Architectural Support for Containment-based Security

Hansen Zhang*
Princeton University

Soumyadeep Ghosh*†
Princeton University

Jordan Fix‡
Princeton University

Sotiris Apostolakis
Princeton University

Stephen R. Beard
Princeton University

Nayana P. Nagendra
Princeton University

Taewook Oh‡
Princeton University

David I. August
Princeton University

Abstract

Software security techniques rely on correct execution by the hardware. Securing hardware components has been challenging due to their complexity and the proportionate attack surface they present during their design, manufacture, deployment, and operation. Recognizing that external communication represents one of the greatest threats to a system's security, this paper introduces the TrustGuard containment architecture. TrustGuard contains malicious and erroneous behavior using a relatively simple and pluggable gatekeeping hardware component called the Sentry. The Sentry bridges a physical gap between the untrusted system and its external interfaces. TrustGuard allows only communication that results from the correct execution of trusted software, thereby preventing the ill effects of actions by malicious hardware or software from leaving the system. The simplicity and pluggability of the Sentry, which is implemented in less than half the lines of code of a simple in-order processor, enables additional measures to secure this root of trust, including formal verification, supervised manufacture, and supply chain diversification with less than a 15% impact on performance.

CCS Concepts • **Security and privacy → Hardware security implementation**.

Keywords hardware security, containment, pluggable

*These authors contributed equally to this research.
†Work done at Princeton University. At Barefoot Networks at time of publication.
‡Work done at Princeton University. At Facebook at time of publication.

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304020

ACM Reference Format:
Hansen Zhang, Soumyadeep Ghosh, Jordan Fix, Sotiris Apostolakis, Stephen R. Beard, Nayana P. Nagendra, Taewook Oh, and David I. August. 2019. Architectural Support for Containment-based Security. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3297858.3304020

1 Introduction

Users may believe their systems are secure if they run only trusted software; however, trusted software is only as trustworthy as the underlying hardware. Even if users run only trusted software, attackers can gain unauthorized access to sensitive data by exploiting hardware errors or by using backdoors inserted at any point during design, manufacture, or deployment [12, 24, 39, 97, 98, 103]. For example, Biham et al. have demonstrated a devastating attack on the RSA cryptosystem that builds on a multiplication bug that computes the wrong product for only a *single pair* of 64-bit integers [17]. An attacker can use knowledge of this pair to break *any key* used in *any RSA-based software* running on *any device* whose processor has this bug using *a single chosen message* [17, 79].

Due to the complexity of designing and manufacturing hardware, architects and manufacturers have limited confidence that their systems have not been altered maliciously [20, 45, 52]. This confidence is further undermined by the fact that building a computer system often involves different parties across several important stages, from the initial specification all the way to fabrication. For example, manufacturing may be outsourced for economic reasons to companies operating under the jurisdiction of foreign governments [12]. Additionally, many hardware components may include intellectual property restrictions that prevent concerned parties from ensuring their correctness and security. Although prior techniques attempt to ensure hardware integrity during design, manufacture, and deployment, they either do not protect against attacks in all of these stages or are limited in the types of hardware components covered [2, 20, 23, 40, 44, 45, 54, 65, 66, 76, 77, 106].

Given the difficulty in securing hardware, solutions that protect complex computing systems using a single secure hardware component are attractive. A promising class of techniques uses this approach to secure systems by trusting only the processor [25, 86]. Unfortunately, the only processors that have been secured with formal methods are simple and low performance [20, 58, 64, 83]. Generally, processor designers have focused on performance over security. For example, aggressive optimizations (e.g., out-of-order execution) improve processor performance at the cost of design complexity. The complexity of processors with reasonable performance makes their verification far beyond the capabilities of state-of-the-art formal methods [45, 52, 68, 93]. Further, backdoor detection does not scale to complex hardware components such as processors with reasonable performance [1, 39, 44, 84, 89, 97–99, 109, 110]. Thus, securing a system using a single *secure* hardware component means securing a system with a single *simple* hardware component.

Single simple hardware components have successfully provided systems with limited security guarantees. For example, the Trusted Platform Module (TPM) provides secure cryptographic functions and hardware authentication (e.g., a processor manufactured by a known supplier) [37, 48, 92]. Thus, while TPM can serve as the root of trust, it relies on the assumption that a verified identity is sufficient for security. However, as described above, authenticated provenance of a hardware component does not ensure that it is secure.

This paper introduces a method to provide a stronger set of security guarantees for a complex system with only a single simple hardware component. This method builds on three key insights. First, checking that a computation is correct can be much simpler than performing the computation. Second, irreparable harm generally involves maliciously or erroneously constructed external communication. Third, checking the correctness of external communication is more practical than checking all state changes within the system.

This paper presents *TrustGuard*, a proof-of-concept architecture that enables the detection and *containment* of malicious behavior by untrusted components before results are externally visible. At the core of TrustGuard is the *Sentry*, a single simple component dedicated to security. The Sentry's simplicity and open design make it amenable to formal verification and allow it to serve as the basis of trust used to secure a system. In TrustGuard, the Sentry is the only path between the system and its external interfaces. Untrusted components must prove to the Sentry that any data sent externally is the result of correct execution of trusted and signed software.[1] This allows the Sentry to *contain* malicious behavior by untrusted hardware and software. While containment does not provide availability guarantees, it assures users that all output is only the result of verified execution.

[1]The correctness of signed software is orthogonal to the problem addressed by this paper. Extensions of this work can help secure software (§9).

The feasibility of containment-based security depends upon the simplicity of the trusted components and its impact on system performance. While the form of containment may change for different types of architectures, this work establishes the feasibility of containment-based security with a TrustGuard prototype protecting a system with a single-core processor. The key insight is that the untrusted processor and memory can do almost all of the work, including acting as control for the Sentry, and hold almost all of the state without compromising any containment guarantees.

The Sentry cannot independently execute programs. Instead, it relies on information sent by the processor to check program execution. Thus, the Sentry avoids much of the complexity of aggressive processor optimizations. The Sentry detects any erroneous or malicious behavior by untrusted components without trusting any information sent by the processor. TrustGuard checks the execution information sent by the processor using a combination of functional unit re-execution (§5.1) and a cryptographic memory integrity scheme (§5.3). The functional unit re-execution is similar to DIVA [9], which adds a checker pipeline stage to the processor to provide reliability guarantees (§3).

The execution information sent by the processor removes dependences between instructions and allows the Sentry to efficiently check the correctness of instructions in parallel (§5.2). In fact, doing so enables the Sentry to offer containment while operating at clock frequencies much lower than the frequency of the untrusted processor with minimal impact to system performance.

In summary, the contributions of this paper are:

- The containment model, a model in which a single simple, trusted hardware element, the Sentry, quarantines the malicious effects of untrusted components;
- TrustGuard, a proof-of-concept design (§5) that shows the viability of the Sentry in terms of performance (§7.1), energy (§7.4), and design complexity (§8); and
- A characterization of the threat model and security assurances provided by TrustGuard (§2 and §6).

2 Motivation

Today, users must take on faith that their hardware and software providers have built a system that will not betray them to malicious parties. While significant research has focused on securing the software stack, all such techniques rely on correct execution by the hardware. This means that hardware threats, found both in theory and in practice, can bypass any software security guarantees. Thus, a system is only as trustworthy as the underlying hardware.

2.1 Hardware Threats

One class of hardware threats come from malicious hardware backdoors inserted into the processor during design

or fabrication stages. The Illinois Malicious Processors contain shadow circuitry inserted at the design phase to enable privilege escalation, backdoor login, and password theft [47]. Becker et al. demonstrated how a hardware Trojan inserted during manufacture could compromise Intel's cryptographically secure Random Number Generator [15].

The problem does not end with malicious hardware modification. Inadvertent bugs within hardware components like functional units, coprocessors, memory, or on-chip networks also pose security threats to the system. For example, as stated in §1, a correctness bug can be exploited to weaken encryption and lead to leakage of sensitive data [17].

Memory-related bugs are also a threat. Kim et al. showed, in an attack called Rowhammer, that repeated accesses to an address can cause data corruption in nearby addresses in DRAM modules from three major manufacturers [46]. After the discovery of Rowhammer, Google's Project Zero team developed two proof-of-concept exploits of this vulnerability [78]. These exploits achieved privilege escalation and underlined the security implications of such hardware bugs.

There are also threats that exploit performance-enhancing features in the processor. For example, Meltdown [57] and Spectre [50] extract information across protection boundaries.

TrustGuard focuses on preventing any harmful effects of incorrect output in the face of all hardware threats mentioned above (§6). In general, TrustGuard can protect against incorrect program output caused by hardware Trojans, bugs, and other hardware security vulnerabilities (known and not-yet-known). The Sentry even detects the results of internal side-channel exploits, such as Meltdown and Spectre, if and when they influence the output of the system.

2.2 Threat Model

TrustGuard ensures that all output from the system is only the result of correctly executed signed software. The Sentry allows only such output to pass while blocking all other output. Thus, the output of non-signed software, such as malware, is not permitted to pass through the Sentry. The Sentry also prevents any errors in the execution of signed software caused by malicious interference or system errors from leaving the system.

TrustGuard does not provide any guarantees about availability or internal correctness of a system. Rather than preventing hardware and software from maliciously or incorrectly altering the computation of results, TrustGuard instead prevents any such interference from escaping the system via explicit external communication. TrustGuard considers all hardware components in the system other than the Sentry–including processor, memory, and peripherals–as untrusted and vulnerable to compromise.

TrustGuard does not protect against communication channels out of the system other than through the Sentry. TrustGuard does not protect against information leaked through the Sentry via covert channels or side channels (e.g., encoding of sensitive information in an energy usage pattern, long duration timing encodings, implicit information leaked by failures). Moreover, malicious adversaries are assumed not to have physical access to the Sentry nor the physical gap.

TrustGuard does not give any guarantees about vulnerabilities (or a lack thereof) in the signed software itself. The Sentry ensures that all output from the system is only the result of correctly executed signed software. The Sentry prevents results from the execution of unsigned software, including malicious interference with signed software, from being communicated externally. (See §9 for further discussion about extending this work to cover software vulnerabilities.)

3 Background

This section introduces various existing proposals, each of which possesses some desirable properties for providing a basis of trust in a complex system.

Redundant Execution for Security and Reliability. One traditional approach to building trustworthy systems from untrustworthy components employs redundant execution [11, 14]. In this approach, several untrustworthy components redundantly perform computation, and the system uses majority voting to detect erroneous behavior. Design diversity of redundant components makes a hardware bug or backdoor escaping detection less likely. However, the cost of creating such a system is quite high, making it attractive only for high-assurance and high-security systems, such as aircraft and military systems.

