)**.** *For all programs P, host states H and H', and sentry states σ, if $P \vdash_{host} H \xrightarrow{\tau} H'$ and $H \sim \sigma$, then there exists σ' such that either $H' \sim \sigma'$ and*

$P \vdash_{sentry} \sigma \xrightarrow{\tau} \sigma'$*, or* $\tau = \epsilon$ *and* $\sigma = \sigma'$

Informally, Lemma 1 states that if the host, H, can take a step to H' and the host is simulated by the Sentry, then the Sentry will take a corresponding step to maintain the simulation (either by stepping through the next command or by taking no step if $\tau = \epsilon$). Thus, such a host is able to convince the Sentry that it has properly executed, inducing the Sentry to produce external communication on its behalf. Theorem 1 follows from Lemma 1.

Proof of Lemma 1 goes by induction on the structure of host transition relation. The initial states of H and σ are equal by definition. A selection of interesting cases of the induction follow.

UNTRUSTED MODE: Suppose we have $P \vdash_{host} H \xrightarrow{\epsilon} H'$, then $\exists\, \sigma' = \sigma$ by definition.

ADD: Suppose we have $P \vdash_{host} H \xrightarrow{\tau} H'$ from the **add** rule of $P \vdash_{host}$. It follows that

- $H = \langle R[r_d = n_0, r_t = n_1], M, 1, \mathcal{Q}_g, pc\rangle$ with $n_2 = n_0 + n_1$,
- $H' = \langle R[r_d = n_2], M, 1, \mathcal{Q}_g, pc + 1\rangle$, and
- $\tau = (P[pc], n_0, n_1, n_2)$,

Let σ' be $\langle R[r_s = n_0, r_t = n_1, r_d = n_2], \gamma, \mathcal{Q}_g, \mathcal{Q}_p, pc+1\rangle$. $H' \sim \sigma'$ holds with $pc_H = pc_\sigma$, $R_H(r_d) = R_\sigma(r_d)$, and $M, \gamma,$ and $\mathcal{Q}_g$ unchanged, and and we can conclude that $P \vdash_{sentry} H \xrightarrow{\tau} H'$ from the **add** rule of $P \vdash_{sentry}$.

BGTZ.G: Suppose we have $P \vdash_{host} H \xrightarrow{\tau} H'$ from the **bgtz.g** rule of $P \vdash_{host}$. It follows that

- $H = \langle R[r_s = n_1], M, 1, \mathcal{Q}_g, pc\rangle$ with $n_1 > 0$,
- $H' = \langle R, M, 1, \mathcal{Q}_g, n_0\rangle$, and
- $\tau = (P[pc], n_1, n_0)$,

Let σ' be $\langle R[r_s = n_1], \gamma, \mathcal{Q}_g, \mathcal{Q}_p, n_0\rangle$. $H' \sim \sigma'$ holds with $n_0 = pc_H = pc_\sigma$, $R_H(r_s) = R_\sigma(r_s)$, and $M, \gamma,$ and $\mathcal{Q}_g$ unchanged, and and we can conclude that $P \vdash_{sentry} H \xrightarrow{\tau} H'$ from the **bgtz.g** rule of $P \vdash_{sentry}$.

LOAD: Suppose we have $P \vdash_{host} H \xrightarrow{\tau} H'$ from the **load** rule of $P \vdash_{host}$. It follows that

- $H = \langle R[r_s = n_0], M[n_0 = n_1], 1, \mathcal{Q}_g, pc\rangle$,
- $H' = \langle R[r_d = n_1], M, 1, \mathcal{Q}_g, pc + 1\rangle$, and
- $\tau = (P[pc], n_0, n_1)$.

Let σ' be $\langle R[r_s = n_0, r_d = n_1], \gamma[n_0 = n_2], \mathcal{Q}_g, \mathcal{Q}_p, pc + 1\rangle$, with ***hash***$(n_1) = n_2$. $H' \sim \sigma'$ holds with $pc_H = pc_\sigma$, $R_H(r_d) = R_\sigma(r_d)$, and M, γ, and $\mathcal{Q}_g$ unchanged, and we can conclude that $P \mid\!\underline{}_{sentry}\; H \xrightarrow{\tau} H'$ from the **load** rule of $P \mid\!\underline{}_{sentry}$.

STORE: Suppose we have $P \mid\!\underline{}_{host}\; H \xrightarrow{\tau} H'$ from the **store** rule of $P \mid\!\underline{}_{host}$. It follows that

- $H = \langle R[r_d = n_0, r_s = n_1], M, 1, \mathcal{Q}_g, pc\rangle$,
- $H' = \langle R, M[n_0 = n_1], 1, \mathcal{Q}_g, pc + 1\rangle$, and
- $\tau = (P[pc], n_0, n_1)$.

Let σ' be $\langle R[r_d = n_0, r_s = n_1], \gamma[n_0 = n_2], \mathcal{Q}_g, \mathcal{Q}_p, pc + 1\rangle$ and ***hash***$(n_1) = n_2$. $H' \sim \sigma'$ holds with $pc_H = pc_\sigma$, ***hash***$(M(n_0)) = \gamma(n_0)$, and R_H, R_σ and $\mathcal{Q}_g$ unchanged, and we can conclude that $P \mid\!\underline{}_{sentry}\; H \xrightarrow{\tau} H'$ from the **store** rule of $P \mid\!\underline{}_{sentry}$.

GET: Suppose we have $P \mid\!\underline{}_{host}\; H \xrightarrow{\tau} H'$ from the **get** rule of $P \mid\!\underline{}_{host}$. It follows that

- $H = \langle R, M, 1, n :: \mathcal{Q}_g, pc\rangle$,
- $H' = \langle R[r = n], M, 1, \mathcal{Q}_g, pc + 1\rangle$, and
- $\tau = (P[pc], n)$.

Let σ' be $\langle R[r = n], \gamma, \mathcal{Q}_g, \mathcal{Q}_p, pc + 1\rangle$. $H' \sim \sigma'$ holds with $pc_H = pc_\sigma$, $R_H(r) = R_\sigma(r)$, $\mathcal{Q}_{g,H} = \mathcal{Q}_{g,\sigma}$ and M and γ unchanged, and we can conclude $P \mid\!\underline{}_{sentry}\; H \xrightarrow{\tau} H'$ from the **get** rule of $P \mid\!\underline{}_{sentry}$.

PUT: Suppose we have $P \mid\!\underline{}_{host}\; H \xrightarrow{\tau} H'$ from the **put** rule of $P \mid\!\underline{}_{host}$. It follows that

- $H = \langle R[r = n], M, 1, \mathcal{Q}_g, pc\rangle$
- $H' = \langle R, M, 1, \mathcal{Q}_g, pc + 1\rangle$, and
- $\tau = (P[pc], n)$.

Let σ' be $\langle R[r = n], \gamma, \mathcal{Q}_g, \mathcal{Q}_p :: n, pc + 1\rangle$. $H' \sim \sigma'$ holds with $pc_H = pc_\sigma$, and R_H, R_σ, M, γ and $\mathcal{Q}_g$ unchanged, and we can conclude that $P \mid\!\underline{}_{sentry}\; H \xrightarrow{\tau} H'$ from the **put** rule of $P \mid\!\underline{}_{sentry}$.

5.4.1 Beyond Synchronous One Way Host-Sentry

The host and Sentry models of Sections 5.2 and 5.3 serve well as simple base models, with the Sentry maintaining its own storage of Merkle tree data (metadata) and operating in lockstep with the host. Although this is a valid CAVO model, it requires the Sentry to have access to a large, trusted memory, and any implementation is only as fast as the Sentry, greatly reducing its usefulness. The following sections discuss more sophisticated models that overcome these limitations.

