Defeating AutoLock: From Simulation to Real-World Cache-Timing Exploits against TrustZone

Quentin Forcioli[1], Sumanta Chaudhuri[1] and Jean-Luc Danger[1]

Telecom Paris, Palaiseau, FRANCE, name.surname@telecom-paris.fr

Abstract.
In this article, we present for the first time a cross-core *Prime+Probe* attack on ARM TrustZone, which bypasses the *AutoLock* mechanism. We introduce our simulation-driven methodology based on gem5 for vulnerability analysis. We demonstrate its utility in reverse engineering a SoC platform in order to study its microarchitectural behavior (caches, etc.), inside a simulator, in spite of hardware protection. We present a novel vulnerability analysis technique, which takes into account the cache set occupancy for targeted victim executable. This proves to be essential in identifying information leakage in presence of *AutoLock*. The above tool also identifies the cache lines leaking a maximum amount of information. A cross-core *Prime+Probe* attack is then mounted on these max-leakage cache lines both in simulation for fine-tuning, and in real hardware. We validate our analysis and attack method on *OP-TEE*, an open-source trusted execution environment running on *RockPi4* a board based on *RK3399* SoC. More specifically we target the RSA subroutine in the *MbedTLS* library used inside *OP-TEE*. Despite the presence of *AutoLock*, multiplier obfuscation, and assuming a cross-core attack, we are able to retrieve 30% of the key bits, which can later be used in *Branch-and-Prune* methods to recover the full key.

Keywords: SoC, gem5, Security, Virtual Platform, Penetration Testing, Co-Simulation,ARM, Trusted Execution, Trusted OS, TEE, Cache Timing Attacks, Micro-Architectural Attacks, Reverse Engineering

1 Introduction

Trusted execution environments (TEEs) have evolved to become the mainstay of security-related tasks in System-on-Chip (SoC) architectures. More and more security-related tasks, such as key storage and Digital Rights Management (DRM), are being executed within special security frameworks such as SGX [CD16] for x86, TrustZone [NMB+16] for ARMv8, and Keystone [LKS+20] for RISC-V architecture. Trusted execution environments (TEEs) are a special type of operating system (OS) that manage these security frameworks and provide cryptographic isolation between the 'Secure' and 'Non-Secure' worlds. In this article, we limit our scope to TrustZone [NMB+16] and ARMv8. Some well-known TEEs based on TrustZone include QSEE (Qualcomm), Toppers [fERtS], Trusty [And16], Samsung Knox [Sam15], SierraTEE, QSEE, and *OP-TEE* [YL20].

In a similar vein, attacks against TEEs have increased, becoming more vicious in nature. In this article, we focus on cache timing attacks against TEEs, particularly the RSA implementation within *OP-TEE* running on ARMv8 architecture, which targets a large class of devices, especially in the embedded and mobile domain. *OP-TEE*, being an open-source TEE, is robust against various attacks compared to obscure TEEs. Our

Table 1: State of the Art: Cache Timing Attacks on Trustzone, NA denotes that no information is found on this subject.

Ref. Attack	**Attacker**	**Victim**	**Type**	**Cross-Core/ LLC**	**Auto-Lock**
Ryan et. al. (2019) [Rya19]	Kernel	TrustZone	Interrupt-driven	NO	NO
Lapid et. al.(2018) [LW18]	Kernel	TrustZone	Prime- Probe	NO	NO
Nin Zhang et. al. (2016 TruSpy) [ZSS+16]	App/ Kernel	TrustZone	Prime-Probe	NO	NO
Xiaokuan Zhang et.al (2016) [ZXZ16]	App/ Kernel	Kernel	Flush-Reload	YES	NA
Kou et. al. (Load-Step 2021) [KHSZ21]	App/ Kernel	TrustZone	Interrupt-driven	YES	NA
Kou et. al. (2023) [KSHZ22]	App/ Kernel	TrustZone	Interrupt-driven (SF)	YES	NO
Lipp. et. al. (ArmaGeddon 2016) [LGS+16]	App	Kernel	Evict+ Reload	NO	NO
Ours	Kernel	TrustZone	**Prime-Probe**	**YES**	**YES**

method is unique in that we develop and fine-tune our attack in an architectural simulator (*gem5* [LPAA+20]) before porting it to real devices.

