Secure TLBs

Shuwen Deng, Wenjie Xiong, Jakub Szefer
{shuwen.deng,wenjie.xiong,jakub.szefer}@yale.edu
Yale University

ABSTRACT

This paper focuses on a new attack vector in modern processors: the timing-based side and covert channel attacks due to the Translation Look-aside Buffers (TLBs). This paper first presents a novel three-step modeling approach that is used to exhaustively enumerate all possible TLB timing-based vulnerabilities. Building on the three-step model, this paper then shows how to automatically generate micro security benchmarks that test for the TLB vulnerabilities. After showing the insecurity of standard TLBs, two new secure TLB designs are presented: a Static-Partition (SP) TLB and a Random-Fill (RF) TLB. The new secure TLBs are evaluated using the Rocket Core implementation of the RISC-V processor architecture enhanced with the two new designs. The three-step model and the security benchmarks are used to analyze the security of the new designs in simulation. Based on the analysis, the proposed secure TLBs can defend not only against the previously publicized attacks but also against other new timing-based attacks in TLBs found using the new three-step model. The performance overhead is evaluated on an FPGA-based setup, and, for example, shows that the RF TLB has less than 10% overhead while defending all the attacks.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; • **Computer systems organization** → *Embedded hardware*; Reduced instruction set computing.

KEYWORDS

timing side and covert channel attacks, timing attack defenses, TLBs

ACM Reference Format:
Shuwen Deng, Wenjie Xiong, Jakub Szefer. 2019. Secure TLBs. In *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3307650.3322238

1 INTRODUCTION

Research on timing-based attacks (and defenses) in processors has a long history. To-date, researchers have focused mainly on the memory subsystem when showing the different timing-based attacks, and have, for example, demonstrated a plethora of timing-based channels in caches, e.g. [1–3, 10, 20]. All the attacks have shown the possibilities to extract sensitive information via the timing-based channels, and often the focus is extracting cryptographic keys.

Originally, many timing-based attacks may not have gained much attention as they were not considered practical. However, the recent Spectre [14] and Meltdown [17] attacks have shown that timing-based channels are more dangerous than previously thought. The Spectre and Meltdown attacks consist of two parts: first, speculative execution is used to access sensitive information, second, a timing-based channel is used to actually transfer the information to the attacker. Whether by themselves, or in combination with speculative execution, the timing-based channels in microarchitecture pose threats to system security, and should be mitigated.

Most mitigations of timing-based attacks in the memory subsystem have focused on design of secure caches, e.g. [4, 6, 16, 18, 27–29, 31–33]. Meanwhile, this work focuses on preventing timing-based attacks due to TLBs, and presents the first hardware defenses for TLBs.

Similar to caches, and all cache-like structures, timing variations due to hits and misses exist in TLBs and can be leveraged to build practical timing-based attacks [8, 12]. For example, in [8], it is shown that TLB timing channels can be used to extract the cryptographic key from the RSA public-key algorithm with a 92% success rate. Timing-based channels in TLBs are distinct in that they are triggered by memory translation requests, not by direct accesses to data. They also have a different granularity (pages vs. cache lines for data or instruction caches), and, in commercial processors, TLBs have more complicated logic, compared to caches, due to support for various memory page sizes. Further, defending cache attacks does not protect against TLB attacks [8]. Moreover, there has not been a systematic security analysis of the TLB vulnerabilities, nor concrete proposals for secure TLB design. This paper provides both.

This work starts by providing a novel three-step modeling approach to exhaustively enumerate all possible TLB timing-based vulnerabilities. Similar to the recent work on modeling cache attacks [5], this work develops a novel model of the attacker and the victim behavior in relation to the TLB states. Rather than modeling software attacks, the three-step approach analyzes possible victim or attacker behaviors that affect the TLB state. All possible combinations of the attacker and victim behaviors are evaluated, and systematically reduced to only three-step behaviors that can result in timing-based attacks. In total, 24 possible vulnerabilities were found, of which only 8 map to existing attacks [8, 12]. We believe that the other 16 are new attack types not previously considered.

Based on the three-step model, micro security benchmarks are then semi-automatically generated. Specifically, each vulnerability type in the three-step modeling approach is translated into concrete sets of assembly instructions which can be executed on the processor simulator to test the TLB.

Armed with the three-step model and the security benchmarks, the security of different typical configurations of TLBs are tested

ISCA '19, June 22–26, 2019, Phoenix, AZ, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6669-4/19/06…$15.00
https://doi.org/10.1145/3307650.3322238

using Rocket Core implementation of the RISC-V processor architecture. *Standard TLBs*, i.e. Fully-Associative (FA) or Set-Associative (SA) TLBs, which include process IDs, e.g. ASID in RISC-V architecture, are shown to be vulnerable to many of the attacks. Consequently, this work presents new defenses. Especially, we present two new secure TLB designs in hardware: a Static-Partition (SP) TLB and a Random-Fill (RF) TLB, latter of which is more complex but can defend all of the attacks. These are the first hardware defenses to TLB attacks.

Our security evaluation shows that the standard SA TLBs are able to defend 10 types of external hit-based attacks (labeled EH in Table 2) and attacks requiring obtaining a TLB hit between different processes (i.e. process IDs). The SP TLB is able to further prevent 4 more external miss-based vulnerabilities (labeled EM in Table 2). In total it can defend 14 out of the 24 vulnerabilities. Meanwhile, the RF TLB is able to prevent all of the 24 possible timing-based vulnerabilities in TLBs.

To help understand the impact of the new secure TLBs on the system performance, a RISC-V Rocket Core based processor with the new secure TLBs are synthesized on the Zynq ZC706 and ZedBoard FPGAs. This allows for running security software alongside SPEC 2006 benchmarks and a full Linux system. Based on our evaluation, for example, the SP TLB has 3x misses per kilo-instructions (MPKI) compared to the standard SA TLB, while the RF TLB has 9% more MPKI than the standard TLB. For the RF TLB, the hardware cost of the defenses is about 8% more logic.

1.1 Contributions

This work's contributions are:

- Present the first three-step modeling approach that can be used to reason about all hit-based and miss-based timing-based attacks on TLBs.
- Develop a semi-automated method to generate micro security benchmarks for each vulnerability from the TLB three-step model.
- Design the first hardware defense for TLB attacks: the new SP TLB and the new RF TLB, and realize them in a Rocket Core implementation of a RISC-V processor.
- Evaluate the security of the standard and secure TLBs using the micro security benchmarks running on RISC-V simulation, and show that the results match with theoretical channel capacity calculation.
- Test performance by synthesizing the hardware on FPGAs, and running RSA decryption tests alongside SPEC 2006 benchmarks under Linux on the FPGAs.

The Chisel code for the Rocket Core and the secure TLB designs will be released under open-source license and will be available at http://caslab.csl.yale.edu/code/securetlbs/.

2 RELATED WORK

This section reviews existing work on caches (most closely related to TLBs) and the few existing works on TLBs. Dedicated surveys of microarchitectural timing-based attacks and defenses, e.g., [23], cover details of other types of attacks and defenses.

2.1 Timing-Based Attacks and Caches

In modern processor caches there are timing differences between cache hits (fast) and cache misses (slow), and these variations in timing have been exploited to leak sensitive information. Especially, a large number of different cache timing-based side-channel and covert-channel attacks have been presented in literature, e.g. [1–3, 10, 20]. And, there are many secure hardware cache designs which aim to prevent these different attacks, e.g. [4, 6, 16, 18, 27–29, 31–33]. However, even if the cache-based attacks are mitigated, TLB-based attacks are the next attack vector that malicious attackers might use – and hence are the focus of this work.

2.2 Timing-Based Channels in TLBs

Compared with caches, there are only two published TLB-based timing attacks [1] [2]. TLBleed attack [8] uses timing-based channels combined with machine learning to create attack which is able to leak bits of secret keys from the RSA algorithm (they also show attack for the EdDSA algorithm). They leverage the Prime + Probe [19] attack strategy previously applied in processor caches.

Prior to TLBleed, the Double Page Fault attack [12] leverages the Cache Collision [3] attack strategy previously applied in processor caches. It requires the victim to access some kernel memory pages twice, and uses the fact that an access to a previously allocated kernel virtual pages will bring in TLB entries, even if page fault is generated and accesses permission checks failed. The timing of the second access thus reveals information on whether a inherent TLB hit happened.

Beyond these individual attacks, there are neither exhaustive categorizations nor models of possible TLB timing-based attacks – as are proposed in this work.

2.3 Existing Approaches to Securing TLBs

Currently, we are only aware of five approaches (mostly software-based) that can help mitigate some TLB attacks, but are not as effective as our hardware secure TLBs.

First, today's Linux system makes use of virtual addresses and process identifiers, e.g. ASID on RISC-V, to identify different processes in the SA TLBs. When the attacker and the victim are separate processes that are assigned different process identifiers in hardware, external hit-based vulnerabilities (labeled as *EH* vulnerability macro types in Table 2) are not possible. This can defend 10 of the 24 attack types in our categorization.

Second, in the Sanctum [4] secure processor design, the per-core SA TLBs are flushed by a *security monitor* software whenever a core switches between enclave and non-enclave code. This defense adds protection for 4 more external miss-based attacks (labeled as *EM*

[1]The Leaky Cauldron [26] attack is also related to TLB and targets Intel SGX. However, it does not depend on hits and misses in the TLB, but instead it relies on the assumption that the attacker can evict the enclave entries in the TLB, so an enclave's memory access will trigger a page table walk, and the malicious OS can get the page access pattern trace.

[2]The Malicious Management Unit [24] attack makes use of the Memory Management Unit (MMU) to build eviction set of virtual addresses to allow the page table entries to map to certain cache sets in the CPU caches (especially in the Last-Level Cache). In this case, eviction sets which can bypass the software-based defenses are formed and can trigger cache timing-based attacks in LLC. However, similar to the Leaky Cauldron [26] attack, this attack also does not depend on hits and misses in the TLB.

vulnerability macro types in Table 2) over SA TLB without flushing by a security monitor, for a total prevention of 14 out of 24 attacks.

Third, Intel SGX also flushes SA TLBs during switching between enclave and non-enclave code [13]. The flushing in Intel is presumably done in hardware, as opposed to software in Sanctum. It can also defend the 14 attacks, same as Sanctum.

Fourth, the InvisiSpec [30] work proposes to prevent observable changes to D-TLBs during speculation, by delaying any TLB updates and page table walks until the speculative loads reach point of visibility. It focuses on speculative channels, but cannot defend against conventional ones.

