Load-Step: A Precise TrustZone Execution Control Framework for Exploring New Side-channel Attacks Like Flush+Evict

Zili KOU
Hong Kong University of Science and Technology
zkou@connect.ust.hk

Wenjian HE
Hong Kong University of Science and Technology
wheac@connect.ust.hk

Sharad Sinha
Indian Institute of Technology Goa
sharad@iitgoa.ac.in

Wei ZHANG
Hong Kong University of Science and Technology
wei.zhang@ust.hk

***Abstract*—Trusted execution environments (TEEs) are imported into processors to protect sensitive programs against a potentially malicious operating system (OS), though, they are announced not effective in defending microarchitecture (μarch) side-channel attacks. Furthermore, TEE attackers often utilize their high privilege to strengthen attacks by interrupting the execution of victim programs. Maximum temporal resolution is achieved on the x86 platform, which interrupts and measures by every instruction. However, the capability of μarch side-channel attacks and the precision a kernel-privileged attacker can achieve in the TrustZone system are still unexplored. In this paper, we propose Load-Step, a precise framework that periodically interrupts the victim program in the TrustZone system and then conducts μarch side-channel attacks. Our self-designed benchmark shows that Load-Step can invoke interrupts with load-instruction precision. Based on Load-Step, we present Flush+Evict, a new side-channel attack detecting the Arm Cache Coherent Interconnect (Arm-CCI). It outperforms Prime+Probe with much higher precision and 282% of the profiling speed. When attacking the RSA decryption in the latest MbedTLS library, Load-Step can recover the full key by only a single trace in 7.5 seconds. Our work thus breaches the exponent blinding, which aims to defend RSA decryption against side-channel attacks in the MbedTLS library.**

***Index Terms*—μarch attack, Embedded system, Arm TrustZone**

I. Introduction

Personal privacy and sensitive information in computing systems are receiving increasing attention. However, software vulnerabilities are frequently discovered since some privileged system programs contain insecure code that can be utilized to achieve privilege escalation. TEEs are thus imported into modern central processing units (CPUs) to offer isolation between enclaves and the outside, claiming that even a malicious OS cannot compromise the enclave programs. Till now, Intel Software Guard Extensions (SGX) [1] is the most common TEE on the x86 platform and TrustZone [2] is built in almost every Arm device.

However, recent research [3, 4, 5, 6] reveals that μarch side-channel attacks are effective in leaking secrets of enclave programs. In the upgraded threat model of systems with TEEs, attackers are considered to be the kernel-privileged software or even the malicious OS. Thus, TEE attackers usually take advantage of their high privilege in the OS to significantly improve traditional μarch side-channel attacks. For instance, cache side-channel attacks conducted by kernel-privileged attackers require fewer observation traces to recover the keys of cryptography algorithms [3]. Furthermore, attackers [3, 4, 5] in the SGX usually exploit the interrupt mechanism of the OS to repeatedly preempt the execution of enclave programs. In such cases, the temporal resolution of the attacks is determined by the interrupt precision. The per-instruction precision is achieved by SGX-Step [5] with which attackers can even expose the execution flow of the victim program [4]. However, there is no systematic analysis of μarch side-channel attacks in the TrustZone system. The improvement of the μarch side-channel attacks and the maximum temporal resolution the attacker can obtain are still unknown.

978-1-6654-3274-0/21/$31.00 ©2021 IEEE

This paper systematically analyzes the TrustZone system and claims that the TrustZone is weak to μarch side-channel attack as a malicious OS can exhaustively strengthen itself to obtain fine-grained information. In detail, we propose a framework named Load-Step, which lies in the normal world that can interrupt a secure world program with high-precision and low-noise. We present Flush+Evict, which is the first attack to exploit the side-channel from the scope of the whole TrustZone system instead of only within a core. Summarized, we make the following contributions:

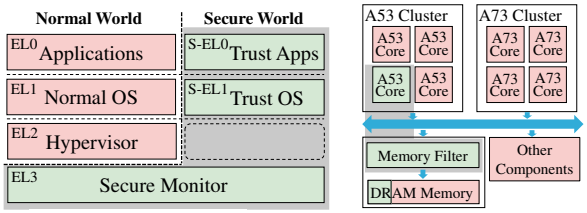
- Our systematic analysis of the TrustZone system points out that some Arm-specific properties like the power constraint and the requirement of memory coherency can be utilized to enhance the μarch side-channel attacks.
- We implement Load-Step as a Linux kernel driver that can be installed on rooted Arm devices. Load-Step has a precision up to load-instruction level.
- Based on Load-Step, we present Flush+Evict as a new μarch side-channel attack detecting the Arm-CCI. Compared to Prime+Probe, Flush+Evict achieves much higher throughput and precision.
- We show that a kernel-privileged attacker in the TrustZone system can recover the full key of the RSA decryption in the latest MbedTLS library by a single detection trace. This implies some latest defenses of side-channel attacks, like exponent blinding, are still vulnerable.