Two-Way Communication

The first step is to more closely model the metadata management presented in Chapter 4. This requires the host to store the metadata and send both memory values and metadata to the Sentry on loads. It also requires two-way communication between the host and Sentry to properly update the metadata on stores: the host sends memory values and metadata to the Sentry, the Sentry computes the updated metadata, the Sentry sends the updated metadata back to the host for storage. Toward that end, we move the metadata storage, γ, from the Sentry state to the host state. Additionally, we introduce a channel back from the Sentry to the host, so the Sentry can send the updated metadata, β, back to the host. This is is essentially the reverse of the channel used to transfer τ.

This change only affects the **load** and **store** commands when the host is in trusted mode; it does not change its operation in untrusted mode. For **load**, the host is responsible for loading the metadata and forwarding it to the Sentry. For **store**, the host sends the results and metadata to the Sentry, and the Sentry validates its results. If validation succeeds, the Sentry then updates the metadata and sends it back to the host for storage. After sending τ, the host waits for the response β. For all commands other than **store**, $\beta = \epsilon$ and the host does not need to wait for a response. A sketch of the updated **load** and **store** rules follow:

$$\frac{\begin{array}{c} P[pc] = \text{lw.T}\ r_d, [r_s] \quad R(r_s) = n_0 \quad M(n_0) = n_1 \quad \gamma(n_0) \ = \ n_2 \\ \tau \ = \ (P[pc], n_0, n_1, n_2) \end{array}}{P \vdash_{\mathit{host}} \langle R, M, \gamma, 1, \mathcal{Q}_g, pc \rangle \xrightarrow[\epsilon]{\tau} \langle R[r_d := n_1], M, \gamma, 1, \mathcal{Q}_g, pc+1 \rangle} \ \text{load}$$

$$\frac{\begin{array}{c} P[pc] = \text{sw}\ [r_d], r_s \quad R(r_d) = n_0 \quad R(r_s) = n_1 \\ \tau \ = \ (P[pc], n_0, n_1) \quad \beta = (n_2) \end{array}}{P \vdash_{\mathit{host}} \langle R, M, \gamma, 1, \mathcal{Q}_g, pc \rangle \xrightarrow[\beta]{\tau} \langle R, M[n_0 := n_1], \gamma[n_0 := n_2], 1, \mathcal{Q}_g, pc+1 \rangle} \ \text{store}$$

$$\frac{\tau = (\tau_c, n_0, n_1, n_2) \quad \tau_c = P[pc] = \text{lw}\ r_d, [r_s] \quad R(r_s) = n_0 \quad \textit{\textbf{hash}}(n_1) = n_2}{P \vdash_{\mathit{sentry}} \langle R, \mathcal{Q}_g, \mathcal{Q}_p, pc \rangle \xrightarrow[\epsilon]{\tau} \langle R[r_d := n_1], \mathcal{Q}_g, \mathcal{Q}_p, pc+1 \rangle} \ \text{load}$$

$$\frac{\begin{array}{c} \tau = (\tau_c, n_0, n_1) \quad \tau_c = P[pc] = \text{sw}\ [r_d], r_s \quad R(r_d) = n_0 \quad R(r_s) = n_1 \\ \textit{\textbf{hash}}(n_1) = n_2 \quad \beta \ = \ (n_2) \end{array}}{P \vdash_{\mathit{sentry}} \langle R, \mathcal{Q}_g, \mathcal{Q}_p, pc \rangle \xrightarrow[\beta]{\tau} \langle R, \mathcal{Q}_g, \mathcal{Q}_p, pc+1 \rangle} \ \text{store}$$

Asynchronous Host-Sentry Operation

The second step is to change from a lock-step model to an asynchronous model where the host is able to execute the program and stream values to the Sentry, which could be many hundreds or thousands of instructions behind the host. Such a change requires changing the communication channels that carry τ and β to queues. For instructions that do not access memory, this change is trivial.

The metadata management required to validate memory instructions presents a challenge to an asynchronous model due to the following: the host is responsible for managing both memory and metadata; the host is ahead of the Sentry in terms of program execution; the host updates memory immediately after a store; and the Sentry generates updated metadata after a store and sends it to the host for storage. Combined, these facts mean that the host has an up-to-date view of memory but potentially an out-of-date view of the metadata. Thus, a program can store to a memory location and then load from it before the host has received the updated metadata for that location from the Sentry. In such a situation, the host cannot send the correct instruction trace and metadata values to the Sentry for validation.

The host system is thus required to track when there are outstanding updates to the metadata, for example by marking the corresponding memory location dirty. A naïve implementation of the host system could pause execution if a load occurs to a dirty memory location and wait for the updated metadata. However, this would essentially reduce the system to the lock-step model. One solution, which is the system implemented by this dissertation, is to introduce a trusted cache to the Sentry that is managed by the host (in reality it would be managed by the Sentry Control in the host as in Chapter 4). The trusted cache allows the Sentry to validate load operations by trusting that its internally cached version of the value is correct, thus obviating the Sentry's need for the metadata for validation of cached values.[1] When executing loads from dirty locations, the host can simply send the Sentry the loaded value and direct the Sentry to use its cached value for validation. Upon evicting a value from the cache, the Sentry computes the updated metadata and sends it back to the host for storage. If the host loads a value that has an outstanding eviction, it can again direct the Sentry to use its internal values for validation. Once an eviction has completed and the host has received the updated metadata, it can mark the location as clean and future loads from that location can be validated as described in the section above.

Given the complexity of this system, we do not present its formal semantics and proof of correctness. However, this system is essentially maintaining a kind of cache-coherence between the host and the Sentry using a directory-based protocol, where the host acts as the directory. Many prior works have formally validated the correctness of cache-coherence protocols [62, 93, 96], and proof of the correctness of this system would follow those works.

[1]The cached values can either be validated before being placed into the cache, or speculatively used during validation as described in Chapter 2.3.2 [54, 120].

Chapter 6

Two DSC Case Studies

In order to test the difficulty in writing DSCs and test the capabilities of the Sentry to protect real programs, we evaluate two case studies, an in-memory database and a SAT solver. The in-memory database serves as a strenuous test of the networking capabilities of CAVO. The DSC for the in-memory database shows the potential for protecting a wide variety of programs whose primary functionality is storage, such as databases and filesystems. The SAT solver serves as a strenuous test for a program that requires a large amount of computation and shows how to adapt programs with well known verification methods to CAVO.

6.1 Redis

Redis is a commercial grade, open source, in-memory data structure store with persistence features. It is used as a database, cache, and message broker [97]. Redis or its variants are used in many datacenter applications and are offered by cloud services such as Amazon Web Services and Microsoft Azure. The integrity of production databases, like Redis, is incredibly important to virtually every organization that uses computer systems. Databases that return incorrect results and/or leak data can cause irreparable harm, especially in sensitive environments such as hospitals and financial institutions. However, the complexity of system components and of their interactions makes it almost impossible for a Redis server to guarantee the correctness of its returned results. Apart from bugs and vulnerabilities in Redis, bugs in the system itself can have adverse effects on the integrity of the database, including bugs in disk firmware (e.g. lost write [95]), bus controllers (e.g. not reporting write failure or corrupt data [48]), device drivers (e.g. issues disk requests with bad parameters or data [33]), filesystems (e.g. ZFS vulnerable to memory corruptions [125]). Thus, this thesis presents a prototype DSC for Redis to explore the ability for the CAVO model to protect a database server.