1.1 Motivation

AutoLock [GRLZ+17] is an undocumented feature in ARM processors that prevents cross-core eviction of cache lines from Last-Level Caches (LLC), making cache timing attacks from another core considerably difficult. Particularly in the context of TEEs, where the secure thread can be forced to run on a different core. In this article, we address the issue of circumventing *AutoLock* in a unique manner. We reproduce the ARM architecture in an open-source architecture simulator *gem5* [LPAA+20], along with *AutoLock*. Based on this simulator capable of running the same binary as the real board, we devise and fine-tune our attack to bypass *AutoLock*. It is worth noting that in the original *AutoLock* article [GRLZ+17], the authors used the DSTREAM probe from ARM with ARM DS-5 Debugger. This is not always possible in commercial devices as the JTAG port is password protected.

1.2 Related Work

Cache-Timing Attacks [Per05] are well-known in the literature and have been used against various implementations. They are also the backbone of various transient execution attacks such as Spectre [KHF+19] and Meltdown [LSG+18]. By measuring the access times related to cache hits and misses, a process can guess the data access patterns or instructions of a different process running on the same processor or another core that shares the same cache. Well-known cache timing attack techniques are listed below:

- Flush+Reload [YF14, LYG+15]: The shared caches are flushed by the attacker, before the victim execution. Then target addresses are reloaded and the time is measured.

- Prime+Probe [LYG+15]: The attacker fills all possible slots (prime-set) for the target address (victim). After the victim's execution, the attacker probes those slots to see if one of the slots is missing.
- Flush+Flush [GMWM16]: The target addresses are flushed by the attacker once before the victim's execution, and once after. The address usage is guessed from the timing of the second flush operation.
- Evict+Reload [GSM15]: Similar to Flush+Reload, but the initial flush operation is replaced by a cache eviction operation. This technique is useful when flush privilege is not available.
- Evict+Time [OST06]: A three-step process where, first, the victim is executed and the time required is measured. Next, the attacker evicts a target address and reruns the victim. If the time taken is longer than the first execution, the target address was probably used.

Trusted Execution Environments benefit from a smaller attack surface than classical operating systems, although they are still vulnerable to cache timing attacks. Some examples of cache timing attacks against TEEs are:

1. Ryan et al. (2019) [Rya19] use cache timing attacks to analyze cache traces from L1D and BTB in order to attack ECDSA in Qualcomm TEE. This relies on the TEE and the attack sharing the same CPU at different time slices. Their tool *CacheGrab* uses interrupts to halt the secure thread and to transfer execution to the normal world without any cache flush.

   Note: In *OP-TEE* interrupt driven attacks can be disabled using the config CFG_CORE_WORKAROUND_NSITR_CACHE_PRIME. It also relies on same core operation which is difficult to achieve.

2. Lapid et al. (2018) [LW18] use Prime+Probe and Flush+Reload techniques to attack *Samsung TrustZone Keymaster* in Trustonic's Kinibi secure OS. They reverse-engineered the Galaxy S6 BootROM to study the AES implementation in the *Keymaster* truslet.

   Note: It relies on the attack running on the same time-sliced CPU and the shared memory between the *normal* and the *secure world*.

3. Ning Zhang et al. (Truspy, 2016) [ZSS+16] present Prime-Probe attacks from the app or kernel privilege level to a victim (T-table-based AES implementation in OpenSSL) running inside TrustZone.

   Note: It targets L1 cache and works on same core only.

4. Xiaokuan Zhang et al. (2016) [ZXZ16] present a systematic exploration of vectors for flush-reload attacks. They demonstrate a novel construction of flush-reload side channels on last-level caches of ARM processors, which particularly exploit return-oriented programming techniques to reload instructions.

   Note: Flush+Reload attacks are not possible on TEEs since the Rich OS and secure OS do not share memory space.

5. Kou et al. (Load-Step 2021, 2023) [KHSZ21] present an interrupt-driven framework where the *trusted application* (TA) is halted, and control returns to the normal world. Then, a new method Flush+Evict is used to read the cache state.

   Note: Although the attacks are on LLC, there is no mention of *AutoLock*. In *OP-TEE*, interrupt-driven attacks like *CacheGrab* [Rya19] can be disabled using the configuration:

CFG_CORE_WORKAROUND_NSITR_CACHE_PRIME.

6. Kou et al. (Attack Directory 2023) [KSHZ22] present an interrupt-driven framework. The Snoop Function of ARM Cache-Coherent Interconnect CCI 500 is used to read the cache state of the *trusted application* (TA) in a halted state.

   Note: Same as above, interrupt-driven attacks like *CacheGrab* are too easy to detect and can be disabled using an *OP-TEE* build config. (see above)

7. Armageddon, by Lipp et al. [LGS+16], presents several *Prime+Probe* attacks in a cross-core scenario, however, not on TEEs. There is no mention of *AutoLock*.