The redundant execution approach has also been used to build systems resilient to transient faults [6–10, 26, 36, 60, 62, 69, 70, 74, 81, 82, 87, 100, 102, 104, 112, 114]. DIVA [9] showed that it is possible to build a simple, redundant checker to detect errors in a processor's functional units and its communication channels with the register file and data cache.

While the introduction of a simple checker presents a promising approach, DIVA was not designed for and is not trivially extended to security. Architecturally, DIVA's checker is embedded in the processor's commit path, and thus both the checker and processor must be manufactured jointly. This makes the checker vulnerable to malicious changes during the processor's manufacturing. DIVA also relies on the processor to correctly communicate trace information to the checker. Consequently, the checker cannot tell if the instruction execution stream it receives is modified, for example by insertion or modification of instructions.

Additionally, DIVA does not provide any protections for memory and register files. It instead relies on ECC to detect any transient faults that may occur in these modules. This is obviously insufficient for security. Finally, simply moving DIVA off-chip is not a straightforward process, as there are many issues to consider, including the potentially high off-chip bandwidth required between the processor and checker.

Trusted Hardware Elements. One approach for building trustworthy systems is to incorporate a hardware monitoring element that acts as the foundation of trust upon which the security of the system is based. For example, instruction granularity monitoring co-processors such as FlexCore [33, 43], Raksha [32], and log-based lifeguards [27–29] utilize additional hardware to monitor software execution and detect software vulnerabilities. All these techniques either trust that the processor configured the monitoring hardware correctly or trust that the information flow through the hardware is correct. Thus, these techniques are all vulnerable to the effects of any backdoors introduced into the processor during manufacturing.

Many researchers have also proposed designs for secure processors as the basis of trust for the system [25, 30, 31, 56, 73, 85]. These works prevent attacks by relying on features provided by the secure processor. While securing processors is indeed easier than securing all hardware in the system, modern processor designs are too complex to be reliably verified [20, 52, 58]. Adding security features to processors, such as SGX [59] and TrustZone [4], increases their design complexity and makes them even more difficult to verify. Additionally, these solutions do not address malicious modifications made to these secure processors during manufacture.

Chip Integrity Verification. Hardware backdoors, often referred to as hardware Trojans, can be inserted at any phase of the chip manufacturing process—specification, register-transfer level (RTL) design, IP integration, physical design, and fabrication. Various defenses have been proposed against hardware Trojans including post-fabrication detection [1, 12, 21, 44, 51, 105, 107], run-time monitoring [98], and design-time deterrence [22, 41, 75, 84, 99, 110]. However, these techniques are typically not comprehensive. For instance, post-fabrication detection techniques that rely on logic testing cannot detect backdoors designed to stay dormant during post-fabrication testing [89, 98]. Furthermore, some of these techniques may also have a high runtime cost [98].

An alternate approach, called verifiable outsourcing, verifies the correct execution of an untrusted hardware component by using a trusted processor or ASIC as the verifier [94, 95]. However, this approach incurs a large performance overhead and has limited applicability. More generic protocols apply to a broader class of applications but have an even larger performance overhead ($10^5\times$–$10^7\times$) [16, 19, 63, 96].

4 The TrustGuard Approach

The goal of TrustGuard is to serve as a foundation of trust upon which trustworthy systems can be built by assuring users that all communication is verified before leaving the system. As shown in Figure 1, TrustGuard consists of an untrusted system isolated from its external interfaces by a Sentry spanning an otherwise physical gap. Thus, all external communication must pass through this Sentry, which

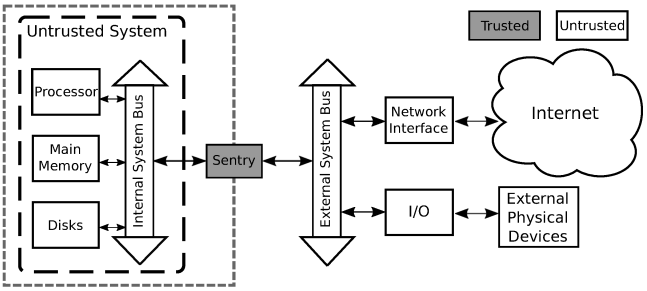

Figure 1. The TrustGuard Approach

verifies that all output from the system is the result of correct execution of signed programs. TrustGuard defines the *correctness of output* with respect to the instruction set architecture (ISA). By preventing the external communication of incorrect results or results of unsigned programs, TrustGuard *contains* within the system the effects of faulty or malicious components.

As checking results of processor execution is simpler than performing full processor execution (§8), a simple Sentry can keep up with a much faster, superscalar processor (§7). This model offers a higher performance alternative to using currently available formally verified processors [58, 83], which have simple, in-order pipelines and cannot provide performance comparable to superscalar processors. Moreover, security processors that have superscalar performance are too complex for formal verification [30, 56, 73, 85]. In TrustGuard, only the Sentry needs to be verified and secured; the complexity of the processor does not affect the security assurances given by TrustGuard.

5 The TrustGuard Architecture

Figure 2 shows a high-level, proof-of-concept design of the TrustGuard architecture. To allow output only from the correct execution of signed software, the Sentry in TrustGuard checks (1) correctness of instruction execution with respect to the ISA (including determining the next instruction in program order); and (2) that each instruction is part of a signed program. Policy dictates what to do upon detection of incorrect execution. TrustGuard supports many different incorrect execution policies including: halting execution; continuing execution while preventing erroneous output; alerting the user; and reporting the incorrect instruction. This proof-of-concept supports a system with a single-core processor running software, including an OS, signed by a trusted authority. Additionally, TrustGuard requires that all I/O originates from explicit I/O instructions.

5.1 Checking Instruction Correctness

Most of a modern processor's complexity resides in components that support aggressive optimizations, such as branch

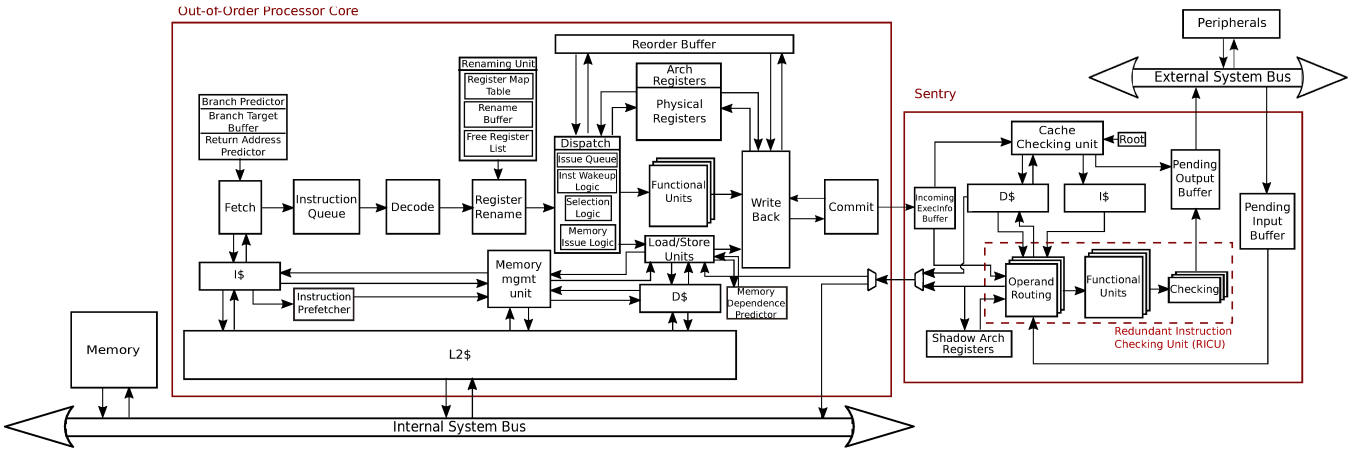

Figure 2. TrustGuard Architecture

prediction, memory dependence speculation, and out-of-order execution. These aggressive optimizations do not perform essential computation. Instead, they merely prepare instructions and data for computation as early as possible. The Sentry's operation more closely resembles functional unit execution than the processor's full instruction pipeline. In TrustGuard, the untrusted processor and memory do almost all of the work, including acting as control for the Sentry, and hold almost all of the state without compromising the Sentry's containment guarantees. To accomplish this, the untrusted processor is required to send information about committed state changes to the Sentry. This includes the results of all committing instructions. Sending information about committing state obviates the need to check the speculative work done by the processor.

Additionally, the sent execution information allows the Sentry to efficiently check the correctness of instruction execution with respect to the ISA. The method by which the Sentry checks correctness of instruction execution depends on the type of the instruction. For non-memory instructions, the Sentry uses its functional units to re-execute the operation specified by the instruction using the execution information provided by the processor. We call this *functional unit re-execution*. For memory instructions, however, functional unit re-execution would be quite expensive due to the cost of replicating the memory subsystem entirely. Instead, TrustGuard relies upon a cryptographic memory integrity scheme to transform memory operations into computations that can be easily checked (§5.3).

To ensure that the processor does not misreport execution information, the Sentry maintains a shadow register file, containing all architectural register state produced by the verified instruction sequence. Note that the Sentry's register file need only maintain reference values for the set of architectural registers. This set of architectural registers is typically much smaller than the set of physical registers used in an out-of-order processor [38]. The Sentry cannot execute programs independently but can verify the correctness of the received program execution information.

5.2 Redundant Instruction Checking Unit

The Redundant Instruction Checking Unit (RICU) is part of the Sentry, shown in Figure 2. The RICU performs functional unit re-execution of arithmetic, logic, and control instructions. The checking process itself is composed of three stages: (1) *Operand Routing* (OR), which retrieves the next instruction result to be checked from the `ExecInfo` buffer and determines the operands to be used for re-execution of that instruction; (2) *Value Generation* (VG), which re-executes the instructions using the Sentry's own functional units; and (3) *Checking* (CH), which compares the results to determine if the processor reported the correct value.

One cannot assume that trusted fabrication plants can produce chips using state-of-the-art fabrication technology. Thus, the Sentry is designed to be manufactured with older and slower technology while minimizing the performance impact on processors manufactured with the latest technology. The RICU in the Sentry supports this by checking multiple instructions in parallel regardless of dependences among those instructions. The Sentry breaks dependences by speculating that the *processor executes instructions correctly*, using unchecked values from the untrusted processor to check dependent instructions without delay. Misspeculation occurs when the untrusted system delivers malicious or erroneous results to the Sentry. Misspeculation is detected by the CH stage before the communication of those results are sent to the external interfaces (§5.5).