The Redis DSC along with the Sentry can prevent such irreparable harm. For example, an attacker could compromise the server and attempt to covertly inject private data from the server into the Redis database as a data value. The attacker could then query for that data in an attempt to exfiltrate the private data. However, as the insertion of the value was not properly authenticated by the DSC, the outward communication of the value would be prevented by the DSC and Sentry. As confirmed by the proof of concept, CAVO prevents attempts to manipulate the results of queries, such as by manipulating the data directly or returning old values via replay.

6.1.1 Redis Validation

The DSC for Redis is realized as a proxy layer that sits between the Redis server and the client, as shown in Figure 6.1. The proxy layer receives all requests from the client, forwards them to Redis, and then validates the responses returned before sending them to the client. This proxy-based approach eliminates the need to change or even examine the Redis server code. Note that the proxy-based approach is just one of many ways that DSCs can be implemented.

The Redis DSC follows prior work done to authenticate queries from outsourced databases [76, 77, 124] and uses Authenticated Data Structures (ADS) [41, 82, 108] to check the integrity of database operations. ADS allows for the outsourcing of data maintenance and process-

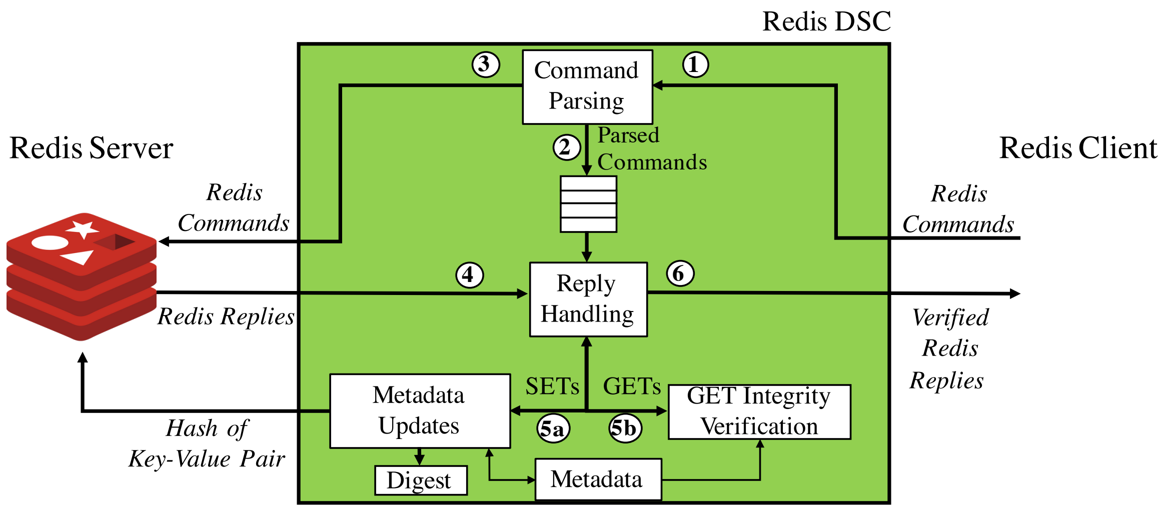


Figure 6.1: .

ing tasks to untrusted parties without loss of integrity. The results of database operations can be efficiently verified using a short digest, which can be viewed as a summary of the current contents of the data and is kept by the verifier. By using ADS, a computationally bound malicious entity cannot fool the verifier into accepting an incorrect result.

To ensure that the response to a request is the correct value, the DSC augments key-values stored to the database with hashes and updates an authenticated search tree (AST) on top of the hashes. The root of this AST represents the state of the database at any time and is updated on every store (i.e. Redis SET operation). As in a Merkle tree, the AST root secures the entire structure. Using this AST, replies to reads (i.e. Redis GET operations) from the Redis Server that involve stale or invalid values will be detected.

After initializing, the DSC waits to receive client requests. Upon receiving a client request (1), the DSC first parses then places the request into the request queue (2). Next the DSC forwards the request to the Redis Server (3). The reply from Redis (4) is then matched to the parsed command. If the command was a SET (5a), the AST is updated and the hash of the new key-value pair is stored in the DSC's metadata and also back into Redis. If the command was a GET (5b), the hash of the key-value response is compared against the stored hash in the DSC's metadata. Note that because the DSC's memory integrity is ensured by the Sentry, validation up to the root of the AST does not need to be performed. Simply validating the key-value hash is sufficient to ensure the validity of the response. This is because the Sentry will ensure that the DSCs entire AST is correct in memory. Thus, validation of a single node is sufficient to ensure correctness. This process is similar to the way the Sentry does not need to perform memory validation for values already checked and stored in its cache. This demonstrates how DSCs can be simplified by taking advantage of the protections offered by the Sentry.

The current prototype implementation for the Redis server node DSC is $\sim$3K lines of C code (LOC). When combined with the $\sim$4k lines of Verilog of the Sentry, the DSC and Sentry provide integrity for a complete Redis server using only $\sim$7K LOC. These $\sim$7K LOC protect an untrusted system of hardware and software totaling more than 500M LOC. This dramatically reduces formal verification efforts by *five orders of magnitude*.

Planned future work is to enable the Sentry to provide the programmer access to the Sentry in order to provide persistent protection. This could either be in the form a non-volatile root register stored on the Sentry, or exposing the Sentry's encryption ability to the programmer. Such functionality will also useful for ensuring other persistence state such as the file system and other storage integrity.

With this functionality, the Redis DSC AST root could be stored so that it can survive restarting the DSC process and full system reboots. During initialization, the DSC could recover the AST root from the Sentry. The DSC could then either read the contents of the database to validate its integrity, or validate the database on demand during subsequent SETs and GETs.

6.2 SAT

The Boolean Satisfiability Problem (SAT) is the problem of determining if there exists an assignment of variables in a given propositional logic formula such that the formula evaluates to true. SAT is a well studied and important problem. It was the first problem that was proven to be NP-Complete [35] and has important applications in areas such as circuit design [55], bounded model checking [27], and automatic theorem proving [23]. SAT solvers typically take a formula as input and produce an output that claims the formula to be either satisfiable (SAT) or unsatisfiable (UNSAT). Modern solvers typically require the input to be in Conjunctive Normal Form (CNF) to simplify their operation. While there are many SAT solving algorithms, the Davis-Logemann-Loveland (DLL) algorithm [39], commonly referred to as the DPLL algorithm in recognition of the algorithm basis in the earlier Davis-Putnam algorithm [40], forms the basis of most modern SAT solvers.

Given its power and importance across many fields, developing fast SAT solvers has long been a priority for many and has spawned conferences and yearly international SAT competitions [63]. While many recent advances have greatly increased the performance of SAT solvers, the need for speed had also introduced bugs from the additional complexity of the performance increasing features. Given that SAT solvers are frequently used in mission critical applications, it is important to ensure that the solver produces correct results [121].

6.2.1 SAT Validation

A SAT solver DSC must be able to validate both SAT and UNSAT responses. Validating SAT is relatively straight forward. It is possible for the SAT solver to produce the variable assignment that shows the formula is satisfiable with relatively little additional overhead. A validator can then verify the solution in linear time given the assignment and the original problem in CNF form. However, validating an UNSAT response is usually not trivial.