II. Background and Related Work

A. Cache Side-channel Attacks

The cache is an essential component in modern CPUs that saves much time in loading data. The cache is implemented in fixed-size lines and is usually set-associative, i.e. an N way set-associative cache means every set can hold N lines. Data with the same pattern of physical addresses will map to the same cache set. If new data needs to be cached into a saturated cache set, one old cache line should be evicted. Based on the properties of the cache, adversaries exploit cache side-channels to breach cryptography algorithms [7, 8]. Here we present two types of cache side-channel attacks.

Evict-based: Prime+Probe [8]. The attacker elaborates a group of data called an eviction set that maps to the same cache set. When loading the eviction set, the specific cache set is fully occupied by the attacker. If the victim loads data that also maps to this cache set, the data of the attacker is evicted.



(a) Software (b) Hardware

Fig. 1: Control domains of the TrustZone system.

The attacker therefore needs more time to reload the eviction set, indicating this cache set has been accessed by the victim in the past.

Flush-based: Flush+Reload [7] and Flush+Flush [9]. The attacker flushes a cache line of a specific address, causing this data to be evicted from the cache. If the victim loads that data, the data is cached and remains in the cache. The attacker therefore needs less time to reload/re-flush this data. In flush-based attacks, the attacker needs to share the same memory space with the victim.

B. Side-channel Attacks against Enclaves

TEEs are claimed to defend the enclaves against even a malicious OS. However, recent research has revealed that TEE attackers can conduct high-resolution side-channel attacks with the help of upgraded privileges. On the Intel SGX platform, kernel-privileged attackers explore the OS's built-in functions, like the interrupt mechanism, to enhance their side-channel attacks [3, 4, 5, 10]. In detail, the attacker periodically raises interrupts to preempt the core previously used by the victim enclave. Then, attackers can observe side-channels including caches [3, 10], page tables [4, 5], and branch prediction units [11] every time when interrupting the enclave programs. In such cases, the temporal resolution of the attack is determined by the precision of the interrupt mechanism, i.e., how frequently the interrupts are raised. SGX-Step [5] uses the private hardware timer to invoke interrupts with per-instruction precision. Copycat [4] indicates a kernel-privileged attacker with per-instruction precision can even expose the balanced and page-aligned *switch case* statement.

C. Arm TrustZone

The Arm TrustZone aims to provide isolation for sensitive programs by hardware and software co-design. In software, the whole system is divided by exception levels (ELs). The EL3 and secure exception levels (S-EL0 and S-EL1) are initiated before the normal OS boots. Programs in the EL0-EL2 cannot access the memory of the secure world, and all communications are governed by the TrustZone-specific ***smc*** instruction. In hardware, components like the memory filter can block insecure memory accesses. Moreover, when a core switches to the secure world, the whole core, together with its μarch resources, are out of the control of the normal world. The shadow regions in Fig. 1 depict the control domains of the TrustZone in both software and hardware.

The TrustZone technology has been widely used in both academic projects [12, 13] and commercial products [14]. Despite the hot atmosphere of the SGX, only a few works discuss the μarch side-channel attacks in the TrustZone system. In detail, works in [6], [15], and [16] analyze the traditional

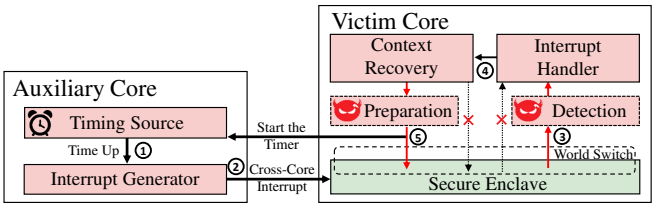


Fig. 2: Structure of Load-Step.

cache side-channel attacks on Arm devices and then present limited examples to claim that cache side-channel attacks are possible of leaking secrets from the TrustZone. Their threat models are in user-privilege without the exploration of the new upgraded threat model in TEEs. The work in [17] attacks Arm TrustZone in the malicious OS scenario, using the OS's capabilities to invoke interrupts and Prime+Probe to extract the keys of Qualcomm's ECDSA algorithm. Rather than analyzing the μarch vulnerabilities in the TrustZone system, this work aims more to breach a real cryptography algorithm, for which it takes hours to recover the key. To attack the TrustZone at the μarch level, a more systematic analysis that stands in a general point of view is expected and new Arm-specific attack methods with higher performance are promising to be discovered.

