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

JANUARY 2018
Abstract

All layers of today’s computing systems, from hardware to software, are vulnerable to attack. Market and economic pressure generally result in companies focusing on performance and features, leaving security and correctness as post-deployment concerns. Thus, systems must frequently be patched or augmented after deployment to fix security vulnerabilities. While this non-virtuous exploit-patch-exploit cycle is not the most secure, it is practical enough for companies to use. Formal methods in both software and hardware can give true guarantees as to the security they provide. Ideally, modern systems would be comprised of entirely 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, are a major factor in clean slate solutions being unrealistic. Thus, a practical solution should rely on a few, simple components and should be adoptable incrementally.

As the first implementation of the Containment Architecture with Verified Output (CAVO) model developed at Princeton, TrustGuard 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, TrustGuard’s 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. The first extension is to make 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. The second extension is to give 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 similar to the way that the small, trusted Sentry can ensure correctness of execution of a large amount of complex, untrusted hardware.
Acknowledgments

Todo
Contents

Abstract ......................................................... iii
Acknowledgments .............................................. v
List of Tables .................................................. viii
List of Figures .................................................. ix

1 Introduction ................................................ 1
  1.1 TrustGuard ............................................... 3
  1.2 Dissertation Contributions .............................. 4
  1.3 Dissertation Organization .............................. 7

2 Background and Motivation ................................. 8
  2.1 Trust through Attestation .............................. 8
  2.2 Trust through Enclaves ................................ 10
  2.3 Trust through Containment ............................ 11
    2.3.1 Parallel Redundant Instruction Checking .......... 12
    2.3.2 Memory Validation .................................. 13
    2.3.3 Simplicity of the Sentry Design .................. 14
    2.3.4 Performance Evaluation of TrustGuard ............ 15

3 Towards a Practical Containment Architecture ........... 17
  3.1 Threat Model .......................................... 18
  3.2 CAVO .................................................. 19
4 CAVO Prototype Implementation

4.1 Programming Model

4.1.1 Supporting DSRs

4.1.2 Network Communication

4.2 Compiler Based Software Toolchain

4.2.1 Sentry Emulation and Program Instrumentation

4.2.2 Library and System Call Verification

4.3 Runtime Verification

4.3.1 Program Loader

4.3.2 Sentry Control

4.3.3 Sentry 2.0

4.3.4 External Communication

5 Evaluation

5.1 Exploration of a DSR: Redis

5.2 Evaluation

6 Related Work

7 Conclusion

7.1 Conclusion

7.2 Future Research Directions
List of Tables

3.1 Various levels of protection offered by the Sentry depending upon the level of developer effort in DSR creation and program verification. . . 23
List of Figures

1.1 (a) TrustGuard. A Sentry, with help from a modified CPU, contains erroneous (malicious or not) actions of untrusted hardware components. (b) CAVO (this dissertation). 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. ................................................................. 3

2.1 TrustGuard Architecture ......................................................... 11
2.2 Effect of adding Merkle Tree accesses to untrusted processor .... 15
2.3 Parameter sweep through processor-Sentry bandwidth .............. 15
2.4 Parameter sweep through parallel checking pipelines ............... 15
2.5 Parameter sweep through Sentry clock frequency ................... 15

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. ................................. 20

4.1 Simplified view of the first prototype implementation of a CAVO based system. ................................................................. 26
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, 17], damage to enterprise assets [6, 23], operational disruption [1, 18], industrial and military espionage [2, 8], and even physical harm to people and their environment [3, 5, 17]. In most cases, designers, manufacturers, and developers do not possess an effective and practical way to secure their products. Thus, security and privacy breaches are still daily news.

System engineers and software developers have traditionally approached security as a 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 in the future. While patching software can be expensive, annoying, and disruptive (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 [24].

The story is different for hardware, where only a subset of vulnerabilities can be repaired through mechanisms such as microcode patches or firmware updates [7]. 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 [31, 33, 38, 66]. However, the complexity of modern hardware design has simply outpaced our ability to ensure their correctness, as demonstrated by the recent Spectre [37] and Meltdown [45] 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. Worse, these methods currently do not scale to the large and complex designs of modern computing systems [34, 35, 36, 40, 49]. 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 [41, 55]. 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.

Recent processor based secure enclaves (such as Intel’s SGX [32], and ARM’s TrustZone [10]) are a type of TCB. 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 performing 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 tasks, they ultimately still require placing trust in a complex processor (made even more complex by the enclave) that cannot be properly secured. Furthermore, the sensitive data cannot be used unencrypted outside the enclave without putting it at risk, which means that a user can never directly interact with it (§2.2).

1.1 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 [28]. 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.

Figure 1.1: (a) TrustGuard. A Sentry, with help from a modified CPU, contains erroneous (malicious or not) actions of untrusted hardware components. (b) CAVO (this dissertation). 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.