Fifth, some processors employ FA TLBs, which by design do not have different TLB sets (there is only one set). Miss-based vulnerabilities (labeled as $*M$ vulnerability macro types in Table 2) do not apply to FA TLBs. Such TLBs can defend 18 of the attacks.

There are also software techniques that can help mitigate some TLB timing-based attacks. For example, the global bit, in x86 processors, will prevent pages from being flushed. However, there are other ways to invalidate a page, e.g. using TLB coherence, to make invalidation related attacks possible. Using large pages for the crypto libraries can also be one possible software defense to TLB timing-based attacks.

Unlike all the existing work, this work presents two new hardware secure TLB designs, including the RF TLB which can prevent all types of timing-based attacks according to the three-step model, and has about same performance as a SA TLB.

3 MODELING TLB TIMING-BASED VULNERABILITIES

To analyze all the possible timing-based TLB attacks, this section presents a three-step modeling approach which can be used to reason about the behavior of the TLB logic and to derive all the possible timing-based vulnerabilities.

3.1 Threat Model and Assumptions

A TLB timing-based attack involves an attacker and a victim. In many cases they are executing on the same processor core, a set of cores, or a set of hyper-threads which share same physical TLB, but this is not required for all types of the attacks. In this paper, we use A and V to denote the attacker and the victim with different process IDs. For the attacks where the attacker and the victim are in the same address space, attacker is able to trigger some known address memory operation as if it were the victim, e.g. states V_a and $V_{a^{alias}}$ in Table 1 can be actually attackers.

We assume, in hardware, all memory operations are identified by the virtual memory address, *vaddr* (including null address in case of certain TLB flush related operations) and the process ID (including null process ID in case of certain TLB flush related operations), e.g., ASID in RISC-V.

The victim is assumed to have some security critical memory range, x, within which the access pattern depends on the secret the attacker wants to learn. An example of a security critical region is the set of page entries accessed during execution of the RSA functions of *libgcrypt*, where the value of the key bit (either 0 or 1) determines which specific memory pages are accessed. The timing of the accesses to the security critical memory range is affected by the timing of TLB related operations, and it can reveal information such as cryptographic keys.

The attacker is assumed to know the victim software, e.g. what implementation of a cryptographic algorithm it uses, but not the secret cryptographic keys. He or she is assumed to know the size, *ssize*, and the location, *sbase* (in virtual memory) of the security critical memory range x. And, the attacker is assumed to know the TLB state machine logic; although during run-time of the processor the attacker cannot access the internal state of the hardware TLB – he or she can only observe the timing of the memory operations and try to deduce the state of the TLB from the timing.

The attacker can measure the timing of its own memory operations or operations of the victim; but cannot access the actual sensitive data being processed by the victim. In most cases, the attacker can also force the victim to execute specific operations, e.g. force the victim to perform decryption while the attacker measures timing. Thus, even if some operation is done by the victim, it is under control of the attacker so attacker can measure the timing. The timing can be identified by the attacker as $fast$ or $slow$.

The timing attacks can be both side-channel attacks and covert-channel attacks. The difference between the two is that the victim in the side-channel scenario is the sender in the covert-channel scenario. Regardless of the channel type, we use V for victim (or sender) and A for attacker (or receiver).

Our threat model assumes that high-level OS page table related channels are already mitigated. E.g. TLB miss can take variable amount of time depending on whether there already exists a page table translation, or whether the OS has to create a new translation entry during a page fault. We focus on address translation data of the TLB structure. We do not consider possible TLB timing channels that are due to port contention, LRU replacement, or any directory structures. We also do not consider Page Walk Cache [9, 25] effect on storing intermediate translations of memory pages[3].

3.2 Introduction of the Three-Step Model

One observation we make is that all existing TLB timing-based attacks take three steps. In $Step$ 1, a memory operation is performed, placing the TLB block (also called TLB slot or TLB entry) in a known initial state (e.g. a new translation is put into the block or block is invalidated). Then, in $Step$ 2, a second memory operation alters the state of the TLB block from the initial state. Finally, in $Step$ 3, a final memory operation is performed, and the timing of the final operation reveals some information about the relationship among the addresses from $Step$ 1, $Step$ 2 and $Step$ 3. Attacks with more than three steps can be reduced to a three-step attack, as shown in Appendix A.

We write the three steps as: $Step\ 1 \rightsquigarrow Step\ 2 \rightsquigarrow Step\ 3$ which indicates a sequence of steps taken by the attacker or the victim. Table 1 lists all the 10 possible states of the TLB block for each step of our three-step model.

Each step in the model represents a state of a TLB block, since all the TLB blocks are updated following the same TLB state machine logic, it is sufficient to consider only a TLB block as it is the smallest unit of the TLB. Different implementations of TLBs involve different mapping functions for the TLB blocks. However, this does not affect

[3]So far Page Walk Cache does not exist in RISC-V architecture.

Table 1: The 10 possible states for a single TLB block in our three-step vulnerability modeling procedure.

States	Description
V_u	The TLB block contains translation for a memory address u, translation which is placed in the TLB block due to a memory access by the victim. Attacker does not know u, but u is from a range x of memory locations, range which is known to the attacker. The address u may have same page index as A_a or V_a and thus conflict with them in the TLB block. The goal of the attacker is to learn the page address or index of V_u.
A_a or V_a	The TLB block contains translation for a memory address a. The translation is placed in the TLB block due to a memory access by the attacker, A_a, or the victim, V_a. The attacker knows the address a, independent of whether the access was by the victim or the attacker themselves. The address a is from within the range of sensitive locations x. The address a may or may not be the same as the address u.
$A_{a^{alias}}$ or $V_{a^{alias}}$	The TLB block contains translation for a memory address a^{alias}. The translation is placed in the TLB block due to a memory access by the attacker, $A_{a^{alias}}$, or the victim, $V_{a^{alias}}$. The address a^{alias} is within the range x. It is not the same as a, but it has same page index and can map to the same TLB block, i.e. it "aliases" to the same block.
A_{inv} or V_{inv}	The TLB block previously containing translation for a memory address is now invalid. The translation is "removed" from the TLB block by the attacker A_{inv} or the victim V_{inv} as the result of TLB block being invalidated, e.g. due to synchronization updates to in-memory memory-management data structures or due to context switch between processes which causes OS to flush per-core TLB entries.
A_d or V_d	The TLB block contains translation for a memory address d. The translation is placed in the TLB block due to a memory access by the attacker, A_d, or the victim, V_d. The address d is not within the range x.
$\star$	Any data, or no data, can be in the TLB block. The attacker has no knowledge of page address in this TLB block.

the model, as the steps target on only one single TLB block. Different TLBs may make it more difficult in practice for attacker and victim to access the same block, but once they can achieve that – qualifying the practical difficulty of achieving certain steps is not part of the model, the model shows if there is a possibility of an attack or not, independent of the practical difficulty.

3.3 Derivation of All TLB Vulnerabilities

Based on the states possible in each step there are in total $10 * 10 * 10 = 1000$ combinations of possible three-steps. We developed an algorithm that can process the list of all the three-steps, and eliminates ones which cannot lead to an attack. A three-step combination cannot become a vulnerability if it satisfies one of the below rules:

(1) A $\star$ is not possible in *Step* 2 or *Step* 3, having $\star$ in the step means the TLB is in an unknown state and this removes useful information for the attacker.

(2) A V_u must be in one of the steps. If there is no unknown u in the steps, there is nothing for the attacker to learn.

Table 2: The table shows all the timing-based TLB vulnerabilities. *Attack Strategy* column gives our common name for each set of one or more specific vulnerabilities that would be exploited in an attack in a similar manner (many of the names are borrowed from cache timing-based attacks in literature). *Vulnerability Type* column gives the three steps that define each vulnerability. For *Step* 3, *fast* indicates a TLB hit must be observed, while *slow* indicates a TLB miss must be observed. *Macro Type* column proposes the categorization the vulnerability belongs to. "E" is for external interference vulnerabilities. "I" is for internal interference vulnerabilities. "M" is for miss-based vulnerabilities. "H" is for hit-based vulnerabilities. *Attack* column shows if a type of vulnerability has been previously presented in literature.

Attack Strategy	Vulnerability Type			Macro Type	Attack
	Step 1	*Step* 2	*Step* 3		
TLB Internal Collision	A_{inv}	V_u	V_a (fast)	IH	(1)
	V_{inv}	V_u	V_a (fast)	IH	(1)
	A_d	V_u	V_a (fast)	IH	(1)
	V_d	V_u	V_a (fast)	IH	(1)
	$A_{a^{alias}}$	V_u	V_a (fast)	IH	(1)
	$V_{a^{alias}}$	V_u	V_a (fast)	IH	(1)
TLB Flush + Reload	A_{inv}	V_u	A_a (fast)	EH	**new**
	V_{inv}	V_u	A_a (fast)	EH	**new**
	A_d	V_u	A_a (fast)	EH	**new**
	V_d	V_u	A_a (fast)	EH	**new**
	$A_{a^{alias}}$	V_u	A_a (fast)	EH	**new**
	$V_{a^{alias}}$	V_u	A_a (fast)	EH	**new**
TLB Evict + Time	V_u	A_d	V_u (slow)	EM	**new**
	V_u	A_a	V_u (slow)	EM	**new**
TLB Prime + Probe	A_d	V_u	A_d (slow)	EM	(2)
	A_a	V_u	A_a (slow)	EM	(2)
TLB version of Bernstein's Attack	V_u	V_a	V_u (slow)	IM	**new**
	V_u	V_d	V_u (slow)	IM	**new**
	V_d	V_u	V_d (slow)	IM	**new**
	V_a	V_u	V_a (slow)	IM	**new**
TLB Evict + Probe	V_d	V_u	A_d (slow)	EM	**new**
	V_a	V_u	A_a (slow)	EM	**new**
TLB Prime + Time	A_d	V_u	V_d (slow)	IM	**new**
	A_a	V_u	V_a (slow)	IM	**new**

(1) Double Page Fault attack [12].
(2) TLBleed attack [8].

(3) A $\star$ in one step, followed by V_u in next step cannot lead to an attack, since the TLB block needs to be in some known state before V_u is placed into it.

(4) Three-step patterns with two adjacent steps repeating, or both known to the attacker, can be eliminated[4].

(5) Steps involving a known address a and an alias to that address a^{alias}, give same information, thus three step combinations which only differ in use of a or a^{alias} cannot represent different attacks, and only one combination needs to be considered, e.g., $V_u \rightsquigarrow A_{a^{alias}} \rightsquigarrow V_u$ is a repeat type of $V_u \rightsquigarrow A_a \rightsquigarrow V_u$, and one of the two can be eliminated.