Figure 3 highlights the performance impact of the Sentry's parallel checking on the code in Figure 3(a) and its dependence graph in Figure 3(b). Figure 3(d) shows the Sentry's checking schedule with speculation-enabled parallel

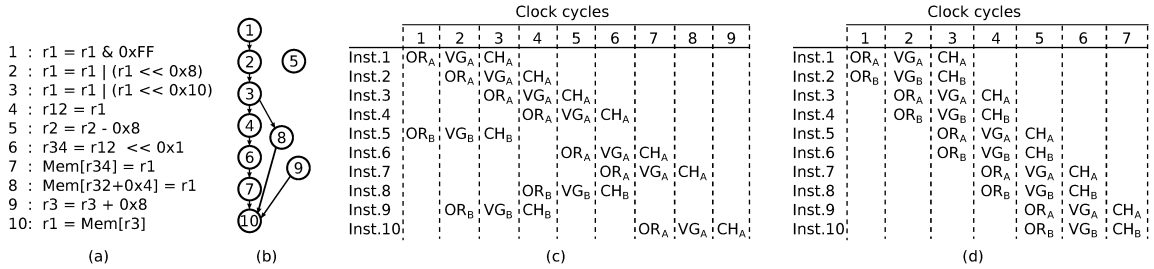

Figure 3. Example of speculation enabled parallel checking for a 2-wide Sentry: (a) Trace example from 456.hmmer; (b) Dependences between instructions in the trace; (c) Checking schedule with dependences respected; and (d) Checking schedule for speculation enabled parallel checking.

checking, a significant improvement over the schedule with dependences respected shown in Figure 3(c).

5.3 Memory Checking

TrustGuard checks the correctness of memory values accessed by untrusted system components using a memory integrity scheme based on Message Authentication Codes (MACs). Doing so allows TrustGuard to transform memory operations into computations that can be easily checked. To reduce complexity, the Sentry does not interface with a memory controller. Instead, the untrusted processor performs all memory accesses (including accesses to MACs and other metadata) and forwards data to and from the Sentry.

Bonsai Merkle Tree. Prior work in memory integrity assumes a secure processor that faithfully performs the cryptographic functions to ensure integrity [72, 85]. In TrustGuard, however, the sensitive cryptographic functions must be performed by the Sentry, as it is the only trusted component. With every cache line, TrustGuard associates a MAC—a keyed cryptographic hash of the address of the cache line, the data itself, and counters that maintain the version of the cache line. Additionally, TrustGuard builds a Bonsai Merkle Tree [72] on the counter blocks to protect the integrity of the counters, thereby preventing replay attacks. The root of the Merkle tree is stored in a special register in the Sentry. Figure 4 shows the structure of the Bonsai Merkle tree used by TrustGuard. We collectively refer to the metadata needed to verify integrity (i.e., MACs, counters, and intermediate Merkle tree nodes) as *shadow memory*.

Instead of using a single counter per data cache line, TrustGuard uses a split counter [108]. The counter is split into two, with one smaller minor counter per-cache line, and a larger major counter that is shared by a group of cache lines. Cache lines are divided into 2KB groups. The overflow of a minor counter requires incrementing the major counter and re-MACing of all the cache lines in the group. If the group counter overflows, the entire memory must be re-MACed with a different key. In TrustGuard, the minor counters are 14 bits long, and the major counters are 64 bits long. This configuration achieves a balance between counter size and

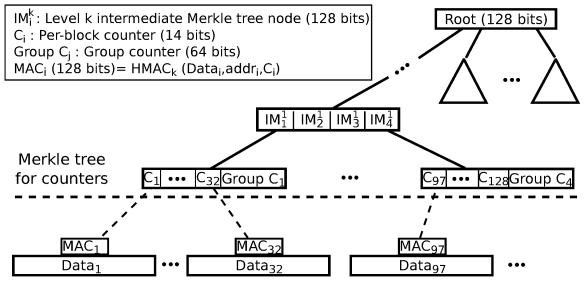

Figure 4. Bonsai Merkle Tree [72] used by TrustGuard to protect memory integrity.

the number of re-MACing operations; it also means that a group of 32 minor counters and their major counter fit in a single 64-byte cache line.

Cache Mirroring. While the use of the Bonsai Merkle tree adds protection against replay attacks, it also massively increases the number of memory accesses required to verify memory integrity as well as the bandwidth needed between the processor and the Sentry. To reduce the number of memory accesses, TrustGuard uses a *cache mirroring technique*.[2]

In cache mirroring, the processor and the Sentry have L1 data and instruction caches of the same configuration.[3] The untrusted processor forwards all fill and eviction information for its L1 caches to the Sentry. This allows the processor to act as an oracle prefetcher for the Sentry, ensuring that memory values are always available in the Sentry when needed. The processor is responsible for the fill, replacement, timing, and coordination with the rest of the memory subsystem. The Sentry merely checks that the processor's actions are correct and uses the cache as its local store.

In addition to program data and instructions, the Sentry requires the processor to send it the shadow memory needed to verify an incoming cache line's integrity. When the processor fetches a new line into its L1 cache, it also fetches any shadow memory not currently present in the cache for that line and forwards them to the Sentry. Many of the counters

[2]Other cache configurations are possible, mirroring was selected for both its effectiveness and its simplicity in implementation.

[3]The Sentry only has L1 caches, regardless of other levels in the processor.

and MACs in the Merkle tree nodes are likely already available in the processor's (and therefore also the Sentry's) L1 caches due to memory locality and adjacent placement of counters and MACs. These effects further reduce the communication bandwidth between the processor and Sentry.

Cache Checking Unit (CCU). Since the cache in the Sentry is trusted, each access to the Sentry's cache does not need to be verified by cryptographic operations, work which would dramatically reduce the performance of the system. Instead, only cache line insertion and eviction require cryptographic operations.

The CCU validates the integrity of inserted cache lines. Upon receiving new data or instruction cache lines from the processor, the Sentry speculatively stores the new cache line in the data or instruction cache marked as unchecked, allowing for the RICU to proceed with instruction checking without waiting for integrity to be verified. The Pending Output Buffer (§5.5) holds all subsequent output instructions until this speculative assumption is confirmed correct.

Next, the CCU re-computes the MACs for the cache lines using the data received and the counter values that were either already cached or received. It then checks the calculated MAC against the one reported by the processor. Similarly, if the delivered cache line contains counters, the Sentry must also check the MACs of the counters ($\texttt{IM}_{\texttt{i}}^{\texttt{1}}$ nodes in Figure 4), as well as all the intermediate Merkle tree nodes toward the root until a cached ancestor is found or the root node stored on the Sentry is reached. Once verification is complete, a confirmation signal is sent to the Pending Output Buffer (§ 5.5) to alert speculative output operations that they are no longer dependent on this instance of speculation.

The CCU updates the shadow memory state for evicted cache lines. Whenever a dirty data cache line is evicted from the Sentry's cache, the CCU increments counters, creates new MACs, and sends this new shadow memory state to the processor to be stored back to memory.

5.4 Link Compression

To reduce the communication between the processor and the Sentry, TrustGuard uses a hybrid of Significance-Width Compression (SWC) and Frequent Value Encoding (FVE) [90]. As the compression and decompression do not need to be trusted, it adds no complexity to the trusted logic of the Sentry.

5.5 Discussion of Other Issues

Program Loading The TrustGuard architecture requires programs to be signed by a trusted authority to communicate externally. To ensure the correct execution of signed programs, the Sentry contains a trusted program loader, similar to systems in previous software integrity work [34, 49]. Upon creating the Sentry, the manufacturer will generate the Sentry's secret key and the Merkle Tree metadata for the trusted program loader. The root of this tree will then be fused into a private static register on the Sentry. Upon initialization, the Sentry will load this value into the Merkle Tree root register to verify the trusted program loader. The Sentry checks the trusted program loader. In turn, the trusted program loader verifies the signatures of trusted programs as it loads them and their metadata into memory before starting their execution.

Interaction with Peripherals As shown in Figure 2, the Sentry resides physically between the untrusted processor and the peripherals. The Sentry prevents the results of unverified instructions from communicating to the peripherals via the *Pending Output Buffer* (POB). In TrustGuard, all I/O is the result of explicit I/O instruction execution. Only values that have been verified as correct by the RICU are stored in the POB. However, these checked output operations may be dependent on a cache line that is in the process of having its integrity checked, as described in §5.3. Therefore, the POB confirms that all speculatively filled cache lines that the output is dependent upon have been verified before allowing the output to proceed to a peripheral. Direct memory access (DMA) is compatible with containment-based security, but DMA is not implemented in this proof-of-concept.

Changes to Processor Design The nature of the interaction between the untrusted processor and the Sentry in TrustGuard requires several modifications to be made to the design of the untrusted processor. The processor and the Sentry must support the same ISA as TrustGuard defines correctness of instructions with respect to the ISA specifications. They must also have the same number of architectural registers. The untrusted processor must also support loading and storing of MACs, counters, and Merkle tree nodes from and to cache lines and memory. The processor additionally must send a trace of its execution information (cache lines, results, shadow memory accesses, and the processor's condition flags) to the Sentry. This communication is synchronized through the addition of two buffers to the processor's design: one `ExecInfo` buffer for the outgoing communication to the Sentry, and the other for shadow memory values received from the Sentry.

6 Attack Scenarios

To evaluate its containment capabilities, we modeled TrustGuard in the gem5 simulator [18] and implemented the following attack scenarios.

Incorrect Arithmetic Instruction Execution. These attacks involve the untrusted processor manipulating an arithmetic instruction's execution. In particular, we implemented the following cases: (1) Incorrect execution of arithmetic instruction, e.g., the multiplier bug compromising RSA [17]; (2) Modification of values in the register file. In the example from Figure 5, if the untrusted processor manipulates the result of `r1 = r2 * r1` to be any other value than `0x6000`, the Sentry's Redundant Instruction Checking

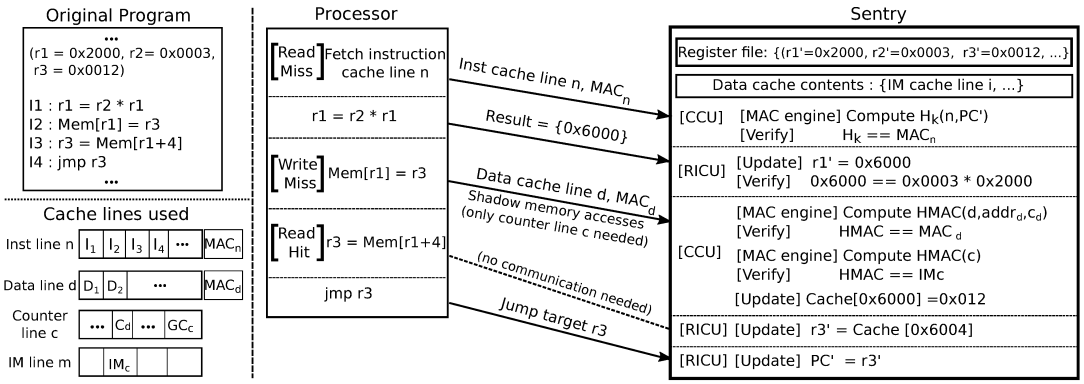

Figure 5. Example code checking by the Sentry. `rx'` in the figure indicates the shadow register in the Sentry for the register `rx` in the untrusted processor. H_k is the keyed cryptographic hash function used with key k.

