TEESec: Pre-Silicon Vulnerability Discovery for Trusted Execution Environments

Moein Ghaniyoun
ghaniyoun.1@osu.edu
The Ohio State University
Columbus, Ohio, USA

Kristin Barber
kristinbarber@google.com
Google
Mountain View, California, USA

Yuan Xiao
yuan.xiao@intel.com
Intel
Santa Clara, California, USA

Yinqian Zhang
yinqianz@acm.org
SUSTech
Shenzhen, China

Radu Teodorescu
teodores@cse.ohio-state.edu
The Ohio State University
Columbus, Ohio, USA

ABSTRACT

Trusted execution environments (TEE) are CPU hardware extensions that provide security guarantees for applications running on untrusted operating systems. The security of TEEs is threatened by a variety of microarchitectural vulnerabilities, which have led to a large number of demonstrated attacks. While various solutions for verifying the correctness and security of TEE designs have been proposed, they generally do not extend to jointly verifying the security of the underlying microarchitecture. This paper presents TEESec, the first pre-silicon framework for discovering microarchitectural vulnerabilities in the context of trusted execution environments. TEESec is designed to jointly and systematically test the TEE and underlying microarchitecture against data and metadata leakage across isolation boundaries. We implement TEESec in the Chipyard framework and evaluate it on two open-source RISC-V out-of-order processors running the Keystone TEE. Using TEESec we uncover 10 distinct vulnerabilities in these processors that violate TEE security principles and could lead to leakage of enclave secrets.

CCS CONCEPTS

• **Security and privacy**; • **Computer systems organization** → **Architectures**;

KEYWORDS

Security, Trusted Execution Environments, Verification

ACM Reference Format:
Moein Ghaniyoun, Kristin Barber, Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. 2023. TEESec: Pre-Silicon Vulnerability Discovery for Trusted Execution Environments. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3579371.3589070

ISCA '23, June 17–21, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0095-8/23/06…$15.00
https://doi.org/10.1145/3579371.3589070

1 INTRODUCTION

Trusted Execution Environments (TEEs) [2, 3, 5, 14, 17, 18, 25, 27, 33, 51] were designed to provide confidentiality and integrity guarantees to security-critical applications, even when running on untrusted operating systems or hypervisors. TEEs provide isolated execution environments, dubbed *enclaves*, for applications to run sensitive code on private data. The hardware-enforced isolation ensures that no privileged code can access the memory of the enclave or any enclave data that are temporarily stored in microarchitectural elements. However, in spite of the increased isolation enabled by TEEs, vulnerabilities in the underlying microarchitectural implementations have led to numerous demonstrated attacks and enclave data leakage [10, 12, 23, 24, 37, 45, 47, 56, 64]. One of the main reasons is that the security afforded by the TEEs relies to a significant extent on complex microarchitectural implementations.

While the correctness of the TEE hardware/firmware implementation can be more easily verified [21, 50, 52], the interaction with the underlying microarchitecture is complex and challenging to thoroughly test. For example, recent work by Cheang et al.[15] formally verifies the RISC-V PMP checker hardware module in the RISC-V Rocket Chip against formalized functional properties of physical memory protection (PMP) rules. It concludes that this hardware implementation passes the functional verification. However, as we show in this work, when the same PMP module is used in an out-of-order processor, it is vulnerable to data leakage.

This paper presents TEESec, the first pre-silicon framework aimed at comprehensive joint verification of TEE and microarchitecture, with the goal of discovering vulnerabilities (such as leakage of enclave data or metadata across isolation boundaries) that may impact TEE security assumptions or guarantees. To that end TEESec performs a detailed, register transfer level (RTL) verification of a processor's microarchitecture and its TEE hardware/software implementation.

The TEESec framework consists of three main components: a Verification Plan, a Test Gadget Constructor, and TEESec Checker. The verification plan systematically profiles the processor design and enumerates all microarchitectural structures which may contain enclave data or metadata. The TEE hardware/software interface and security guarantees are also profiled in this phase. The verification plan enumerates all memory access modalities covering all possible paths through which the enclave data/metadata can be accessed. In the next phase, TEESec exercises all the memory access

modalities by crafting and assembling verification gadgets. In the final phase, the automatically assembled verification codes are run through a cycle-accurate register transfer level (RTL) simulation of the design-under-test, which emits a comprehensive simulation log containing the state of all microarchitectural structures identified in the verification plan.

The simulation log is automatically analyzed to look for any traces of enclave data or metadata that violate desired TEE security principles. In our threat model we define two TEE security principles: (1) No enclave data should be fetched into or remain in the CPU microarchitectural state when the CPU is not in trusted enclave execution mode, and (2) Microarchitectural state that is modified by the enclave code should not affect the execution of any non-enclave code. Any violation of these two security principles may lead to microarchitectural vulnerabilities that are potentially exploitable. While we believe these security principles serve a broad threat model, TEESec can be adapted to verify other security needs under different threat models.

We implement TEESec using the Chipyard framework [4] for designing full-system hardware and we evaluate our work on two different open source RISC-V processors: the BOOM [13, 70] and XiangShan [62], [65] designs. Both processors are equipped with Keystone [27], an open-source framework for designing trusted execution environments in RISC-V systems. Using TEESec, we discover enclave data leakage in processor internal buffers (LFB), register file and metadata leakage via performance counters and branch prediction unit (BPU). Overall, we discover and identify 10 distinct design or implementation issues in these two RISC-V processors that enable secret leakage from enclaves or trusted firmware to the untrusted world. We note that the Keystone TEE offers no out-of-the-box side-channel resistance, leaving it up to the microarchitectural implementation to provide such protection. TEESec can be used to verify that such deployments are correct and meet isolation and security guarantees. We show that the naive deployment of Keystone (or similar TEEs) on out-of-order processors such as BOOM and XiangShan cannot guarantee isolation.

This paper makes the following contributions:

- Presents TEESec, the first pre-silicon framework for co-verification of software and hardware implementations of trusted execution environments.
- Summarizes two fundamental security principles that TEE designs should follow and presents a systematic approach to generating test cases to verify these security principles in the CPU microarchitecture.
- Implements TEESec in the Chipyard framework and evaluates it on two open-source RISC-V CPU cores, running the Keystone TEE.
- Demonstrates the effectiveness of TEESec by identifying 10 distinct vulnerabilities in the evaluated designs.

The rest of this paper is organized as follows: Section 2 provides background on TEEs designs, existing attacks and verification approaches. Section 3 outlines the threat model and a set of security principles. Sections 4 and 5 present the TEESec framework design and implementation. Section 6 details the experimental methodology. Section 7 presents a number of case studies based on vulnerabilities detected by TEESec. Section 8 discusses possible countermeasures. Section 9 concludes.

2 BACKGROUND AND RELATED WORK

2.1 Trusted Execution Environments

A Trusted Execution Environment (TEE) provides isolation and increased security guarantees to security-critical code. This is usually achieved with help from both hardware and software primitives that define and enforce isolation boundaries. There have been a number of TEE designs proposed by both industry and academia with the aim of protecting sensitive code and data from untrusted elements of the the software stack, including the operating system and hypervisor. Some of the most popular commercial TEE designs include: Intel SGX [25], AMD SEV [2], ARM TrustZone [5].

Several TEE frameworks have targeted the open-source RISC V ISA. Sanctum [17] was the first such framework, prototyped on the RISC-V Rocket architecture. In Sanctum, enclave setup and management are implemented in a Security Monitor, which runs in machine mode – the highest privilege level for a RISC-V system. The Security Monitor is responsible for verifying and attesting enclaves. Enclave isolation is enforced through hardware support in the Page Table Walker (PTW), which prevents unauthorized access to enclave memory.

MI6 [9] extends Sanctum's isolation guarantees to out-of-order processors through hardware support added to the RiscyOO [67] out-of-order core and by expanding the threat model to include side-channel and speculative execution attacks. In order to mitigate speculative execution and timing side-channel attacks, MI6 introduces a *purge* instruction with hardware support to flush sensitive data from microarchitectural buffers and the L1 Data Cache before context switches.

We evaluate TEESec on Keystone [27], an open-source framework for designing trusted execution environments in RISC-V systems. Keystone uses RISC-V Physical Memory Protection (PMP) to define security domains and isolate these domains from each other. Unlike other TEEs, Keystone requires no additional hardware support beyond PMP.

2.2 TEE Attacks Exploiting Microarchitectural Channels

Many demonstrated attacks have exploited microarchitectural side channels to break TEE security guarantees. For example, in addition to general cache side channel attacks such as Prime+Probe [1, 35, 36, 40–42, 55, 68] and Flush+Reload [8, 8, 69], several side-channel attacks specifically target TEE designs like Intel SGX [10, 23, 24, 45, 56, 71] and AMD SEV [29, 30, 38, 39, 60].

Other attacks exploit the near-total control of the untrusted OS over TEE instances to extract enclave data [12, 16, 28, 32, 47, 64]. Some force page faults in order to infer the control flow of the enclave code and leak secrets in applications with secret-dependent memory access patterns [47, 64]. Other attacks [12] exploit the side effects of accessing unprotected page table memory during address translation. The side effects of each page table walk can then be further examined to learn the memory access pattern of the enclave. Translation Lookaside Buffer (TLB) poisoning attacks [32] exploit the fact that the hypervisor is responsible for flushing the TLB in AMD SEV virtual machines (VMs). An attacker process can interfere with the execution of the victim process in the same VM by skipping

TLB flushes with a malicious hypervisor. Branch shadowing attacks [28] infer enclave control flow by inducing collisions in the Branch Target Buffer (BTB) of the CPU. SgxPectre [16] leaks enclave secrets by inducing speculative execution of instructions that lead to cache state changes. This is achieved by manipulating the branch predictor state (shared by enclave and non-enclave code) from outside the TEE.