D. Threat model

Our attack assumes that a cryptography library is implemented in the secure world and offers normal world services to invoke. Meanwhile, the OS in the normal world is assumed malicious, which means the attacker can install any external kernel module or software driver to the OS. The attacker can assign any specific core to run a cryptography program in the secure world. These assumptions are common for launching attacks on TEEs [18] and are still within the assumptions made by the TrustZone.

III. Load-Step Framework Design

In this section, we first analyze the control domain of the TrustZone and then present our framework to handle the main difficulties when developing high-precision and low-noise interrupt-based attacks in the TrustZone system.

Unlike that of the SGX, the TrustZone's control domain is stricter in both software and hardware aspects. In software, it implements a separate trust OS in the S-EL1, and the enclave programs never rely on the normal OS for page walking. Thus, it is impossible to leak information by monitoring page tables. In hardware, when a core is assigned to run a program in the secure world, the normal OS loses control of all the hardware resources of this particular core. However, the components outside the control domain can still be thoroughly utilized. In software, attackers can edit the kernel code of the normal OS to increase the precision while reducing the noise. In hardware, any other components on the bus can be used by attackers to enhance the μarch side-channel attacks.

A. Structure

In the TrustZone system, once a core switches to the secure world, the normal OS would completely lose control of it. This renders the conventional interrupt manipulation methods [3, 5] not applicable to the TrustZone, as they require that some hardware resources such as the private hardware timer can survive during the world switch.

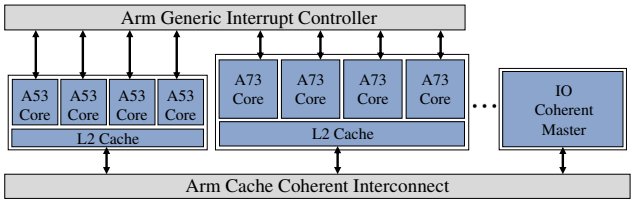


Fig. 3: The big.LITTLE architecture of Arm CPUs.

We then involve two cores in Load-Step: one as the auxiliary core and the other as the victim core. The attack procedure is shown in Fig. 2. ① The auxiliary core receives a time-up event from its timing source. ② It then controls the Arm generic interrupt controller (Arm-GIC) to generate a cross-core interrupt forwarding to the victim core. ③ The normal OS is responsible for handling cross-core interrupts. Thus, once the victim core receives the interrupt, the secure world is forced to save its context and switch to the normal world. Immediately following is the detection step designed for attackers to do various μarch side-channel attacks. ④ Then, we let the original OS routines proceed. ⑤ Before the victim core resumes to the secure world, we allow attackers to prepare the μarch states, after which Load-Step informs the auxiliary core to start the timer for the next attack epoch.

Note that Load-Step is a generic attack framework that different detection methods can be adopted to detect different μarch side-channels. Taking Prime+Probe on L2 cache as an example, when the victim core is interrupted and switches to the normal world, the victim core immediately does the *Probe* to gather the secrets and does the *Prime* just before it switches to the secure world. The detection methods in Load-Step are more powerful than those in traditional attacks, as the victim programs are suspended for every interrupt epoch during which the detection methods have sufficient time and privilege to detect, analyze, and prepare the μarch side-channels.

B. Installation

We implement Load-Step as an external kernel module that can be installed into the Linux OS. In the normal world OS, alongside the world switch function are the interrupt handler function and the context recovery function. To get a stable run-time environment with minimum noise, Load-Step firstly searches the memory location of the two functions and then replaces them with the malicious functions. Some hijack tricks such as direct memory modification are needed, but it is still within the privilege assumption of the threat model and feasible for a kernel module. Now, the detection method, as well as its preparation, are executed in the highest priority in the normal world, as shown in Fig. 2.

C. Timing Source

The precision of the interrupt invoking determines the temporal resolution of attacks, implying that the timing source is essential. We now discuss the possible timing sources, and we will present their performance results in Section V-A.

Modern CPUs usually implement private hardware timers for each core, which can deliver hardware interrupts for OS scheduling. When attacking enclave programs, the timer can be a reliable timing source with few software jitters. We build the necessary software driver for the private hardware timer on Arm devices so that the timer of the auxiliary core can periodically generate time-up events in a dedicated interval. However, the frequency of the timer on Arm devices is usually fixed and relatively slow, like 2 MHz of the platform we use, which is much slower compared to the core's speed.