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 integrity 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 an 8% impact on system performance and energy consumption [28]. 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 Dissertation Contributions

This dissertation generalizes TrustGuard’s concept of containment to a broader, more practical and effective *Containment Architecture with Verified Output (CAVO)* model that en-
sures full system privacy and integrity. The CAVO model frees the Sentry from direct
coupling to the customized CPU from prior work, replacing it with efficient run-time sup-
port in software, and reduces the trusted code base from the entire application stack to a
small Dynamic Specification Check (DSC). To encourage adoption, CAVO must be flex-
ible 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).

To decouple the Sentry from a modified CPU, this dissertation proposes moving the
modified processor functionality that supports the Sentry into software. Removing the
need for the modified processor has the added benefit of allowing 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. Furthermore, this increases the practicality of the CAVO model
by enabling to protect existing systems that are in use today.

In the same way that the Sentry serves as a small, trusted verifier for a large amount of
untrusted hardware, CAVO creates a programming model that allows developers to create
small DSCs for their programs or systems. DSCs are pieces of code that 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. By trusting only the
Sentry and the DSC, the overall amount of trusted code in the system goes from over 500M
lines of code (LOC) to just ∼4K lines of RTL for the Sentry and ∼3K LOC for an prototype
key-value DSC (§5.1). Sentry verification is a one time cost, 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.
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.

To allow for environments with different levels of trust in components the CAVO model is composed of several flexible and exchangeable 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 high security military application where only the Sentry is trusted) to lower security (e.g., a low security 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 will be flexible enough to support a wide range of DSC types (§3.2.2). Ideally, the DSC is 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 (§5.1). 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 a prototype implementation of the Sentry on an FPGA card. Together, these components enable a Sentry protected program to run on a commodity system, with the added Sentry FPGA card.

In summary, the contributions of this dissertation are:
• Design of the CAVO model, which provides the flexibility to enable a number of different instantiations to suit a desired effort-to-security trade-off.
• 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
  – Prototype FGPA Sentry Implementation
• Evaluation of a DSC designed to protect the Redis key-value store

1.3 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 evaluates the CAVO model using a DSC for the Redis key-value store. Chapter 6 discusses other related work. Finally, Chapter 9 concludes and discusses areas of future exploration for the CAVO model.
Chapter 2

Background and Motivation

Modern computer systems architecture is layered, with applications at the top and hardware at the bottom. 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.” [21] Thus, trust in a system must ultimately be based in hardware.

Given the difficulties in securing hardware (§6), 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 [27], with many modern systems implementing both local attestation, sometimes referred to as secured boot, and remote attestation [67], for establishing the identity of remote hosts in a network environment. Attestation assumes that there is a secure and trusted set of hardware and software that can be used to establish a trusted state of a 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. Remote attestation can be used to prove this to third parties.

While attestation can be useful, especially in relatively simple embedded 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 [47, 64]. Recent software efforts have formally verified a C compiler [42], a simple operating system kernel [30, 49], a deterministic random bit generator [72], and an HMAC algorithm used for TLS [25]. Unfortunately, they currently cannot scale to the complexity of full, production systems (§6). 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... [27]

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... [26]
2.2 Trust through Enclaves

Recognizing that it is too difficult to ensure trust in the system at large, others use a trusted hardware module to build a secure enclave or trusted execution environment inside the system. The enclave protects trusted code from other hardware and software in the system. 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 [32] and ARM’s TrustZone [10]) or as secure co-processors (such as the IBM 4758 [21]). The first approach necessitates trusting a modern complex processor, a risky proposition given their complexity and the recent Spectre [37] and Meltdown [45] vulnerabilities. In fact, some researchers have already found ways to apply Spectre to attack SGX enclaves [19]. Co-processors typically come with a large performance overhead, as they are usually much slower than the main processor and require off-chip communication.

While the enclave approach is useful for ensuring 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 can not 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 is never directly used, 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 [28]. In TrustGuard, the trusted hardware element, called 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 uni-processor 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...
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 [12, 13, 14, 16, 29, 48, 57, 58, 61, 63, 69, 74]. Additionally, the Sentry uses a Bonsai Merkle Tree-based memory integrity scheme [60] 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 simplified (§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 results 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 used 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 [60]. 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 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 are 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 [60, 65]. 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 processors 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 only released from the Pending Output Buffer 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 [47, 64]. 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 Performance Evaluation of TrustGuard

TrustGuard [28] was modeled in the gem5 simulator using an out-of-order (OoO) ARM-core based untrusted processor to perform the analysis. The evaluation was performed to determine the Sentry’s effect on the processors performance during steady state operation. Performance was evaluated against 8 SPEC INT2006 and three SPEC FP2006 workloads.