Most of the aforementioned vulnerabilities result from the complex interaction between TEE security and isolation expectations and the reality of the underlying microarchitectural implementation. This makes comprehensive verification of the TEE and its implementation very challenging.

2.3 Existing Pre-Silicon Verification Approaches

Each TEE design defines a set of security guarantees with respect to its threat model. We survey existing work on design-time (pre-silicon) verification of hardware designs in general and TEEs in particular. Prior work on pre-silicon verification falls under two broad categories: static (formal) and dynamic verification.

Formal Verification. Formal verification [15, 19, 21, 48, 50, 52] can be used to mathematically prove that formally specified requirements of a TEE design are met. For example, TAP [50] formally verifies TEE platforms by first defining the requirements of enclave secure remote execution and then creating a trusted abstract platform which is an idealization of the TEE framework with respect to a threat model. The enclave platform is verified by machine-checked proofs to ensure it satisfies the integrity, confidentiality and secure measurement requirements.

The hardware implementation component of TEE designs can also be formally verified. For example, an implementation of the RISC-V Physical Memory Protection (PMP), which provides the core security guarantees for the Keystone TEE framework, is formally verified in [15]. The PMP rules are first formalized based on the RISC-V ISA specifications. A hardware implementation of PMP is then translated to the Uclid5 [46] verification language. Finally, by encoding the formal specification in Uclid5, the hardware implementation of PMP is verified against formalized rules. In a similar approach, UPEC [19] introduces a hardware property that is formulated based on the formal definition of security with respect to transient execution attacks.

Formal verification is a powerful tool for ensuring the functional correctness or security of a design. However, formal verification has limitations that prevent it from being universally applicable. For example, formal verification can only be applied where there exists formal descriptions for both design specification and implementation. Moreover, the complexity of formal specifications increases exponentially with that of the design, making formal verification impractical for most components of a modern high-performance processor.

Dynamic Verification Pre-silicon verification is extensively applied for functional correctness verification as well as security. In this approach, the design is simulated using cycle-accurate representations and exercised using test stimuli. Simulation-based test environments usually consist of generators, driver monitors, and checkers to produce the stimuli, transform the stimuli to actual inputs that run on the design-under-test, record the state of design and its outputs and validate the output against the expected result. Prior work has proposed several improvements to this process to make it more effective for security verification. For example, SecVerilog [66] uses a new type system extension of the Verilog hardware description language (HDL), which allows information flow policies to be described and verified at compile-time. GLIFT [53] instruments the design at gate-level and enforces the information flow policies at run-time. CellIFT [49] proposes a similar dynamic information flow tracking approach, but improves scalability to larger models by leveraging macrocell abstractions (a higher design abstraction than the gate-level representation used in GLIFT). In [7] pre-silicon simulation is used to detect violations of constant-time execution in security-critical code.

Other prior work has focused on fuzzing-based verification of hardware designs [22, 54, 63]. SpeechMiner [63] leverages a fuzzing framework to identify Meltdown-type vulnerabilities in existing processors. IntroSpectre [22] uses gadget fuzzing and register-transfer level (RTL) simulation to detect transient execution vulnerabilities in out-of-order processor designs. TEESec leverages the IntroSpectre logging framework. Our approach, however, goes beyond transient execution leakage to capture all paths for data and metadata leakage across enclave boundaries. Unlike prior work, TEESec enables the comprehensive and joint verification of both TEE hardware and software on complex, high-performance processor designs.

3 TEE SECURITY PRINCIPLES

In this section, we first outline the threat model we consider in this paper and then discuss the security principles the TEE design should follow to offer strong security in the considered threat model.

3.1 Threat Model

TEESec targets microarchitectural vulnerabilities that can potentially leak secrets across the boundary between an enclave and the untrusted world. Specifically, the secrets considered in this work are data and code protected by the enclave, and metadata created during enclave execution. Enclave data can leak either directly or as new values derived from enclave data, and can be found in microarchitectural structures such as caches, line fill buffers (LFB), load queues and store buffers, etc. Metadata include state of the branch prediction units and values of hardware performance counters and generally any enclave execution information except for timing. We assume the adversary is in control of any software component outside the targeted enclave, which includes the application code, the system software, or other enclaves running on the system.

We do not target specific attack methods with which the enclave secrets may be exfiltrated. These methods could include access-driven side-channel attacks that steal secrets through shared microarchitectural state, or transient execution attacks that exploit speculation features. Timing-based attacks that infer secrets by observing the execution time of certain instructions [43, 57] are out of scope. Off-chip memory attacks and in general any type of physical attacks that require physical accesses of the machine (and hence the processor chip) are also not considered in this work. Attacks against

encrypted off-chip enclave memory [31] are also out of scope. Finally, we do not consider any tampering with trusted firmware or any attestation-related bypassing techniques [11, 16, 58].

3.2 Security Principles

The large number of demonstrated attacks that have exploited microarchitectural vulnerabilities in existing TEE implementations highlight the need for precise definition of the expected security principles that the microarchitecture underlying the TEE implementation should adhere to. With the aforementioned threat model in mind, we identify the following high-level security principles that the microarchitecture should follow in order to prevent leakage across the TEE boundary.

- **P1:** No enclave data should be fetched into or remain in the CPU microarchitectural state when the CPU is not in trusted enclave execution mode.
- **P2:** The microarchitectural state that is influenced by the enclave code should not affect the execution of any non-enclave code.

The reasons behind **P1** are twofold: First, the untrusted world should not be able to access data in enclave memory. A memory access can be either **explicit** or **implicit**. An **explicit** access is a memory access initiated directly by the execution of memory-related instructions such as loads and stores. Any explicit illegal accesses to enclave memory regardless of the execution privilege should raise an exception and be promptly handled. An **implicit** memory access is initiated by the hardware and can serve different purposes including memory management (e.g., page table walks or page walker cache refills), performance optimizations (e.g., data prefetching), etc.

Second, enclave data should be cleansed from the microarchitecture state when the processor is not running in the enclave mode. This means flushing of all microarchitectural buffers, such as caches, TLBs, LFBs, etc., has to be enforced during context switches from the enclave to the untrusted world. It also implies that the enclave and the untrusted world cannot share microarchitectural components (e.g., last-level cache) simultaneously. We do acknowledge that in practice vendors might be reluctant, for performance reasons, to enforce some of these isolation mechanisms (e.g. full cache flush). In such cases additional hardware support might be used to provide the same guarantees with lower performance impact.

Principle **P2** ensures that enclave metadata cannot be indirectly leaked from microarchitectural structures. While microarchitectural states are not directly visible to software, secrets can still be leaked through various side-channels. **P2** suggests a reset of the microarchitectural state is required when a context switch from enclave mode to non-enclave mode takes place. It also precludes sharing of microarchitectural components by enclave and non-enclave code.

We note that, while a violation of these security principles does not always lead to exploitable vulnerabilities, a design following these principles is guaranteed to mitigate all known attacks under our threat model.

4 THE TEESEC FRAMEWORK

TEESec is a pre-silicon framework designed to verify register-transfer level (RTL) implementations of TEE-protected CPUs. The framework could be deployed stand-alone or as part of the functional verification process of the design. We choose RTL-level verification as opposed to post-silicon for two important reasons: (1) The RTL design fully represents the functionality of the CPU (unlike higher-level architectural simulations which include necessary approximations/simplifications) and (2) the RTL simulation infrastructure provides full visibility to the complete set of microarchitectural structures (unlike the final hardware, in which much of the microarchitectural state is invisible to direct observation). TEESec is currently implemented using the RISC-V based Chipyard framework. In our case studies we choose the Keystone TEE [27] as a verification target, but stress that TEESec can be equally applied to other TEE designs.

The TEESec framework consists of three main components: (1) A **Verification Plan**, (2) **Test Gadget Constructor** and (3) the **TEESec Checker**. In the initial phase, we analyze the design and specifications of the target system to construct a verification plan, which enumerates the relevant components that will be part of the verification, as well as the TEE software API and security specifications. The verification plan is used to generate a set of testing gadgets that follow the verification plan and assemble those gadgets into test code sequences that will run on the processor under test. The TEESec checker is integrated into an RTL simulator, uses test inputs generated from the testing gadgets, logs detailed execution state specified by the verification plan and verifies that the security principles and TEE API contracts are enforced. Figure 1 illustrates a high-level view of TEESec framework and its three components, which are described in detail next.

4.1 Constructing a Verification Plan

The verification plan consists of a comprehensive analysis of the relevant microarchitectural features of the processor under test, including a complete enumeration of all memory access paths implemented by the processor, all architectural and microarchitectural data and metadata storage elements and the software API of the TEE. Some of these steps are automated while others rely on designer input.

4.1.1 Enclave Memory Access Paths. A comprehensive approach for identifying the potential for secret leakage requires first identifying all paths through which data, metadata and code can be retrieved from memory into the processor. This must include all possible paths to access memory. While most memory accesses are explicit (initiated by load and store instructions), many are implicit requests needed to service other functions (prefetcher requests, page table walks, etc.). These implicit memory access paths are often overlooked by vulnerability detection tools.