[4]Some of the possible attacks involve only two steps, but these attacks are represented by three-step model where first step is an explicit $\star$, i.e., they are represented by patterns $\star \rightsquigarrow \cdots$.

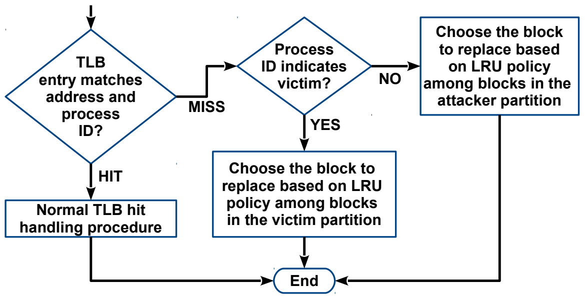
TLB entry matches address and process ID?
MISS
HIT
Process ID indicates victim?
NO
YES
Choose the block to replace based on LRU policy among blocks in the attacker partition
Choose the block to replace based on LRU policy among blocks in the victim partition
Normal TLB hit handling procedure
End

Figure 1: SP TLB access handling procedure flow chart.

(6) An *inv* related state cannot be in *Step* 2 or *Step* 3 because it is so far not possible for most ISAs to allow flushing of the TLB from user space. (See more discussion in Appendix B).

(7) If measured timing corresponds to more than one possible sensitive address translation of the victim, the corresponding vulnerability is removed. E.g., $\star \rightsquigarrow A_a \rightsquigarrow V_u$ is removed because when observing a fast timing, u can possibly map to a, or first step's potential u that is included in $\star$.

After applying the script which implements our simplification algorithm, 34 three-step access patterns remain as candidates for possible timing-based TLB attacks. These 34 access patterns are further manually reduced to a list of 24 types of timing-based TLB vulnerabilities, listed in Table 2. Due to space limitation, details on why the 10 patterns cannot form vulnerabilities are not included in the paper.

To summarize all the vulnerability types, Table 2 shows the list of all the 24 vulnerability types, along with a more coarse-grained attack strategies, which cover one or more vulnerability types. The list of vulnerability types can be further collected into four simple macro types: internal interference miss-based (IM), internal interference hit-based (IH), external interference miss-based (EM), external interference hit-based (EH).

All types of vulnerabilities only involving the victim, V, in the states in *Step* 2 and *Step* 3 are called internal interference vulnerabilities (I). The remaining ones are called external interference (E). Some vulnerabilities allow attacker to learn that the page address of the victim maps to the TLB set of the attacker by observing *slow* timing due to a TLB miss. we call these miss-based vulnerabilities (M). The remaining ones leverage observation of $fast$ timing due to a TLB hit, and are called hit-based vulnerabilities (H).

Most of the vulnerability types have not been explored before, except for two groups. The Double Page Fault attack [12] is effectively based on the Internal Collision, and it maps to types labeled (1) in the Attack column in Table 2. The TLBleed attack [8] is effectively based on the Prime+Probe strategy, and it maps to types labeled (2) in the Attack column in Table 2. All other vulnerability types correspond to new attacks not previously discussed.

4 SECURE TLB DESIGNS

In order to prevent timing-based vulnerabilities, we designed two secure TLBs, the SP TLB and the RF TLB. Designs of the secure TLBs follow the threat model discussed in Section 3.1. We focus on L1 D-TLB in this work, but it can be applied to instruction TLBs as well as other levels of TLB.

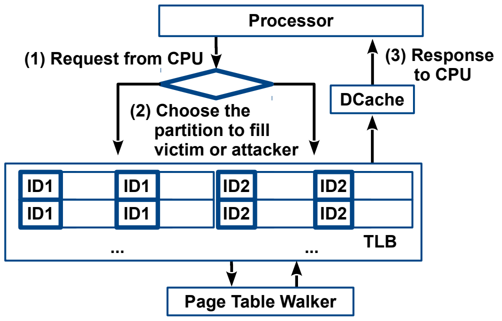

Figure 2: Sample block diagram of SP TLB with victim (ID1) and attacker (ID2) part being allocated 50% of TLB space.

4.1 Static-Partition (SP) TLB

SP TLB is a SA TLB where certain ways are assigned to a victim process and other ways are assigned to all remaining processes, which by default are assumed to be potential attacker processes. The process ID, e.g. ASID in RISC-V, is used to differentiate the victim and the attacker. The number of ways assigned to each is set at design time, but could be further extended to be dynamic at run time.

4.1.1 SP TLB Access Handling Procedure. SP TLB isolates the accesses between the victim and the attacker. TLB hits are identical to SA TLB, where both address and process ID must match. For TLB misses, the victim's address translations cannot cause replacement in the attacker's partition, and the attacker's address translations cannot cause replacement in the victim's partition. Each partition maintains its own LRU policy, which can prevent some LRU attacks, but defense of LRU related attacks is not focus of this work as discussed in the threat model. The SP TLB access handling procedure is shown in Figure 1.

4.1.2 SP TLB Logic. The SP TLB (Figure 2) partitions the victim and the attacker by cache ways. The allocation of different partitions is configurable during the design time. Assuming there are M ways in total. The victim partition will take N ($0 < N < M$) ways while the attacker partition will take the remaining $M - N$ ways. In our implementation, the victim and the attacker part are allocated 50% of TLB ways by default. Process ID field of each entry in the TLB is reused by the SP TLB to determine whether a partition is victim's or attacker's.

SP TLB requires minimal changes to the TLB logic, and protects 14 out of the 24 vulnerabilities shown in Section 5.

4.2 Random-Fill (RF) TLB

To protect all the vulnerabilities, we propose Random-Fill TLB, which is able to de-correlate the requested memory access from actual TLB entries that are brought into the TLB, making the attacker's observations non-deterministic. For TLB hits, the behavior is the same as the SA TLB. For TLB misses, depending on the memory address region, a random address translation will be fetched

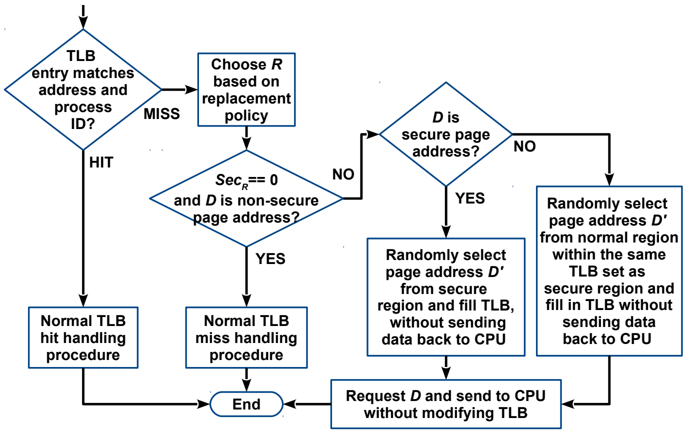
TLB entry matches address and process ID?
MISS
HIT
Choose *R* based on replacement policy
Sec_R == 0 and *D* is non-secure page address?
NO
YES
D is secure page address?
NO
YES
Normal TLB hit handling procedure
Normal TLB miss handling procedure
Randomly select page address *D'* from secure region and fill TLB, without sending data back to CPU
Randomly select page address *D'* from normal region within the same TLB set as secure region and fill in TLB without sending data back to CPU
End
Request *D* and send to CPU without modifying TLB

Figure 3: RF TLB access handling procedure flow chart.

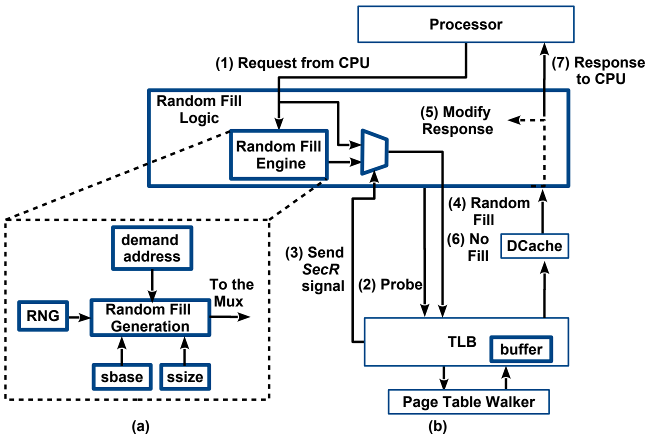

Figure 4: (a) Random Fill Engine, (b) RF TLB block diagram.

into the TLB ("random fill"), while the originally requested address is directly sent back to the CPU without filling the TLB ("no fill"). The RF TLB also introduces the *Sec* bit which is used to identify certain memory translation entries are belonging to secure data.

4.2.1 RF TLB Access Handling Procedure. RF TLB access handling procedure is shown in Figure 3. D denotes the requested address translation. D' is a random address translation to be filled in the TLB (D and D' are possibly the same because of the randomization). R is the TLB entry that would be evicted by D, in TLB set S, according to the LRU replacement policy. R' is the TLB entry evicted by D'. Sec_R and Sec_D is the Sec bit of R and D, respectively, indicating whether the page address is in the secure region.

If D maps to an existing entry in the TLB (page address and process ID matches), a normal TLB hit handling procedure will occur. Otherwise:

- If Sec_R is 0 and Sec_D is 0, normal TLB miss occurs.
- If Sec_R is 1 and Sec_D is 0, then D' is chosen as a random virtual non-secure page address, within the same sets of TLB entries as the secure region, and filled in TLB, evicting R'. Meanwhile, R will not be evicted and results of D request will be sent to the processor directly. Thus an attacker cannot deterministically evict the secure address chosen by the replacement policy.
- If Sec_D is 1, then D' is chosen as a random virtual address within the secure region, and filled in TLB, evicting R'. Results of D request will be sent to the processor directly. Thus, an attacker cannot observe TLB state changes due to secure page address D, but he or she instead observers TLB state changes due to random page address D'.

RF TLB uses the randomization approach to randomly bring in data from specified memory ranges to confuse the attacker. It does not randomize all of the TLB accesses so as to limit the performance impact. The RF TLB is able to prevent all types of timing-based vulnerabilities shown in Table 2, which are discussed in Section 5.

4.2.2 RF TLB Logic. RF TLB block diagram is shown in Figure 4b. All the bold lines and blocks are the added hardware and logic extension. In the TLB array, an extra field (a secure bit *Sec*, either 0 or 1) is added to each of the TLB entries to indicate whether it contains an address translation within the secure region. In addition, the existing process ID field (e.g., ASID in RISC-V) in each TLB entry is used to differentiate the victim and the attacker process. By default, we set specific process ID 1 for the victim program and all other ASIDs to be attackers.