Early work in verifying SAT solvers proved that such an approach is feasible [52, 59], but were based on older SAT solving algorithms. zChaff provided the first validator that can easily check the correctness of state-of-the-art DPLL SAT solvers [122]. The zChaff validator is based on the fact that showing that an empty clause can be generated is sufficient to show that a formula in CNF is unsatisfiable. The validator requires that the SAT solver generate a trace of its resolution process. Using the trace and the original CNF formula, the validator attempts to find a resolution sequence that generates an empty clause from the original clause. If such a sequence is found, the UNSAT solution is accepted. If such a sequence is not found, the validator has proven that the solver has produced either an incorrect result or an incorrect trace.

The validator in zChaff is the perfect example of a DSC. Using minor modifications to the untrusted code (the solver), the validator is able to determine the correctness of the result provided. Thus, we use the zChaff solver and validator, with the below modifications, as a second case study in the creation of DSCs.

The zChaff SAT validator serves as the basis for the trusted CAVO program. First, the validator was converted from C++ to C in order to allow it to be compiled by the CAVO toolchain. This involved replacing the use of the standard template vector and set libraries with custom C versions. Next, the validator was converted into a server that accepts connections from clients, receives a SAT formula in CNF from the client, forwards the response to the untrusted zChaff solver, receives a response, validates the response, and forwards the result to the client. No modifications (besides the C++ to C conversion) needed to be made the actual validation algorithm and code.

6.3 Evaluation

Evaluation of the two DSC case studies has four primary goals. First is to ensure that the original goal of simplicity of the trusted components in CAVO has been maintained. Second is to ensure that the functionality of the original programs are maintained. Third is to show the ability of CAVO to prevent erroneous output from escaping the system. Fourth is to determine the current performance of the unoptimized system in order to provide guidance for future optimization efforts.

All evaluation was performed on an Ubuntu 16.04 Linux server with an Intel(R) Core(TM)

	Trusted Code	Untrusted Code
Redis	-	$\sim$20k
Redis DSC	$\sim$2000	$\sim$200
zChaff DSC	$\sim$700	$\sim$100
zChaff validator	$\sim$800	-
zChaff	-	$\sim$2000
Sentry Library	$\sim$3000	$\sim$500

Table 6.1: Approximate lines of code for different trusted and untrusted components for the Redis and zChaff SAT DSCs, and the programs they validate.

i5-6500 CPU @ 3.20GHz and 32GB memory. The CAVO toolchain is based on the RISC-V toolchain [9] (commit ad9ebb8557e32241bfca047f2bc628a2bc1c18cb), with modifications to the Clang front end (to support the Sentry pragmas) and the GNU assembler (to support the creating the Sentry binary). gcc v4.8.4 serves as the untrusted host compiler. The Sentry is instantiated on a NetFPGA SUME Board [7] operating at 100 MHz and uses only a single instruction checking pipeline and encryption engine (to perform the hashes need for Merkle tree operations) for simplicity.

6.3.1 Simplicity

Table 6.1 shows the line counts for various components of the CAVO system used to evaluate the two DSC case studies, which serves as a proxy for complexity of the components. Ideally, trusted components are of minimal complexity, but should never be more complex than the untrusted components they protect. Furthermore, converting an existing validator to a DSC should not be an overly complex process.

The Redis DSC compares favorably to Redis itself, $\sim$2000 LOC vs $\sim$20k, and even more favorably to the rest of the system, which consists of $\sim$400M LOC in the OS, $\sim$20M LOC in the kernel, $\langle$100M LOC for the processor and hardware. The majority of the Redis DSC code ($\sim$500 LOC) is simple code that handles the parsing and processing of the received Redis commands. The SAT validator DSC is actually shorter than the original SAT validator. This is primarily due to the fact that the networking library for the Sentry is currently simpler than its pure C counterpart. Additionally, the code that handles the communication with the SAT solver has become untrusted. In total, the validator required only five additional lines to become a DSC: two lines to send the UNSAT core and its length to the Sentry, and three lines to check for a failure to validate the core and trigger a Sentry `alert`. Finally, the Sentry Library (which contains small DSCs for various C standard library calls and the Sentry networking functions §4) is currently $\sim$3000 lines of code. Since this library is used across all CAVO programs, it represents a one time cost in terms of formally verifying its correctness. The Sentry Library also contains $\sim$500 lines of untrusted code to setup the DSC for native execution and communicate with the Sentry FPGA.

6.3.2 Base Functionality and Attack Testing

CAVO ensures that any external communication is the result of the correct execution of programs. To ensure that CAVO did not prevent correct communication, unmodified versions of Redis and zChaff were evaluated. Redis was evaluated using its provided benchmarking tool (redis-bench) and zChaff was evaluated using well known SAT inputs. Both programs were able to successfully communicate correct results.

To simulate errors and malicious activity by the untrusted software and hardware components, two sets of experiments were performed. The first set simulated errors in the untrusted program. For Redis, this involved randomly changing values in the database without going through the DSC. Upon retrieving this value, the DSC triggered an alert. For zChaff, this meant randomly changing values in the trace. This change was detected by the DSC during trace validation and triggered an alert. To simulate errors in any part of the host system, including the Sentry control, errors were injected into the trace sent from the host to the Sentry for both the Redis and SAT programs by randomly flipping bits in the data stream. All errors were detected by the Sentry, causing the program to halt. Future instantiations of CAVO could include user configurable policies to handle errors, such as

	DSC	Values Sent	Sentry Control	TCP/IP Server	Redis Set	Redis Get	SAT
1	Native				1.05×	1.8×	1×
2	Native			Native	1.05×	1.38×	1.02×
3	CAVO	IO Only	Forward IO	Native	1.1×	7.7×	1.04×
4	CAVO	All Results	Forward IO	Native	1.1×	27×	1.8×

Table 6.2: Overhead of CAVO components evaluated on the Redis and SAT DSCs.

alerting the user or attempting to repair the errors.

6.3.3 Unoptimized Toolchain Performance

As discussed in Chapter 4.3, there are several major components that make up the runtime of CAVO. The focus of this dissertation is to create a complete, working, CAVO system. In order to determine areas for future study and optimization, we evaluate the overhead of multiple components. Table 6.2 shows the results of testing the Redis and SAT DSCs. Redis is evaluated using the benchmarking tool provided by Redis using a 40 client testing environment. All overheads are normalized to the baseline throughput of 770 SET/sec and 138k GET/sec for the 40 client environment. SAT is evaluated using zChaff SAT solver and validator DSC converted into a server. The baseline is the combined native solver and validator time. The input formula is the unsatisfiable 1dlx_c_mc_ex_bp_f test case, which is the Boolean condition for the correctness of a single-issue pipelined DLX processor with multicycle functional units, exceptions, and branch prediction [46]. The baseline solver and validator time for this formula is 0.122 seconds.

Row 1 shows the overhead of using a native version of the DSC without any Sentry protections. For Redis, this introduces a modest amount of overhead from the extra work done by the DSC, as well as the additional communication between the DSC and the Redis server. Note that the GET operations slowed down much more than the SET operations, 1.79× for GETs and 1.05× for SETs. Redis is designed to be a fast, in-memory database, SET operations are highly optimized and thus more impacted by overhead compared to the relatively heavy weight SET operations and the even heavier SAT solver. In some sense, Redis SET can be thought of as trusting the entire program, rather than trusting a DSC that is lightweight in compared to the untrusted calculations.

Recall from Chapter 4.3 that CAVO uses an external TCP/IP server between the Sentry and the network to avoid requiring the Sentry to validate the TCP/IP protocol. Row 2 shows the results of adding in an external TCP/IP server (§4.3) on the host system. Adding in the external TCP/IP server has a minor impact to SETs. It actually speeds up SET operations in comparison to the native Redis DSC alone as it is able to decouple the client management from the DSC operations.