As an example of an implicit memory access, let us consider a load instruction that causes a TLB miss in the RISC-V BOOM [70] design. In this case, the processor has to request a hardware-managed "page table walk". The page table walker unit accesses the root page table and depending on the virtual memory implementation, there may be multiple accesses to memory to resolve the mapping and update the TLB. All these accesses are *implicit* and essentially invisible to the program/user. Depending on the implementation of the page table, multiple page table accesses can miss in the cache, leading to several additional implicit requests

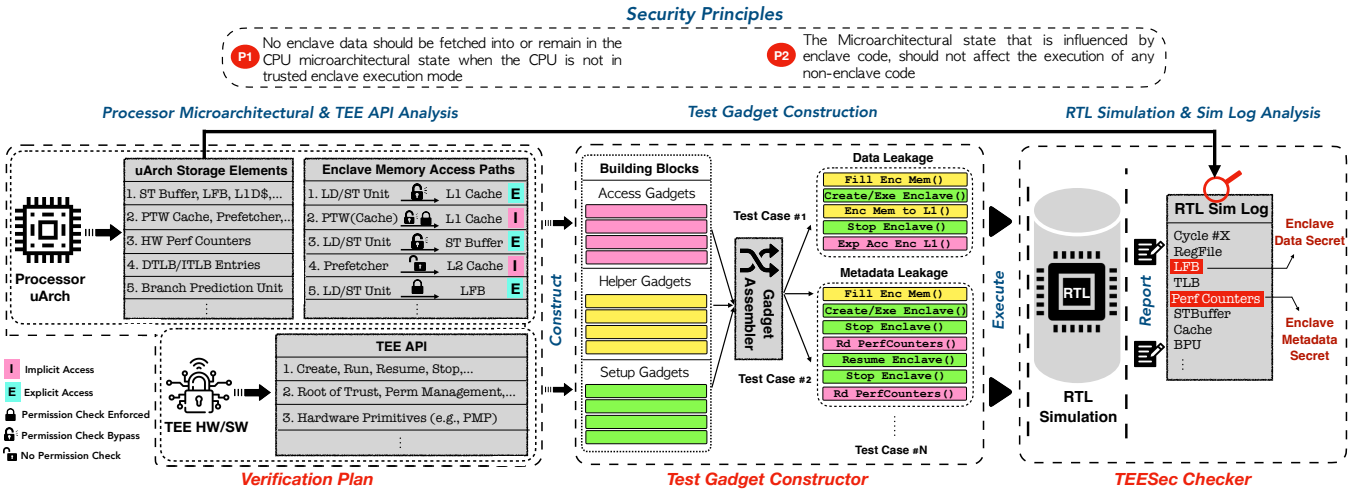

Figure 1: TEESec architecture and workflow. TEESec consists of three main components: a Verification Plan, a Test Gadget Constructor and TEESec Checker which is designed to run in conjunction with an RTL simulator.

to the memory hierarchy. Other examples of implicit accesses are prefetch requests. The BOOM processor is equipped with a simple next-line prefetcher which prefetches the next 64 bytes (cache line size) of memory in case of a cache miss. In this case, the hardware prefetcher logic initiates this implicit access.

As these examples illustrate, implicit accesses involve distinct mechanisms with different access paths that need to be included in the verification plan to ensure full coverage of all possible leakage channels. TEESec relies on the complete list of memory access paths for verification. This can be constructed from design specifications or direct RTL code analysis. This list will be used to generate access gadgets that exercise all possible paths through which data, metadata and code can be accessed.

4.1.2 Permissions Check Handling. Correct and secure access permission enforcement is fundamental to secure TEE designs. For example, the first variant of Meltdown [34] exploits a speculative permissions check implementation, which occurs in parallel with the memory access. This implementation choice may seem benign at first glance because the illegal access and all instructions executed after it are eventually squashed. However, the potential to leak the secret still exists because, before the instructions are squashed, the accessed secret can be forwarded to a transmitter gadget and leaked through a side channel.

It is also important to note that not all memory accesses undergo access permissions checks. Implicit memory accesses frequently forgo such checks, for example those initiated by the prefetcher. In these cases an attacker can craft code to induce the prefetcher to access memory regions inaccessible to the attacker process. Since hardware prefetching does not check permissions, no exceptions will be raised for these illegal accesses. Similarly, accesses generated by the hardware page table walker (PTW) undergo speculative checks that can be bypassed by Meltdown-like attacks, or no permission checks at all.

In order to verify TEE vulnerabilities to these and other types of attacks, TEESec enumerates all permission check policies for all memory access paths. This information is used to generate/select test gadgets that exercise all these policies for verification.

4.1.3 Microarchitectural Data/Metadata Storage Elements. Modern out-of-order processor designs conduct complex operations with data stored in program-invisible microarchitectural structures. This complexity and lack of visibility into the implementation details has lead to multiple vulnerabilities. For example, data was leaked from the line fill buffer (LFB), an otherwise obscure structure, in [44]. Depending on the processor design, multiple paths may exist to service a load instruction. These include the store queue, the line fill buffer, L1 data cache, etc. Accesses through all these paths can leave secret data traces that can lead to exploits.

In addition to data, some microarchitectural structures can leak metadata that can be used in an attack. For example, branch target buffers (BTB) provide the predicted target address for branch instructions. The BTB is usually implemented as a set-associative structure which makes it susceptible to collision-based attacks (Section 7.2.2). Another set of storage elements containing metadata are hardware performance counters that hold information such as number of L1 cache misses, TLB misses, store-to-load forwarding, exceptions, etc. This information can be used by the untrusted host to infer the enclave program flow and memory accesses, which could be exploited by an attacker to leak enclave secrets. In Section 7.2.1, we describe a case study where the inaccessible performance counters can be read by an underprivileged user.

In order to detect such vulnerabilities TEESec seeks to exhaustively identify all microarchitectural structures that have state and capture that state in the execution log used for leakage verification. The goal is to capture all potential paths inside the microarchitecture that can be traveled by enclave data, code or metadata as a result of both explicit and implicit memory accesses. The lists of microarchitectural storage elements can be compiled automatically using tools such as the the Yosys Verilog RTL synthesis tool [61], which can be used to identify all HDL constructs that could be mapped to memory objects and eventually to memory cells. This

mapping happens as a pass in Yosys synthesis steps and can be extended to compile a record of all the modules containing storage elements along with their interfaces, to help with integration into the logging mechanism of the TEESec Checker.

4.1.4 The TEE Software API. TEESec verifies that the TEE API functions follow the outlined security principles. This requires identifying all relevant TEE APIs (e.g Enclave Create(), Resume(), Destroy(), etc.) and summarizing their expected behavior. TEESec will then verify that the microarchitectural state is leakage-free with respect to the security principles defined in Section 3.2 following privilege transition boundaries (e.g. after Enclave Destroy() calls). Note that TEESec is focused on identifying data/metadata leakage, not the functional correctness of the TEE API, which is orthogonal to this work. The TEE API is also used to establish the test boundaries, generate setup gadgets to initialize the system state to the desired parameters before each test.

4.2 Test Gadget Construction

The second main component of TEESec is the test gadget generation framework, illustrated in the middle section of Figure 1. The Verification Plan is used to enumerate all data/metadata memory access paths and microarchitectural storage structures. It also outlines the TEE API functions and expected behavior with respect to enclave data/metadata protections. Using this plan we generate test sequences to reproduce all the access paths, both explicit and implicit. We use a modular, gadget-based test generation approach to increase the scalability, coverage and re-usability of the test code. The use of targeted gadgets also limits the number of possible test sequences in order to keep testing time reasonable. Each gadget is composed of a few assembly instructions with parameterized variables. Three types of gadgets are used in TEESec:

Helper Gadgets are responsible for setting up the microarchitectural state and seeding secret data in enclave memory. These gadgets are used in cases that require a specific state for enclave data/metadata. As an example, the `Fill_Enc_Mem()` gadget includes a few store instructions in a loop, which populate enclave memory with secret values. These secrets are computed as a hash function of the memory address where they are stored. The memory address is provided to the gadget as an input. This way, any secret leakage identified in the simulation log can be traced back to the original memory access. Other gadget examples include bringing certain enclave data to cache, causing cache misses in enclave data, triggering hardware prefetch of certain addresses, etc.

Setup Gadgets provide the necessary set of instructions to set up the execution environment and a means of communication between enclave and non-enclave world. Setup gadgets are constructed to follow the TEE Software API specifications outlined in the verification plan. Gadgets such as `Create_Enclave()`, `Exe_Enclave()` and `Stop_Enclave()` setup and/or destroy enclaves and perform context switches between enclave and non-enclave execution. TEESec uses these gadgets to verify that leakage does not occur following TEE API calls, for example when an enclave is created, stopped, resumed and stopped again, exited, destroyed, etc.

Access Gadgets reproduce and exercise all the memory access paths identified in the verification plan. This is achieved with assistance from helper gadgets that insert/remove enclave data/metadata in the cache and processor buffers. For example, access gadget `Exp_Acc_Enc_L1()` requires the accessed enclave data to be present in the L1 cache to reproduce one of enclave memory accesses in Figure 1. Similarly, `Imp_Acc_Pref()` access gadget requires the targeted enclave data to be present in L2 and not in L1 so that it can be brought to the L1 by an implicit access initiated by the prefetcher. Depending on the microarchitectural state at the time of each gadget's execution, we can have multiple permutations of the same gadget. For example, `Exp_Acc_Enc_L1()` can be executed when the accessed data is only in L1, L1 and LFB, L1 and store buffer, etc.