An extra set of registers is added to store the process ID of the victim process and the start address, $sbase$, and the size, $ssize$ of the secure region (the base and size are defined in terms of pages, usually 4KiB). The registers can be managed by a trusted OS to change the victim process ID and secure regions when different victim programs need protection.

An extra *buffer* is added which stores equivalent of one TLB entry. It is used as temporary storage for translation data that is returned to the CPU, but which should not be placed in the TLB. It will be cleaned up after the address is returned.

The Random Fill Engine (RFE), shown in Figure 4a is used to generate addresses which should be used for TLB updates[5]. In Figure 4b, (1-2), the "no fill" *fill_type* will first be sent to TLB. On a TLB miss, the TLB will *probe* the page address without filling TLB entries to see if the chosen entry has a valid secure page address translation. Then, (3) the Sec_R bit is set and sent back. Next, (4) if it is a request to the secure region or the Sec_R bit is one, a random fill request will be triggered. If the original request is in the secure region, a random virtual page address is derived from RFE within the secure region [$sbase$, $sbase + ssize$], and a translation will be put into the TLB entry. If the original request comes from the non-secure region, most of the higher bits of the requested address are remain the same while the bits that correspond to the TLB set index[6] will be randomized to make the eviction indeterministic. Next, (5) the *Random Fill Logic* will modify the response and prevent the random fill result from being sent to the processor. Then, (6)

[5]We assume the OS has pre-generated page table entries that may correspond to the random virtual address generated by the RFE, which may not be actually used by the original program, to prevent OS or software-based timing attacks due to page faults when a page entry for a random address is looked up by the TLB.

[6]The TLB set index to be randomized has bit size $S_n = log_2[min(ssize, nsets)]$, where $nsets$ is the number of sets in TLB. A random set index will be generated within the region $[sbase[S_n - 1, 0], sbase[S_n - 1, 0] + min(ssize, nsets)]$ for random fill.

```
1. void _gcry_mpi_powm  (gcry_mpi_y_  res,
          gcry_mpi_t  base,  gcry_mpi_t  expom gcry_mpi_t_mod)
2. {
3.     mpi_ptr_t  rp,  xp;  /* pointers  to MPI data */
4.     mpi_ptr_t  tp;
5.     ...
6.     for(;;)  {
7.         /* For every  exponent  bit  in  expo*/
8.         _gcry_mpih_sqr_n_basecase(xp,  rp);
9.         if(secret_exponent  || e_bit_is1)  {
10.            /* unconditional  multiply  if exponent  is
11.             * secret to mitigate  FLUSH+RELOAD
12.             */
13.           _gcry_mpih_mul(xp,  rp);
14.        }
15.        if(e_bit_is1)  {
16.             /*e bit  is 1,  use  the result*/
17.             tp = rp;  rp = xp;  xp = tp;
18.             rsize  = xsize;
19.        }
20.    }
21. }
```

Figure 5: Code sample for one of the variants of modular exponentiation from *libgcrypt* version 1.8.2. Pointers rp, xp and tp are defined (blue dashed square). rp and xp are used for both *e_bit* is 1 or 0 (green dashed square). tp will only be accessed when *e_bit* is 1 (red square).

the original page address is finally requested, and "no fill" *fill_type* will be sent to the TLB to obtain the translation. Finally, (7) this address will be stored in the *buffer*, without modifying TLB entries, and be sent back to the processor.

4.2.3 RF TLB vs. RF Cache. As a possible alternative, the "random fill" part can be done asynchronously during the idle cycles, as has been proposed for secure caches, e.g., [18]. However, using this way, programs which are TLB intensive for accesses to the secure region will starve "random fill" and result in no random entries being put into the TLB, which will negatively impact the security offered by the TLB.

The proposed RF TLB differentiates victim and attacker, secure and non-secure region and has a different random fill scheme for pages within or outside of the secure region for both attacker and victim. That helps RF TLB prevent all types of TLB timing-based vulnerabilities. Meanwhile, RF Cache [18] only differentiates victim and attacker and cannot prevent all types of cache timing-based vulnerabilities [11].

5 SECURITY EVALUATION

5.1 Micro Security Benchmarks

Most of the types of the attacks derived with the three-step model do not correspond to already known attacks. Some exceptions include, for example, the TLBleed attack [8], which was demonstrated using the *libgcrypt*'s RSA cryptographic implementation. In the RSA implementation, whether the pointer *tp* is accessed depends on the secret *e_bit* in *_gcry_mpi_powm* function (line 17, Figure 5). In TLBleed, the attacker can use the TLB Prime + Probe attack strategy to deploy an attack which allows them to learn whether *tp* is accessed, to know the secret bit. However, such examples for most of the other attacks do not exist.

Thus, we developed micro security benchmarks which can be used to test TLBs to check if they are vulnerable to each of the

```
                 1.  #include  "riscv_test.h"
                 2.  #include  "test_macro.h"
                 3.  RVTEST_RV64U             # Define  TVM  used by program.
                 4.  # Test  code region.
                 5.  RVTEST_CODE_BEGIN        # Start  of  test  code.
                 6.
secure         { 7.  csrw sbase,  3           # Set page base of secure  region
region         { 8.  csrw ssize,  3           # Set page size of secure  region
info           { 9.  ...
               { 10. # Attacker  primes  the  whole  TLB/specific  set
               { 11. csrw process_id,  0      # Set current  process  for simulation
A_d            { 12.                          # 0 is attacker;  1 is victim
               { 13. la x1,  tdat2048
               { 14. ldnorm  x2,  0(x1)
               { 15. ...
               { 16. # Victim  does  serect  data  access/secure  address  translation
               { 17. csrw process_id,  1
V_u            { 18. la x1,  tdat1024
               { 19. ldrand  x2,  0(x1)
               { 20. ...
                 21. csrr  x3,  tlb_miss_count  # Read  TLB  miss  counter
               { 22. # Attacker  probe  the  TLB  set
A_d            { 23. csrw process_id,  0
               { 24. la x1,  tdat2048
               { 25. ldnorm  x2,  0(x1)
               { 26. ...
check          { 27. csrr  x4,  tlb_miss_count  # Read  TLB  miss  counter  again
TLB            { 28. beq  x3,  x4,  no_tlb_miss  # Compare  and  see  if  there  is  TLB
miss           { 29.                          #  miss  (slow  access)
/hit             30. ...
                 31. RVTEST_PASS              # Signal  success.
                 32.no_tlb_miss:
                 33. RVTEST_FAIL              # Output  info  for  no  TLB  miss
                 34. RVTEST_CODE_END          # End  of  test  code.
                 35.
                 36. # Data  section.
                 37. RVTEST_DATA_BEGIN        # Start  of  test  data  region.
                 38.      TEST_DATA
data           { 39. tdat00:                  # A  big  array  is  initialized
array          { 40. tdat0:     .dword  0
for            { 41. ...
testing        { 42. tdat16489: .dword  16489
               { 43. RVTEST_DATA_END          # End  of  test  data  region.
```

Figure 6: Code sample for TLB Prime + Probe micro security benchmark $A_d \rightsquigarrow V_u \rightsquigarrow A_d$ (slow) variant, used in simulation testing of Rocket Core-based RISC-V.

attack types. To generate the micro security benchmarks, we leverage a Python script that follows a three-step template to generate assembly code of all the types of vulnerabilities showed in Table 2.

Figure 6 is a micro security benchmark example of the $A_d \rightsquigarrow V_u \rightsquigarrow A_d$ (slow) variant of TLB Prime + Probe vulnerability. Inside the benchmark, first there is standard prologue with include statements (line 1-5), then the secure region (*sbase*, *ssize*) is set (line 7-8). For the specific vulnerability, the three steps are executed in the order of A_d (line 10-15), V_u (line 16-20) and A_d (line 22-26). Out-of-secure-address-region d will be accessed using the *norm* type of memory access while inside-secure-address-region u will use *rand* type of memory access, corresponding to the non-secure and secure page address accesses illustrated in Section 4.2.2, respectively. Attacker measures the final step's timing (line 21, 27-29). The same script can be used to generate assembly tests for all SA TLB, SP TLB, and RF TLB.

5.2 Channel Capacity

An attacker gains knowledge about the secret address translation through TLB timing channel by observing the timing of address translation in a TLB block. The observed timing may depend on the victim's prior behavior.

There are two possible victim's behaviors B: whether the victim's secret-dependent memory access results in address translation, V_u, which maps to the TLB block tested by the attacker or not. There are also two possible attacker's observations O: whether attacker

Table 3: Probabilities of different victim behaviors B and attacker observations O.

		Attacker's observation O	
		Miss	Hit
Victim's behavior B	Memory access (or invalidation) maps to the same address/index attacker tests	p_1	$1 - p_1$
	Memory access (or invalidation) does not map to the same address/index attacker tests	p_2	$1 - p_2$

observes slow access due to a TLB miss or fast access due to a TLB hit.

To evaluate the relation between the victim's behaviors and the attacker's observations, we define p_1 and p_2 as listed next, and shown in Table 3: When the victim behavior triggers a translation of an address that maps to the TLB block the attacker tests, we use p_1 to denote the probability the attacker observes a TLB miss, and $1 - p_1$ as the probability the attacker observes a TLB hit. When the victim's behavior triggers a translation of an address that does not map to the TLB block the attacker tests, we use p_2 to denote the probability the attacker observing a TLB miss, and $1 - p_2$ for observing a hit.

To provide the optimal scenario for attacker, we assume the probability of victim's access V_u mapping to the TLB block tested by the attacker to be the same as the probability of V_u not mapping to the block, i.e. both are $\frac{1}{2}$.

We use channel capacity [7] to quantify the amount of information about the secret address translation that the attack gains from a specific timing-based attack as follows:

$$\begin{aligned} C \equiv I(B;O) &\equiv \sum_{b,o} p(b,o) log \frac{p(b,o)}{p(b)p(o)} \\ &\equiv \frac{p_1}{2} log \frac{2p_1}{p_1 + p_2} + \frac{p_2}{2} log \frac{2p_2}{p_1 + p_2} \\ &+ \frac{1-p_1}{2} log \frac{2(1-p_1)}{2 - p_1 - p_2} + \frac{1-p_2}{2} log \frac{2(1-p_2)}{2 - p_1 - p_2} \end{aligned} \tag{1}$$

where $I(B;O)$ denotes the mutual information between victim's behavior B and attacker's observation O. The $p(b)$ and $p(o)$ are the marginal probability distribution functions of victim's behavior B and attacker's observation O. The $p(b,o)$ is the joint probability function of victim's behavior B and attacker's observation O.