The DSC in row 3 has gone through the CAVO toolchain and been transformed into a Sentry compatible program and sends only the results of the Sentry `put` and `get` operations to a simplified version of the Sentry Control that only handles these IO operations. The relatively small decline in GET and SAT performance shows that the toolchain, even in its current unoptimized form, is a viable approach for some DSCs. Row 4 shows the results of sending all results to the Sentry Control, of which only the IO operations are processed and forwarded to the external TCP/IP server. This shows the overhead of extracting all results from the program, 1.8$\times$ for SET, 41.8$\times$ for GET, and 2.4$\times$ for SAT. Recall that the instrumentation is responsible for storing instruction results so that they may be sent to the Sentry. Given that these results can be batched, the amortized cost of storing them is a memory store and register increment. Naïvely, this triples the number of dynamic instructions executed. However, the actual effects are more complex. These instructions will always serve to increase the size of the basic block they are located in. Given that modern superscaler processors are more easily able to find instruction parallelism in large blocks, it is possible that adding these instructions can have a less than 3$\times$ impact to performance. However, the additional store instructions will fill the processor's memory queue and pollute the cache, negatively impacting performance. Making use of modern processor features, such as streaming store instructions to avoid polluting the cache, can reduce the overhead further.

6.3.4 Prototype FPGA Performance

The following experiments evaluated the performance of the prototype software Sentry Control and the functional prototype FPGA Sentry implemented on a PCIe network card, provided by Zhang [119]. These prototypes were implemented to demonstrate the full functionality of a CAVO system, but are still under development and optimization. First, an experiment to determine the overhead of sending of all results from the DSC as well as all operations in the Sentry Control except for the Merkle tree operations showed overheads of 5.6$\times$ for SETs, 209$\times$ for GETs, and 9.01$\times$ for SAT. Recall that the Sentry Control is responsible for determining the instruction type for the next instruction, managing the Sentry's cache, generating Merkle tree operations, and forwarding the result to the Sentry based on the instruction type. The sizeable increase in overhead for this experiment indicates that the overhead of sending all values and the full processing of the Sentry Control requires substantial improvement. Moving the Sentry Control to hardware would greatly reduce its overhead. The Sentry Control is essentially emulating the Sentry in software, thereby creating a substantial amount of overhead.

The next experiment evaluated the PCIe FPGA card, a simple Sentry without memory validation, and placed the TCP/IP server on the external softcore system. Introducing the PCIe card again has a relatively small impact on the SET operations compared to the previous experiment (5.6$\times$ vs 7$\times$) and a noticeable impact on GET operations (209$\times$ vs 405$\times$) and SAT (9.01$\times$ vs 135$\times$). The final experiment evaluated the full system, including the Merkle tree memory validation operations. The 154$\times$, 8117$\times$, and 677$\times$ slowdowns for GET, SET, and SAT respectively show that the Merkle tree operations introduce serious overheads. This overhead comes from both the additional operations required by the Sentry Control and the additional operations performed by the Sentry itself. This overhead is magnified in the case of GET as it has a relatively higher ratio of memory to normal instructions. This shows the need for both a hardware Sentry Control, more instruction checking pipelines in the Sentry, and the need for additional or pipelined hash engines.

The prototype Sentry used for this evaluation is running at 100 MHz, with a single instruction checking pipeline, and a single hash checking engine. In a production system, the Sentry would need to more closely mirror the Sentry evaluated in TrustGuard [54, 120], i.e. 8 instruction checking pipelines, multiple hash checking engines, and an operating frequency of at least 1 GHz. The Sentry would need to be fabricated as an ASIC to support these additional features. These enhancements, which are part of on-going work, represent an improvement of several orders of magnitude over the current prototype, which would put the overhead of the Sentry in line with the results reported by TrustGuard of $\sim$15% performance overhead. Thus, the Sentry would have only a minor impact to performance and the performance of the full system would be limited by the Sentry Control. The full system performance would then be expected to be of the same magnitude as the system without the Sentry, with the additional overhead of the Merkle tree management. To roughly estimate the performance of such a system, we replaced the Sentry with simple logic to manage IO and saw overheads of roughly $\sim$10$\times$ for GET, $\sim$500$\times$ for SET, and $\sim$180$\times$ for SAT.

Given that the Sentry Control is untrusted, its implementation does not impact the security of the system. Thus, it could be placed as an untrusted component, along with the Sentry and a cache for Merkle tree data, on a fabricated circuit to reduce its performance overhead. Furthermore, the Sentry Control represents an ideal component to move to hardware to improve performance. Recall from Chapter 4.3.2 that the Sentry Control has four primary runtime functions: receive program results, decode the current instruction, forward results and Merkle tree data based on the instruction type, and manage the Sentry's cache. Each of these are simple routines that represent very little overhead in hardware but are costly in software. For example, determining the instruction type in software requires multiple instructions to execute in the host, whereas instruction decode in hardware is a single pipeline stage. Similarly, the cache and Merkle tree management both require

100s of instructions in the Host, but are highly efficient in hardware. Prior work has shown that hardware based Merkle tree mechanisms can have as low as $\sim$2% performance impact [101]. Thus, when moved to hardware, the Sentry Control should have performance impact similar to adding another pipeline stage to the Sentry, slightly increasing latency but not grossly impacting throughput. When combined with the Sentry improvements above, it is expected that the performance bottleneck will be the instrumentation inserted into the program to gather results and the bandwidth required to transfer the results to the Sentry. TrustGuard showed that a host-Sentry bandwidth of 10 GB/s, well below modern host-PCIe bandwidths, is sufficient to limit overhead to below 10% [54, 120]. Thus, with adequate bandwidth, overall system performance should be of the same order of magnitude as Row 4, where the instrumentation gathers all results but the Sentry and Sentry Control do not impact performance.

Chapter 7

Other Related Work

Chapter 2 motivated the CAVO model through comparison to other techniques that use minimal trusted hardware to secure systems. This chapter gives an overview of select other related work, much of which is complimentary to CAVO.

Industrial Adoption of Formal Methods Industry is increasingly adopting formal methods to ensure that critical applications are free from bugs and vulnerabilities. For example, Amazon has used formal methods to ensure correctness in several critical algorithms within the Amazon Web Services infrastructure [49, 88] and Microsoft has used them to design and validate Cosmos DB [74]. This shows a willingness of companies to ensure the security and correctness of critical pieces of software. However, the security of these pieces assumes the security and correctness of all the layers beneath them, including the OS and hardware, to execute the application correctly. Many companies are unwilling to change critical components, such as the OS, and replace their entire computing base due to the economic and operational costs in such changes [28]. Thus, the CAVO model, which needs only a secure DSC and the Sentry to ensure the security of the entire system, is a natural way for companies to continue to adopt formal methods to secure applications without needing to additionally ensure the security of the entire system that sits beneath those applications.

Verifiable Outsourcing. The idea of a trusted entity outsourcing execution to an untrusted entity and then verifying the returned results, without needing to re-execute the original computation, has long existed in many different forms and has seen increasing interest with the rise of cloud computing. For example, work in outsourced databases [42, 76, 77, 85] seeks to ensure the authenticity, integrity, and non-repudiation of database result queries when the database is hosted by an untrusted third party. Such systems typically have trusted local hosts querying data from a trusted publisher that is stored on in an untrusted database. Work in verifiable outsourced computation seeks to generalize the concept to enable a trusted local machine to validate the correctness of the remote execution of any generic program [92, 110, 112, 113, 114]. Such techniques typically involve the use of probabilistic proof systems in which the prover (untrusted worker) generates a proof in the form of a mathematical assertion that the given program and input generate the provided output. The verifier can then efficiently validate the correctness of the proof. While these systems have seen dramatic improvement recently, they are still limited by factors such as applicability (typically they only work for programs with static loop bounds and no indirect memory access) and efficiency (in terms of the costs of setting up and checking the proof). The CAVO model can greatly benefit both sets of techniques by bringing true trustworthiness to the local host to correctly perform validation. In fact, ideas from the outsourced database community helped to guide the Redis DSC.