   Note: According to [GRLZ+17], it seems that the attacks were carried out on ARM variants without *AutoLock*, such as the OnePlus One, the Alcatel One Touch Pop 2, and the Samsung Galaxy S6."

1.3 Contributions

Given the current state of the art, our contributions in this article are as follows:

- We designed a new methodology called *PyDevices* within the *gem5* simulator to integrate system devices directly using Python configuration scripts. By leveraging Ghidra [Roh19] alongside this interface, we established a rapid prototyping approach. (please see section 2). This new method i.e. *PyDevices* allows us to reverse-engineer currently unsupported platforms by *gem5*, and associated binary *bootroms*. This is possible regardless of password-protected JTAG ports found in commercial platforms which can not be probed using DSTREAM and ARM DS-5 (pl see. section 5.3).
- We present a novel vulnerability analysis technique called VictimScan, which takes into account the cache set occupancy and replacement policies for targeted victim executables instead of cache lines only. (pl. see subsection 5.5) This proves to be essential in identifying information leakage in presence of *AutoLock*. The above tool also identifies the cache lines leaking a maximum amount of information. Although multiple cache leakage analysis tools, both static and dynamic [GVR+23] exist, to our knowledge, none of them take into account the cache set occupancy for target indices, and thus unsuitable for devices with *AutoLock*.
- For the first time, we demonstrate a cross-core attack on TrustZone that bypasses *AutoLock*. Unlike most attacks in the literature, which utilize mechanisms like *CacheGrab* [Rya19] relying on interrupt-driven methods, our approach cannot be easily disabled in recent versions of *OP-TEE* using the configuration:

  CFG_CORE_WORKAROUND_NSITR_CACHE_PRIME

  Additionally, these attacks often assume same-core operation, which is challenging to achieve in practice and can be disabled by the TEE.

  .

1.4 Organization

The rest of the article is organized as follows: In section 2, we introduce our simulation platform based on *gem5* and discuss the associated GDB instrumentation, booting TEEs, and the *PyDevices* method. In section 3, we illustrate a simulation-driven method of vulnerability analysis which takes into account the cache set occupancy and replacement policies. In section 4 we show how the vulnerabilities detected in section 3 can be exploited in presence of *AutoLock*. Section 5 presents the experimental setup, the attacker model, the real hardware platform *RockPi4* and attack results. Finally, Section 6 concludes the article with an emphasis on future works.

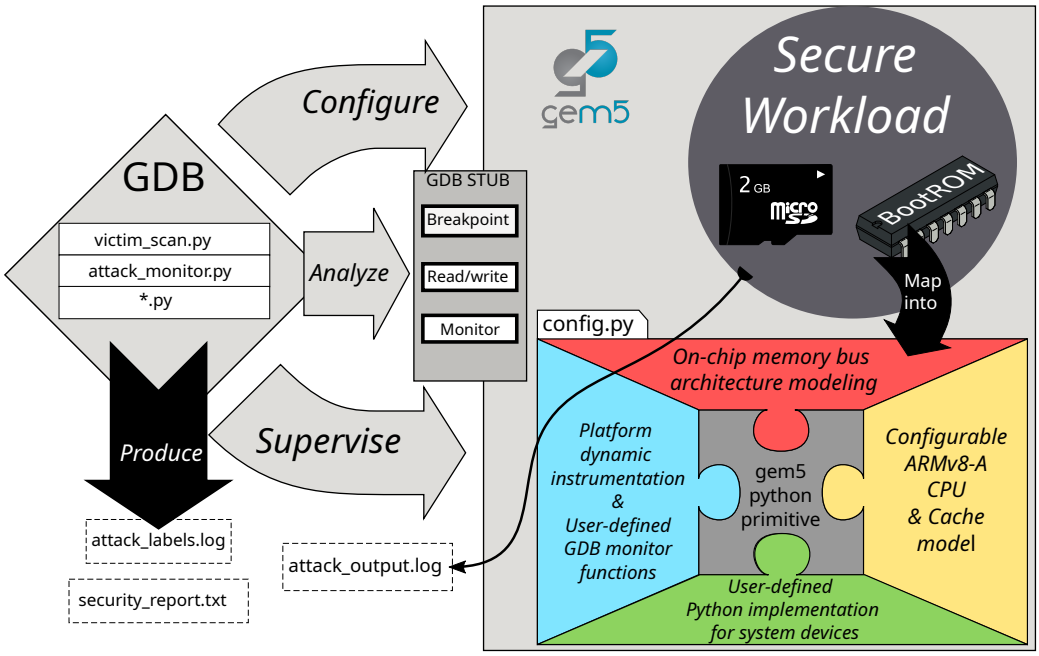

Figure 1: Overview of the Archisec platform, instrumentation tools and simulation capabilities. We added functionalities in *gem5* compiled binary to be then used in run-time loaded configuration files, in addition to *gem5* primitives, in order to simulate and instrument real ARMv8 hardware with their unmodified secure workloads.