The p_1 and p_2 will have different values for each type of vulnerability, and also depend on the type of the TLB. Especially, if a TLB is able to defend against a specific type of an attack, the mutual information C should be zero for that attack type. Otherwise, the attacker can gain some knowledge about the victim's behavior. Below we analyze the C for different TLB types, and compare with theoretical calculations.

5.3 Theoretical Result and Security Evaluation of the TLBs

We implemented SP TLB and RF TLB, as illustrated in Section 4, as well as a SA TLB, in Chisel code and integrated them into the Rocket Core-based RISC-V processor[7]. In addition to implementing the TLBs, new TLB miss performance counters were implemented and are used by the simulation to determine *slow* or *fast* TLB accesses based on whether miss occurs or not, respectively. The Chisel code for the whole processor with the new TLBs was used to generate cycle-accurate simulations.

The simulated hardware was used to execute the micro security benchmarks previously discussed in Section 5.1. For each benchmark, it was run 500 times each for "mapped" or "not mapped" (shown in Table 3) victim address for the tested TLB block, therefore in total 1000 times. Multiple runs are needed as the RF TLB leverages randomization and we need to average results over many runs. For each TLB type, each of the benchmarks was run, thus 24 vulnerability types × 1,000 simulations = 24,000 runs.

The security evaluation focused on 8-way 32-entry SA TLB as the example. With this setup, the system software will take 4 out of 32 entries and distribute the 4 entries in different sets, so 28 different user pages are sufficient to prime the TLB. We assume two cases for the victim: one has 6 contiguous pages (3 pages out of the 6 are secure), another has 31 contiguous secure pages (to simulate contention between secure address translations).

5.3.1 *Security of SA TLB, SP TLB and RF TLB.*

The theoretical and the simulation results of all the TLBs are listed and compared in Table 4.

SA TLB. SA TLB has simulated and theoretical $C = 0$ for TLB Flush + Reload, TLB Evict + Probe, and TLB Prime + Time attacks, thus it defends these attacks. SA TLB is not able to prevent internal TLB Collision ($p_1 = 0, p_2 = 1, C = 1$) and TLB Evict+Time, TLB Prime+Probe and TLB Bernstein's Attack ($p_1 = 1, p_2 = 0, C = 1$).

SP TLB. For SP TLB, all the vulnerabilities that SA TLB can prevent are also prevented by SP TLB. Further, TLB Evict + Time and TLB Prime + Probe vulnerability can be prevented by SP TLB. For these two types of vulnerabilities, $p_1 = p_2 = 0, C = 0$.

On the other hand, SP TLB is still vulnerable to TLB version of Bernstein's Attack ($p_1 = 1, p_2 = 0, C = 1$) and TLB Internal Collision vulnerability ($p_1 = 0, p_2 = 1, C = 1$) since victim's own address contention and TLB hit due to its own accesses cannot be defended by partitioning.

RF TLB. The RF TLB can defend all the vulnerabilities that SA TLB can defend. There are then 14 vulnerabilities left to consider, which can be further reduced to simplify the analysis: V_a and A_a belong to a, similarly, $V_{a^{alias}}$ and $A_{a^{alias}}$ are a^{alias}, V_d and A_d are d. Following this way, we can simplify the 14 patterns into 6 patterns for RF TLB, which are listed below. The *sec_range* stands for secure region, its value is 3 in the first 3 cases and 31, to simulate contention between secure address translations, in the last 3 cases. The *nset* and *nway* stands for the number of sets and ways, whose value is 4 and 8 in the simulation tests, respectively. *prime_num* stands for the virtual page address number that can prime the whole

[7]Chisel commit ID: 980778b, Rocket Chip commit ID: aca2f0c

Table 4: Comparison of SA TLB, SP TLB and RF TLB simulation and theoretical results. $p_1{}^*$ and $p_2{}^*$ represent probabilities based on simulation. p_1 and p_2 are theoretical calculations. C^* and C represent mutual information based on simulation and theoretical calculation, respectively. $n_{M,M}$ and $n_{N,M}$ denote number of misses when the victim's secret address and test address map and do not map to each other, respectively. Bold C^* and C are the ones with value 0 or about 0, indicating that this TLB is able to prevent the corresponding vulnerability. Small numbers are rounded up.

		SA TLB								SP TLB								RF TLB							
Attack Category	Vulnerability Type	$n_{M,M}$	p_1*	p_1	$n_{N,M}$	p_2*	p_2	C^*	C	$n_{M,M}$	p_1*	p_1	$n_{N,M}$	p_2*	p_2	C^*	C	$n_{M,M}$	p_1*	p_1	$n_{N,M}$	p_2*	p_2	C^*	C
TLB Internal Collision	$A_d \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	0	0	0	500	1	1	1	1	2	0.01	0	500	1	1	0.98	1	343	0.69	0.67	333	0.67	0.67	**0.01**	**0**
	$V_d \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	0	0	0	500	1	1	1	1	0	0	0	500	1	1	1	1	328	0.66	0.67	338	0.68	0.67	**0.01**	**0**
	$A_{a^{alias}} \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	10	0.02	0	500	1	1	0.93	1	3	0.01	0	500	1	1	0.97	1	485	0.97	0.97	483	0.96	0.97	**0.01**	**0**
	$V_{a^{alias}} \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	9	0.02	0	500	1	1	0.94	1	2	0.01	0	500	1	1	0.98	1	489	0.98	0.97	486	0.97	0.97	**0.01**	**0**
	$A_{inv} \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	0	0	0	500	1	1	1	1	0	0	0	500	1	1	1	1	322	0.65	0.67	353	0.71	0.67	**0.01**	**0**
	$V_{inv} \rightsquigarrow V_u \rightsquigarrow V_a$ (fast)	1	0.01	0	500	1	1	0.99	1	0	0	0	500	1	1	1	1	328	0.66	0.67	349	0.70	0.67	**0.01**	**0**
TLB Flush + Reload	$A_d \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**
	$V_d \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**
	$A_{a^{alias}} \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**
	$V_{a^{alias}} \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**
	$A_{inv} \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**
	$V_{inv} \rightsquigarrow V_u \rightsquigarrow A_a$ (fast)	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**
TLB Evict +Time	$V_u \rightsquigarrow A_d \rightsquigarrow V_u$ (slow)	500	1	1	0	0	0	1	1	0	0	0	26	0.05	0	**0.03**	**0**	0	0	0.01	0	0	0.01	**0**	**0**
	$V_u \rightsquigarrow A_a \rightsquigarrow V_u$ (slow)	500	1	1	0	0	0	1	1	0	0	0	0	0	0	**0**	**0**	2	0.01	0.01	7	0.01	0.01	**0.01**	**0**
TLB Prime +Probe	$A_d \rightsquigarrow V_u \rightsquigarrow A_d$ (slow)	500	1	1	1	0.01	0	0.99	1	0	0	0	20	0.04	0	**0.02**	**0**	167	0.33	0.33	158	0.32	0.33	**0.01**	**0**
	$A_a \rightsquigarrow V_u \rightsquigarrow A_a$ (slow)	500	1	1	1	0.01	0	0.99	1	0	0	0	2	0.01	0	**0.02**	**0**	135	0.27	0.26	148	0.30	0.26	**0.01**	**0**
TLB Bernstein's Attack	$V_u \rightsquigarrow V_a \rightsquigarrow V_u$ (slow)	500	1	1	1	0.01	0	0.99	1	500	1	1	1	0.01	0	0.99	1	0	0	0.01	10	0.02	0.01	**0.01**	**0**
	$V_u \rightsquigarrow V_d \rightsquigarrow V_u$ (slow)	500	1	1	0	0	0	1	1	500	1	1	31	0.06	0	0.83	1	0	0	0.01	0	0	0.01	**0**	**0**
	$V_d \rightsquigarrow V_u \rightsquigarrow V_d$ (slow)	500	1	1	0	0	0	1	1	500	1	1	0	0	0	1	1	160	0.32	0.33	163	0.33	0.33	**0**	**0**
	$V_a \rightsquigarrow V_u \rightsquigarrow V_a$ (slow)	500	1	1	0	0	0	1	1	500	1	1	0	0	0	1	1	35	0.07	0.09	22	0.04	0.09	**0.01**	**0**
TLB Evict +Probe	$V_d \rightsquigarrow V_u \rightsquigarrow A_d$ (slow)	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**
	$V_a \rightsquigarrow V_u \rightsquigarrow A_a$ (slow)	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**
TLB Prime +Time	$A_d \rightsquigarrow V_u \rightsquigarrow V_d$ (slow)	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**
	$A_a \rightsquigarrow V_u \rightsquigarrow V_a$ (slow)	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**	500	1	1	500	1	1	**0**	**0**

4-way 32-entry TLB in RISC-V. The theoretical probabilities p_1 and p_2 for the 6 combined patterns are:

- $V_u \rightsquigarrow d \rightsquigarrow V_u$ (slow): p_1=p_2=$\frac{1}{sec\_range}\times\frac{1}{min(nset,sec\_range)\times nway}=\frac{1}{3}\times\frac{1}{3\times 8}$=0.01.
- $d/inv \rightsquigarrow V_u \rightsquigarrow a$ (fast): p_1=p_2=1-$\frac{1}{sec\_range}$=$1-\frac{1}{3}$=0.67.
- $d \rightsquigarrow V_u \rightsquigarrow d$ (slow): p_1=p_2=$\frac{1}{sec\_range}=\frac{1}{3}$=0.33.
- $V_u \rightsquigarrow a \rightsquigarrow V_u$ (slow): p_1=p_2=$(\frac{nway}{sec\_range})^{nway}=(\frac{8}{31})^8$=0.01.
- $a^{alias} \rightsquigarrow V_u \rightsquigarrow a$ (fast): p_1=p_2=1-$\frac{1}{sec\_range}$ $= 1-\frac{1}{31}$=0.97.
- $a \rightsquigarrow V_u \rightsquigarrow a$ (slow): Because V_u cannot get hit due to A_a, there are two cases:
  - $A_a \rightsquigarrow V_u \rightsquigarrow A_a$: p_1=p_2=$\frac{nway}{sec\_range}=\frac{8}{31}$=0.26.
  - $V_a \rightsquigarrow V_u \rightsquigarrow V_a$: p_1=p_2=$\frac{sec\_range-prime\_num}{sec\_range}=\frac{31-28}{31}$=0.09.

All the mutual information derived derived based on the above probabilities for the RF TLB is 0 for the theoretical calculations and about 0 for the simulation results as shown in Table 4, indicating RF TLB is secure against these attacks.