Gadget Assembler. The final step of the test gadget construction is accomplished by the Gadget Assembler, which automatically generates complete test sequences that cover all valid permutations that exercise the entire range of both implicit and explicit memory access paths with different permissions, and for all TEE API calls. In order to guide the assembler to generate valid test sequences, an execution model is constructed automatically to capture the expected microarchitectural state following gadget execution. The execution model ensures the expected pre-conditions exist for each access gadget to exercise the targeted memory access path.

4.3 TEESec Checker

The final component of our framework is the TEESec Checker, illustrated in the third panel of Figure 1. The Checker is responsible for executing the verification code sequences and analyzing the system for potential leakage. The TEESec Checker is integrated into an RTL simulator. The source code is instrumented to log the content of all microarchitectural storage elements outlined in the verification plan, at cycle granularity. In order to identify data leakage, the checker searches the log for verbatim enclave secrets in processor buffers and microarchitectural structures when the CPU is not executing in enclave mode. Similarly, to identify potential for metadata leakage, the TEESec checker searches for enclave metadata residue such as branch prediction history in CPU buffers. This metadata leakage is a potential security vulnerability as it might lead to exploits that leak enclave data or control flow.

5 TEESEC IMPLEMENTATION

We implement TEESec in the Chipyard framework, with deep integration into the Verilator RTL simulator. TEESec is designed to be used by microprocessor architects and security verification engineers with a high-level understanding of the design. It does not require a detailed knowledge of the underlying microarchitectural implementation. It requires awareness of which processor structures that can initiate memory accesses, in order to be included in the verification plan. This process could potentially be automated through a profiling pass. TEESec also requires enumeration of all possible privilege levels, but it does not require knowledge of how permissions are enforced by the microarchitecture. Permission enforcement is part of the automatic verification process of TEESec.

As TEESec is still a prototype, some of the steps that are now manual could be automated in a production system. In the current TEESec prototype the identification of data/metadata storage elements is automatic. Based on the storage elements discovered such as line-fill buffer (LFB), write-back buffer, store buffer, etc. and design specifications, enumerating all memory access paths is

automatable, but currently a manual pass. Generating access gadgets to exercise each possible access path is also a manual process. Since gadgets are parameterized, we rely on fuzzing for gadget assembly and to generate varied test cases. After the verification plan is established and the gadgets constructed, the gadget fuzzing process, the RTL simulation log analysis, and leakage discovery are all automatic. Table 1 shows a summary of TEESec components highlighting, for each component, whether it is currently automatic, automatable in a production system, or manual for the time being.

	TEESec Components	Manual	Automatic
Verification Plan	Identifying Storage Elements	-	✓
	Listing Memory Access Paths	(✓)	-
	Listing TEE HW/SW APIs	(✓)	-
Test Gadget Constructor	Access Gadgets Targeting Memory Access Paths	✓	-
	Test Case Assembly	-	✓
TEESec Checker	RTL Simulation Log Analysis	-	✓
	Leakage Discovery	-	✓

Table 1: The main components of the TEESec framework. ✓ indicate components that are automatic, and (✓) indicate steps that are automatable but currently implemented as a manual pass.

In the implementation of the TEESec prototype we used 8 setup gadgets, 12 helper gadgets, 2 metadata access gadgets and 13 data access gadgets (one for each memory access path). Each gadget is parameterized allowing the fuzzer to generate multiple test cases in order to maximize coverage. TEESec generated 585 test cases, which cover all access paths. Generating, running and analyzing each test case takes approximately 5 minutes on our systems. Table 2 summarizes the number of gadgets used in our evaluation, as well as the time overhead of the main TEESec steps (manual or automatic). Currently the most time-intensive component of TEESec is the development of the verification plan for a new TEE/microarchitecture, at around 40 person-hours. Once a verification plan is developed for a given microarchitecture, deploying it to a new TEE is relatively straightforward, with only incremental changes. The main cost of the verification plan is ensuring coverage of all memory access paths.

6 EVALUATION METHODOLOGY

We evaluated two open-source, dynamically scheduled, out-of-order processor designs: XiangShan [62] and BOOM [13, 70]. While they are both RISC-V processors, they have substantially different architectures and implementations. For BOOM, we evaluated two versions which includes the latest release, SonicBOOM (v3.1), and the last stable release (v2.3) before SonicBOOM.

We deploy Keystone, an open-source framework for designing TEEs and managing secure hardware enclaves on RISC-V systems

Gadgets	**Setup**	**Helper**	**Access**	**Total Test Cases**
No.	8	12	15	585
TEESec	**Verification Plan**	**Gadget Constructor**	**Checker**	**Avg. Time**
Time	40* person-hour	~1min	~4min	~5min

Table 2: Total number of test cases generated along with the average time to generate/execute/analyze each test case. *one-time cost.

to both BOOM and XiangShan. Keystone uses a security monitor that runs at the highest execution privilege level on RISC-V systems (Machine Mode) and is responsible for creating, running, stopping, attesting and destroying hardware enclaves. Keystone requires limited hardware modification to be deployed to a RISC-V core. Similar to other RISC-V TEEs [6, 20], Keystone utilizes RISC-V Physical Memory Protection (PMP) to define security domains and isolate these domains from each other. RISC-V PMP provides a set of control and status registers (CSRs) which can be configured to specify the access privileges of each physical memory region. The size and address boundaries of each region is also managed by a similar set of PMP CSRs.

To test TEESec on both XiangShan and BOOM cores, we used the Keystone-enabled version of Berkeley Bootloader included with the standard security monitor. For our OS needs, we developed a modified version of *riscv-pk* (proxy kernel) that supports RISC-V *sv39* virtual memory management and provides basic exception handling features. Also, our modified *riscv-pk* interacts with the security monitor via the Supervisor Binary Interface (SBI) which is triggered by executing an ECALL instruction that generates an environment call exception. Depending on the value stored in register *a*0 at the time of ECALL execution, different functions in the security monitor such as create/stop enclave can be called. The created enclaves support two levels of execution privilege and all context switches from/to enclaves to the host go through the security monitor.

We run all tests on a machine equipped with an Intel Xeon E5-2440, 2.40GHz CPU, 32GB of RAM, running RHEL 7.9. For the RTL simulation, we opted for Verilator, an open-source RTL simulator which converts the Verilog code to C++, which is then used to build the simulator.

7 RESULTS AND CASE STUDIES

The deployment of TEESec on the two processors lead to the discovery of 10 distinct violations of the security principles **P1** and **P2** outlined in Section 3. These findings are listed in Table 3. The leakage cases fall into two classes: (A) enclave data leakage (*D1 – D8*) and (B) enclave metadata leakage (*M1* and *M2*). The first class represents violations of *P1* and the second class violations of *P2*.

7.1 Enclave Data Leakage

By following the TEESec Verification Plan and targeting all memory access paths on BOOM and XiangShan, we discovered eight instances where the enclave secret is leaked to the untrusted host

Type		Leaking Cases	Secret Access Path	Source	BOOM	XS
Data	D1	Leaking enclave data via L1D prefetcher abuse	Load Inst (Exp) —> L1 Cache Miss —> Prefetcher (Imp) —> L2 Cache Req —> LFB Refill	LFB	✓	-
	D2	Leaking enclave/SM data through page table walks	Load Inst (Exp) —> TLB Miss —> Page Table Walk (Imp) —> L1 Cache Miss —> L2 Cache Req —> LFB Refill	LFB	✓	-
	D3	Leaking LFB residual data after enclave destroy	Store Inst (Exp) —> L1 Cache Miss —> L2 Cache Req —> LFB Refill (Stale enclave data)	LFB	✓	-
	D4	Leaking enclave data/code to host user/supervisor	Load Inst (Exp) —> TLB/PMP Check —> L1 Cache Hit —> Write-back RF —> Secret Forwarded	RF	✓	✓
	D5	Leaking Keystone SM data/code to host user/supervisor	Load Inst (Exp) —> TLB/PMP Check —> L1 Cache Hit —> Write-back RF —> Secret Forwarded	RF	✓	✓
	D6	Leaking enclave data/code to another enclave	Load Inst (Exp) —> TLB/PMP Check —> L1 Cache Hit —> Write-back RF —> Secret Forwarded	RF	✓	✓
	D7	Leaking host user/supervisor data/-code to enclave	Load Inst (Exp) —> TLB/PMP Check —> L1 Cache Hit —> Write-back RF —> Secret Forwarded	RF	✓	✓
	D8	Leaking enclave data/code through store buffer	Load Inst (Exp) —> TLB/PMP Check —> Store Buffer Hit —> Write-back RF —> Secret Forwarded	RF	-	✓
Metadata	M1	Revealing enclave control-flow/data access patterns via performance counters	Reset Perf Counters —> Enter Enclave —> Stop Enclave —> Read Perf Counters	HPC	✓	✓
	M2	Revealing enclave control-flow via conflicts on branch prediction units	Enter Enclave —> Execute a Cond. Branch —> Stop Enclave —> Execute a Cond. Branch Mapping to the Same uBTB Entry —> Check Cycle Count	BPU	✓	✓

Table 3: Enclave data/metadata leakage cases, the secret source and access path.

or another enclave, as shown in Table 3. In the first three cases (*D1-D3*), enclave secrets were found in the BOOM Line Fill Buffer (LFB) while the processor was executing in non-enclave mode. In the other five case studies (*D4-D8*), enclave secrets were written-back to the register file and/or forwarded to next dependent instruction, even when the CPU was running in non-enclave mode.