2 Our Simulation Platform and Toolbox

Our platform is based on *gem5* [LPAA+20], an open-source computer architecture simulator widely used in the computer architecture community. Various uses of *gem5* for security evaluation can be found in the literature, categorized into hardware and software domains. Firstly, *gem5* has been employed to identify software vulnerabilities related to microarchitecture, such as cache timing. Reference [WGSW18] presents such an approach. CacheD [WWL+17] is another static analysis tool for detecting software cache information leakage, which employs *gem5* simulation to validate the results.

Table 2 shows simulation capabilities of *gem5*. As we can see all popular ISA/CPU families are supported, with advanced out-of-order cores, and all the elements of the memory hierarchy. Through its Python-based scripting interface, a SoC model can be described using its unique component model primitives called *SimObjects* (currently supporting over 300 parameterized models). All *SimObjects* are compiler c++ binary, python is used

Table 2: gem5 capabilities (version 20.0)

Feature	Options
System Modes	SysCall, Full-System
ISA	ARM, x86, RISC-V, SPARC, POWER, MIPS
CPU Models	Simple, In-order Pipeline, Out-of-order Pipeline, KVM
Cache Models	Classic, Ruby
DRAM Models	DDR3, DDR4, Wide-IO
I/0	Disk, NIC, PCI
External API	SST, SystemC TLM 2.0
Accelerator Models	GPU

only for configuration. It is possible to run unmodified binaries from a working SoC in *full-system* mode. It is also possible to use a faster simulation method where operating system routines (system-call) are emulated, called *syscall mode*.

Figure 1 depicts *gem5* capabilities and an overview of our platform. Apart from traditional *gem5* capabilities, which supports *armv8* cores, on-chip bus and memory hierarchy, we have added the capability to boot *Secure Workloads* such as GlobalPlatform-compatible TEEs. We have also added a method to model various I/O devices in a SoC quickly with the help of a Python script. This is used later to model real-life SoCs in *gem5* easily.

On the left of the figure 1, we show a toolbox for vulnerability analysis based on *gdb*. This toolbox helps the attacker to stop the execution flow of a secure workload at will and carry out a detailed inspection of its micro-architectural states. This toolbox is later used to do dynamic cache analysis of *secure workloads (TEEs)*.

2.1 *gdb* Instrumentation

It is already possible to configure *gem5* to closely follow all the execution states during binary execution. Indeed, with the appropriate debug flag activated, all executed instructions can be traced, and micro-architectural states at each cycle can be dumped. However, the amount of data required to trace all the cache states rapidly increases as the binary becomes more complex. In *full system mode*, it can easily take 1 Terabyte of data just to follow all the instructions executed during the booting of Linux.

To mitigate this problem in *gem5*, it is possible to add specific *gem5* instructions to isolate key parts of the workload. However, it requires modifying the binary to integrate the instructions. This is not suitable in our case because we want to use the same unmodified binary both in simulation and real hardware.

We propose a method based on *gdb* monitor implementation. With our modifications to *gem5*, it is possible to halt an executable, inspect micro-architectural states, and resume simulation. This method drastically reduces the amount of data produced(few megabytes), as we can focus more closely on the key parts of our program without sacrificing accuracy.