5.3.2 Comparison of the Different TLBs. As can be seen from Table 4, the simulation results match the theoretical values closely, indicating the actual hardware Chisel implementation of TLBs matches the theoretical calculations we presented.

For the TLBs, normal SA TLB can prevent 10 types of timing-based vulnerabilities due to requirement of deriving TLB hit for both address and process ID. For the SP TLB, it is able to prevent external interference by partitioning but is weak at preventing internal interference. Therefore, SP TLB is able to prevent 4 more types of vulnerabilities. However, internal hit-based vulnerabilities, such as $V_{inv} \rightsquigarrow V_u \rightsquigarrow V_a$ (fast), can still happen in SP TLB. For RF TLB, random fill technique is able to de-correlate the TLB fill with the memory access. This is able to prevent all classes of timing-based vulnerabilities listed in Table 2.

6 PERFORMANCE EVALUATION

6.1 Hardware FPGA Setup

The SP TLB and RF TLB were implemented in Chisel hardware construction language and realized in the Rocket Core-based RISC-V processor. SP TLB related logic is about 300 lines of Chisel code, while RF TLB related logic is about 500 lines of Chisel code. The same hardware code was used for simulation (Section 5) and the performance evaluation (this Section), with minor changes for the FPGA version. Further, to allow for performance evaluation under realistic settings and with use of Linux, the RISC-V with the secure TLBs was synthesized on the Xilinx ZC706 and ZedBoard Evaluation FPGA boards.

6.2 Performance Evaluation Setup

To enable performance measurements, a TLB miss counter was added, and cycle counter and instruction counters were enabled in user mode. The counters are used to collect data during execution of the cryptographic program and benchmarks. The collected data were: instructions per cycle (IPC) and TLB misses per kilo instructions (MPKI).

Two TLB sizes were selected for evaluation. 32-entry, 4-way SA TLB corresponds to TLBs used in Intel's Haswell processors [22], while the 128 entry corresponds to TLBs used in Intel's Nehalem microarchitecture [15].

The following L1 D-TLB configurations were tested. FA TLB with 32 entries (labeled *FA 32* in figures), SA TLB with 32 entries, 2-way (labeled *2W 32* in figures), SA TLB with 32 entries, 4-way (labeled *4W 32* in figures), and the same configurations, but for 128 entries (labeled *FA 128*, *2W 128*, and *2W 128*, respectively in figures). All of these were used for baseline Standard TLB, SP TLB and RF TLB. In addition, a naive solution to prevent all TLB attacks is to disable the TLB. While in our RISC-V setup it is not possible to fully disable the TLB, we include TLB with 1 entry (labeled *1E* in figures) as closest possible configuration to show its impact on performance. In total, 19 TLB configurations were tested on our FPGA setup.

The SP TLB was tested with half the ways to be set victim partition. The RF TLB was tested where the secure region was set by the software, see *SecRSA* discussion below.

For performance evaluation, we use the RSA implementation from TLBleed attack [8][8]. Further, we selected TLB-intensive SPEC 2006 integer and floating point benchmarks to evaluate the overheads introduced by the secure TLB designs. The four selected benchmarks are: 453.povray, 471.omnetpp, 483.xalancbmk, and 436.cactusADM[9]. The different configurations are listed below.

RSA. The *libgcrypt*'s RSA decryption routine was run 50, 100 and 150 times in series to simulate multiple uses of secret cryptographic key that the attacker may want to learn via the timing channels. Each time the same hard-coded key was used. No security is enabled for this configuration.

SecRSA. This is same as RSA configuration, except for SP and RF TLBs the security features are enabled to protect the RSA. For SP TLB, the SecRSA's process ID is set as the "victim", and all the address translations will be put in the victim partition in the SP TLB, while other processes' address translation will be in the attacker partition. For the RF TLB, SecRSA's *.data* section pages including the ones referenced by the *tp*, *rp* and *xp* pointers (the number of these pages is 3, and the pointers are previously discussed in Section 5.1) are protected and accesses are randomized (see Section 4.2).

RSA with povray, **omnetpp**, **xalancbmk** and **cactusADM**. In order to better see the performance impact on the whole system when a secure program is running, the RSA as discussed above, was run in parallel with each of the selected TLB-intensive SPEC benchmarks. The RSA continuously performs the decryption (50, 100 and 150 times), while the SPEC benchmark runs in background.

SecRSA with povray, **omnetpp**, **xalancbmk** and **cactusADM**. Same as above, but security is enabled for RSA, as discussed in SecRSA case.

6.3 Standard TLB Performance

Standard TLB's IPCs and MPKIs are shown in Figure 7a and Figure 7d. Larger TLB has smaller MPKI and better IPC. RSA routine is relatively small, so it experiences very few MPKIs. When SPEC benchmarks are included, the MPKIs increase and IPC drops. Interestingly, although cactusADM was specified as TLB-intensive in [21], it is not affected much by TLB size. Additionally, in most of the cases, IPC and MPKI give similar result for 50, 100 and 150 runs. This is the same for SP TLB and RF TLB.

Note, the *1E* configuration approximates no TLB scenario. This has on average 38.3% worst performance, based on IPC. Thus, achieving security by disabling the TLB will impact system performance significantly.

Further, FA TLB (i.e. *FA 32* and *FA 128*) have as expected better performance than SA configurations, and can prevent 8 more types of attacks compared with 10 types that SA TLB can prevent. However, FA TLBs have area impact of about 0.6% more Slice LUTs for 32 entries, and 3.3% more Slice LUT for 128 entries, see Section 6.6. The FPGA runs slow enough at 50MHz for ZC706 and at 25MHz for ZedBoard that the impact of FA configuration on the critical path is not observed.

[8]RSA from Libgcrypt 1.8.2: https://gnupg.org/ftp/gcrypt/libgcrypt/

[9]The "ref" or "train" inputs to SPEC benchmarks were used, the "train" inputs were used if the benchmark was not able to run with "ref" inputs on the FPGA setup due to memory size limitation.

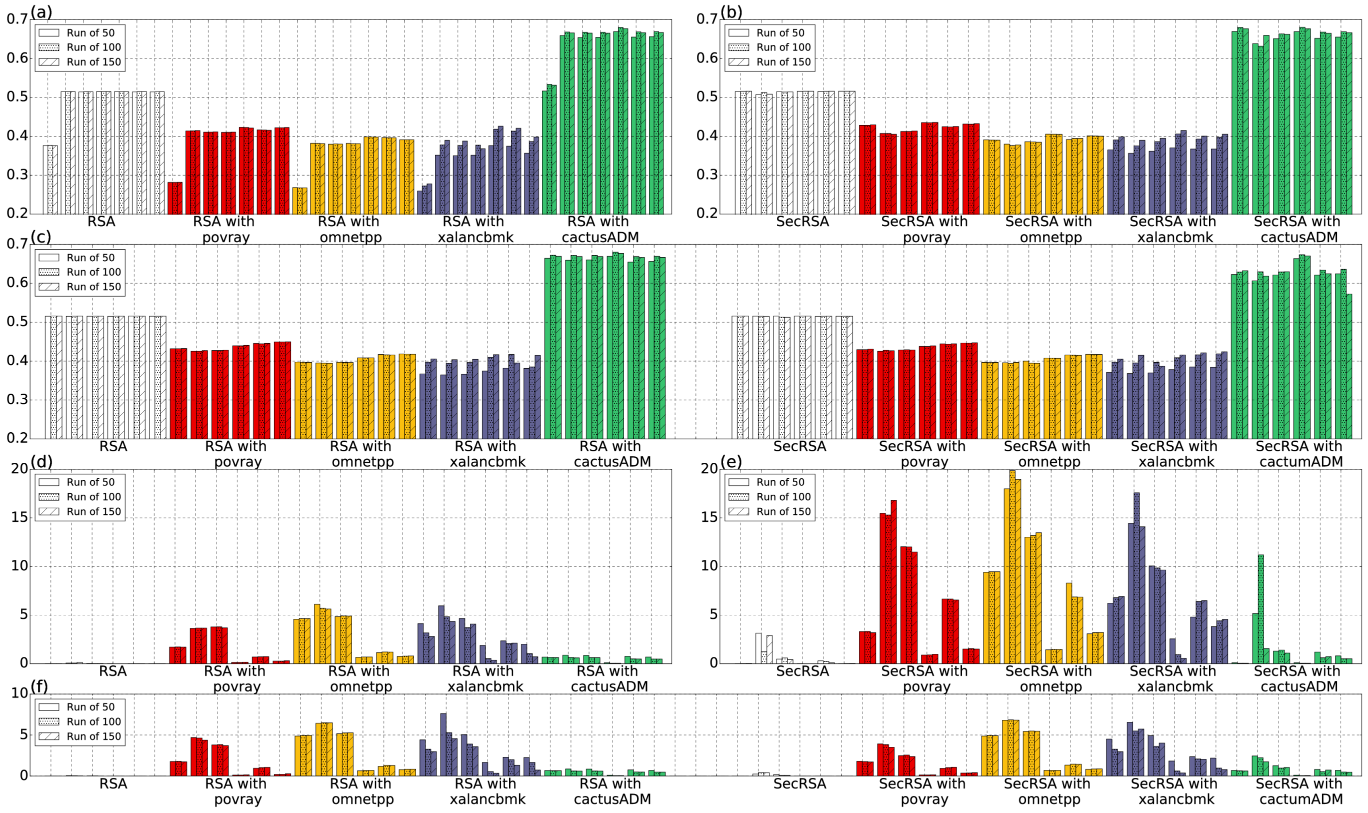

Figure 7: Evaluation of different configuration of TLBs. (a)-(c) IPC of SA TLB, SP TLB and RF TLB, respectively. (d)-(f) MPKI of SA TLB, SP TLB and RF TLB, respectively. Every set of bars in the graph follows the order: (1E only for IPC of SA TLB), FA 32, 2W 32, 4W 32, FA 128, 2W 128 and 4W 128.

6.4 SP TLB Performance

Performance evaluation results for the SP TLB are shown in Figure 7b and Figure 7e. For the SP TLB, half the ways are set to the victim partition. When victim program RSA (SecRSA) is run alone or run with a SPEC benchmark, the secure data of RSA is allocated to half of the ways in the victim partition, and all other data and all SPEC data is in the attacker partition. Overall, IPC is about 0.5% better compared to standard TLB. This may be due to the system code getting invoked more often for the SP TLB, than for the standard TLB, and the system code may have better overall IPC. From the evaluation result, SP additions for the TLB do not influence IPC too much.