Unit (RICU) detects the manipulation when it re-executes the `multiply` operation. Similarly, the untrusted processor could manipulate the instruction by changing the value of an operand. However, this illegal change in the value of the operand will not appear in the Sentry's shadow register file. Therefore, the result of the operation produced by the Sentry will differ from that reported by the processor and the manipulation will be detected.

Insertion of Malicious Instructions. These attack scenarios involve the following cases: (1) insertion of non-program instructions; (2) skipping execution of program instructions; and (3) reordering instructions in the processor's instruction stream incorrectly. Case (1) was described by King et al. in the context of Illinois Malicious Processors (IMP) [47]. All these cases were detected using the mechanism described below using the example from Figure 5.

Assume that the untrusted processor inserts `r2 = r2 + 0x1` just before `r1 = r2 * r1` to maliciously increment the value of `r2` as part of an attack. The processor can choose whether to send this instruction's result or not to the Sentry. If the processor sends the incorrect result (`0x0004`), the Sentry will re-execute the next instruction `r1 = r2 * r1` and detect a mismatch between the produced result (`0x6000`) and the received result (`0x0004`). If the processor does not send the instruction's result, on the next instruction the processor will send its multiplication result (`0x8000`) to the Sentry, and the attack will still be detected because it differs from the Sentry's result (`0x6000`). **The Sentry detects incorrect control flow by the processor in a similar fashion.**

Modification of Values in Memory. In the example of Figure 5, the processor (or some other malicious actor such as a rogue program exploiting the Rowhammer bug) could corrupt the data at `Mem[0x2000]` and change the value to be values other than 0x0012. If the cache line containing `Mem[0x2000]` is not in the Sentry's cache, the processor will send the cache line and the corresponding shadow memory to the Sentry. However, the attacker is not able to generate the correct shadow memory values, and the Sentry's cache checking unit (CCU) will detect this modification as it verifies the integrity of the incoming cache lines against its shadow memory.

If the cache line is in the Sentry's cache, the processor may continue executing using the wrong value it stored to memory; however, the Sentry will execute the subsequent instructions using the *correct* value recorded in its internal state. Thus, any output depending on the corrupted memory will be communicated correctly as long as the cache line is present in the Sentry. Upon cache line eviction, the Sentry will send a MAC of the cache line, assuming that the value stored was `0x0012`. The next time the line is loaded back into memory, the Sentry's CCU will discover the maliciously executed store due to a mismatch in the MAC values. **The Sentry detects modification of instructions in a similar fashion.**

Replay Attack Using Old Memory, Counter, and MACs. Another attack could be attempted by replaying old data, counter, and MACs already seen for a location in memory. However, upon cache line eviction, the counter is incremented, and the MAC is regenerated using the new counter. The Bonsai Merkle Tree scheme allows the Sentry to use its internal state to verify the correctness of the counter. Replaying old shadow memory values will result in a mismatch in either the counters or the MACs and thus be detected by the Sentry's CCU.

7 Performance Analysis

For performance analysis, we modeled TrustGuard in the gem5 simulator [18] using an out-of-order (OoO) ARM-based untrusted processor. Table 1 shows the architectural parameters used in the performance analysis. The default processor frequency is 2GHz (common on current Intel and ARM processors [3, 42]). The default bandwidth between the processor and the Sentry is 16GB/s (achievable using an interconnect such as 16-lane PCIe). We simulated the execution of all 8 SPECINT 2006 workloads that work with gem5, three

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

Table 1. Architectural parameters for simulation

additional SPECFP benchmarks: 450.soplex, 453.povray, and 470.lbm, as well as cryptographic operations from OpenSSL (AES & RSA encryption/decryption). For programs with short execution times – 445.gobmk, 450.soplex, 462.libquantum, and OpenSSL – we simulate whole programs. For all other benchmarks, we sample five random simulation checkpoints. For each checkpoint, benchmarks run in the simulation for 25 million instructions to warm up the microarchitectural state prior to a cycle-accurate simulation for 200 million instructions to collect performance data. For each experiment, the baseline is the out-of-order, superscalar processor-based system without any TrustGuard modifications (OoO only).

7.1 Performance of TrustGuard

Two factors impact the IPC of the untrusted processor in TrustGuard. The first factor is the increased cache and memory pressure from additional Merkle tree accesses. On average, the benchmarks performed 31.3% more accesses to memory. The second factor is the introduction of two new kinds of stalls in the untrusted processor: (1) *Slow Sentry Stalls*, due to the Sentry's inability to check instructions as fast as execution by the processor; and (2) *Bandwidth Stalls*, where the Sentry is kept waiting for execution information due to bandwidth limitations on the channel to the processor.

To demonstrate the performance implications of TrustGuard, we evaluated the effects of various design parameters, such as the RICU width (Figure 6), frequency of the processor and the Sentry (Figure 8), and the bandwidth between the processor and the Sentry (Figure 10), on the IPC of the untrusted processor.

Sentry Parallelism. Figure 6 shows the effect of varying the number of instructions checked in parallel by the Sentry (RICU width) on the IPC of the untrusted processor. The geomean decline in IPC for RICU widths of 2, 4, 6, and 8 was 41.6%, 16.8%, 15.4%, and 15.2% respectively. The higher RICU width resulted in higher checking throughput on the Sentry, leading to improved performance. This is borne out by the decrease in the number of Slow Sentry Stalls as the RICU width increases (Figure 7). The average percentage of Slow Sentry Stalls experienced was 30.76%, 2.13%, 0.30%,

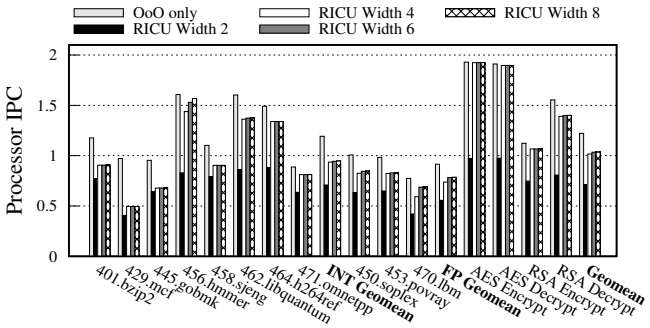

Figure 6. IPC while varying the Sentry's RICU widths (Processor Frequency=2GHz, Sentry Frequency=1GHz, Bandwidth=16GB/s).

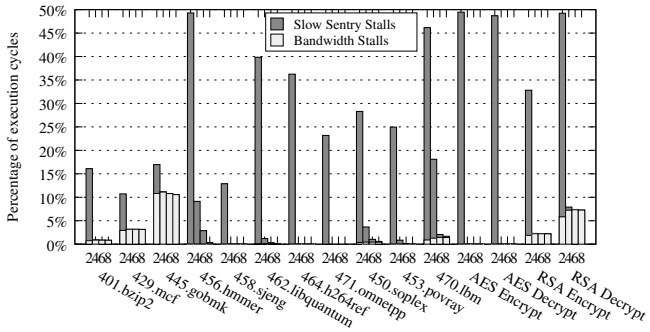

Figure 7. Stalls induced by the Sentry while varying the Sentry's RICU widths (same configuration as Figure 6). X-axis labels indicate the RICU widths.

and 0.05% respectively for RICU widths of 2, 4, 6, and 8. The effect of increasing the RICU width was especially visible for benchmarks with higher baseline IPCs. For example, for 456.hmmer, going from RICU width 2 to 4 reduced the percentage of Slow Sentry cycles from 49.29% to 9.12%.

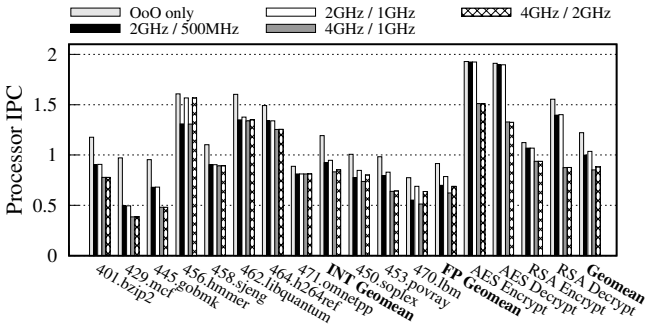

Figure 8. IPC while varying the Processor/Sentry frequency (RICU width=8, Bandwidth=16GB/s).

Processor/Sentry Clock Frequency. One of the main insights behind TrustGuard is that a Sentry running at lower clock frequency can verify the execution of instructions by

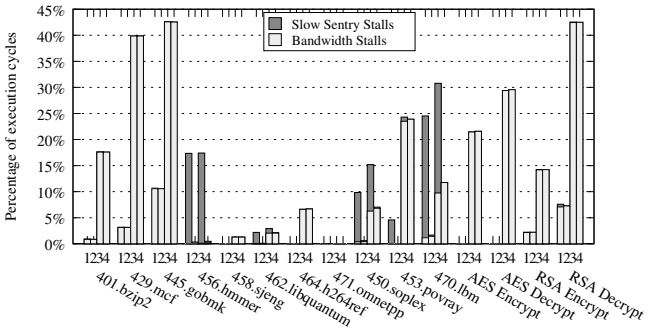

Figure 9. Stalls induced by the Sentry while varying the frequencies (same configuration as Figure 8). X-axis labels indicate the Processor/Sentry frequency: 1=2GHz/500MHz, 2=2GHz/1GHz, 3=4GHz/1GHz, 4=4GHz/2GHz.

the untrusted processor without impacting its performance too adversely. Figure 8 shows the effect of varying the clock frequency at which the processor and the Sentry operates. The Sentry's throughput increased at higher Sentry to processor frequency ratio, leading to better performance. Compared to the baseline, the 15 benchmarks showed a geomean IPC reduction of 18.3% at 2GHz Processor & 500MHz Sentry, 15.2% at 2GHz / 1GHz, 30.2% at 4GHz / 1GHz, and 27.9% at 4GHz / 2GHz.