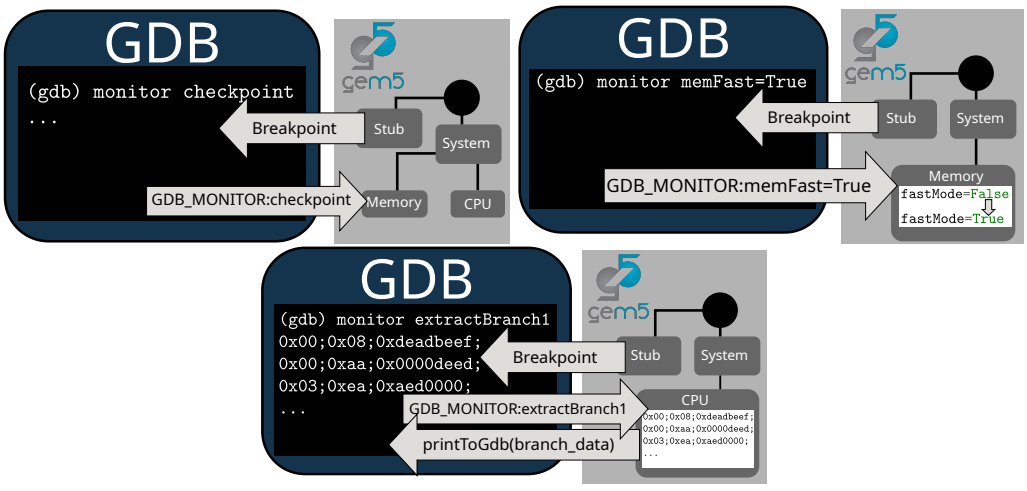

Figure 2: Typical use case for the interactive debug session between *gdb* and *gem5*: using *gem5* native functionality from *gdb* to control, configure, and analyze the simulation

Figure 2 describes our method in detail. There are two elements in the interactive session:

- gem5 simulation comprising of various *SimObjects* corresponding to SoC components described in the configuration file.
- *gdb* with a real user or with *gdb-python* running a specific script file, which can fully automate a monitoring process

We modified gem5 to provide access to the internal state of all *SimObjects*. With the help of *gdb-monitor calls*, we can control and modify the *SimObject* states from a *gdb-python* script. Notably:

- Change the precision and the speed of the simulation.(e.g by switching to simpler CPU models)
- Modify simulation parameters without having to change the disk images.
- Access to performance counters (normally requiring a specific CPU instruction)
- Flush caches.
- Synchronize information between *gdb* and the simulator.
- Automatically connect to a terminal for text I/O.

As we can see in figure 2, on the top left we are saving a checkpoint, in top right, we are switching to fast simulation for memory *SimObjects*, and in the bottom we are extracting branch predictor states corresponding to a particular branch in the simulated binary.

2.2 TrustZone and OP-TEE support in *gem5*

TrustZone is the commercial name for ARM's security framework needed to implement a secure enclave in ARM architecture. It mainly relies on specific execution modes. On ARMv8-A, these modes are called **Exception Levels** (EL).

There are multiple TEEs developed for ARMv7-A and ARMv8-A architectures. They leverage the TrustZone framework to deploy a secure execution environment. For our project, we chose OP-TEE [YL20], an open-source TEE that follows the GlobalPlatform API [lea21]. It is currently maintained directly by Arm as part of the TrustedFirmware-A Project [Lin23].

2.2.1 TEE Internals

As described in figure 3, TEEs is deployed in secure memory, using TrustZone hardware isolation. As the *rich OS* (e.. Linux) can not access secure memory, *secure OS* and *rich OS* exchange through the *secure monitor* running in EL3. For that, they use secure monitor calls(SMC) to switch CPU exception level between EL1 (OS) and EL3 (*secure monitor*). Through this mechanism, the OP-TEE kernel module can exchange with a TEE running in TrustZone protected EL1S. This TEE kernel module allows *linux client applications* to launch and communicate with *trusted applications* running inside TEE. For that, they use the *Global Platform* TEE client API following *GlobalPlatform*[lea21] specifications. In our examples, *trusted applications* are loaded from the *rich OS* filesystem, using the *tee daemon* which undergoes an integrity check using a stored key. *trusted applications* also have access to services provided through TEE internal core API implemented with service calls (SVC). They gave access to libraries directly implemented inside TEEs such as *mbedTLS*[Lin24] for OP-TEE.