The MPKI is significantly higher than standard TLB (207.5% more or 3.07x), as again the effective TLB size is one half. Assignment of different number of ways for victim and attacker partitions, and its impact on performance could be further explored. Further, ideas of coalescing in TLBs [21] could be explored to improve the effective TLB size for victim and attacker partitions.

6.5 RF TLB Performance

Performance evaluation results for the RF TLB are shown in Figure 7c and Figure 7f. The IPC is about 1.4% higher compared to SA TLB. Again, RF TLB may involve even more system code, in which case a better IPC is derived. Meanwhile, comparing the corresponding configurations, the MPKI of RF TLB is about 64.5% better than SP TLB, while 9.0% worse than SA TLB. Thus, RF TLB provides both better performance and better security than SP TLB, while being as good as standard TLB in performance. It has about 39.4% better IPC than disabling TLB (approximated by the *1E* configuration) while providing the same security.

6.6 Area Overhead

We further evaluate the area overhead of the new secure additions. We use the number of Slice Look-Up Table (LUT), Slice Registers, Block RAMs and DSPs from the FPGA synthesis reports for the Xilinx ZC706 FPGA as a proxy for the area. For all the configurations, the Block RAM usage is 24 and the DSP usage is 15. Slice LUT and Registers numbers are shown in Table 5. The baseline is again 32-entry, 4-way SA L1 D-TLB.

Comparing to 4-way 32-set SA TLB, 4-way 32-set SP TLB has 0.4% more Slice LUTs and 0.1% more Slice Registers. 4-way 32-set RF TLB has 6.2% more Slice LUTs and 5.5% more Slice Registers. On average for all the configurations of TLBs, SP TLB has about 0.2% less Slice LUTs and 1.3% less Slice Registers compared with SA TLB, while RF TLB has about 6.5% more Slice LUTs and 7.9% more Slice Registers compared with SA TLB.

Table 5: Area overhead of the new secure additions. Δ Slice LUT and Δ Slice Registers columns show the the difference from the 32-entry, 4-way SA TLB baseline.

	Configu-rations	Slice LUTs	Δ Slice LUTs	Slice Registers	Δ Slice Registers
	1E	35266	-777	18359	-4406
SA TLB	FA 32	36395	352	22199	-566
	2W 32	36298	255	23513	748
	4W 32	**36043**	–	**22765**	–
	FA 128	40177	4134	33815	11050
	2W 128	39684	3641	38630	15865
	4W 128	38107	2064	35694	12929
SP TLB	FA 32	36499	456	22251	-514
	2W 32	36387	344	23523	758
	4W 32	**36183**	**140**	**22798**	**33**
	FA 128	40568	4525	33824	11059
	2W 128	38609	2566	38521	15756
	4W 128	38049	2006	35659	12894
RF TLB	FA 32	38281	2238	22697	-68
	2W 32	38510	2467	25643	2878
	4W 32	**38266**	**2223**	**24018**	**1253**
	FA 128	42740	6697	34252	11487
	2W 128	42509	6466	45823	23058
	4W 128	41259	5216	39538	16773

7 CONCLUSION

This paper proposed a novel three-step modeling approach that exhaustively enumerates all possible TLB timing-based vulnerabilities. It showed how to automatically generate micro security benchmarks that test for the TLB vulnerabilities. It gave details of two new hardware secure TLB designs: a Static-Partition (SP) TLB and a Random-Fill (RF) TLB. The simulations confirmed the theoretical channel capacity calculations and full system performance on FPGA showed that the new secure TLBs are as good as regular TLBs, while protecting against the various attacks. The proposed secure TLBs can defend not only against the previously publicized attacks, but also other possible timing-based attacks in TLBs found using our new three-step modeling approach.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported by NSF grants 1651945 and 1813797, and through SRC award number 2844.001.

REFERENCES

[1] Onur Acıiçmez and Çetin Kaya Koç. 2006. Trace-Driven Cache Attacks on AES (Short Paper). In *International Conference on Information and Communications Security*. Springer, 112–121.

[2] Daniel J Bernstein. 2005. Cache-Timing Attacks on AES. (2005).

[3] Joseph Bonneau and Ilya Mironov. 2006. Cache-Collision Timing Attacks against AES. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 201–215.

[4] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security Symposium*. 857–874.

[5] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. 2018. Cache Timing Side-Channel Vulnerability Checking With Computation Tree Logic. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2.

[6] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2012. Non-Monopolizable Caches: Low-Complexity Mitigation of Cache Side Channel Attacks. *ACM Transactions on Architecture and Code Optimization (TACO)* 8, 4 (2012), 35.

[7] Andrea J Goldsmith and Pravin P Varaiya. 1997. Capacity of Fading Channels with Channel Side Information. *IEEE Transactions on Information Theory* 43, 6 (1997), 1986–1992.

[8] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium*. USENIX, 955–972.

[9] Part Guide. 2011. Intel® 64 and Ia-32 Architectures Software Developer's Manual. *Volume 3B: System programming Guide, Part* 2 (2011).

[10] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache Games–Bringing Access-Based Cache Attacks on AES to Practice. In *IEEE Symposium on Security and Privacy*. IEEE, 490–505.

[11] Zecheng He and Ruby B Lee. 2017. How Secure is Your Cache against Side-Channel Attacks?. In *International Symposium on Microarchitecture (MICRO)*. ACM, 341–353.

[12] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical Timing Side Channel Attacks Against Kernel Space ASLR. In *IEEE Symposium on Security and Privacy*. IEEE, 191–205.

[13] Intel Intel. 64. IA-32 Architectures Software Developer's Manual. *Volume 3A: System Programming Guide, Part* 1, 64 (64), 64.

[14] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01203.

[15] Nasser Kurd, Jonathan Douglas, Praveen Mosalikanti, and Rajesh Kumar. 2008. Next Generation Intel® Micro-Architecture (Nehalem) Clocking Architecture. In *Symposium on VLSI Circuits*. IEEE, 62–63.

[16] Ruby B Lee, Peter Kwan, John P McGregor, Jeffrey Dwoskin, and Zhenghong Wang. 2005. Architecture for Protecting Critical Secrets in Microprocessors. In *ACM SIGARCH Computer Architecture News*, Vol. 33. IEEE Computer Society, 2–13.

[17] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01207

[18] Fangfei Liu and Ruby B Lee. 2014. Random Fill Cache Architecture. In *International Symposium on Microarchitecture (MICRO)*. IEEE, 203–215.

[19] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: the Case of AES. In *Cryptographers' Track at the RSA Conference*. Springer, 1–20.

[20] Colin Percival. 2005. Cache Missing for Fun and Profit.

[21] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. Colt: Coalesced Large-Reach TLBs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 258–269.

[22] Efraim Rotem and Senior Principal Engineer. 2015. Intel Architecture, Code Name Skylake Deep Dive: A New Architecture to Manage Power Performance and Energy Efficiency. In *Intel Developer Forum*.

[23] Jakub Szefer. 2018. Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses. *Journal of Hardware and Systems Security* (2018), 1–16.

[24] Stephan Van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think. In *USENIX Security Symposium*. 937–954.

[25] AMD Virtualization. 2008. AMD-v Nested Paging. *White paper* (2008).

[26] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. 2017. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *Conference on Computer and Communications Security*. ACM, 2421–2434.

[27] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C Myers, and G Edward Suh. 2016. SecDCP: Secure Dynamic Cache Partitioning for Efficient Timing Channel Protection. In *Design Automation Conference*. ACM, 74.

[28] Zhenghong Wang and Ruby B Lee. 2007. New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks. In *ACM SIGARCH Computer Architecture News*, Vol. 35. ACM, 494–505.

[29] Zhenghong Wang and Ruby B Lee. 2008. A Novel Cache Architecture With Enhanced Performance and Security. In *International Symposium on Microarchitecture (MICRO)*. IEEE, 83–93.

[30] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W Fletcher, and Josep Torrellas. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *International Symposium on Microarchitecture (MICRO)*. IEEE, 428–441.

[31] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. 2017. Secure Hierarchy-Aware Cache Replacement Policy (SHARP): Defending Against Cache-Based Side Channel Attacks. In *International Symposium on Computer*

Architecture. ACM, 347–360.

[32] Danfeng Zhang, Aslan Askarov, and Andrew C Myers. 2012. Language-Based Control and Mitigation of Timing Channels. *ACM SIGPLAN Notices* 47, 6 (2012), 99–110.

[33] Danfeng Zhang, Yao Wang, G Edward Suh, and Andrew C Myers. 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 503–516.

APPENDIX A: SOUNDNESS ANALYSIS OF TLB THREE-STEP MODEL

In this section we analyze the soundness of the three-step model to demonstrate that the three-step model can cover all possible SA TLB timing-based vulnerabilities. If there is a vulnerability, it can always be reduced to a model that requires only three steps.

Let β denote the number of memory page related operations in a vulnerability.

When $\beta = 1$, there is only one memory page related operation, and it is not possible to create interference between memory page related operations since two memory page related operations are the minimum requirement for an interference. Furthermore, $\beta = 1$ corresponds to the three-step pattern with both *Step* 1 and *Step* 2 to be $\star$ since the TLB state $\star$ gives no information. These types are not listed in Table 2, which shows all the effective vulnerabilities. Therefore, attack cannot happen when $\beta = 1$.

When $\beta = 2$, it satisfies the minimum requirement of an interference for memory page related operations and corresponds to the three-step cases where *Step* 1 is $\star$. Three-step cases where *Step* 1 is $\star$ does not have corresponding effective vulnerabilities shown in Table 2. So $\beta \neq 2$.

When $\beta = 3$, we exhaustively list all possible three-step memory page related operations in Section 3.3 and we conclude that there are in total 24 types of effective vulnerabilities, of which 16 are new compared to what is known in literature. So $\beta = 3$.