Algorithm 1 Software timing sources

Variant-A: a Finite Loop
Temporal parameter: T_a
$t \leftarrow T_a$
do
  $t \leftarrow t - 1$
while $t > 0$

Variant-B: Detect Cycle Counter
Temporal parameter: T_b
$t_o \leftarrow read(PMCCNTR) + T_b$
do
  $t \leftarrow read(PMCCNTR)$
while $t < t_o$

TABLE I: Trade-off between reliability and resolution

	Reliability	Resolution
Hardware Timers	Few jitters	200 ns to 1000 ns
Software Methods	More software jitters	1 ns in a 1 GHz core

To overcome the limited precision of the hardware timing source, we exploit the possible software methods and provide two variants, as shown in Algorithm 1. In Variant-A, the auxiliary core runs a finite loop where the parameter T_a determines the interrupt interval. Variant-B takes advantage of the cycle count register (PMCCNTR) of Arm cores to get the time. The precisions of both the software timing sources are therefore as high as the executing speed of the core. However, software methods incur more jitters than hardware. A comparison between hardware and software timing sources is presented in Table I. The best choice varies on devices since the frequency of the hardware timer differs from that of chips.

D. Arm-specific Optimization

Most Arm CPUs are designed as the big.LITTLE architecture [19], which involves a high-performance cluster and a power-efficient cluster, as shown in Fig. 3. Dynamic voltage and frequency scaling (DVFS) is built in the Linux kernel to manage power consumption. To meet the power efficiency constraints, the frequency of little cores is much lower than that of big cores. For example, the frequency of little cores on Kirin 960 System on Chip (SoC) [20] ranges from 533 MHz to 1844 MHz, while the big cores can run up to 2362 MHz. As the TrustZone system relies on the normal world to invoke and schedule the trusted services, we then choose a little core as the victim core to run the sensitive program and set its frequency to the lowest while choosing a big core as the auxiliary core with its maximized speed. As a result, the relative precision of the software timing source is increased. Additionally, when the system is idle, it is more common for an Arm device to shut down all other cores, only leaving a little core on to meet the minimum demand. We utilize this feature to greatly reduce the noise as we can shut down the other three little cores that share the targeted L2 cache with the victim core.

IV. Flush+Evict

Traditional side-channel attackers usually need many observation traces to enhance their confidence of secrets guessing because every single trace unavoidably involves some noise or only when gathering enough traces can the attacker have the full information to guess the secret. The precision and the throughput of a side-channel attack thus determine the capability and the practicality of the attack. After all, it is practically infeasible to launch a side-channel attack that requires millions of traces in hours to figure out one cryptography

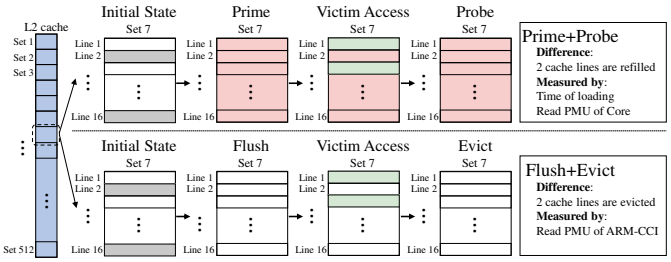


Fig. 4: Methodology of Prime+Probe and Flush+Evict.

key. TEE attackers [3, 17] usually adopt Prime+Probe to monitor the victim program's data flow by the cache side-channel. However, evict-based attacks have lower precision and throughput than flush-based attacks due to the costly loading of the eviction set. In this section, we present Flush+Evict and its design challenges. Flush+Evict is the first flush-based attack targeting the TrustZone system.

a) Sharing memory: The main constraint of flush-based attacks is that the attacker must share the memory space with the victim, which contradicts the TrustZone's isolation. Flushing the victim's cache lines by the addresses of the data is impossible in the TrustZone system, however, there exist other Arm-specific methods that can achieve the same effects. The Armv8 [19] instruction set architecture (ISA) implements more special cache maintenance instructions than the ISA on x86. ***DC CISW*** denotes cleaning and invalidating the data cache by *set* and *way*. This flexible instruction offers us the ability to precisely flush a specific cache set instead of loading the eviction set to occupy it.

b) Timing difference: For a flush-based attack to work, it is fundamental to have a special flush instruction, which exhibits timing difference between cached and uncached data. Unfortunately, ***DC CISW*** does not demonstrate a clear timing difference on a cache line regardless of whether it is empty or cached. Additionally, no performance counter of the Arm core can detect such a difference. We address this challenge by leveraging the Arm-CCI to detect the state of the cache. As shown in Fig. 3, there is no shared cache connecting two clusters except the Arm-CCI. To maintain cache coherency between clusters, the Arm-CCI adopts a snoop-based policy to synchronize two clusters [21]. For instance, when one cluster evicts data from its cache, it must send an *evict transaction* to the Arm-CCI. Moreover, the kernel-privileged attacker is able to access the Arm-CCI's performance monitor units (PMUs) that can count the number of *evict transactions*.