7.1.1 D1: Leakage via the Hardware Prefetcher. In order to verify the prefetcher implicit memory access path, an access gadget is designed to access data in a memory page adjacent to an enclave page protected by PMP. When the accessed data is located at the boundary-straddling addresses of the accessible page, the prefetcher triggers memory requests for the next cache line that falls in the enclave memory region, as illustrated in Figure 2. Since no permission checks are performed for the prefetch accesses, the Line Fill Buffer receives 64 bytes (cache line size) of enclave data (`Refill` from L2 to LFB). The TEESec Checker identifies these traces of PMP-protected memory in the LFB as a potential vulnerability. The same test does not identify a vulnerability in XiangShan as this processor lacks an L1 prefetcher.

7.1.2 D2: Leakage through Page Table Walks. TLBs store the mappings of virtual memory pages to physical pages to speed up address translation. A TLB miss occurs when the processor requests an address translation that is not present in the TLB. For performance reasons many processors implement the miss handling mechanism in hardware, generally using a page table walker (PTW). On a TLB miss, the page table walker first looks up the root page table and then generates multiple memory accesses to traverse the page table hierarchy and reach the targeted page table entry. All memory accesses to page tables are implicit and invisible to software.

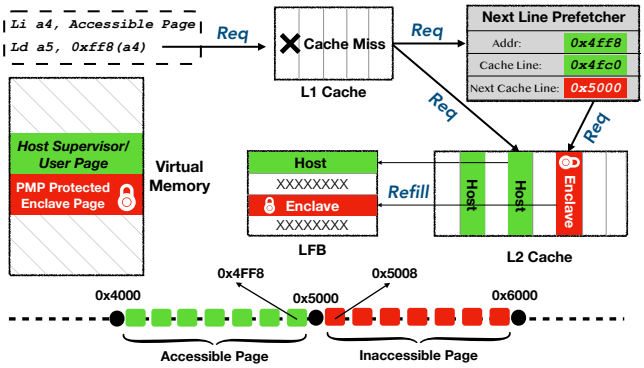

Figure 2: Abusing L1 cache next-line prefetcher to bring enclave data into LFB in BOOM.

According to most TEE threat models, the malicious OS can modify the root page table address of the host to point to the enclave memory. TEESec verifies all permutations of relevant state under which this can occur, including different states for the cache, Miss Status Holding Register (MSHR), and other microarchitectural components.

In BOOM, when a TLB miss occurs, the processor always initiates a memory request to the root page table to start the address translation process. While the implicit page table walk request eventually raises an Access Fault exception, the access to the invalid page table is not squashed. Thus, it results in the LFB being filled with 64 bytes (cache line size) of enclave data. The TEESec Checker identifies this case as potential leakage. Indeed, by setting the root

page table address to point to enclave memory, an attacker can speculatively load the entire enclave page into the LFB.

The steps of the attack are shown in Figure 3. ❶ The host root page table address in the Supervisor Address Translation and Protection (SATP) CSR is modified to point to enclave memory. ❷ A load instruction is issued to an arbitrary page for which the TLB does not have a translation. ❸ A TLB request will be followed by ❹ a page table walk request. ❺ The PTW hardware initiates a request to the L1 cache to retrieve the root page table entry (pointing to enclave memory), leading to a cache miss. ❻ Next, a request is initiated from L1 to L2 to fetch the missing cache line. ❼ The L2 responds by refilling the LFB with enclave secrets.

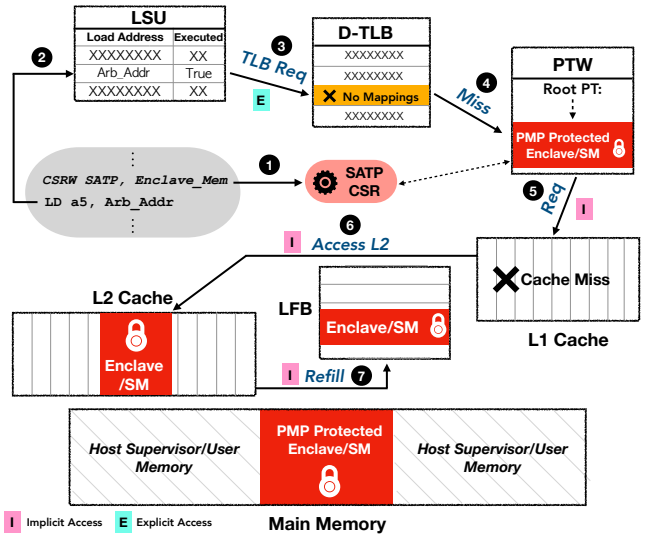

Figure 3: Manipulating host root page table to point to the enclave/SM memory and forcing a page walk by executing a load instruction that misses in TLB

Interestingly, the same vulnerability does not exist in XiangShan. By closely examining the simulation log and investigating the Chisel source code of XiangShan, we found that the PTW cache refill signals are not sent over the typical L1D request channel. Instead, they are sent as TileLink requests directly to L2 over the *'A'* channel. Responses are returned as TileLink messages over the *'D'* channel. Moreover, before creating a request, the refill address is checked by PMP CSRs. If the address falls in an inaccessible region, no request will be created at all. In the simulation log, we observed that in spite of raising a PTW cache miss, there is no valid request sent over the *'A'* channel by the PTW cache. To validate the test sequence, we experimented with exactly the same steps, but with the manipulated page table entry pointing to an accessible memory region rather than enclave memory. In this case we were able to identify the valid request and response and also the refill values in the PTW cache. This showed that the test was working as intended and that indeed XiangShan does not exhibit the PTW vulnerability.

7.1.3 D3: Leakage of LFB Residual Data. Keystone provides a RISC-V Supervisor Binary Interface (SBI) for the host supervisor to destroy an enclave and free its memory. Destroying an enclave is only allowed when the enclave is in either *'stopped'* or *'exit'* state. One of the setup gadgets, `Destroy_Enclave()`, is executed by the host supervisor to destroy the target enclave. The gadget sets register `a0` to `enclave_id` and register `a7` as the function specifier (in this case `mcall_sm_enclave_destroy`), followed by an `ECALL` instruction to raise an exception to be handled by the security monitor (SM) (❶ in Figure 4). Next, the control is transferred to the SM exception handler. Depending on the values in registers `a0` and `a7`, the appropriate SM function is executed. In addition to a few checks at the beginning of `sm_enclave_destroy`, which make sure the requested enclave can be destroyed, a `memset()` that zeros-out the entire enclave physical memory is executed ❷. `memset()` is implemented with a set of store (`sd`) instructions that are executed in a loop until the entire enclave memory is zeroed out.

In the BOOM processor, when a store instruction is committed, a cache request is sent to L1D. In case of an L1 cache miss, a fill request is sent to the L2 cache to fetch the missing line, which will be serviced through the LFB. According to security principle **P1**, all data residues should be cleansed after the CPU leaves the enclave mode. However, TEESec Checker detects enclave secrets in the LFB brought in by the `memset()` stores. These secrets persist in the LFB even after the context switch from the security monitor to host supervisor ❸. In the current BOOM microarchitecture the LFB is not used to directly service load requests, which means LFB data cannot leak. However, the LFB has been shown to be very vulnerable to leakage in Intel processors [59]. As a result, we flag leakage to the LFB as a potential concern.

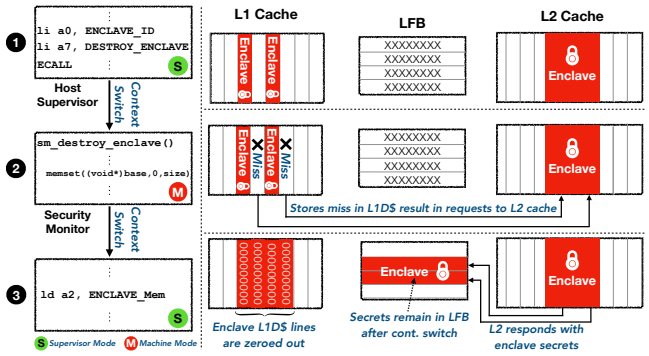

Figure 4: Enclave secrets are brought into LFB when SM cleanse the enclave memory upon enclave destroy.

7.1.4 D4-D7: Leaking Secrets from L1D. All leakage instances in this category originate in the L1 data cache and are Meltdown-type [34] vulnerabilities. We discover these vulnerabilities using extensive testing of the explicit memory access path, using gadget permutations that exercise a wide range of execution states, including secret present or not in the L1D and LFB, MSHRs utilization, etc. Out tests also include all types of memory instructions, e.g. load byte, load word, load double word, load half word, missaligned load, etc. The TEESec Checker finds multiple instances in which a PMP protected secret can be transiently leaked through the register file and a dependent instruction, even though an Access Fault exception is raised.