When $\beta > 3$, the pattern of memory related operations for a vulnerability can be deducted using the following rules:

- *Rule 1*: If there is a sub-pattern such as $\{ \ldots \rightsquigarrow \star \rightsquigarrow \ldots \}$, the longer pattern can be divided into two separate patterns, where $\star$ is assigned as *Step* 1 of the second pattern. This is because $\star$ gives no timing information, and the attacker loses track of the cache state after $\star$. This rule should be recursively checked until there are no sub-patterns with a $\star$ in the middle or last step ($\star$ in the last step will be deleted).
- *Rule 2*: If there is a sub-pattern such as $\{ \ldots \rightsquigarrow A_{inv}/V_{inv} \rightsquigarrow \ldots \}$, the longer pattern can be divided into two separate patterns, where A_{inv}/V_{inv} is assigned as *Step* 1 of the second pattern. This is because A_{inv}/V_{inv} will flush all the timing information of the current block and it can be used as the flushing step for *Step* 1, e.g., vulnerability $\{ A_{inv} \rightsquigarrow V_u \rightsquigarrow A_a(fast) \}$ shown in Table 2. It cannot be candidate for middle steps or the last step because it will flush all timing information, making the attacker unable to correspond the final timing with victim's sensitive address translation information. This rule should be recursively checked until there are no sub-patterns with a A_{inv}/V_{inv} in the middle or last step (A_{inv}/V_{inv} in the last step will be deleted).
- *Rule 3*: If the remaining memory page related operations have a sub-pattern that has two adjacent steps both related to known addresses or both related to unknown address (including repeating states), the two adjacent steps can be reduced to only one.
  - For two unknown adjacent memory page related operations (containing u, denoted as *u_operation*), although u is unknown, both of the accesses target on the same u so can be reduced. E.g., $\{ \ldots \rightsquigarrow V_u \rightsquigarrow V_u \rightsquigarrow \ldots \}$ can be reduced to $\{ \ldots \rightsquigarrow V_u \rightsquigarrow \ldots \}$.
  - For two known adjacent memory related operations (denoted as *not_u_operation*), these two operations result in a deterministic state of the cache block, so these two steps can be reduced to only one step. E.g., $\{ \ldots \rightsquigarrow A_d \rightsquigarrow V_a \rightsquigarrow \ldots \}$ can be reduced to $\{ \ldots \rightsquigarrow V_a \rightsquigarrow \ldots \}$.

  The *Rule 3* should be recursively checked until there are no two adjacent steps both related to known addresses or both related to unknown address, i.e., the resulting pattern must be of a format of *u_operation* and *not_u_operation* alternating.
- *Rule 4*: After recursive reductions of *Rule 1*, *Rule 2* and *Rule 3*, either $\beta \leq 3$ holds, or the following sub-pattern still exists:
  - $\ldots \rightsquigarrow u\_operation \rightsquigarrow not\_u\_operation \rightsquigarrow u\_operation \rightsquigarrow \ldots$

  If the pattern contains this sub-pattern, the three-step sub-pattern must be an effective vulnerability according to Table 2 and reduction rules shown in Section 3.3. The corresponding pattern can be treated effective and the checking is done.

Algorithm 1 β-Step ($\beta > 3$) Pattern Reduction

```
Require: $\beta$: number of steps of the pattern
    $step\_list$: a two-dimensional dynamic-size array. $step\_list[0]$ contains the states of each step of the original pattern in order. $step\_list[1]$, $step\_list[2]$, ... is empty initially.
Ensure: $reduce\_list$: reduced effective vulnerability pattern(s) array. It will be an empty list if the original pattern does not correspond to an effective vulnerability.
1: while $step\_list$.contain($\star$) and $\star$.index not 0 do
2:     $Rule\_1$ ($step\_list$)
3: end while
4: while ($step\_list$.contain($A_{inv}$) and $A_{inv}$.index not 0) or ($step\_list$.contain($V_{inv}$) and $V_{inv}$.index not 0) do
5:     $Rule\_2$ ($step\_list$)
6: end while
7: while $step\_list$.contain(adjacent $not\_u\_operation$ or $u\_operation$) do
8:     $Rule\_3$ ($step\_list$)
9: end while
10: $reduce\_list$ = $Rule\_4$ ($step\_list$)
11: return $reduce\_list$
```

We make use of the four *Rule*s in the way shown in Algorithm 1 either i) to reduce a β-step ($\beta > 3$) pattern to be within three steps or ii) demonstrate that the β-step pattern can be mapped to one or more three-step vulnerabilities if it is effective.

In conclusion, the three-step model can model all possible timing-based SA TLB vulnerability with any β steps. Attacks which are represented by more than three steps can be always reduced to one

Table 7: The table shows additional possible timing-based TLB vulnerabilities when different types of TLB invalidations are possible. The column headings are the same as in Table 2.

Attack Strategy	Vulnerability Type			Macro Type	Attack
	$Step$ 1	$Step$ 2	$Step$ 3		
TLB Internal Collision	A_a^{inv}	V_u	V_a (fast)	IH	(1)
	V_a^{inv}	V_u	V_a (fast)	IH	(1)
TLB Flush + Reload	A_a^{inv}	V_u	A_a (fast)	EH	**new**
	V_a^{inv}	V_u	A_a (fast)	EH	**new**
TLB Reload + Time	V_u^{inv}	A_a	V_u (fast)	EH	**new**
	V_u^{inv}	V_a	V_u (fast)	IH	**new**
TLB Flush + Probe	A_a	V_u^{inv}	A_a (slow)	EH	**new**
	A_a	V_u^{inv}	V_a (slow)	IH	**new**
	V_a	V_u^{inv}	A_a (slow)	EH	**new**
	V_a	V_u^{inv}	V_a (slow)	IH	**new**
TLB Flush + Time	V_u	$A_{a^{inv}}$	V_u (slow)	EH	**new**
	V_u	$V_{a^{inv}}$	V_u (slow)	IH	**new**
TLB Internal Collision Invalidation	A^{inv}	V_u	V_a^{inv} (slow)	IH	**new**
	V^{inv}	V_u	V_a^{inv} (slow)	IH	**new**
	A_d	V_u	V_a^{inv} (slow)	IH	**new**
	V_d	V_u	V_a^{inv} (slow)	IH	**new**
	$A_{a^{alias}}$	V_u	V_a^{inv} (slow)	IH	**new**
	$V_{a^{alias}}$	V_u	V_a^{inv} (slow)	IH	**new**
TLB Flush + Flush	A_a^{inv}	V_u	V_a^{inv} (slow)	IH	**new**
	V_a^{inv}	V_u	V_a^{inv} (slow)	IH	**new**
	A_a^{inv}	V_u	A_a^{inv} (slow)	EH	**new**
	V_a^{inv}	V_u	A_a^{inv} (slow)	EH	**new**
TLB Flush + Reload Invalidation	A^{inv}	V_u	A_a^{inv} (slow)	EH	**new**
	V^{inv}	V_u	A_a^{inv} (slow)	EH	**new**
	A_d	V_u	A_a^{inv} (slow)	EH	**new**
	V_d	V_u	A_a^{inv} (slow)	EH	**new**
	$A_{a^{alias}}$	V_u	A_a^{inv} (slow)	EH	**new**
	$V_{a^{alias}}$	V_u	A_a^{inv} (slow)	EH	**new**
TLB Reload + Time Invalidation	V_u^{inv}	A_a	V_u^{inv} (slow)	EH	**new**
	V_u^{inv}	V_a	V_u^{inv} (slow)	IH	**new**
TLB Flush + Probe Invalidation	A_a	V_u^{inv}	A_a^{inv} (fast)	EH	**new**
	A_a	V_u^{inv}	V_a^{inv} (fast)	IH	**new**
	V_a	V_u^{inv}	A_a^{inv} (fast)	EH	**new**
	V_a	V_u^{inv}	V_a^{inv} (fast)	IH	**new**
TLB Evict + Time Invalidation	V_u	A_d	V_u^{inv} (fast)	EM	**new**
	V_u	A_a	V_u^{inv} (fast)	EM	**new**
TLB Prime + Probe Invalidation	A_d	V_u	A_d^{inv} (fast)	EM	**new**
	A_a	V_u	A_a^{inv} (fast)	EM	**new**
TLB Bernstein's Invalidation Attack	V_u	V_a	V_u^{inv} (fast)	IM	**new**
	V_u	V_d	V_u^{inv} (fast)	IM	**new**
	V_d	V_u	V_d^{inv} (fast)	IM	**new**
	V_a	V_u	V_a^{inv} (fast)	IM	**new**
TLB Evict + Probe Invalidation	V_d	V_u	A_d^{inv} (fast)	EM	**new**
	V_a	V_u	A_a^{inv} (fast)	EM	**new**
TLB Prime + Time Invalidation	A_d	V_u	V_d^{inv}(fast)	IM	**new**
	A_a	V_u	V_a^{inv} (fast)	IM	**new**
TLB Flush + Time Invalidation	V_u	A_a^{inv}	V_u^{inv} (fast)	EH	**new**
	V_u	V_a^{inv}	V_u^{inv} (fast)	IH	**new**

(1) Double Page Fault attack [12].

Table 6: The 7 specific-address-invalidation-related states for a single TLB block.

States	Description
V_u^{inv}	The TLB block previously containing translation for a memory address u is now invalid. Attacker does not know u, but u is from a sensitive memory range x of memory locations, range which is known to the attacker. The address u may have same page index as a and thus conflict with them in the TLB block.
A_a^{inv} or V_a^{inv}	The TLB block previously containing translation for a memory address a is now invalid. The attacker knows the address a, independent of whether the access was by the victim or the attacker themselves. The address a is from within the range of sensitive locations x. The address a may or may not be the same as the address u.
$A_{a^{alias}}^{inv}$ or $V_{a^{alias}}^{inv}$	The TLB block previously containing translation for a memory address a^{alias} is now invalid. The address a^{alias} is within the sensitive memory range x. It is not the same as a, but it has same page index and maps to the same TLB block, i.e. it "aliases" to the same block.
A_d^{inv} or V_d^{inv}	The TLB block previously containing translation for a memory address d is now invalid. The address d is not within the sensitive memory range x.

(or more) vulnerabilities from our three-step model; and thus, using more than three step is not necessary.

APPENDIX B: ADDITIONAL ATTACKS

Table 7 shows additional possible timing-based TLB vulnerabilities when different types of TLB invalidations are possible, which are listed in Table 6. The translation can be "removed" from the TLB block by the victim or the attacker as the result of TLB block being invalidated, e.g., through a dedicated TLB flush instruction. We are unaware of existing RISC-V ISAs or systems which have such features, but future extensions may add such features and they could cause security bugs. For example, invalidating a specific TLB entry for some processors on Linux is possible by using *mprotect()* system call, which changes the access protection bits for the calling process's memory pages.

If it is possible for the attacker or the victim to trigger invalidation of a specific address or entry in the TLB, then attacks such as TLB Flush + Time become possible. Invalidation of a specific address or entry is needed in $Step$ 2 to derive information about V_u in the last step.

If invalidation of TLB can be for a specific address or entry and has variable timing, then attacks such as TLB Flush + Flush become possible. One performance improvement to TLB could be that for each invalidation, check the TLB first. If TLB entry is already invalid, the invalidation is done. If it is valid, then during the second cycle, update the TLB entry to mark it as invalid. This may shorten each cycle, but would introduce $fast$ or $slow$ timing differences that lead to the further attacks.