We then present Flush+Evict, a high precision and throughput side-channel attack, to distinguish whether the victim has previously accessed the data belonging to a specific cache set, as shown in Fig. 4. In the ***Flush*** step, the attacker flushes a cache set by ***DC CISW***, making it empty for the victim. In the ***Evict*** step, the attacker flushes the set again and checks the increment of the *evict transaction* by Arm-CCI's PMU. The increment ranges from zero to the set-associativity of the cache, implying the extent of access by the victim.

V. Evaluation

We evaluate Load-Step on a real Arm platform, Hikey 960 with Kirin 960 SoC [20]. The frequency of its private hardware timers is 2 MHz. It has eight Armv8 cores in the big.LITTLE architecture, four Cortex A73s [22] in the big cluster and four Cortex A53s [23] in the little cluster, as shown in Fig. 3. Linux kernel 5.5 is running in the EL1 and the open-source TrustZone design called OPTEE [24] is implemented in the S-EL1 as the trust OS. We choose an A53 core as the victim core and an A73 core as the auxiliary core. We perform Flush+Evict side-channel attack on the A53's L2 cache, which is 16-way set-associative and has 512 cache sets in total.

Algorithm 2 Benchmark Program

Step 1: Elaborate a bunch of data $D[512][16]$ given the targeted cache has 512 cache sets that are 16-way set-associative.
$D[i][j]$ maps to the i^{th} cache set for all j.
Step 2: Access cache sets sequentially.
for all $i < 512$ **do**
for all $j < 16$ **do**
Load $D[i][j]$
end for
end for
Step 3: Release the memory and clear the context.

A. Temporal Resolution

Load-Step aims to interrupt the victim programs and conduct μarch side-channel attacks with high temporal resolution. Therefore, selecting the right timing source as well as the best parameters is important. Algorithm 2 describes our benchmark program, which firstly allocates a bunch of data whose size is the same as the L2 cache size, and then sequentially accesses the data maps to every cache set. Load-Step periodically interrupts the benchmark program and conducts Flush+Evict to profile the memory access of the entire L2 cache. We visualize the experiment results with the format of a heat map, where the Y-axis denotes the cache sets, the X-axis denotes the interrupt epochs, and the color denotes the extent of the memory access.

Fig. 5(a) shows that Load-Step invokes interrupts by every 16 load-instructions when using the private hardware timer as the timing source. In this case, the memory access pattern of the benchmark program looks like a straight line, starting from the first cache set to the last. Fig. 5(b) is the result with a shorter interrupt interval, which interrupts the benchmark program by 4 load-instructions on average. The best precision the private hardware timer can achieve is shown in Fig. 5(c), which invokes interrupts by 2 load-instructions on average. The program would fall into endless embedded interrupts if the parameter were set too small. This is because the limited frequency of the private hardware timer cannot help distinguish any more load-instructions and the further reduction of the parameter causes the interrupt interval to be smaller than the world switch period.

We then try the software timing sources, and the results show that software timing sources can easily defeat the hardware timing source with higher precision, as shown in Fig. 5(d)-5(f). Variant-A performs better than Variant-B due to its simple but effective design. In Fig. 5(e), Load-Step can almost achieve the load-instruction precision as there are only a few interrupt epochs that have two memory accesses. By slightly reducing the parameter, Load-Step finally distinguishes every load-instruction of the benchmark program, as shown in Fig. 5(f). In both software and hardware timing source, we cannot get perfectly even patterns due to the unavoidable jitters of the whole system. This also suggests that the jitters from other parts cancel out the intrinsic stability of the hardware timer. Therefore, the software timing sources are always prior unless the hardware timer has a very high frequency.