We categorize the leakage instances based on the presence/absence of the targeted data in L1D:

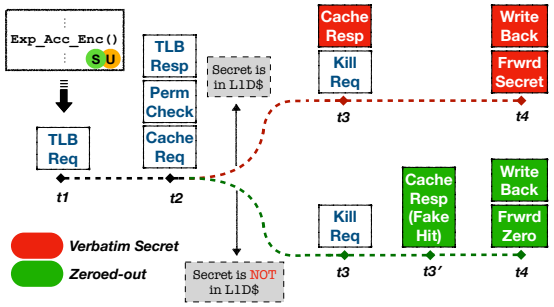

Figure 5: Accessing PMP protected data on XiangShan. Two different access paths are shown: the data present and not present in L1D.

a) PMP-protected data is present in L1D: Simulation log analysis for both BOOM and XiangShan shows that as soon as the physical address is resolved from a TLB response, a memory request is created and sent to the L1 cache. In the same cycle, an Access Fault exception is raised as a result of the illegal access. However, because the memory request is sent to the cache at the same time, the request cannot be squashed. Due to the lazy handling of the exception, the cache responds with the secret value in the next few cycles. The dispatcher forwards the secret to the next dependent instruction and eventually writes it to the physical register file.

b) PMP-protected data is not present in L1D: When the targeted secret data is not in the L1D we observe different behaviors in BOOM vs. XiangShan. In BOOM, the memory access is not squashed after the exception is raised. This means the request goes all the way to the L2 and eventually results in an LFB fill with 64 bytes of secret data, which we flag as potential leakage.

In XiangShan, the L1D miss response to the processor's LSU is slower than the hit response ($t3'$ vs. $t3$ in Figure 5). This gives the L1D cache additional time to handle the access exception on a miss. If an exception occurs, the L1D returns a "fake hit" response with a 0 data value, and does not generate a fill request to the L2. As a result, XiangShan is not vulnerable to leakage if the PMP-protected data is not in the L1D.

To cover all possible combinations of isolation boundary crossing, we tested the following scenarios, which all resulted in secret leakage on both processors:

- *D4*: Leaking enclave data/code to host user/supervisor.
- *D5*: Leaking Keystone Security Monitor data/code to host user/supervisor.
- *D6*: Leaking enclave data/code to another enclave.
- *D7*: Leaking host user/supervisor data/code to enclave.

7.1.5 D8: Leaking Secrets from the Store Buffer. XiangShan includes a store buffer that holds the pending writes to the L1D of committed store instructions. In order to test leakage through the store buffer, we used the TEESec Gadget Constructor to create instruction sequences that are executed right before the context switch from enclave to host. These instructions include multiple stores to different addresses inside enclave memory that fall in different cache lines. Next, we generated load instructions to access the same addresses right after the context switch to host user/supervisor. To make sure that none of these accesses are resolved by the L1D, we chose addresses that are not cached. We observed that the load requests are simultaneously sent to the L1D, store queue, and store buffer. The TEESec Checker reveals that the store buffer resolves several loads, transiently supplying enclave data to dependent instructions outside the enclave, in spite of an Access Fault exception being raised.

7.2 Enclave Metadata Leakage

In addition to accessing verbatim enclave data, enclave secrets can also be leaked through microarchitectural structures containing potential enclave metadata. Two metadata leakage instances were identified with TEESec. In the first one, hardware performance counters (HPC) were exploited to leak enclave secrets. In the second one, the Branch Target Buffer (BTB) is used to leak the control-flow of the enclave program.

7.2.1 M1: Exploiting HPCs. Hardware performance counters are special-purpose registers which track a wide range of microarchitectural events. Both BOOM and XiangShan offer a variety of HPCs grouped in different event sets, such as memory events and branch prediction events.

We explored multiple ways an attacker can leak enclave secrets through HPCs. The TEESec Checker identified multiple instances of HPC leakage across enclave boundaries in both BOOM and XiangShan. This occurs because none of the performance counters are reset at context switches and Keystone framework does not provide a software mechanism to clear these counters after entering or before exiting enclaves. This design shortcoming can be exploited by a malicious host in a Prime-and-Probe type attack: it first primes micro-architectural states (e.g, by resetting the HPCs, flushing cache, TLB, etc.) before entering the enclave and repeatedly interrupting enclave execution and probing the counter readings.

Even if mitigations are added by limiting certain sensitive HPCs to be only accessible by the security monitor, we demonstrate in Figure 6 that exploitation is still possible on XiangShan processor by utilizing a malicious interrupt service routine. Although reading the CSR is not allowed by the current execution privilege ($t2$-$t4$ in Figure 6), the value of the CSR can still be transiently written back to the register file. If an interrupt arrives after the write-back ($t3$) and before the faulting instruction is flushed from ROB ($t5$), the current state of the context including the logical register file will be stored in memory, filling up the store buffer. As previously mentioned in Section 7.1.5, in XiangShan values in the store buffer can be leaked to unprivileged users. Using the same technique, an attacker can retrieve the performance counter data pending at the store buffer. The BOOM processor is not vulnerable to such attacks because it waits for the privilege check and writes nothing to the register file when the check fails.

7.2.2 M2: Abusing Branch Prediction Units. The Branch Target Buffer (BTB) is one of the CPU internal structures responsible for resolving the target address of branch instructions. Similar to the cache, BTBs are set-associative structures with multiple levels. XiangShan, for example, utilizes a small directly mapped uBTB with 1024 entries in conjunction with a larger 4-way associative FTB holding a total of 4096 entries. The number of bits used for tags and set indexes is determined by the BTB size and associativity. For performance considerations, only a partial set of address bits is

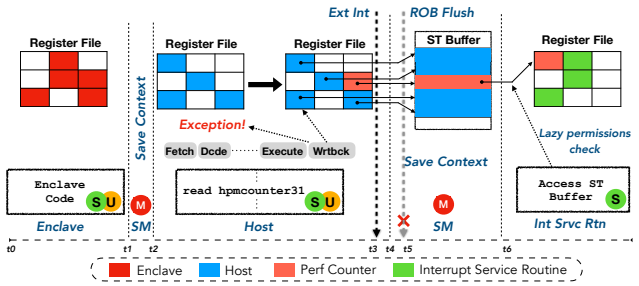

Figure 6: Leaking HPC data through the store buffer.

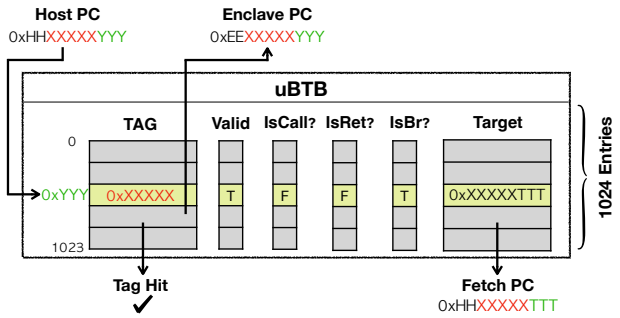

Figure 7: Host and enclave branches map to same uBTB entry.

used for tag matches. This allows an attacker to induce conflicts in BTB sets by mapping a shadow branch to a BTB entry with the same tag but at a different address.

In Figure 7, both *Host PC* and *Enclave PC* are mapped to the same uBTB entry as they only differ in the most significant bits which are excluded in tag lookups. As a result, the attacker can prime a BTB entry before entering the enclave, and probe the entry after exiting the enclave, to determine whether the branch instruction in the enclave has been taken or not. Such information reveals the program control flow and can be exploited to reveal enclave secrets as demonstrated in [28]. Both BOOM and XiangShan are exposed to these attack primitives because BTB structures are not flushed on context switches to/from enclave. There are also no software mechanisms deployed by Keystone to ensure no enclave metadata is left at enclave context switches.

8 COUNTERMEASURES

Possible countermeasures to the vulnerabilities we identified include selective flushing of microarchitectural structures on context switch, serializing permission checks with memory accesses, or sanitizing returned data. Table 4 summarizes the possible mitigations for each vulnerability. Some of these countermeasures have already been implemented in prior work. For example, MI6 [9] introduced a special *flush* instruction in the RiscyOO core to clear microarchitectural buffers and the L1D in enclave context switches. We note, however, that their design does not mitigate the *D1* and *D2* vulnerabilities, since they cannot be eliminated through flushing.

Hardware design changes may be needed to mitigate some of the vulnerabilities we identified. For example, an alternative to flushing is tagging shared microarchitectural structures with enclave IDs, and enforcing ID checks on every access. This can be done by extending tagged BTBs such as Intel eIBRS [26] to more CPU structures.

The PMP module can also be modified to serialize the permissions check and the memory access, thus mitigating cases *D4-D9*. XiangShan already performs PMP checks before the memory access to the PTW caches, which is why it is not vulnerable to *D2*. Alternatively, a lower-overhead solution is to allow the data access and permissions check in parallel, but zero-out returned data in case of a permission check failure (Table 4 column 4).

The PMP module could also be extended to initiate the flushing of all/select microarchitectural buffers at every PMP reconfiguration (performed on enclave context switch). We note that not all mitigations need to be deployed in all systems depending on threat models and other factors. While effective, some of the proposed countermeasures can have a significant performance penalty. We leave it to future work to evaluate the performance impact and examine possible optimizations.

	Flush L1D$	Flush ST Buffer	Clear Illegal Data Returns	Flush LFB	Flush/Tag BPU/HPC	Flush Everything
D1	-	-	-	-	-	-
D2	-	-	✓	-	-	-
D3	-	-	-	✓	-	✓
D4	✓*	-	✓	-	-	✓
D5	✓*	-	✓	-	-	✓
D6	✓*	-	✓	-	-	✓
D7	✓*	-	✓	-	-	✓
D8	-	✓	✓	-	-	✓
M1	-	-	-	-	✓	✓
M2	-	-	-	-	✓	✓

Table 4: Possible mitigations for the identified leakage cases. * items are only effective on XiangShan

9 CONCLUSION

We designed and implemented TEESec, the first pre-silicon framework for the joint verification of TEE software and microarchitectural implementation, with the goal of discovering complex security vulnerabilities in trusted execution environments. We evaluated TEESec on the BOOM and XiangShan RISC-V cores and identified 10 distinct instances of security violations. We believe TEESec demonstrates the importance of joint verification of TEE software and hardware, and the effectiveness of integrating security verification into pre-silicon RTL simulation.