Figure 9 shows the number of bandwidth and slow Sentry stalls experienced by the processor while varying the frequency of the processor and the Sentry. The average percentage of Slow Sentry Stalls reduced from 3.84% at 2GHz / 500MHz to 0.05% at 2GHz / 1GHz Sentry and from 3.28% at 4GHz / 1GHz Sentry to 0.04% at 4GHz / 2GHz.

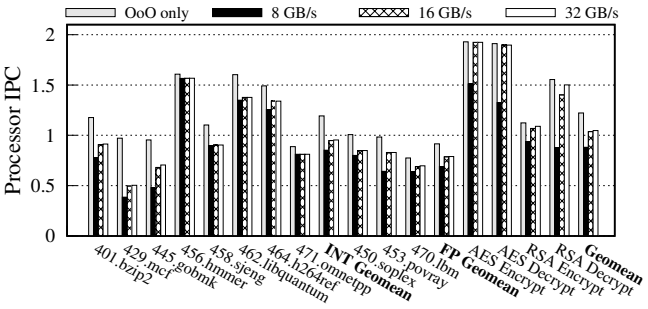

Figure 10. IPC while varying the bandwidth (Processor Frequency=2GHz, RICU width=8, Sentry Frequency=1GHz).

Bandwidth. As shown in Figure 7 and Figure 9, the average percentage of bandwidth stalls remains quite stable because the communication depends on program characteristics. Figure 10 shows the effect of varying bandwidth on the IPC of the untrusted processor, while Figure 11 presents the percentage of Slow Sentry and Bandwidth Stalls incurred for the resulting configurations. The geomean IPC decline was 27.9% at 8GB/s, 15.2% at 16GB/s and 14.2% at 32GB/s. The

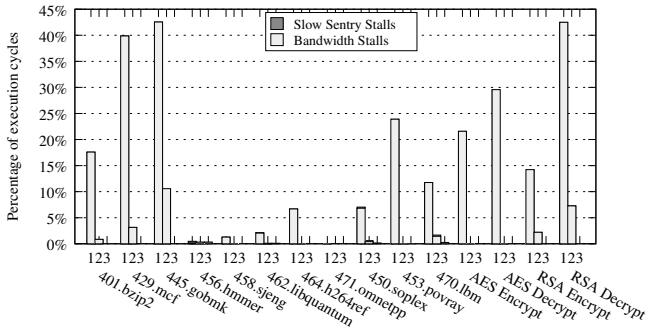

Figure 11. Stalls induced by the Sentry while varying the bandwidth (same configuration as Figure 10). X-axis labels indicate the bandwidth: 1=8GB/s, 2=16GB/s, 3=32GB/s.

corresponding average of the percentage of bandwidth stalls was 17.36% at 8GB/s, 1.73% at 16GB/s, and 0.00% at 32GB/s. With cache mirroring, the processor need not send cache data to the Sentry on L1 cache hits. Therefore, programs with greater cache locality will save on communication. For example, at 16GB/s, 445.gobmk with an L1 data cache hit rate of 83.5% incurred bandwidth stalls for 10.57% of execution cycles. By contrast, 456.hmmer with 98.7% data cache hit rate incurred bandwidth stalls for only 0.004% of execution cycles.

7.2 Link Utilization

When the processor-Sentry link is 16GB/s, the geomean bandwidth usage across the eleven benchmarks is 9.2GB/s. The highest usage is in 445.gobmk (12.4GB/s) while the lowest is in 471.omnetpp (4.84GB/s). As for instantaneous bandwidths (bandwidth used in a particular cycle), 445.gobmk uses more than 12GB/s of instantaneous bandwidth for 75% of execution cycles while benchmarks like 471.omnetpp and 458.sjeng use less than 4GB/s for more than 60% of execution cycles.

7.3 Instruction Verification Latency

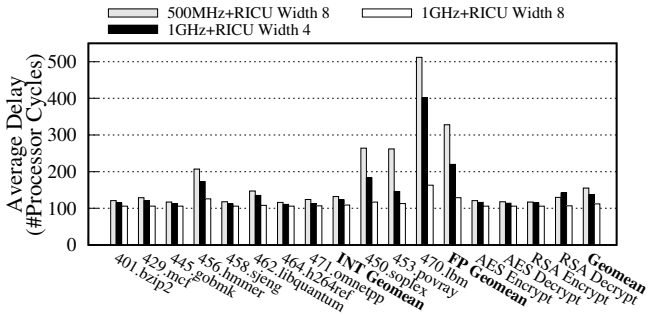

Figure 12. Average latency between committing of an instruction in the processor and its checking by the Sentry (Processor Frequency=2GHz, Bandwidth=16GB/s).

Output operations in TrustGuard cannot proceed until the Sentry verifies them. Figure 12 shows the average latency for each instruction from when untrusted processor commits it to when the Sentry verifies it. This metric is the average output operation delay. With increased Sentry parallelism and frequency, the throughput of checking increases, which results in a decline in the average latency. The geomean average latency for each instruction is 155 processor cycles (77.5ns) at 500MHz and RICU width 8, 137 processor cycles (68.5ns) at 1GHz and RICU width 4, and 112 processor cycles (56ns) at 1GHz and RICU width 8. Note that every instruction incurs a latency of at least 100 processor cycles (the latency of off-chip communication).

There is a clear difference between the SPECINT and SPECFP benchmarks. At 1GHz and RICU width 4, SPECINT has a geomean latency of 123 CPU cycles (61.5ns), while SPECFP has a geomean latency of 220 CPU cycles (110ns). SPECFP's higher latency comes from the higher latency of floating point operations compared to integer operations, which is magnified by the fact the Sentry runs at half the frequency of the untrusted processor.

7.4 Energy

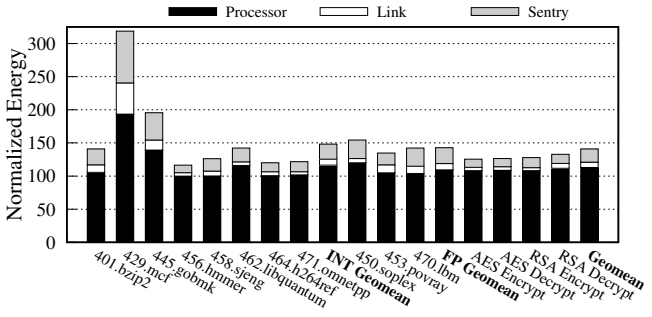

Figure 13. TrustGuard's energy usage. Processor Frequency=2GHz, RICU Width=8, Sentry Frequency=1GHz, Bandwidth=16GB/s

We used McPAT v1.2 [55] to model the energy of the TrustGuard processor and Sentry using execution statistics from the performance simulation. Power for the MAC engines was estimated using an HMAC-MD5 accelerator [101], adapted to our design using technology scaling. Figure 13 shows the energy consumption of TrustGuard, normalized to the energy consumption of the baseline untrusted processor. The geomean energy consumption for TrustGuard was 41.0% greater than the baseline, while instantaneous power was 16.0% greater than the baseline. The untrusted processor in TrustGuard showed a geomean 13.0% higher energy consumption than the baseline processor. The geomean energy consumption of the Sentry itself is 19.9% of the energy consumption of the baseline processor, which is significantly lower than the 100+% increase that would come from a second redundant processor. The main factors for Sentry's lower energy consumption compared to the untrusted processor are its lower frequency, the absence of an L2 cache, and the absence of the OoO support structures. Furthermore, the link–modeled as a PCIe interconnect [88] consumes a geomean 8.0% of the energy of the untrusted processor. 429.mcf saw a large energy overhead due to its memory intensive nature and, the majority of the overhead comes from the Merkle tree accesses and hash computations.

8 Simplicity of the Sentry

Only the Sentry must be secured to ensure containment of the entire system. The simpler the Sentry, the more confidence there will be in its containment guarantees. To evaluate the design complexity of the Sentry, we use the observation by Bazeghi et al. [13] that lines of HDL code serve as a good approximation of a design's complexity. For this purpose, we built an FPGA prototype of the Sentry supporting the RISC-V user-level ISA [71]. We chose RISC-V due to the unavailability of a full, open-source ARM processor. We synthesized the Sentry design onto a NetFPGA SUME FPGA Board using Xilinx Vivado 17.2.

Table 2 shows the lines of code (LoC) our prototype Sentry, along with the reported size of various open-source processors. The Sentry's design complexity compares favorably to in-order processors, some of which have been formally verified [20, 58, 64, 83]. Table 2 shows that the Sentry prototype's LoC is an order of magnitude less than that of the out-of-order (OoO) processors. Concerning area, we found that a single-core BOOM configuration uses ~4× the number of LUTs and ~3× the number of flip-flops compared to the Sentry [71].

The Sentry is simpler than the processor in many ways. It lacks both cache and memory controllers. Functional unit re-execution in the Sentry is simpler than instruction execution in a processor. The Sentry does not include out-of-order execution, branch predictors, memory dependence predictors, register renaming units, dispatch units, reorder buffers, multiple cache levels, load/store queues, inter-stage forwarding logic, bypass networks, memory control, and misspeculation recovery support. Note that most of these components optimize overall performance rather than perform the actual execution of instructions. The Sentry is incapable of initiating instruction execution on its own. Instead, it relies on the processor to direct its work.

The cache checking unit (CCU) and parallel checking are optimizations to the Sentry worth the added complexity. The size and design complexity of the MAC engines in the CCU is comparable to other functional units already present on both the processor and the Sentry, and MAC engines have been formally verified [91]. The processor performs much of the Merkle tree and cache control logic for the Sentry,

Processor	**Leon3** [13]		**PUMA** [13]		**IVM** [13]		**BOOM** [71]		**Sentry**	
Description	In-Order		OoO no FP		OoO no FP		OoO		This Work	
Language	VHDL		Verilog		Verilog		Chisel		Verilog	
Lines of Code (LoC) for various components	Pipeline	2814	Fetch	1490	Fetch	4972	Fetch	1974	OR	425
	Memory	4456	Decode	3416	Decode	963	Decode	650	VG	392
			ROB	913	Rename	2519	ROB	709	CH	60
			Execute	9613	Issue	2704	Rename	456	CCU	1030
			Memory	2251	Execute	4083	Issue	356	FPU	1636
					Retire	2278	Execute	3898		
					Memory	5308	Memory	7407		
	Total	**7270**	**Total**	**17683**	**Total**	**22827**	**Total**	**15450**	**Total**	**3543**

Table 2. Comparison of implementation complexities in terms of lines of code (LoC) of various open-source processor designs against that of the Sentry prototype. Note: Processor proof of correctness would not secure memory nor program integrity.

thus further reducing the Sentry's complexity (§5.3). Parallel checking requires relatively simple forwarding logic between functional units compared to pipeline forwarding in OoO engines.