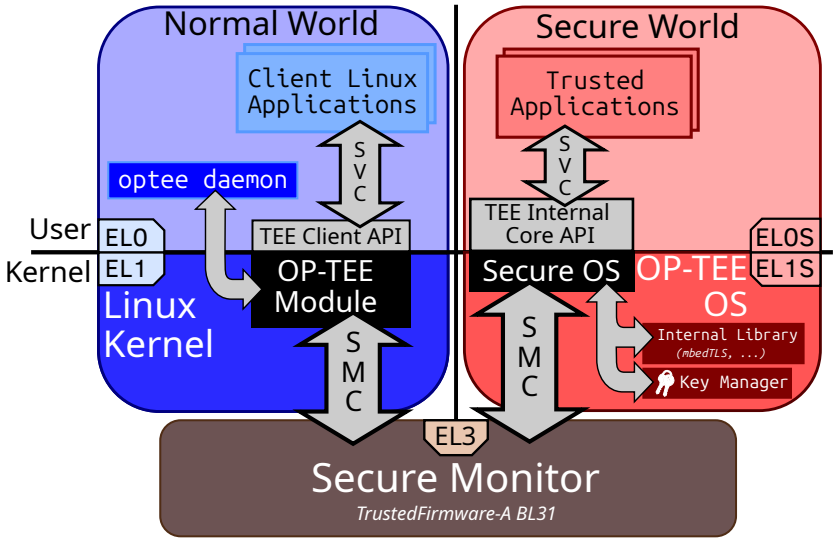

Figure 3: Typical OP-TEE scenario on ARMv8-A: A client application running in Linux, uses the TEE client API to interact with a Trusted Application (TA) running in OP-TEE OS. While **SVC** represents classical service calls used by Rich and *secure OS*, **SMC** represents Secure Monitor Calls. SMCs are used by the two OS to communicate via the Secure Monitor.

2.2.2 Booting TEEs in *gem5*

Current *gem5* ARM models support the EL3 firmware level associated with TrustZone support and EL0S and EL1S necessary to run TEEs and *trusted applications*. Moreover, *gem5*'s ARM MMU supports secure labeling, and it can use the NS bit to access unsecured regions from the secure exception Level.

We did the following modifications to *gem5* to correctly boot a TEE. We fixed some small bugs in the ARM ISA implementation in *gem5* that arose when trying to boot OP-TEE for the first time:

- The deactivation of EL2S was not correctly handled
- The generic ARM interrupt controller was not correctly synchronized when switching between secure and unsecure.

These modifications have been committed to *gem5* stable branch 21.1.

However, the secure labeling in *gem5* creates difficulties when operating it with *gdb*. When bootstrapping a TA, OP-TEE randomly places the TA into the address space. So we need *gdb* to grab from OP-TEE the offset to be able to debug the TA. For that purpose, we fixed the *packet protocol* inside *gem5* and allowed i*gdb* memory translation to be considered as both secure and unsecure.

2.3 PyDevices: fast system devices prototyping in *gem5*

ARMv8-A has a wide variety of platforms with different devices, different memory maps and different boot methods. On *gem5*, Only the Versatile Express-type platform (Vexpress) was implemented. This type of platform is mainly represented as a demonstration board from ARM and also virtual models like the *ARM FastModels*. The Vexpress platform in *gem5* is implemented through specific devices provided as SimObjects (UART, WatchDog, etc..) and in direct assumptions in the ISA (ArmSystem and Vexpress PowerControler are

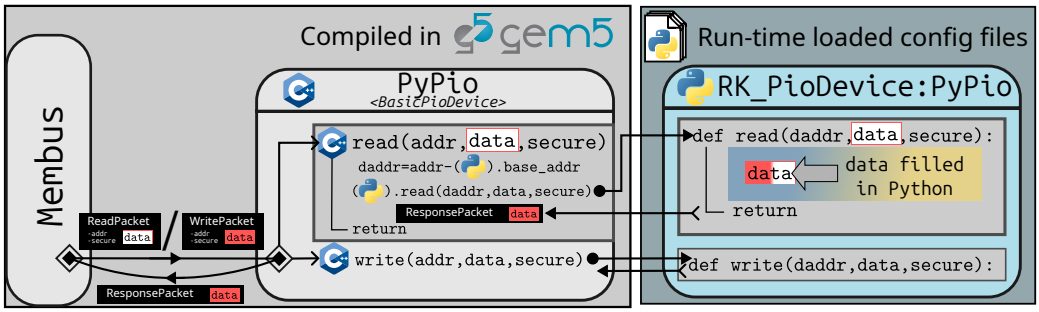

Figure 4: Integration of PyDevices in *gem5*: `RK_PioDevices` is memory-mapped devices implemented in Python thanks to the PyDevice API. Config files are responsible for describing the platform memory layout by specifying memory address for each device instance.

directly linked) and in generic ARM devices (GICs). Implementing a different Platform would require writing a SimObject for each device and each modification would require a lengthy compilation. This is why we designed a faster method: **PyDevices**. With PyDevices, system device implementation such as UART, WatchDogs, Timer, etc. can be directly implemented in Python configuration script files for *gem5* (see figure 4). To implement a new device using PyDevices, only implementations of the `read` and `write` methods corresponding to memory accesses to the devices are needed. This method implies that device implementations no longer need to be compiled directly in *gem5*. Instead, they can be modified at run time as you can see on figure 4. *gem5* at run times transfer packets the device received through its memory interface, to the `read` and `write` which are delegated to their PyDevice implementations in Python. To implement the `read` and `write` methods, PyDevices have access to:

- Specific PyDevices tools for DMA and interrupts activation and deactivation.
- Python tools provided by *gem5*: SimObjects Python methods, *gdb-python* API, Python simulation Events, etc.
- Any classical Python module: numpy, opencv, etc.