Performance degradation in TrustGuard can come from three sources. First, there is the 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 geometric mean 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 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 the geomean IPC decline was 21.1%, 8.5%, and 7.5% respectively, as seen in Figure ?? . 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. Two sets of experiments were performed 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 (1/4th 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, experiments varying the Sentry’s clock frequency were performed, 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.
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, representing decades of development and refinement.
3. Trust should only be placed in components that are small and 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 purposely 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 maybe relaxed for less secure settings.

3.1 Threat Model

The goal of TrustGuard is to ensure that only correctly executed signed software is allowed to communicate. Thus, TrustGuard 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 TrustGuard 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. TrustGuard 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. TrustGuard 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. TrustGuard does not address leakage through side channels that bypass the Sentry, such as timing, power, and availability.

### 3.2 CA VO

The goal of CA VO is ultimately to provide users strong privacy and integrity guarantees for their systems communication, while relying on only a small hardware and software trusted computing base. This is accomplished by separating the system into trusted components, whose first priority is to ensure the security of the system, and untrusted components, which can focus on performance and functionality but whose work is ultimately validated by trusted components before leaving the system. More specifically, trust is placed in Dynamic Specification Checks (DSRs) to ensure that any communication that leaves the system adheres to the intention of the programmer and in the Sentry to ensure that the DSR was faithfully executed by the hardware.

While TrustGuard [28] 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 CA VO systems without requiring substantial hardware modifications to the system under protection, namely requiring a specialized processor and custom bus to interface with the sentry the Sentry. Second, it gives developers the option to use trusted code to verify the execution of untrusted code, thus enabling a reduction of the size of the trusted code base. This is similar to the way that the small, trusted Sentry can ensure correctness of execution of a large amount of complex, untrusted hardware.
3.2.1 CAVO Infrastructure

A 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 DSR 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 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 multiple ways to extract and forward execution results. Some potential infrastructures considered were: an emulator / virtual machine approach, where DSRs 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 thesis presents an infrastructure based upon the compiler based approach, which
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 gives the opportunity 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 thesis makes use of a prototype Sentry on an FPGA PCIe network card due to its ease of testing both the network card and appliance type manifestations.

### 3.2.2 Dynamic Specification Routines

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 Routines (DSRs), thereby reducing the trusted code base to just the DSR and Sentry libraries, rather than the OS, libraries, and whole applications as in TrustGuard.

Ideally, DSRs are small pieces of verification code that ensure a dynamic specification
for an entire untrusted system, such as a database. In some cases it may be more desirable for a DSR to enforce a particular security policy for the application rather than ensuring correctness. Since simple and flexible DSR creation is the key to its adoption, CAVO seeks to give developers flexibility in implementing DSRs 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.

DSRs facilitate formal verification by drastically reducing the size of the trusted code base, similar 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, DSRs 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 can not only reduce amount of validation work the Sentry must perform, but can also offer 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 DSR to perform validation, similar to the way that the untrusted processor in TrustGuard performed additional work to facilitate validation performed by the Sentry.

**Example Applications**

Table 3.1 summarizes various types of DSRs 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 guar-
### Table 3.1: Various levels of protection offered by the Sentry depending upon the level of developer effort in DSR creation and program verification.

<table>
<thead>
<tr>
<th>Protection Type</th>
<th>Effort</th>
<th>Protection</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Sentry Only</td>
<td>None</td>
<td>Malicious HW; and interference from malicious programs</td>
<td>Any Program</td>
</tr>
<tr>
<td>2. Algorithmic Verification</td>
<td>Low</td>
<td>Protections from (1); and ensure correct output of algorithm through verification</td>
<td>Sort; SAT; Gradient Descent; Euler’s Method; Newton’s Method</td>
</tr>
<tr>
<td>3. Security Policy Verification</td>
<td>Low</td>
<td>Protections from (1); and ensure security policy</td>
<td>Data Loss Prevention; Login Restrictions; Mechanical Protections for Physical Systems</td>
</tr>
<tr>
<td>4. Specification Verification</td>
<td>Medium</td>
<td>Protections from (1); and ensure correct output according to specification</td>
<td>Database Operations; Compiler Transformations; Circuit Equivalence Checking</td>
</tr>
<tr>
<td>5. Full Program Verification</td>
<td>High</td>
<td>Protections from (1); and ensure correct execution of entire program</td>
<td>Small Programs</td>
</tr>
</tbody>
</table>

...anteed by the Sentry, namely protection from malicious hardware and interference from untrusted applications. Similarly, programs that have already been formally verified, such as CompCert [42], can be signed and executed with the security guarantees offered by its verification and those of the Sentry. Below are some example areas where DSRs can be used to ensure correctness in programs or enforce security properties.