ACKNOWLEDGMENTS

This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. This work was also supported in part by the Intel Corp. under the Side Channel Academic Program, the Air Force Research Laboratory under the Assured and Trusted Microelectronics Solutions award FA8650-20-C-1719, and the National Science Foundation under award 2018627.

REFERENCES

[1] Onur Aciiçmez. 2007. Yet another microarchitectural attack: exploiting I-cache. In *Proceedings of the 2007 ACM workshop on Computer security architecture*. 11–18.

[2] AMD 2014. *AMD64 architecture programmer's manual volume 2: System programming*. Technical Report. https://www.amd.com/system/files/TechDocs/24593.pdf, [Online; accessed 16-April-2022].

[3] AMD 2021. *SEV-ES Guest-Hypervisor Communication Block Standardization*. Technical Report. https://developer.amd.com/wp-content/resources/56421.pdf, [Online; accessed 21-April-2022].

[4] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* 40, 4 (2020), 10–21.

[5] ARM 2009. *Building a Secure System Using TrustZone Technology*. Technical Report. https://arm.com, Technical Report No. PRD29-GENC-009492C[Online; accessed 16-April-2022].

[6] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. 2021. CURE: A Security Architecture with CUstomizable and Resilient Enclaves. In *30th USENIX Security Symposium (USENIX Security 21)*. 1073–1090.

[7] Kristin Barber, Moein Ghaniyoun, Yinqian Zhang, and Radu Teodorescu. 2022. A Pre-Silicon Approach to Discovering Microarchitectural Vulnerabilities in Security Critical Applications. *IEEE Computer Architecture Letters* 21, 1 (2022), 9–12.

[8] Naomi Benger, Joop Van de Pol, Nigel P Smart, and Yuval Yarom. 2014. "Ooh Aah... Just a Little Bit": a small amount of side channel can go a long way. In *Cryptographic Hardware and Embedded Systems–CHES 2014: 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings 16*. Springer, 75–92.

[9] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, and Srinivas Devadas. 2019. MI6: Secure enclaves in a speculative out-of-order processor. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 42–56.

[10] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical.. In *WOOT*. 11–11.

[11] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. 2021. One glitch to rule them all: Fault injection attacks against AMD's secure encrypted virtualization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2875–2889.

[12] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC, 1041–1056.

[13] Christopher Celio, Pi-Feng Chiu, Borivoje Nikolic, David A. Patterson, and Krste Asanović. 2017. *BOOM v2: an open-source out-of-order RISC-V core*. Technical Report UCB/EECS-2017-157. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-157.html

[14] David Champagne and Ruby B Lee. 2010. Scalable architectural support for trusted software. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE, 1–12.

[15] Kevin Cheang, Cameron Rasmussen, Dayeol Lee, David W Kohlbrenner, Krste Asanović, and Sanjit A Seshia. 2022. Verifying RISC-V physical memory protection. *arXiv preprint arXiv:2211.02179* (2022).

[16] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2019. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*. 142–157.

[17] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX, 857–874.

[18] Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry Ponomarev, Nael Abu Ghazaleh, and Ryan Riley. 2014. Iso-x: A flexible architecture for hardware-managed isolated execution. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 190–202.

[19] Mohammad Rahmani Fadiheh, Johannes Muller, Raik Brinkmann, Subhasish Mitra, Dominik Stoffel, and Wolfgang Kunz. 2020. A Formal Approach for Detecting Vulnerabilities to Transient Execution Attacks in Out-of-Order Processors. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[20] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2021. Scalable Memory Protection in the PENGLAI Enclave. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 275–294.

[21] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software. Association for Computing Machinery, New York, NY, USA.

[22] Moein Ghaniyoun, Kristin Barber, Yinqian Zhang, and Radu Teodorescu. 2021. INTROSPECTRE: A pre-silicon framework for discovery and analysis of transient execution vulnerabilities. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 874–887.

[23] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. Association for Computing Machinery, New York, NY, USA.

[24] Marcus Hähnel, Weidong Cui, and Marcus Peinado. 2017. High-Resolution Side Channels for Untrusted Operating Systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA, 299–312.

[25] Intel 2014. *Intel Software Guard Extensions Programming Reference*. Technical Report. https://www.intel.com/content/dam/develop/external/us/en/documents/329298-002-629101.pdf, [Online; accessed 16-April-2022].

[26] Intel 2018. *Intel. Indirect Branch Restricted Speculation*. Technical Report. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html, [Online; accessed 18-Feb-2023].

[27] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. 2020. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*.

[28] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC, 557–574.

[29] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. 2022. A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP. In *2022 IEEE Symposium on Security and Privacy (SP)*. 337–351.

[30] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. 2019. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA, 1257–1272.

[31] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *30th USENIX Security Symposium (USENIX Security 21)*. 717–732.

[32] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. TLB Poisoning Attacks on AMD Secure Encrypted Virtualization. In *Annual Computer Security Applications Conference*. 609–619.

[33] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. 2000. Architectural support for copy and tamper resistant software. *Acm Sigplan Notices* 35, 11 (2000), 168–177.

[34] 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 *27th USENIX Security Symposium (USENIX Security 18)*. 973–990.

[35] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. 2016. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 406–418.

[36] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy*. 605–622.

[37] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. Cachezoom: How SGX amplifies the power of cache attacks. In *Cryptographic Hardware and Embedded Systems–CHES 2017: 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*. Springer, 69–90.

[38] Mathias Morbitzer, Manuel Huber, and Julian Horsch. 2019. Extracting secrets from encrypted virtual machines. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*. 221–230.

[39] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. 2018. Severed: Subverting amd's virtual machine encryption. In *Proceedings of the 11th European Workshop on Systems Security*. 1–6.

[40] Michael Neve and Jean-Pierre Seifert. 2006. Advances on Access-Driven Cache Attacks on AES. In *Proceedings of the 13th International Conference on Selected Areas in Cryptography* (Montreal, Canada) *(SAC'06)*. Springer-Verlag, Berlin, Heidelberg, 147–162.

[41] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. *The Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)* (2006), 1–20.

[42] Colin Percival. 2005. Cache missing for fun and profit.

[43] Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan Capkun. 2021. Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend. In *30th USENIX Security Symposium (USENIX Security 21)*. 663–680.

[44] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS '19 Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 753–768.

[45] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware guard extension: Using SGX to conceal cache attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings 14*. Springer, 3–24.

[46] Sanjit A. Seshia and Pramod Subramanyan. 2018. UCLID5: integrating modeling, verification, synthesis and learning. In *Proceedings of the 16th ACM-IEEE International Conference on Formal Methods and Models for System Design*. 1–10.

[47] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. 2016. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. 317–328.

[48] Rohit Sinha, Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani. 2015. Moat: Verifying confidentiality of enclave programs. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 1169–1184.

[49] Flavien Solt, Ben Gras, and Kaveh Razavi. 2022. {CellIFT}: Leveraging Cells for Scalable and Precise Dynamic Information Flow Tracking in {RTL}. In *31st USENIX Security Symposium (USENIX Security 22)*. 2549–2566.

[50] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit A Seshia. 2017. A formal foundation for secure remote execution of enclaves. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2435–2450.

[51] G Edward Suh, Charles W O'Donnell, Ishan Sachdev, and Srinivas Devadas. 2005. Design and implementation of the AEGIS single-chip secure processor using physical random functions. In *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 25–36.

[52] Haiyong Sun and Hang Lei. 2020. A design and verification methodology for a trustzone trusted execution environment. *IEEE Access* 8 (2020), 33870–33883.

[53] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. 2009. Complete information flow tracking from the gates up. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, Vol. 44. 109–120.

[54] Timothy Trippel, Kang G Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. 2022. Fuzzing hardware like software. In *31st USENIX Security Symposium (USENIX Security 22)*. 3237–3254.

[55] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology* 23 (2010), 37–71.

[56] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings for the 27th USENIX Security Symposium*. USENIX Association, 991–1008.

[57] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 178–195.

[58] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2020. SGAxe: How SGX Fails in Practice. https://sgaxeattack.com/.

[59] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 88–105.

[60] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. 2019. The severest of them all: Inference attacks against secure virtual enclaves. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. 73–85.

[61] Clifford Wolf. [n. d.]. Yosys Open SYnthesis Suite. http://www.clifford.at/yosys/.

[62] XiangShan 2021. *XiangShan Official Documentation*. Technical Report. https://xiangshan-doc.readthedocs.io, [Online; accessed 21-April-2022].

[63] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. 2020. SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities. *Network and Distributed System Security Symposium (NDSS)* (2020).

[64] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy*. 640–656.

[65] Yinan Xu, Zihao Yu, Dan Tang, Guokai Chen, Lu Chen, Lingrui Gou, Yue Jin, Qianruo Li, Xin Li, Zuojun Li, et al. 2022. Towards Developing High Performance RISC-V Processors Using Agile Methodology. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1178–1199.

[66] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, Vol. 43. 503–516.

[67] Sizhuo Zhang, Andrew Wright, Thomas Bourgeat, and Arvind Arvind. 2018. Composable building blocks to open up processor design. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 68–81.

[68] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. 2012. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 305–316.

[69] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. 2014. Cross-tenant side-channel attacks in PaaS clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 990–1003.

[70] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. (May 2020).