PyDevices are completed by *PyPowerState* to implement power-domain and CPU sleep/wake-up process. This easily accessible and modifiable implementation can be compared with QEMU'S[Bel05] which uses C to implement new system devices and components, which then need to be compiled directly in QEMU.

2.4 AutoLock integration in *gem5*

AutoLock, as described by [GRLZ+17], is an ARM-specific replacement policy designed to enforce inclusivity by preventing the eviction of L2 cache entries if they are present in a connected L1 cache (see figure 5). This policy ensures that L2 entries remain locked until they are evicted from the L1 caches. Only after their eviction from the L1 caches can they be considered for eviction from the L2 cache. Consequently, up to one entry per associativity of L1 caches can be locked in the L2 cache (illustrated in figure 5). For a complete eviction of an L2 cache set (i.e., all ways not locked), the L1 entries corresponding to the current L2 entries must have been evicted by lines that share the same L1 index but not the same L2 index. This scenario is much less likely to occur with AutoLock, resulting in L2 sets rarely being fully evictable.

Using *gem5*'s classical interface for cache replacement policies, we integrated an AutoLock implementation into our platform.

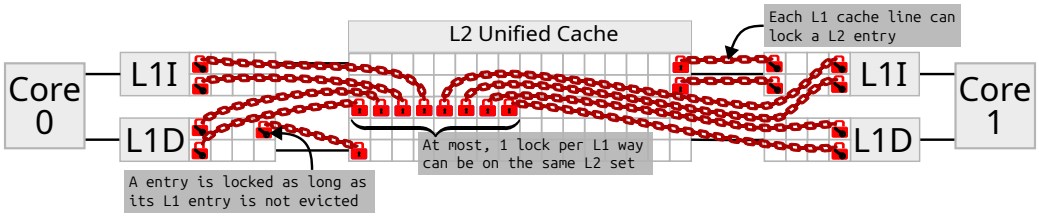

Figure 5: AutoLock: This cache replacement policy prevents eviction of L2 lines that are still present in a cache L1. This lock is set up when a cache L1 receive a *miss* response from the L2. This lock on the L2 line is opened when all the L1 lines that lock an L2 line are evicted.

3 Vulnerability analysis with a simulation platform

We use our simulation platform to study micro-architectural states during TEE execution which is not always possible in real hardware due to hardware protection. However, to efficiently use a simulator, which is slower compared to a real environment, we created methodologies that automatically analyze cache states. For this methodology, we need to define the points-of-interest in the algorithm in advance, called **Key Execution Point** (KEP). Our tool will then automatically highlight key cache states, called **Key Detectable States (KDS)**, associated with each *KEP* to detect them using cache timing attacks.

3.1 Overview of our simulation-based security analysis

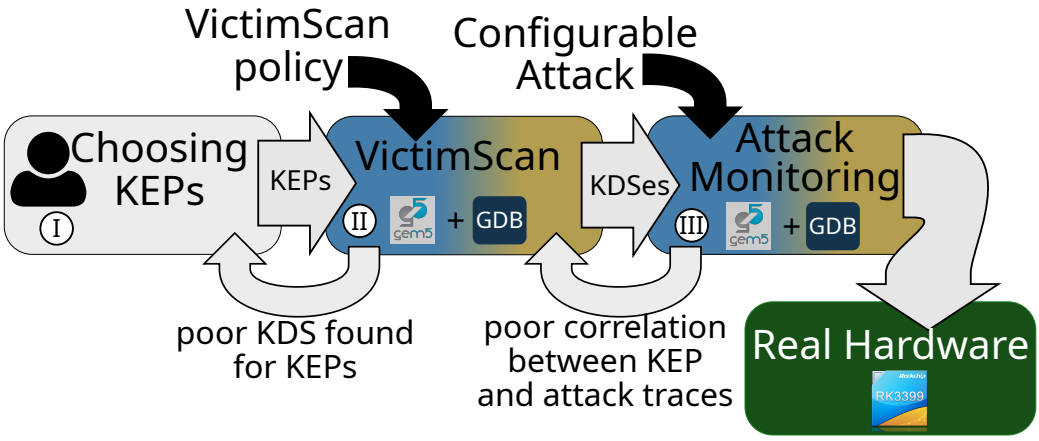

Figure 6: Overview of the simulation-based analysis: with this process we use our simulation platform (*gem5+gdb*) to craft an attack that we can test on real hardware.