Security Reference and Program Monitors. Dynamically enforcing policies at runtime has taken many forms, such as execution monitors [14, 104] and inline reference monitors [44, 78]. To be effective in any form, reference monitors must fulfill three requirements. First, they must provably implement the policy they are enforcing by being simple enough to either fully test or formally prove correct. Next, they must fully control the resource for the policy they are enforcing, otherwise an attacker could simply bypass the monitor. Finally, they must be tamper-proof, otherwise an attacker could attack the monitor itself. Examples of program monitors include enforcement of the Brewer-Nash Chinese Wall security policy [29] to ensuring that Microsoft Word cannot execute macros [44].

Reference monitors originated in a 1972 U.S. Air Force report [14]. The original motivation was to ensure the protection of various levels of classified and unclassified information that could exist simultaneously on a multi-user system. The report proposed reference monitors that would capture and validate all resource access by a user, which would later go on to form the basis of modern multi-user systems.

Schneider later defined the class of enforcement mechanisms that monitor the execution of a system as EM, Execution Monitoring. Examples of EM systems include security kernels, firewalls, and reference monitors. He also defined the first formal models for understanding EM and showed that EM systems are capable of enforcing safety properties (nothing bad happens during execution), but not other properties such as liveness (availability) nor enforce information-flow policies [104].

Later work sought to enforce security policies by inlining the monitors into the program, rather than externally monitoring it. Such inline reference monitors were more powerful as they could enforce security properties unique to the program under protection, rather then simply enforcing system level properties such as memory access [25, 44, 78]. The Polymer system [25, 78] extended the capabilities of monitors by allowing them to "insert actions on behalf of, and suppress actions of, untrusted target applications." These more general monitors, called edit automata, are more powerful than EM type systems in that they can enforce liveness properties in addition to safety properties.

The CAVO model can be thought of as a type of Program Monitor, where DSCs implement security policies and the Sentry ensures their correct, tamper-free execution while controlling all access to the protect resource (network communication). Currently the DSCs presented in this thesis enforce only safety properties. However, there is no limitation to that system that prevents them from enforcing other properties as well. For example, one could imagine a DSC that is able to repair incorrect results returned from untrusted code.

Chapter 8

Conclusion

This dissertation has proposed the CAVO model to enable full system containment of the erroneous and malicious effects of untrusted hardware and reduce the size of the trusted software base. It proposed and implemented a programming model, prototype software toolchain, and runtime capable of producing Sentry-protected programs consisting of trusted Dynamic Specification Checks and untrusted program code. This dissertation also proposed an augmented design for the Sentry capable of supporting the programming model and external communication. Using an implementation of this design on an FPGA [119], the dissertation also evaluates two DSC case studies to show the feasibility and capability of the CAVO model.

8.1 Conclusion

This dissertation seeks a middle ground behind the ad-hoc nature of current security practices and the impracticality of abandoning all current computing systems as required by clean slate approaches. It proposes the *Containment Architecture with Verified Output* model, where a simple hardware module, the Sentry, and an application specific *Design Specification Check* combine to ensure the only output from a system is the result of the correct execution of trusted software. By separating the security of the system into these two components, CAVO makes formal verification of the TCB practical.

CAVO's practicality is enabled through an enhanced machine-independent Sentry (thus allowing it to protect existing commodity machines) and a simple programming model and semantics (to facilitate the creation of simple DSCs). CAVO's feasibility is demonstrated through the implementation of the first prototype CAVO system and evaluation on a Redis DSC.

Evaluation of the Redis DSC showed that CAVO is capable of protecting a commercial grade database system including over 500 million lines of code in just $\sim$4K lines of RTL for the Sentry and $\sim$2K lines of code for the DSC. Evaluation of the zChaff SAT DSC showed the ease of converting an existing program validator to a DSC, requiring just 5 additional lines of code and changing $\sim$20 lines of of C networking code into a few lines of Sentry networking library calls. The performance of the software system for CAVO shows its potential, 1.8$\times$ for GET and 2.4$\times$ for SAT. While the performance of the prototype hardware and software version of the Sentry Control are not yet at acceptable levels (slowdowns ranging from 100-8000$\times$), their architectural similarities to high-performance prior work gives confidence that their performance impact can be brought in line with 15% performance overhead reported by TrustGuard [54, 120].

8.2 Future Research Directions

This dissertation has proposed extensions to the CAVO model that bring it closer to practically securing real systems. Below are future research directions to further enhance the types of applications that CAVO can protect and increase its efficiency.

Federation of Sentries CAVO gives assurances as to the integrity of execution and communication from a given machine. However, in a networked environment, it is insufficient to secure individual nodes without giving them a method to confirm the security of the nodes with which they are communicating. Thus, to be a viable solution security solution in production environments and extend the applications supported, TrustGuard requires some form of remote attestation. There are many different design trade offs that can be made depending upon the desired properties guaranteed by the attestation, the complexity of the protocol used, the relative computational power of the devices, etc. Thus, exploring different models such as the collective containment among groups of protected devices for distributed applications and containing a group of devices with a single Sentry for IoT environments are interesting areas of future research.

Hardware Sentry Control As discussed in Chapter 6.3, the Sentry Control is a major source of overhead in the current implementation as it must perform many actions to support the evaluation of a single trusted instruction. However, while there are many actions it performs, they are all relatively simple and algorithmic (for example, cache control). Thus, moving the Sentry Control to an untrusted hardware module on the Sentry's board would greatly increase its performance. Additionally, this would facilitate research into more interesting cache configurations, such as a tiered metadata cache to better match the structure of the Merkle Tree.

Extending Sentry Protections The Sentry can currently protect the execution of a single application at a time. Extending the Sentry to have the capabilities to protect concurrent execution and persistent storage would further increase the range of applications that can be secured by CAVO. As stated in Chapter 6.1.1, this could potentially be accomplished by exposing the Sentry's encryption functionality or a nonvolatile root register on the Sentry to the programmer.

Sentry Protected Values Homomorphic encryption allows systems to process data without giving them access to the underlying data [53]. The CAVO model could be used to provide similar functionality without requiring expensive homomorphic operations. Extensions could be added that protect sensitive data by never releasing the true values to the host system. Instead, dummy values could be used in the host system while the protected values remain only on the Sentry. The special Sentry operations would know to ignore the computed values sent by the host and instead use its own values. These operations would be slightly more expensive than standard operations, as the host values cannot be used to break dependencies between operations in the Sentry, but would be significantly cheaper than fully homomorphic operations.

Multicore Support A natural next step for a single-core design is the extension to multicore. The selective checking functionality provided by the CAVO Programming Model allows for a single threaded application to validate the execution of a multi-threaded execution. For example, the single threaded SAT DSC presented in Chapter 6.2 could easily validate the execution of a multi-threaded SAT Solver. However, some environments may require multiple Sentry protected programs to run concurrently on a multicore machine. One potential solution for this situation would have a Sentry per CPU core. This would require an almost infeasible amount of bandwidth per core. Using die-stacking technology, with one layer consisting of the CPU cores and one layer consisting of the Sentries for each core, could provide the required bandwidth to make such an approach feasible and is worth future exploration.