9 Conclusion and Future Work

This paper proved the viability of the containment-based security approach. The TrustGuard proof-of-concept implementation showed that a separate, simple Sentry can validate the execution of a processor with less than 15% geomean impact on performance. These results motivate further exploration of containment-based security techniques, tools, and implementations.

Programmer-Enabled Selective Checking Not all of a trusted program needs to be validated by the Sentry to ensure the correctness of external communication. For example, the Sentry does not need to check the execution of any program that does not produce external communication.

Prior work has shown that a small piece of trusted code can be used to validate the result of a large program [53, 63, 111, 113]. This has inspired the exploration of a Sentry programming model, where the validation code is used to validate the execution of untrusted parts of the program and the Sentry checks the validation code itself. The creation of such a Sentry programming model would allow programmers, perhaps with the help of tools, to divide their programs into trusted and untrusted regions or to create checking code to validate results produced by untrusted code prior to output. Initial exploration has shown that SAT solvers [61], filesystems [35], databases [67, 111, 113], and iterative algorithms [80] are good candidates for applying such selective checking mechanism to improve performance.

Multicore Support A natural next step of any single-core feature is the extension to multicore. There are many ways to support multicore, and we leave the exploration of that space, for now, as work inspired by the success reported in this paper. This space includes supporting multicore with a Sentry per core. This method would increase the required bandwidth between the processor and the Sentries to the point of infeasibility. Since the Sentries would likely need to be placed on the same die, the threat model of the system will change.

Another possibility is to use selective checking to support validation of multithreaded programs, where trusted code running on a single thread can validate the result of a multithreaded program. Witness generating SAT solvers provide an example for such a system. Such SAT solvers produce a witness that is validated by a verifier [61]. A single threaded verifier, checked by the Sentry, could ensure correct output by checking the proof witness generated rapidly by an untrusted multithreaded SAT solver.

Cache Policy and Sentry Independence The current L1 cache mirroring scheme works well to simplify the interface between the processor and Sentry. It also reduces the overhead of managing the Sentry's cache. A processor-independent Sentry is desirable for many reasons, including portability and reusability, but would require decoupling the cache designs. An independent cache design also creates new optimization opportunities. For example, the processor could serve as an oracle for the Sentry since it is typically hundreds of instructions ahead. Thus, a more optimized design may, without loss of correctness guarantees, replace the cache on the Sentry with a scratchpad memory.

Acknowledgments

We thank the Liberty Research Group for their support and feedback during this work. We also thank the anonymous reviewers for their insightful comments and suggestions. This material is based upon work supported by the National Science Foundation under grant numbers CNS-1441650 and CCF-1814654. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. A patent application related to this work exists [5].

References

[1] Dakshi Agrawal, Selcuk Baktir, Deniz Karakoyunlu, Pankaj Rohatgi, and Berk Sunar. 2007. Trojan Detection Using IC Fingerprinting. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy (SP '07)*. IEEE Computer Society, Washington, DC, USA, 296–310. https://doi.org/10.1109/SP.2007.36

[2] Aharon Aharon, Dave Goodman, Moshe Levinger, Yossi Lichtenstein, Yossi Malka, Charlotte Metzger, Moshe Molcho, and Gil Shurek. 1995. Test Program Generation for Functional Verification of PowerPC Processors in IBM. In *Proceedings of the 32Nd Annual ACM/IEEE Design Automation Conference (DAC '95)*. ACM, New York, NY, USA, 279–285. https://doi.org/10.1145/217474.217542

[3] ANANDTECH.Com. 2018. Arm's Cortex-A76 CPU Unveiled: Taking Aim at the Top for 7nm. Retrieved January 16, 2019 from https://www.anandtech.com/show/12785/arm-cortex-a76-cpu-unveiled-7nm-powerhouse

[4] Arm.Com. 2009. Building a Secure System using TrustZone Technology. Retrieved January 16, 2019 from http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf

[5] David I. August, Soumyadeep Ghosh, and Jordan Fix. 2015. "Trust architecture and related methods", U.S. Provisional Pat. Ser. No. 15/518,681, Filed October 21, 2015.

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

[7] Todd Austin and Valeria Bertacco. 2005. Deployment of Better Than Worst-Case Design: Solutions and Needs. In *Proceedings of the 2005 International Conference on Computer Design (ICCD '05)*. IEEE Computer Society, Washington, DC, USA, 550–558. https://doi.org/10.1109/ICCD.2005.43

[8] Todd Austin, Valeria Bertacco, David Blaauw, and Trevor Mudge. 2005. Opportunities and Challenges for Better Than Worst-case Design. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference (ASP-DAC '05)*. ACM, New York, NY, USA, 2–7. https://doi.org/10.1145/1120725.1120878

[9] Todd M. Austin. 1999. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 32)*. IEEE Computer Society, Washington, DC, USA, 196–207. http://dl.acm.org/citation.cfm?id=320080.320111

[10] Todd M. Austin. 2004. Designing Robust Microarchitectures. In *Proceedings of the 41st Annual Design Automation Conference (DAC '04)*. ACM, New York, NY, USA, 78–78. https://doi.org/10.1145/996566.996591

[11] A. Avizienis. 1985. The N-Version Approach to Fault-Tolerant Software. *IEEE Trans. Softw. Eng.* 11, 12 (Dec. 1985), 1491–1501. https://doi.org/10.1109/TSE.1985.231893

[12] Mainak Banga and Michael S. Hsiao. 2008. A Region Based Approach for the Identification of Hardware Trojans. In *Proceedings of the 2008 IEEE International Workshop on Hardware-Oriented Security and Trust (HST '08)*. IEEE Computer Society, Washington, DC, USA, 40–47. https://doi.org/10.1109/HST.2008.4559047

[13] Cyrus Bazeghi, Francisco J. Mesa-Martinez, and Jose Renau. 2005. uComplexity: Estimating Processor Design Effort. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 38)*. IEEE Computer Society, Washington, DC, USA, 209–218. https://doi.org/10.1109/MICRO.2005.37

[14] Mark Beaumont, Bradley Hopkins, and Tristan Newby. 2012. SAFER PATH: Security Architecture Using Fragmented Execution and Replication for Protection Against Trojaned Hardware. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '12)*. EDA Consortium, San Jose, CA, USA, 1000–1005. http://dl.acm.org/citation.cfm?id=2492708.2492958

[15] Georg T. Becker, Francesco Regazzoni, Christof Paar, and Wayne P. Burleson. 2013. Stealthy Dopant-level Hardware Trojans. In *Proceedings of the 15th International Conference on Cryptographic Hardware and Embedded Systems (CHES'13)*. Springer-Verlag, Berlin, Heidelberg, 197–214. https://doi.org/10.1007/978-3-642-40349-1_12

[16] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Succinct Non-interactive Zero Knowledge for a Von Neumann Architecture. In *Proceedings of the 23rd USENIX Conference on Security Symposium (SEC'14)*. USENIX Association, Berkeley, CA, USA, 781–796. http://dl.acm.org/citation.cfm?id=2671225.2671275

[17] Eli Biham, Yaniv Carmeli, and Adi Shamir. 2008. Bug Attacks. In *Proceedings of the 28th Annual Conference on Cryptology: Advances in Cryptology (CRYPTO 2008)*. Springer-Verlag, Berlin, Heidelberg, 221–240. https://doi.org/10.1007/978-3-540-85174-5_13

[18] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[19] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. 2013. Verifying Computations with State. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 341–357. https://doi.org/10.1145/2517349.2522733

[20] Jerry R. Burch and David L. Dill. 1994. Automatic Verification of Pipelined Microprocessor Control. In *Proceedings of the 6th International Conference on Computer Aided Verification (CAV '94)*. Springer-Verlag, London, UK, UK, 68–80. http://dl.acm.org/citation.cfm?id=647763.735662

[21] M. Bushnell and Vishwani Agrawal. 2013. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Springer Publishing Company, Incorporated.

[22] Rajat Subhra Chakraborty and Swarup Bhunia. 2009. HARPOON: An Obfuscation-based SoC Design Methodology for Hardware Protection. *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 28, 10 (Oct. 2009), 1493–1502. https://doi.org/10.1109/TCAD.2009.2028166

[23] Rajat Subhra Chakraborty and Swarup Bhunia. 2009. Security Against Hardware Trojan Through a Novel Application of Design Obfuscation. In *Proceedings of the 2009 International Conference on Computer-Aided Design (ICCAD '09)*. ACM, New York, NY, USA, 113–116. https://doi.org/10.1145/1687399.1687424

[24] Rajat Subhra Chakraborty, Somnath Paul, and Swarup Bhunia. 2008. On-demand Transparency for Improving Hardware Trojan Detectability. In *Proceedings of the 2008 IEEE International Workshop on Hardware-Oriented Security and Trust (HST '08)*. IEEE Computer Society, Washington, DC, USA, 48–50. https://doi.org/10.1109/HST.2008.4559048

[25] D. Champagne and R. B. Lee. 2010. Scalable architectural support for trusted software. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. 1–12. https://doi.org/10.1109/HPCA.2010.5416657

[26] Saugata Chatterjee, Chris Weaver, and Todd Austin. 2000. Efficient Checker Processor Design. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 33)*. ACM, New York, NY, USA, 87–97. https://doi.org/10.1145/360128.360139

[27] Shimin Chen, Babak Falsafi, Phillip B. Gibbons, Michael Kozuch, Todd C. Mowry, Radu Teodorescu, Anastassia Ailamaki, Limor Fix, Gregory R. Ganger, Bin Lin, and Steven W. Schlosser. 2006. Log-based Architectures for General-purpose Monitoring of Deployed Code. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability (ASID '06)*. ACM, New York, NY, USA, 63–65. https://doi.org/10.1145/1181309.1181319

[28] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, and Todd C. Mowry. 2011. Log-based Architectures: Using Multicore to Help Software Behave Correctly. *SIGOPS Oper. Syst. Rev.* 45, 1 (Feb. 2011), 84–91. https://doi.org/10.1145/1945023.1945034