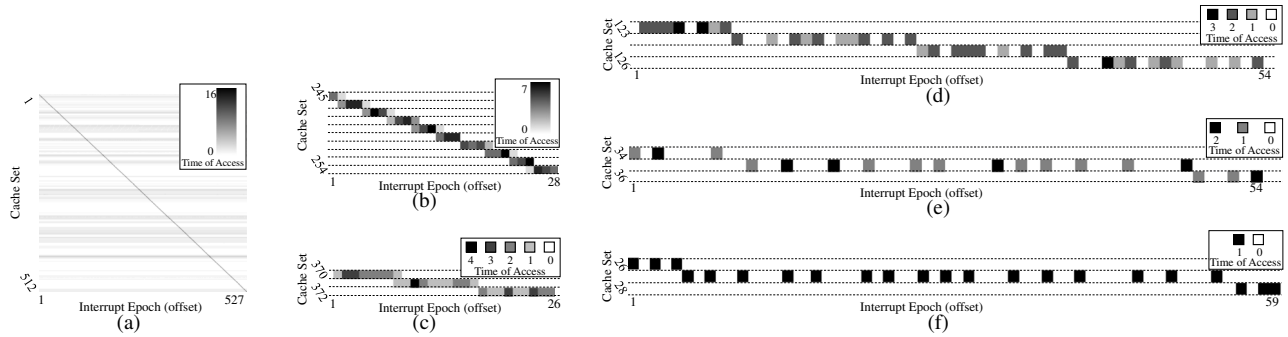


Fig. 5: Results of benchmark experiments. (a) Hardware timing source with $T_H = 51{f_p}^{-1}$. (b) Hardware timing source with $T_H = 46{f_p}^{-1}$. (c) Hardware timing source with $T_H = 41{f_p}^{-1}$, where T_H is the temporal parameter and $f_p = 2$ MHz is the frequency of the private hardware timer. (d) Software timing source Variant-B with $T_b = 680$. (e) Software timing source Variant-A with $T_a = 515$. (f) Software timing source Variant-A with $T_a = 505$.

Algorithm 3 RSA sliding-window algorithm

Given: exponent d, window size S, ciphertext C, modulo n
Compute: plaintext $M \leftarrow C^d \bmod n$
Step 1: pre-compute multipliers $W[2^{S-1}]$ to $W[2^S - 1]$
Step 2: exponentiation
  $P \leftarrow 1$
  for i from 1 to $length(d)$ **do**
    if $[d_i d_{i+1} ... d_{i+S-1}]_2$ matches any $j \in (2^{S-1}, 2^S - 1)$ **then**
      do $P \leftarrow P \times P \bmod n$ **for** S **times**
      $P \leftarrow P \times W[j] \bmod n$ //do a multiplication
      $i \leftarrow i + S$
    else
      $P \leftarrow P \times P \bmod n$ //do a square
    end if
  end for

B. RSA Single Trace Attack

In this experiment, we demonstrate that, with Load-Step, our Flush+Evict side-channel attack can break the security of the RSA decryption implemented by MbedTLS 2.22.0. We first introduce the implementation of the RSA algorithm and then present our attack.

1) Sliding-window Algorithm: In the latest MbedTLS library, the RSA decryption adopts the sliding-window algorithm as shown in Algorithm 3. A window slides from the first bit of the exponent d, which is exactly the private key, and does a square for each bit. When the bits in the window match a specific pattern, the program performs a multiplication with the corresponding multiplier. For instance, when $window\_size$ is 6, the program pre-computes multipliers $W[32]$ to $W[63]$. If the 6-bit value in the window matches "101000", which is 40 in decimal, the program performs a multiplication with $W[40]$. Note that both the square and the multiplication are done by the same function $montmul\_mpi()$. The multipliers are all such very large numbers that loading them would usually occupy more than 5 successive cache sets. Therefore, by detecting when and which cache sets are accessed, we can know the position and value of those multipliers and finally recover the private key.

In our attack, rather than building a simplified prototype to prove the concept, we target a real 4096-bit RSA implemented by the MbedTLS under its default configuration. The $window\_size$ is set to 6, meaning the attacker must monitor the 32 multipliers simultaneously, and the exponent blinding [25] is added to randomize the exponent.

2) Attacking the RSA decryption: We use Load-Step to periodically invoke interrupts, and for every interrupt epoch, we detect the whole L2 cache by either Flush+Evict or Prime+Probe. The raw profiling result of one RSA decryption collected by Flush+Evict is shown in Fig. 6. This trace has 42841 interrupt

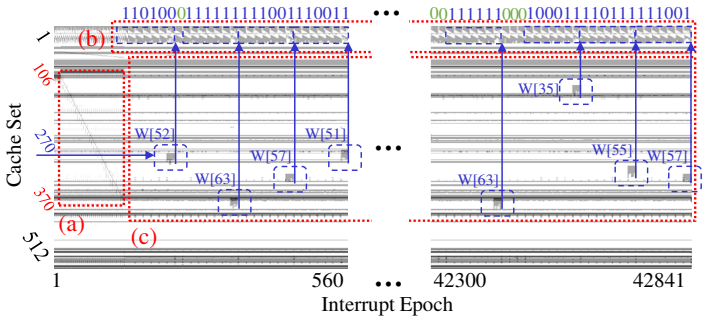


Fig. 6: Memory access pattern of RSA decryption.