Bibliography

[1] 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.

[2] Black Hat Europe: Hacking to spy & remotely control video conferencing systems. http://www.computerworld.com/article/2474852/cybercrime-hacking/black-hat-europe–hacking-to-spy—remotely-control-video-conferencing-systems.html.

[3] A cyberattack has caused confirmed physical damage for the second time ever. https://www.wired.com/2015/01/german-steel-mill-hack-destruction/.

[4] Cybersecurity and manufacturers: what the costly Chrysler Jeep hack reveals. http://www.welivesecurity.com/2015/07/29/cybersecurity-manufacturing-chrysler-jeep-hack/.

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

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

[7] NetFPGA. https://netfpga.org.

[8] Pentium chip flaw could leave Intel liable for damages. http://articles.latimes.com/1994-12-16/business/fi-9735_1_pentium-chip-flaw.

[9] RISC-V Foundation. https://riscv.org.

[10] Trusted Computing Group. http://trustedcomputinggroup.org.

[11] 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.

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

[13] Sorin Lerner Alan Leung, Dimitar Bounov. C-to-Verilog translation validation. In *Proceedings of the 2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign*, MEMOCODE '15, pages 42–47, Washington, DC, USA, 2015. IEEE Computer Society.

[14] James P. Anderson and James P. Anderson. Computer security technology planning study. Technical report, Air Force Electronic Systems Division, 1972.

[15] R. Anderson, M. Bond, J. Clulow, and S. Skorobogatov. Cryptographic processors-a survey. *Proceedings of the IEEE*, 94(2):357–369, Feb 2006.

[16] ARM. ARM Security Technology – Building a Secure System using Trust-Zone Technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2009.

[17] Michael Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In *Proceedings of the First International Conference on*

Certified Programs and Proofs, CPP'11, pages 135–150, Berlin, Heidelberg, 2011. Springer-Verlag.

[18] David I. August, Stephen Beard, and Soumyadeep Ghosh. ”pluggable trust architecture”, U.S. provisional pat. ser. no. 62/539,586, filed july 31, 2018., July 2018.

[19] David I. August, Soumyadeep Ghosh, and Jordan Fix. ”trust architecture and related methods”, U.S. provisional pat. ser. no. 15/518,681, filed october 21, 2015., October 2015.

[20] 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.

[21] Todd M. Austin. DIVA: a reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 196–207. IEEE Computer Society, 1999.

[22] Todd M Austin. Designing robust microarchitectures. In *Proceedings of 41st Design Automation Conference*, pages 78–78. IEEE, 2004.

[23] Thomas Ball, Shuvendu K. Lahiri, and Madanlal Musuvathi. Zap: Automated theorem proving for software analysis. In Geoff Sutcliffe and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 2–22, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[24] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Yun Wang, and Y. Zemach. IA-32 execution layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 191–201, Dec 2003.

[25] Lujo Bauer, Jay Ligatti, and David Walker. Composing security policies with Polymer. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 305–314, New York, NY, USA, 2005. ACM.

[26] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen. Scalable sequential equivalence checking across arbitrary design transformations. In *2006 International Conference on Computer Design*, pages 259–266, Oct 2006.

[27] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[28] Robert Bowman. Windows XP Is Extinct – So Why Are So Many Companies Still On It? https://www.forbes.com/sites/robertbowman/2014/05/12/windows-xp-is-extinct-so-why-are-so-many-companies-still-on-it/, 2014.

[29] D. F. C. Brewer and M. J. Nash. The Chinese Wall security policy. In *Proceedings. 1989 IEEE Symposium on Security and Privacy*, pages 206–214, May 1989.

[30] 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.

[31] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash Hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 18–37, New York, NY, USA, 2015. ACM.

[32] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. Kami: A platform for high-level parametric hardware specification and

its modular verification. *Proc. ACM Program. Lang.*, 1(ICFP):24:1–24:30, August 2017.

[33] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 73–88, New York, NY, USA, 2001. ACM.

[34] Cisco. IoT threat environment. Technical report. `https://www.cisco.com/c/dam/en/us/products/collateral/se/internet-of-things/C11-735871.pdf`.

[35] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[36] 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.

[37] Dan O'Keeffe, Divya Muthukumaran, Pierre-Louis Aublin, Florian Kelbert, Christian Priebe, Josh Lind, Huanzhou Zhu and Peter Pietzuch. Spectre attack against SGX enclave. https://github.com/lsds/spectre-attack-sgx, 2018.

[38] Jared Davis and Magnus O. Myreen. The reflective Milawa theorem prover is sound (down to the machine code that runs it). *Journal of Automated Reasoning*, 55:117–183, 08 2015.

[39] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[40] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.

[41] Premkumar Devanbu, Michael Gertz, Charles Martel, and Stuart Stubblebine. Authentic third-party data publication. *Data and Application Security*, pages 101–112, 2002.

[42] Premkumar T. Devanbu, Michael Gertz, Charles U. Martel, and Stuart G. Stubblebine. Authentic third-party data publication. In *Proceedings of the IFIP TC11/ WG11.3 Fourteenth Annual Working Conference on Database Security: Data and Application Security, Development and Directions*, pages 101–112, Deventer, The Netherlands, The Netherlands, 2001. Kluwer, B.V.

[43] 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.

[44] Úlfar Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Ithaca, NY, USA, 2004. AAI3114521.

[45] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. Security analysis of emerging smart home applications. 2016.

[46] John Franco. Formal Methods. http://gauss.ececs.uc.edu/Courses/c626/, July 2017. University of Cincinnati.

[47] Stefan Frei, Bernhard Tellenbach, and Bernhard Plattner. 0-day patch - exposing vendors (in)security performance, BlackHat Europe, 2008, http://www.techzoom.net/papers/blackhat 0day patch 2008.pdf.

[48] G. Green. EIDE Controller Flaws Version 24. http:// mindprod.com/jgloss/eideflaw.html.

[49] Galois. Verifying s2n HMAC with SAW. https://galois.com/blog/2016/09/verifying-s2n-hmac-with-saw/, 2016.

[50] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 193–206, New York, NY, USA, 2003. ACM.

[51] Morrie Gasser, Andy Goldstein, Charlie Kaufman, and Butler Lampson. The digital distributed system security architecture. National Institute of Standards and Technology, January 1989.

[52] Allen Van Gelder. Extracting (easily) checkable proofs from a satisfiability solver that employs both preorder and postorder resolution. In *In Seventh Int'l Symposium on AI and Mathematics*, 2002.

[53] Craig Gentry. Computing arbitrary functions of encrypted data. *Communications of the ACM*, 53(3):97–105, March 2010.

[54] Soumyadeep Ghosh. *TrustGuard: A Containment Architecture with Verified Output*. PhD thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, United States, January 2017.

[55] E. I. Goldberg, M. R. Prasad, and R. K. Brayton. Using SAT for combinational equivalence checking. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, pages 114–121, March 2001.

[56] 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.

[57] Claire Le Goues. The WHILE and WHILE3ADDR language, 2016. `http://www.cs.cmu.edu/~clegoues/courses/15-819O-16sp/notes/notes01-representation.pdf`.

[58] Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao, and David Costanzo. Certikos: A certified kernel for secure cloud computing. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 3:1–3:5, New York, NY, USA, 2011. ACM.

[59] John Harrison. Stålmarck's algorithm as a HOL derived rule. In *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '96, pages 221–234, Berlin, Heidelberg, 1996. Springer-Verlag.

[60] 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.

[61] Intel. Intel Software Guard Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, 2014.