[29] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. 2008. Flexible Hardware Acceleration for Instruction-Grain Program Monitoring. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. IEEE Computer Society, Washington, DC, USA, 377–388. https://doi.org/10.1109/ISCA.2008.20

[30] Siddhartha Chhabra, Yan Solihin, Reshma Lal, and Matthew Hoekstra. 2010. Transactions on Computational Science VII. Springer-Verlag, Chapter An Analysis of Secure Processor Architectures. http://dl.acm.org/citation.cfm?id=1880392.1880400

[31] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Proceedings of the 25th USENIX Conference on Security Symposium (SEC'16)*. USENIX Association, Berkeley, CA, USA, 857–874. http://dl.acm.org/citation.cfm?id=3241094.3241161

[32] Michael Dalton, Hari Kannan, and Christos Kozyrakis. 2007. Raksha: A Flexible Information Flow Architecture for Software Security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*. ACM, New York, NY, USA, 482–493. https://doi.org/10.1145/1250662.1250722

[33] Daniel Y. Deng, Daniel Lo, Greg Malysa, Skyler Schneider, and G. Edward Suh. 2010. Flexible and Efficient Instruction-Grained Run-Time Monitoring Using On-Chip Reconfigurable Fabric. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*. IEEE Computer Society, Washington, DC, USA, 137–148. https://doi.org/10.1109/MICRO.2010.17

[34] Milenko Drinic and Darko Kirovski. 2004. A Hardware-Software Platform for Intrusion Prevention. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 37)*. IEEE Computer Society, Washington, DC, USA, 233–242. https://doi.org/10.1109/MICRO.2004.2

[35] Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. 2012. Recon: Verifying File System Consistency at Runtime. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*. USENIX Association, Berkeley, CA, USA, 7–7. http://dl.acm.org/citation.cfm?id=2208461.2208468

[36] Mohamed A. Gomaa, Chad Scarbrough, T. N. Vijaykumar, and Irith Pomeranz. 2003. Transient-Fault Recovery for Chip Multiprocessors. *IEEE Micro* 23, 6 (Nov. 2003), 76–83. https://doi.org/10.1109/MM.2003.1261390

[37] David Grawrock. 2009. *Dynamics of a Trusted Platform: A Building Block Approach* (1st ed.). Intel Press.

[38] John L. Hennessy and David A. Patterson. 2017. *Computer Architecture, Sixth Edition: A Quantitative Approach* (6th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[39] Matthew Hicks, Murph Finnicum, Samuel T. King, Milo M. K. Martin, and Jonathan M. Smith. 2010. Overcoming an Untrusted Computing Base: Detecting and Removing Malicious Hardware Automatically. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP '10)*. IEEE Computer Society, Washington, DC, USA, 159–172. https://doi.org/10.1109/SP.2010.18

[40] Warren A. Hunt, Jr. 1989. Microprocessor Design Verification. *J. Autom. Reason.* 5, 4 (Nov. 1989), 429–460. http://dl.acm.org/citation.cfm?id=83471.83476

[41] Frank Imeson, Ariq Emtenan, Siddharth Garg, and Mahesh V. Tripunitara. 2013. Securing Computer Hardware Using 3D Integrated Circuit (IC) Technology and Split Manufacturing for Obfuscation. In *Proceedings of the 22Nd USENIX Conference on Security (SEC'13)*. USENIX Association, Berkeley, CA, USA, 495–510. http://dl.acm.org/citation.cfm?id=2534766.2534809

[42] Intel.Com. 2017. Intel Xeon Platinum 8176 Processor. Retrieved January 16, 2019 from https://ark.intel.com/products/120508/Intel-Xeon-Platinum-8176-Processor-38-5M-Cache-2-10-GHz

[43] M. Ismail and G. E. Suh. 2012. Fast development of hardware-based run-time monitors through architecture framework and high-level synthesis. In *2012 IEEE 30th International Conference on Computer Design (ICCD)*. 393–400. https://doi.org/10.1109/ICCD.2012.6378669

[44] Yier Jin and Yiorgos Makris. 2008. Hardware Trojan Detection Using Path Delay Fingerprint. In *Proceedings of the 2008 IEEE International Workshop on Hardware-Oriented Security and Trust (HST '08)*. IEEE Computer Society, Washington, DC, USA, 51–57. https://doi.org/10.1109/HST.2008.4559049

[45] Christoph Kern and Mark R. Greenstreet. 1999. Formal Verification in Hardware Design: A Survey. *ACM Trans. Des. Autom. Electron. Syst.* 4, 2 (April 1999), 123–193. https://doi.org/10.1145/307988.307989

[46] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 361–372. http://dl.acm.org/citation.cfm?id=2665671.2665726

[47] Samuel T. King, Joseph Tucek, Anthony Cozzie, Chris Grier, Weihang Jiang, and Yuanyuan Zhou. 2008. Designing and Implementing Malicious Hardware. In *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats (LEET'08)*. USENIX Association, Berkeley, CA, USA, Article 5, 8 pages. http://dl.acm.org/citation.cfm?id=1387709.1387714

[48] Steven L. Kinney. 2006. *Trusted Platform Module Basics: Using TPM in Embedded Systems*. Newnes, Newton, MA, USA.

[49] Darko Kirovski, Milenko Drinić, and Miodrag Potkonjak. 2002. Enabling Trusted Software Integrity. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*. ACM, New York, NY, USA, 108–120. https://doi.org/10.1145/605397.605409

[50] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[51] F. Koushanfar and A. Mirhoseini. 2011. A Unified Framework for Multimodal Submodular Integrated Circuits Trojan Detection. *Trans. Info. For. Sec.* 6, 1 (March 2011), 162–174. https://doi.org/10.1109/TIFS.2010.2096811

[52] William K. Lam. 2008. *Hardware Design Verification: Simulation and Formal Method-Based Approaches* (1st ed.). Prentice Hall PTR, Upper Saddle River, NJ, USA.

[53] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[54] Jie Li and John Lach. 2008. At-speed Delay Characterization for IC Authentication and Trojan Horse Detection. In *Proceedings of the 2008 IEEE International Workshop on Hardware-Oriented Security and Trust (HST '08)*. IEEE Computer Society, Washington, DC, USA, 8–14. https://doi.org/10.1109/HST.2008.4559038

[55] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, New York, NY, USA, 469–480. https://doi.org/10.1145/1669112.1669172

[56] David Lie, John Mitchell, Chandramohan A. Thekkath, and Mark Horowitz. 2003. Specifying and Verifying Hardware for Tamper-Resistant Software. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy (SP '03)*. IEEE Computer Society, Washington, DC, USA, 166–. http://dl.acm.org/citation.cfm?id=829515.830564

[57] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC'18)*. USENIX Association, Berkeley, CA, USA, 973–990. http://dl.acm.org/citation.cfm?id=3277203.3277276

[58] Panagiotis Manolios and Sudarshan K. Srinivasan. 2004. Automatic Verification of Safety and Liveness for XScale-Like Processor Models Using WEB Refinements. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1 (DATE '04)*. IEEE Computer Society, Washington, DC, USA, 10168–. http://dl.acm.org/citation.cfm?id=968878.969049

[59] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '13)*. ACM, New York, NY, USA, Article 10, 1 pages. https://doi.org/10.1145/2487726.2488368

[60] Maher Mneimneh, Fadi Aloul, Chris Weaver, Saugata Chatterjee, Karem Sakallah, and Todd Austin. 2001. Scalable Hybrid Verification of Complex Microprocessors. In *Proceedings of the 38th Annual Design Automation Conference (DAC '01)*. ACM, New York, NY, USA, 41–46. https://doi.org/10.1145/378239.378265

[61] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. 2001. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Annual Design Automation Conference (DAC '01)*. ACM, New York, NY, USA, 530–535. https://doi.org/10.1145/378239.379017

[62] Nahmsuk Oh, Subhasish Mitra, and Edward J. McCluskey. 2002. ED4I: Error Detection by Diverse Data and Duplicated Instructions. *IEEE Trans. Comput.* 51, 2 (Feb. 2002), 180–199. https://doi.org/10.1109/12.980007

[63] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. 2013. Pinocchio: Nearly Practical Verifiable Computation. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. IEEE Computer Society, Washington, DC, USA, 238–252. https://doi.org/10.1109/SP.2013.47

[64] V. A. Patankar, A. Jain, and R. E. Bryant. 1999. Formal verification of an ARM processor. In *Proceedings Twelfth International Conference on VLSI Design. (Cat. No.PR00013)*. 282–287. https://doi.org/10.1109/ICVD.1999.745161

[65] Devendra Rai and John Lach. 2009. Performance of Delay-based Trojan Detection Techniques Under Parameter Variations. In *Proceedings of the 2009 IEEE International Workshop on Hardware-Oriented Security and Trust (HST '09)*. IEEE Computer Society, Washington, DC, USA, 58–65. https://doi.org/10.1109/HST.2009.5224966

[66] S. P. Rajan, N. Shankar, and M. K. Srivas. 1997. Industrial Strength Formal Verification Techniques for Hardware Designs. In *Proceedings of the Tenth International Conference on VLSI Design: VLSI in Multimedia Applications (VLSID '97)*. IEEE Computer Society, Washington, DC, USA, 208–. http://dl.acm.org/citation.cfm?id=523974.834850

[67] Redis.Io. 2018. Redis. Retrieved January 16, 2019 from http://redis.io

[68] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. 2016. *End-to-End Verification of Processors with ISA-Formal*. Springer International Publishing, Cham, 42–58. https://doi.org/10.1007/978-3-319-41540-6_3

[69] Steven K. Reinhardt and Shubhendu S. Mukherjee. 2000. Transient Fault Detection via Simultaneous Multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/339647.339652

[70] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. 2005. SWIFT: Software Implemented Fault Tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '05)*. IEEE Computer Society, Washington, DC, USA, 243–254. https://doi.org/10.1109/CGO.2005.34

[71] RISCV.Org. 2018. RISC-V Foundation. Retrieved January 16, 2019 from https://riscv.org

[72] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. 2007. 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)*. IEEE Computer Society, Washington, DC, USA, 183–196. https://doi.org/10.1109/MICRO.2007.44

[73] Brian Rogers, Milos Prvulovic, and Yan Solihin. 2006. Efficient Data Protection for Distributed Shared Memory Multiprocessors. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT '06)*. ACM, New York, NY, USA, 84–94. https://doi.org/10.1145/1152154.1152170

[74] Eric Rotenberg. 1999. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (FTCS '99)*. IEEE Computer Society, Washington, DC, USA, 84–. http://dl.acm.org/citation.cfm?id=795672.796966