epochs in total and we only show the beginning and the end. In brief, three essential areas offer us sufficient information to recover the exponent, as shown by the red symbols in Fig. 6. **Area (a)**: the pre-computing of 32 multipliers leaves a slash-like pattern at the beginning of the profiling. This hints to us which cache set would be accessed when multipliers are loaded into the L2 cache, i.e., the first multiplier $W[32]$ would be cached in the $106^{th} set$ and the last multiplier $W[63]$ would be cached in the $370^{th} set$. **Area (b)**: the $montmul\_mpi()$ is invoked repeatedly to do either the square or the multiplication, which leaves unique and repeated patterns in some cache sets. These patterns precisely describe the rounds of the algorithm that have a one-to-one relationship with exponent bits. We utilize such patterns to dramatically improve the precision of the key recovery. **Area (c)**: the square-shaped patterns caused by loading the $W[i]$ are the main target to deal with. We implement an automatic recovery program based on such an analysis, and its visualization is shown by the blue symbols in Fig. 6. Area (a) helps us label each $W[i]$. For instance, the first multiplier we discovered in Area (a) lies in several cache sets starting from the $270^{th} set$. Since we know the range of all the multipliers is from the $106^{th} set$ to the $370^{th} set$, we can then calculate that accessing the $270^{th} set$ means loading the $W[52]$, and the corresponding 6-bit value is "110100". Area (b) helps us precisely count the round of the algorithm. For every loading of the $W[i]$, the $montmul\_mpi()$ would be invoked 7 times. We then let every recovered 6-bit value occupy 7 rounds and the remaining rounds can just be recovered to '0's.

3) Key Recovery Performance: We first evaluate the performance of the key recovery when using Flush+Evict. Owing to the load-instruction granularity of Load-Step, we can generate millions of interrupt epochs in a single trace, which offers abundant information to recover the full key. As the noise level is also low, we can speed up the attack by reducing the interrupt frequency. Through our investigation, 12157 interrupt epochs

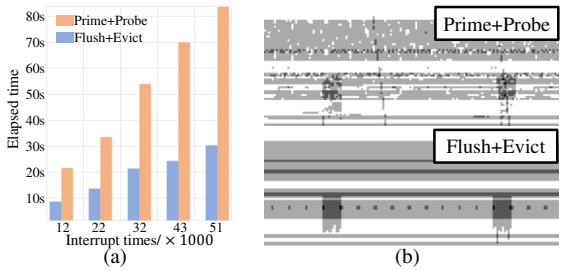


Fig. 7: Comparison between Prime+Probe and Flush+Evict in which (a) shows the profiling speed and (b) shows the profiling noise.

for a single trace are sufficient to 100% correctly recover the key, which takes only 7.4 seconds for profiling and fixed 5 seconds for recovering. We conduct experiments and obtain the same accuracy when applying the CRT [26] optimization.

We then compare the performance of Prime+Probe with Flush+Evict by conducting the same experiments but we replace the detection method with Prime+Probe. Fig. 7(a) shows the elapsed time when interrupting the victim program for the same times. Flush+Evict has 282% of the profiling speed of Prime+Probe on average, so as the throughput. As compared in Fig. 7(b), Prime+Probe incurs much more noise due to the heavy access of the eviction set. For a fair comparison, we apply the same automatic recovery program to process the profiling traces of both methods. The results in Table II indicate that Prime+Probe needs 4 times more time to recover either 88% or 98% of the key using a single trace. Although it may be possible for Prime+Probe to recover more key bits by manually analysis or by a more advanced algorithm in future work, Flush+Evict still prevails because of the attacking speed. Moreover, we also compare our work with a user-privileged work [27], which uses Prime+Probe on x86 devices to attack the 4096-bit RSA algorithm implemented by the MbedTLS 2.3.0. Note that their version of MbedTLS is not hardened by the exponent blinding and the $window\_size$ is set to 1, meaning the difficulty of key recovery is much lower than ours. To summarize, Load-Step, as a kernel-privileged framework, exhibits predominant performance improvement compared with a user-privileged attacker.

4) Exponent Blinding: Exponent blinding is designed to defend against side-channel attacks by randomizing the exponent d to $d + r_i(p-1)(q-1)$, where r_i is a random number in the i^{th} decryption, and the length of r_i denotes the blinding length. With exponent blinding, the collision rate of two of the same exponents dramatically decreases, making the side-channel attacks [6, 17, 27] that require multiple traces impossible. Taking the condition when the blinding length is 28 bytes as an example, the work in [27], which needs 11 traces to recover the full key, now must measure more than 4×10^{30} traces for getting collisions of the 11 same exponents. However, if the attacker can recover the full key by a single trace, the exponent blinding would be completely ineffective in protecting the secret key. The 100% correct guessing, $d+r(p-1)(q-1)$, can just be used to decrypt the ciphertext because of the arithmetic principle of the RSA [28] that $C^d = C^{d+a(p-1)(q-1)} \bmod n$ for any integer a. Additionally, it is computationally feasible to figure out the value of d given $d + r(p-1)(q-1)$, as well as the value of p given $d_p + r(p-1)$ and the value of q given $d_q + r(q-1)$ in the CRT mode [26].