[71] Jiuqin Zhou, Yuan Xiao, Radu Teodorescu, and Yinqian Zhang. 2022. ENCLYZER: Automated Analysis of Transient Data Leaks on Intel SGX. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*. 145–156.

A ARTIFACT APPENDIX

A.1 Abstract

The TEESec framework consists of three main components: a Verification Plan, a Test Gadget Constructor, and TEESec Checker. The verification plan systematically profiles the processor design and enumerates all microarchitectural structures which may contain enclave metadata or data. The verification plan enumerates all memory access modalities covering all possible paths through which the enclave data/metadata can be accessed. This artifact includes BOOM and XiangShan RISC-V cores instrumented according to the verification plan to cover all memory access paths.

In the next phase, TEESec aims to exercise all the memory access modalities by crafting and assembling test gadgets. This artifact includes an automatic Test Gadget Constructor to generate RISC-V assembly test cases that are used by TEESec to identify leakage in the two processors we evaluate.

In the final phase, the automatically assembled test sequences are run through a cycle-accurate register transfer level (RTL) simulation of the design-under-test which emits a comprehensive simulation log containing the state of all microarchitectural structures identified in the verification plan. The TEESec Checker is also included in this artifact to analyze the execution logs and report potential secret leakage instances.

A.2 Artifact Checklist

- **Program:** Python scripts, RISC-V binary compilation, RTL simulation
- **Compilation:**
  - RISC-V GNU Compiler Toolchain (`riscv64-unknown-elf-gcc`)
  - GCC v8.5.0
- **Run-time environment:** Chipyard 1.3.0 framework
- **Hardware:** Compatible with x86 systems
- **Output:** Parsed execution log
- **Experiments:** Leak Keystone enclave data/metadata to host
- **How much disk space required (approximately)?:** Less than 100 GB
- **How much time is needed to prepare workflow (approximately)?:** Few hours including setting up dependencies
- **How much time is needed to complete experiments (approximately)?:** Less than an hour
- **Publicly available?:**
  - https://github.com/MoeinGhaniyoun/TEESec
  - https://doi.org/10.5281/zenodo.7796474

A.3 Description

This artifact consists of three components:

(1) Instrumented BOOM and XiangShan Chisel source code with `printf` statements to print-out microarchitectural state in RTL simulation log.
(2) Customized Berkeley-Bootloader (BBL) and `riscv-pk` to populate host page table entries and provide an interface to setup enclaves.
(3) Test Gadget Constructor and Checker Python files to generate RISC-V assembly test cases to evaluate each data/metadata access path and analyze the simulation log for enclave secret leakage.

A.3.1 How to access.

The entire framework is packaged as a single repository on GitHub at https://github.com/MoeinGhaniyoun/TEESec which includes all three parts above as submodules.

A.3.2 Hardware dependencies.

- x86 compatible system
- 16 GB+ of RAM
- 100 GB of free disk space

A.3.3 Software dependencies.

- Linux system
- Chipyard 1.3.0 framework
- Verilator 4.034
- RISC-V GNU Compiler (can be installed as part of Chipyard)
- Python 3.9+
- GCC 8.5.0

A.4 Building the Simulation Environment

In this section, we build the simulator executables for our test targets, the BOOM and XiangShan cores. Based on the access paths identified in Section 4.1, the source code of these processors is instrumented with `printf` statements to capture the microarchitectural state in a simulation log. If the verification plan is updated and adding a new access path is needed, the logging can be extended to cover the new microarchitectural structures. This is accomplished by updating the instrumentation of the processor's source code.

A.4.1 RISC-V BOOM.

(1) Clone Chipyard 1.3.0:
```
$ git clone https://github.com/ucb-bar/chipyard.git
$ cd chipyard
$ git checkout 1.3.0
```
(2) Modify `.gitmodules` and update the *url* field of *[submodule "generators/boom"]* to point to the instrumented BOOM. The new submodule path should look like this:
```
[submodule "generators/boom"]
path = generators/boom
url=https://github.com/MoeinGhaniyoun/BOOMv3-TEESec.git
```
(3) Run:
```
$ git config -global url."https://github.com/".insteadOf git://github.com
```
(4) Follow Chipyard docs at https://chipyard.readthedocs.io/en/1.3.0/Chipyard-Basics/Initial-Repo-Setup.html#initial-repository-setup and install all the required dependencies.
(5) Initialize the submodules, build for a *normal riscv-tools* setup and set the appropriate environment variables:
```
$ ./scripts/init-submodules-no-riscv-tools.sh
$ ./scripts/build-toolchains.sh riscv-tools
$ source env.sh
```
(6) Compile the customized BOOM simulator executable:
```
$ cd ./sims/verilator
$ make CONFIG=SmallBoomConfig
```
(7) The simulator executable can be found at the same directory by the name `simulator-chipyard-SmallBoomConfig`.

(8) The verification framework can be extended by updating the instrumentation of the BOOM source at `./generators/boom/src` and re-building the simulator as in step *6*.

A.4.2 RISC-V XiangShan.

(1) Follow README instructions at https://github.com/MoeinGhaniyoun/XiangShan-TEESec to prepare the environment
(2) Install mill from https://com-lihaoyi.github.io/mill/mill/Installation.html
(3) Install Verilator 4.034 from https://github.com/verilator/verilator/tree/v4.034
(4) Initialize submodules and create the simulator executable for XiangShan:
```
$ make init
$ make emu CONFIG=MinimalConfig -j10
```
(5) The simulator executable can be found at `XiangShan-TEESec/build/emu`.

A.5 Using Pre-Built Simulators

As an alternative to setting up the Chipyard 1.3.0 framework, we provide the simulator executable in the TEESec main repository, named "BOOM_simulator". This executable is pre-compiled and can accept RISC-V ELF files as its input. Similarly, for XiangShan the simulator executable provided in the TEESec main repository ("XiangShan_simulator") can be used. This executable is pre-compiled and can accept RISC-V bin files as input. In order to use the pre-built simulators, the RISC-V GNU Compiler should be installed, as follows:

A.5.1 Installing RISC-V GNU Compiler.

(1) `$ git clone https://github.com/riscv-collab/riscv-gnu-toolchain.git`
(2) `$ cd riscv-gnu-toolchain`
(3) `$ ./configure –prefix=/installation/path`
(4) `$ make -j10`
(5) Add `/installation/path/bin` to your PATH.

A.6 Setting up the Test Gadget Constructor

The Test Gadget Constructor and TEESec Checker are bundled into a single repository named TestGadgetConstructor-TEESec.

(1) To run the Constructor:
`$ python3 TestGadgetConstructor.py [Access_Gadget] [Secret]`
The first argument specifies which memory access path should be checked for potential vulnerability and the second argument specifies a secret value to be seeded in enclave memory. A full list of available Access Gadgets is included in `TestGadgetConstructor-TEESec/access_gadgets.txt`.
(2) To run the Checker:
`$ python3 Checker.py [Sim_Log] [Secret]`
Here the first arguments points to the location of the generated simulation log and the second argument points to the secret value. In case a secret is found, a snapshot of the processor's state at the exact simulation cycle is generated with the name "CheckerLog.txt".

A.7 Experiment workflow

Next, we walk through leakage case *D4*, described in Section 7.1.4, to illustrate the TEESec experimental workflow. Case *D4* demonstrates that enclave secrets can be leaked to the host user/supervisor. Other leakage scenarios can be reproduced with a similar flow. Here, leakage is defined as ability to access enclave data while executing in host. The secrets can be found in cache response data and, later, in the register file.

A.7.1 Test Case D4: Explicit enclave data leakage.

(1) `$ cd TEESec`
(2) `$ python3 TestGadgetConstructor-TEESec/TestGadgetConstructor.py Exp_Enc_L1 0xdeadbeef`
This will modify the `dummy_entry.S` file at `riscv-pk-TEESec/dummy_payload/dummy_entry.S` and populate it with generated assembly instruction for both enclave and host based on the provided secret and access path. It also starts the simulation with the BOOM_simulator and creates a simulation log file named SimLog.txt in the current directory.
(3) In order to run the Gadget Constructor with a custom version of the instrumented simulator, a third argument needs to be provided to the Constructor to point to the path of the custom simulator.
(4) `$ python3 TestGadgetConstructor-TEESec/Checker.py ./SimLog.txt deadbeef`
This will start the Checker that analyzes the simulation log to locate where the enclave secret is illegally accessed by the host.
(5) Examine CheckerLog.txt and see whether the seeded secret was illegally accessed by the host. Output should look like this:

```
Enclave secret leakage detected!
Secret value: 0xdeadbeef
Microarchitecture structure: Register-file
Sim Cycle No.: 234785
PC of Last Committed Inst.: 0x80004808
```

A.7.2 Pre-Generated Test Cases. The TEESec distribution also includes all the test cases that identified the secret leakage we report in the paper. These can be accessed as assembly code at `riscv-pk-TEESec/pre-generated/dummy_entry.S`. This file also contains all the required test cases to reproduce all access paths discussed in the paper. To use this file as a test case, the `dummy_entry.S` at `riscv-pk-TEESec/dummy_payload/` should be replaced with the pre-generated test case file. Next, the `TestGadgetConstructor.py` should be run with "NO-FUZZING" as the first argument.

A.8 Evaluation and Expected Results

For each testing round in which a test case is generated and simulated on the BOOM or XiangShan processors, if a potential leakage case is found, a `CheckerLog.txt` is generated. It includes information about the secret value, where it was accessed, a comprehensive snapshot of the state of processor when the secret was accessed and the microarchitectural structure where the secret was found.