[75] Jarrod A. Roy, Farinaz Koushanfar, and Igor L. Markov. 2008. EPIC: Ending Piracy of Integrated Circuits. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '08)*. ACM, New York, NY, USA, 1069–1074. https://doi.org/10.1145/1403375.1403631

[76] Hassan Salmani, Mohammad Tehranipoor, and Jim Plusquellic. 2009. New Design Strategy for Improving Hardware Trojan Detection and Reducing Trojan Activation Time. In *Proceedings of the 2009 IEEE International Workshop on Hardware-Oriented Security and Trust (HST '09)*. IEEE Computer Society, Washington, DC, USA, 66–73. https://doi.org/10.1109/HST.2009.5224968

[77] Hassan Salmani, Mohammad Tehranipoor, and Jim Plusquellic. 2012. A Novel Technique for Improving Hardware Trojan Detection and Reducing Trojan Activation Time. *IEEE Trans. Very Large Scale Integr. Syst.* 20, 1 (Jan. 2012), 112–125. https://doi.org/10.1109/TVLSI.2010.2093547

[78] Mark Seaborn. 2015. Exploiting the DRAM rowhammer bug to gain kernel privileges. Retrieved January 16, 2019 from http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html

[79] Adi Shamir. 2018. How Cryptosystems Are Really Broken. Retrieved January 16, 2019 from http://www.forth.gr/onassis/lectures/pdf/How_Cryptosystems_Are_Really_Broken.pdf

[80] A. Shapiro and Y. Wardi. 1996. Convergence Analysis of Gradient Descent Stochastic Algorithms. *J. Optim. Theory Appl.* 91, 2 (Nov. 1996), 439–454. https://doi.org/10.1007/BF02190104

[81] Alex Shye, Tipp Moseley, Vijay Janapa Reddi, Joseph Blomstedt, and Daniel A. Connors. 2007. Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '07)*. IEEE Computer Society, Washington, DC, USA, 297–306. https://doi.org/10.1109/DSN.2007.98

[82] T. J. Slegel, R. M. Averill, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, J. A. Navarro, E. M. Schwarz, K. Shum, and C. F. Webb. 1999. IBM's S/390 G5 microprocessor design. *IEEE Micro* 19, 2 (March 1999), 12–23. https://doi.org/10.1109/40.755464

[83] Sudarshan K. Srinivasan and Miroslav N. Velev. 2003. Formal Verification of an Intel XScale Processor Model with Scoreboarding, Specialized Execution Pipelines, and Impress Data-Memory Exceptions. In *Proceedings of the First ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE '03)*. IEEE Computer Society, Washington, DC, USA, 65–. http://dl.acm.org/citation.cfm?id=823453.823841

[84] Cynthia Sturton, Matthew Hicks, David Wagner, and Samuel T. King. 2011. Defeating UCI: Building Stealthy and Malicious Hardware. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (SP '11)*. IEEE Computer Society, Washington, DC, USA, 64–77. https://doi.org/10.1109/SP.2011.32

[85] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. 2003. AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS '03)*. ACM, New York, NY, USA, 160–171. https://doi.org/10.1145/782814.782838

[86] G. Edward Suh, Charles W. O'Donnell, and Srinivas Devadas. 2007. Aegis: A Single-Chip Secure Processor. *IEEE Des. Test* 24, 6 (Nov. 2007), 570–580. https://doi.org/10.1109/MDT.2007.179

[87] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenburg. 2000. Slipstream Processors: Improving Both Performance and Fault Tolerance. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*. ACM, New York, NY, USA, 257–268. https://doi.org/10.1145/378993.379247

[88] Synopsys.Com. 2018. Synopsys' DesignWare IP for PCI Express (PCIe) Solution. Retrieved January 16, 2019 from http://www.synopsys.com/IP/InterfaceIP/PCIExpress/Pages/default.aspx

[89] Mohammad Tehranipoor and Farinaz Koushanfar. 2010. A Survey of Hardware Trojan Taxonomy and Detection. *IEEE Des. Test* 27, 1 (Jan. 2010), 10–25. https://doi.org/10.1109/MDT.2010.7

[90] Martin Thuresson, Lawrence Spracklen, and Per Stenstrom. 2008. Memory-Link Compression Schemes: A Value Locality Perspective. *IEEE Trans. Comput.* 57, 7 (July 2008), 916–927. https://doi.org/10.1109/TC.2008.28

[91] Diana Toma and Dominique Borrione. 2005. Formal Verification of a SHA-1 Circuit Core Using ACL2. In *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'05)*. Springer-Verlag, Berlin, Heidelberg, 326–341. https://doi.org/10.1007/11541868_21

[92] TrustedComputingGroup.Org. 2018. Trusted Computing Group. Retrieved January 16, 2019 from http://trustedcomputinggroup.org

[93] Muralidaran Vijayaraghavan, Adam Chlipala, Arvind, and Nirav Dave. 2015. Modular Deductive Verification of Multiprocessor Hardware Designs. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*. 109–127. https://doi.org/10.1007/978-3-319-21668-3_7

[94] Victor Vu, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. 2013. A Hybrid Architecture for Interactive Verifiable Computation. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. IEEE Computer Society, Washington, DC, USA, 223–237. https://doi.org/10.1109/SP.2013.48

[95] R. S. Wahby, M. Howald, S. Garg, A. Shelat, and M. Walfish. 2016. Verifiable ASICs. In *2016 IEEE Symposium on Security and Privacy (SP)*. 759–778. https://doi.org/10.1109/SP.2016.51

[96] Riad S. Wahby, Srinath T. V. Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. 2015. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS*.

[97] Adam Waksman and Simha Sethumadhavan. 2010. Tamper Evident Microprocessors. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP '10)*. IEEE Computer Society, Washington, DC, USA, 173–188. https://doi.org/10.1109/SP.2010.19

[98] Adam Waksman and Simha Sethumadhavan. 2011. Silencing Hardware Backdoors. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (SP '11)*. IEEE Computer Society, Washington, DC, USA, 49–63. https://doi.org/10.1109/SP.2011.27

[99] Adam Waksman, Matthew Suozzo, and Simha Sethumadhavan. 2013. FANCI: Identification of Stealthy Malicious Logic Using Boolean Functional Analysis. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. ACM, New York, NY, USA, 697–708. https://doi.org/10.1145/2508859.2516654

[100] Cheng Wang, Ho-seop Kim, Youfeng Wu, and Victor Ying. 2007. Compiler-Managed Software-based Redundant Multi-Threading for Transient Fault Detection. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '07)*. IEEE Computer Society, Washington, DC, USA, 244–258. https://doi.org/10.1109/CGO.2007.7

[101] Mao-Yin Wang, Chih-Pin Su, Chih-Tsun Huang, and Cheng-Wen Wu. 2004. An HMAC Processor with Integrated SHA-1 and MD5 Algorithms. In *Proceedings of the 2004 Asia and South Pacific Design Automation Conference (ASP-DAC '04)*. IEEE Press, Piscataway, NJ, USA, 456–458. http://dl.acm.org/citation.cfm?id=1015090.1015204

[102] N. J. Wang and S. J. Patel. 2005. ReStore: Symptom Based Soft Error Detection in Microprocessors. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN '05)*. IEEE Computer Society, Washington, DC, USA, 30–39. https://doi.org/10.1109/DSN.2005.82

[103] Xiaoxiao Wang, Mohammad Tehranipoor, and Jim Plusquellic. 2008. Detecting Malicious Inclusions in Secure Hardware: Challenges and Solutions. In *Proceedings of the 2008 IEEE International Workshop on Hardware-Oriented Security and Trust (HST '08)*. IEEE Computer Society, Washington, DC, USA, 15–19. https://doi.org/10.1109/HST.2008.4559039

[104] Chris Weaver and Todd M. Austin. 2001. A Fault Tolerant Approach to Microprocessor Design. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (Formerly: FTCS) (DSN '01)*. IEEE Computer Society, Washington, DC, USA, 411–420. http://dl.acm.org/citation.cfm?id=647882.738066

[105] Sheng Wei, S. Meguerdichian, and M. Potkonjak. 2011. Malicious Circuitry Detection Using Thermal Conditioning. *Trans. Info. For. Sec.* 6, 3 (Sept. 2011), 1136–1145. https://doi.org/10.1109/TIFS.2011.2157341

[106] Phillip J. Windley. 1995. Formal Modeling and Verification of Microprocessors. *IEEE Trans. Comput.* 44, 1 (Jan. 1995), 54–72. https://doi.org/10.1109/12.368009

[107] Francis Wolff, Chris Papachristou, Swarup Bhunia, and Rajat S. Chakraborty. 2008. Towards Trojan-free Trusted ICs: Problem Analysis and Detection Scheme. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '08)*. ACM, New York, NY, USA, 1362–1365. https://doi.org/10.1145/1403375.1403703

[108] Chenyu Yan, Daniel Englender, Milos Prvulovic, Brian Rogers, and Yan Solihin. 2006. Improving Cost, Performance, and Security of Memory Encryption and Authentication. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA '06)*. IEEE Computer Society, Washington, DC, USA, 179–190. https://doi.org/10.1109/ISCA.2006.22

[109] Jie Zhang and Qiang Xu. 2013. On hardware Trojan design and implementation at register-transfer level. In *HOST '13*. 107–112. https://doi.org/10.1109/HST.2013.6581574

[110] Jie Zhang, Feng Yuan, Lingxiao Wei, Zelong Sun, and Qiang Xu. 2013. VeriTrust: Verification for Hardware Trust. In *Proceedings of the 50th Annual Design Automation Conference (DAC '13)*. ACM, New York, NY, USA, Article 61, 8 pages. https://doi.org/10.1145/2463209.2488808

[111] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. 2017. vSQL: Verifying Arbitrary SQL Queries over Dynamic Outsourced Databases. In *2017 IEEE Symposium on Security and Privacy (SP)*. 863–880. https://doi.org/10.1109/SP.2017.43

[112] Yun Zhang, Soumyadeep Ghosh, Jialu Huang, Jae W. Lee, Scott A. Mahlke, and David I. August. 2012. Runtime Asynchronous Fault Tolerance via Speculation. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*. ACM, New York, NY, USA, 145–154. https://doi.org/10.1145/2259016.2259035

[113] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2015. IntegriDB: Verifiable SQL for Outsourced Databases. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 1480–1491. https://doi.org/10.1145/2810103.2813711

[114] Yun Zhang, Jae W. Lee, Nick P. Johnson, and David I. August. 2010. DAFT: Decoupled Acyclic Fault Tolerance. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10)*. ACM, New York, NY, USA, 87–98. https://doi.org/10.1145/1854273.1854289