**Algorithmic Verification** There are many algorithms which are easier to validate than to execute. A Developers may wish to protect such algorithms with a small DSR 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 [11, 73]. 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 to reduce the amount of work performed by the small DSR SAT validator. Verifiable computing [68] 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 [22]. For example, many companies are highly concerned with data loss prevention [46, 54, 70]. 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 DSR 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 DSR 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 DSR that integrates with the Sentry was created to Sentry to protect a commercial grade database server, Redis [56]. The DSR code validates database operations, similar to prior work in outsourced databases [20, 43, 44, 50, 51, 53, 71, 75]. A more detailed discussion of the Redis DSR is found in Chapter 5.1.

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 transformation 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 [52, 59, 62]; circuit equivalence checking [15, 39]; and High-Level Synthesis tools, such as C-to-RTL translation [9].

24
Chapter 4

CAVO Prototype Implementation

This chapter presents the first prototype implementation of a full CAVO system. 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 the CAVO Programming Model (§4.1) that breaks an application into two main pieces: the trusted and untrusted code. In this model, an application starts in trusted code and can then call trusted code to perform some work on its behalf. Upon receiving a result from the untrusted code, the trusted code will perform some verification to ensure that the untrusted code has produced an acceptable result. Additionally, the model contains a library that allows the programmer to utilize 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 trusted execution 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.

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 instructions streams results to the Sentry Control, which is currently a daemon running on the untrusted machine, 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, and streaming results to the Sentry. Finally, the Sentry itself validates instructions and communicates with the external interface.

4.1 Programming Model

The goals of the CAVO Programming Model are to support the creation of DSRs and enable trusted communication through the Sentry.
4.1.1 Supporting DSRs

As discussed in Section 3.2.2, one of the core ideas of CAVO is to allow a small, trusted DSR to validate the execution of a large amount of untrusted code. Thus, DSRs require the ability to indicate trusted and untrusted code, call into untrusted code from trusted code, receive untrusted values from untrusted execution, and trigger an alert if validation fails.

4.1.2 Network Communication

Custom communication library. 0-1k are for sockets, first one is to enter send mode, syntax for create connection to this outside, receive from this outside socket. Discuss current blocking design.

Requires additions to the Sentry Get and Put instructions(§4.3.3).

4.2 Compiler Based Software Toolchain

Overall goal of the toolchain and basic flow.

Toolchain figure

4.2.1 Sentry Emulation and Program Instrumentation

Compile to SISA first, then translate that to native x86. Be sure to add communication to the Sentry for the appropriate values. Ensure address space lines up properly. Omit untrusted calls from Sentry version.

Table showing the conversions?
4.2.2 Library and System Call Verification

4.3 Runtime Verification

Overall flow

Runtime Figure, multiple threads, bypass, outside softcore

4.3.1 Program Loader

Include program loader discussion.

Program Loader Flow Figure

4.3.2 Sentry Control

Control handles all communication (sending values) and meta instructions (cache and Merkle management), as in previous work [28].

Figure/Algorithm Sentry Control loop

Includes reset extensions needed for working full system. Clears Sentry state and reloads root of Merkle tree with loader root value.

4.3.3 Sentry 2.0

Discuss the enhancements I made to Sentry and overall architecture design to support communication as shown in the figure.

Figure for communication and bypass. Show the buffers and discuss the speculative lock on communication. Selectively detailed sentry figure

4.3.4 External Communication

TCP/IP stack resides outside the Sentry to eliminate the need to send all the low level info back and forth through the Sentry. Currently this is an embedded kernel running on a
softcore on the FPGA. Would ideally become hardware TCP/IP stack.
Chapter 5

Evaluation

5.1 Exploration of a DSR: Redis

Redis

5.2 Evaluation

Performance Evaluation Breakdown

Instrumentation vs FGPA performance?
Chapter 6

Related Work

Current Ad Hoc security
   Formal Methods
   Security Reference Monitors
   Security CoProcessors
Chapter 7

Conclusion

This dissertation has proposed XXX to enable the containment of erroneous and malicious effects of untrusted hardware and reduce the size of the trusted software base. It proposed, implemented, and evaluated a prototype software toolchain, runtime, FPGA Sentry implementation, and database DSR to show the feasibility and capability of the CAVO model.

7.1 Conclusion

7.2 Future Research Directions

Federation of Sentry / Remote attestation

   Persistent Storage Protection

   Concurrency

   Formal Verification of Sentry and DSRs
Bibliography


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


[48] Maher Mneimneh, Fadi Aloul, Chris Weaver, Saugata Chatterjee, Karem Sakallah, and Todd Austin. Scalable hybrid verification of complex microprocessors. In *Pro-


[55] Lili Qiu, Yin Zhang, Feng Wang, Mi Kyung, and Han Ratul Mahajan. Trusted com-