TABLE II: Performance of key recovery

	Detection Method	Profile Traces	Interrupt Epochs	Elapsed Time	Recovery Accuracy
Load-Step	Flush+Evict	1	12157	7.4 s + 5 s	1.00
		1	10310	6.5 s + 5 s	0.978
		1	9283	5.9 s + 5 s	0.861
	Prime+Probe	1	17714	29.3 s + 5 s	0.978
		1	16637	27.3 s + 5 s	0.885
		1	13363	22.5 s + 5 s	0.823
[27][a]	Prime+Probe	11		$<$ 5 min	1.00

[a]It takes 3min to generate the eviction set, which is not needed for Load-Step

VI. Conclusion

In this paper, we presented Load-Step to claim that kernel-privileged attackers can conduct μarch side-channel attacks with load-instruction precision. We show that μarch side-channel attacks can profit by the low-noise and high-precision environment offered by Load-Step and therefore it is more powerful. For example, Flush+Evict, a new side-channel attack based on the Arm-CCI, performs much better in throughput and precision and can recover the full key of an exponent-blinding-protected RSA algorithm by only a single observation.

References

[1] B. C. Xing *et al.*, "Intel® SGX Software Support for Dynamic Memory Allocation inside An Enclave," in *Proc. of HASP*, 2016.
[2] A. Holding, "Arm Security Technology, Building A Secure System Using Trustzone Technology," 2009.
[3] A. Moghimi *et al.*, "Cachezoom: How SGX Amplifies the Power of Cache Attacks," in *Proc. of CHES*, 2017.
[4] D. Moghimi *et al.*, "CopyCat: Controlled Instruction-level Attacks on Enclaves for Maximal Key Extraction," *arXiv, abs/2002.08437*, 2020.
[5] J. Van Bulck *et al.*, "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control," in *Proc. of SysTEX*, 2017.
[6] M. Lipp *et al.*, "Armageddon: Cache Attacks on Mobile Devices," in *Proc. of USENIX Security*, 2016.
[7] Y. Yarom *et al.*, "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack," in *Proc. of USENIX Security*, 2014.
[8] F. Liu *et al.*, "Last-level Cache Side-channel Attacks Are Practical," in *Proc. of IEEE S&P*, 2015.
[9] D. Gruss *et al.*, "Flush+Flush: A Fast and Stealthy Cache Attack," in *Proc. of DIMVA*, 2016.
[10] M. Hähnel *et al.*, "High-resolution Side Channels for Untrusted Operating Systems," in *Proc. of USENIX ATC*, 2017.
[11] S. Lee *et al.*, "Inferring Fine-grained Control Flow inside SGX Enclaves with Branch Shadowing," in *Proc. of USENIX Security*, 2017.
[12] J. S. Jang *et al.*, "SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment." in *NDSS*, 2015.
[13] C. Marforio *et al.*, "Smartphones as Practical and Secure Location Verification Tokens for Payments." in *NDSS*, vol. 14, 2014.
[14] Samsung, "Samsung Knox," https://www.samsungknox.com/en/.
[15] M. Green *et al.*, "AutoLock: Why Cache Attacks on Arm Are Harder Than You Think," in *Proc. of USENIX Security*, 2017.
[16] N. Zhang *et al.*, "TruSpy: Cache Side-channel Information Leakage from the Secure World on Arm Devices." *Trans. on IACR Cryptol*, 2016.
[17] K. Ryan, "Hardware-Backed Heist: Extracting ECDSA Keys from Qualcomm's TrustZone," in *Proc. of ACM CCS*, 2019.
[18] D. Rosenberg, "Reflections on Trusting Trustzone," *BlackHat USA*, 2014.
[19] Arm, "Arm Architecture Reference Manual ARMv8," F.c.
[20] 96boards, "Hikey960," https://www.96boards.org/product/hikey960/.
[21] Arm, "Arm CoreLink CCI-400 Technical Reference Manual," r1p1.
[22] Arm, "Arm Cortex-A73 Core Technical Reference Manual," r0p2.
[23] Arm, "Arm Cortex-A53 Core Technical Reference Manual," r0p4.
[24] OPTEE, "OPTEE," https://www.op-tee.org/.
[25] P. C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *Proc. of CRYPTO*, 1996.
[26] M. J. Campagna *et al.*, "Key Recovery Method for CRT Implementation of RSA." *Trans. on IACR Cryptol*, vol. 2004, 2004.
[27] M. Schwarz *et al.*, "Malware Guard Extension: Using SGX to Conceal Cache Attacks," in *Proc. of DIMVA*, 2017.
[28] P. Meelu *et al.*, "RSA and Its Correctness through Modular Arithmetic," in *AIP Conference Proceedings*, 2010.