[62] L. Ivanov and R. Nunna. Modeling and verification of cache coherence protocols. In *ISCAS 2001. The 2001 IEEE International Symposium on Circuits and Systems*, volume 5, pages 129–132 vol. 5, May 2001.

[63] Matti Jarvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The international sat solver competitions. *Ai Magazine*, 33, 03 2012.

[64] 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.

[65] 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.

[66] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.*, 32(1):2:1–2:70, February 2014.

[67] 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.

[68] Ralf Kneuper. Limits of formal methods. *Formal aspects of computing*, 9(4):379–394, 1997.

[69] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, January 2018.

[70] 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.

[71] Andreas Kuehlmann and Cornelis A. J. van Eijk. Logic synthesis and verification. chapter Combinational and Sequential Equivalence Checking, pages 343–372. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[72] 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.

[73] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems (TOCS)*, 10(4):265–310, 1992.

[74] Frederic Lardinois. With Cosmos DB, Microsoft wants to build one database to rule them all. https://techcrunch.com/2017/05/10/with-cosmos-db-microsoft-wants-to-build-one-database-to-rule-them-all/, 2017.

[75] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[76] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 121–132, New York, NY, USA, 2006. ACM.

[77] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. Authenticated index structures for aggregation queries. *ACM Trans. Inf. Syst. Secur.*, 13(4):32:1–32:35, December 2010.

[78] Jarred Adam Ligatti. *Policy Enforcement via Program Monitoring*. PhD thesis, Princeton, NJ, USA, 2006. AAI3214569.

[79] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *ArXiv e-prints*, January 2018.

[80] S. Liu and R. Kuhn. Data loss prevention. *IT Professional*, 12(2):10–13, March 2010.

[81] P. Manolios and S.K. Srinivasan. Automatic verification of safety and liveness for XScale-like processor models using web refinements. In *DATE*, 2004.

[82] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. Authenticated data structures, generically. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 411–423, New York, NY, USA, 2014. ACM.

[83] 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.

[84] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: From general purpose to a proof of information flow enforcement. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 415–429, Washington, DC, USA, 2013. IEEE Computer Society.

[85] Einar Mykletun, Maithili Narasimha, and Gene Tsudik. Authentication and integrity in outsourced databases. *Trans. Storage*, 2(2):107–138, May 2006.

[86] Maithili Narasimha and Gene Tsudik. Authentication of outsourced databases using signature aggregation and chaining. In *International Conference on Database Systems for Advanced Applications*, pages 420–436. Springer, 2006.

[87] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 83–94, New York, NY, USA, 2000. ACM.

[88] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services uses formal methods. *Communication of the ACM*, 58(4):66–73, March 2015.

[89] Mizuhito Ogawa, Eiichi Horita, and Satoshi Ono. Proving properties of incremental Merkle trees. In *Proceedings of the 20th International Conference on Automated Deduction*, CADE' 20, pages 424–440, Berlin, Heidelberg, 2005. Springer-Verlag.

[90] HweeHwa Pang, Arpit Jain, Krithi Ramamritham, and Kian-Lee Tan. Verifying completeness of relational query results in data publishing. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 407–418. ACM, 2005.

[91] Ioannis Papagiannis and Peter Pietzuch. Cloudfilter: Practical control of sensitive data propagation to the cloud. In *Proceedings of the 2012 ACM Workshop on Cloud Computing Security Workshop*, CCSW '12, pages 97–102, New York, NY, USA, 2012. ACM.

[92] 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.

[93] Fong Pong and Michel Dubois. Formal verification of complex coherence protocols using symbolic state models. *Journal of the ACM*, 45(4):557–587, July 1998.

[94] Lili Qiu, Yin Zhang, Feng Wang, Mi Kyung, and Han Ratul Mahajan. Trusted computer system evaluation criteria. In *National Computer Security Center*. Citeseer, 1985.

[95] Rajesh Sundaram. The Private Lives of Disk Drives. www.netapp.com/go/techontap/matl/sample/0206tot resiliency.html.

[96] S. Ramírez and C. Rocha. Formal verification of safety properties for a cache coherence protocol. In *2015 10th Computing Colombian Conference*, pages 9–16, Sep. 2015.

[97] Redis. Web site: http://redis.io.

[98] 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.

[99] 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.

[100] Silvain Rideau and Xavier Leroy. Validating register allocation and spilling. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 224–243, Berlin, Heidelberg, 2010. Springer-Verlag.

[101] 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.

[102] 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.

[103] Hanan Samet. Proving the correctness of heuristically optimized code. *Communications of the ACM*, 21(7):570–582, July 1978.

[104] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[105] Alex Shye, Tipp Moseley, Vijay Janapa Reddi, Joseph Blomsted t, 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.

[106] S.K. Srinivasan and M.N. Velev. Formal verification of an Intel XScale processor model with scoreboarding, specialized execution pipelines, and impress data-memory exceptions. In *MEMOCODE '03*.

[107] 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*, pages 160–171. ACM Press, 2003.

[108] Roberto Tamassia. Authenticated data structures. pages 2–5. Springer, 2003.

[109] M Tehranipoor and F Koushanfar. A survey of hardware Trojan taxonomy and detection. *Design Test of Computers, IEEE*, 27(1):10–25, Jan 2010.

[110] 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.

[111] J.D. Tygar and Bennet Yee. Dyad: A system for using physically secure coprocessors. Technical report, Carnegie Mellon University, 1991.

[112] Victor Vu, Srinath Setty, Andrew J Blumberg, and Michael Walfish. A hybrid archi-

tecture for interactive verifiable computation. In *IEEE Symposium on Security and Privacy (SP)*, pages 223–237. IEEE, 2013.

[113] 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.

[114] Michael Walfish and Andrew J. Blumberg. Verifying computations without reexecuting them. *Communications of the ACM*, 58(2):74–84, January 2015.

[115] Chris Weaver and Todd Austin. A fault tolerant approach to microprocessor design. In *International Conference on Dependable Systems and Networks*, pages 411–420. IEEE, 2001.

[116] T. Wchner and A. Pretschner. Data loss prevention based on data-driven usage control. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pages 151–160, Nov 2012.

[117] Yin Yang, Dimitris Papadias, Stavros Papadopoulos, and Panos Kalnis. Authenticated join processing in outsourced databases. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 5–18, New York, NY, USA, 2009. ACM.

[118] Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. Verified correctness and security of mbedTLS HMAC-DRBG. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2007–2020, New York, NY, USA, 2017. ACM.

[119] Hansen Zhang. Sentry NetSume FPGA implementation.

[120] Hansen Zhang, Soumyadeep Ghosh, Jordan Fix, Sotiris Apostolakis, Stephen Beard, Nayana Nagendra, Taewook Oh, and David I. August. Architectural support for containment-based security. pages 361–377, 04 2019.

[121] Lintao Zhang and Sharad Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, pages 10880–, Washington, DC, USA, 2003. IEEE Computer Society.

[122] Lintao Zhang and Sharad Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, pages 10880–, Washington, DC, USA, 2003. IEEE Computer Society.

[123] Yun Zhang, Soumyadeep Ghosh, Jialu Huang, Jae W. Lee, Scott A. Mahlke, and David I. August. Runtime Asynchronous Fault Tolerance via Speculation. *International Symposium on Code Generation and Optimization (CGO)*, March 2012.

[124] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. Integridb: Verifiable SQL for outsourced databases. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1480–1491, New York, NY, USA, 2015. ACM.

[125] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. End-to-end data integrity for file systems: A ZFS case study. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[126] Lantian Zheng and Andrew C. Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 6(2-3):67–84, 2007.