To perform a vulnerability analysis, we use a three-step methodology (see figure 6) to propose a reasonable attack that can be run on our real platform:

I : **Choosing *KEP*s**: Based on the knowledge of the underlying algorithm, we propose points of interest in the algorithm called **Key Execution Point** (*KEP*) which potentially leaks information about the key. These points are regrouped in *KEP* classes that denote similar operations.

II : **VictimScan**: During an automatic/interactive debug session, the victim running in *gem5* is analyzed to extract key features associated with each class of *KEP*s, with respect to an attack information model. We call this information model a *VictimScan*

policy. These key features are called **Key Detectable States (KDS)**, and can be used to configure a cache timing attack.

III : **Attack Monitoring**: During an automatic interactive debug session, the attack scenario (victim + attack) running in *gem5* is monitored to supplement the traces of an attack with victim *KEP* execution. These data can then be processed to confirm the correlation between *KEP*s and attack results.

To use this process, we have to provide settings specific to the type of attack we want to analyze.

3.2 Choosing Key Execution Points

In our analysis process which relies on a precise simulation model, we can not statistically produce traces and then deduce point-of-interest as it is often done for side-channel analysis, for the following reason. As we consider our secure workload not modifiable on our real platform, we can only gather alternative traces in simulation. However, as our simulation model is vastly slower than the real hardware, studying diverse enough traces to construct points of interest would take a huge amount of time. Therefore, our methodology proposes to set the points-of-interest directly in the code instead. We call them **Key Execution Point (KEPs)**. These *KEP*s symbolize potential cache timing weaknesses associated with an execution path and/or a specific variable value. In this article, we denote *KEP*s as ♠,♥, [$\mathbb{S}$] or [$\mathbb{M}$]. *KEP*s sharing the same label are considered in the same class. *KEP*s in the same class represent similar information at the algorithm level: e.g. *which S-box is used for AES*, *which operation is performed by the square and multiply algorithm*, etc. As we are using *gdb* to track execution in our simulations, *KEP*s are implemented as *gdb* breakpoints, covering from single instructions to code segments.

We propose the following function which is reminiscent of the *Square and Multiply Algorithm* as an example to show how our methodology works. It is placed inside a *trusted application* and launched from Linux.

```
17 void crypto_f(big_int_t* A, big_int_t* B,big_int_t* E){
18   for(size_t i=0;i<BIG_N_B;i++){
19     if(bit(E,i)){
20       add(A,B,A);//♠
21     }
22     add(B,B,B);//♥
23 }}
```

KEP	pos
♥	demo_fun.c:22
♠	demo_fun.c:20

Figure 7: Demo Function: This function is placed in TA. It is studied using *VictimScan* and the 2 *KEP*s (♥ & ♠) described in the table.

The *KEP*s in this function, based on typical square and multiply algorithm weaknesses are represented as ♥ and ♠. Because we use the *gdb* format, we can generate them from typical IDEs (like VsCode). They are stored and provided to the following steps as a CSV file.

3.3 VictimScan

Traditional attacks mark target addresses as the branch addresses for the square and multiply sections. We use a different approach since we have reproduced the exact execution scenario in *gem5*. We use *gem5* simulation to find addresses which are susceptible to leak a maximum of information. It could be the branch address, or it would be a heap variable, page-table entry or something else related to that branch. We find out these addresses through our dynamic cache analysis tool *VictimScan* (Step II on figure 6).

VictimScan is a dynamic cache analyzer that identifies weaknesses in an unmodified binary which might lead to a possible cache-timing attack. *VictimScan* runs as Python script in *gdb*. It interacts with *gem5* simulation platform through an interactive debug session in order to configure the attack scenarios and access SoC internal cache states.

3.3.1 Classifying cache states

VictimScan takes the *KEP*s as inputs. It automatically dumps cache states associated with a *KEP* code segment when encountered by the simulation. From these cache dumps, *VictimScan* isolates and organizes cache states in order to detect correlation between them and *KEP*s. *VictimScan* can also use random *KEP*s, denoted with ⚄, which are associated with cache dumps randomly taken during the studied function.

At the end of the studied function, *VictimScan* outputs a set of **Key Detectable States (*KDS*es)** which corresponds to cache states which are the most correlated with each *KEP*s. *KDS*es are also used to select and configure an attack to detect them. We formalize *KDS* definition and their relation with attacks in section 3.5.

The correlation between *KEP* and *KDS*es is evaluated using a score based on accumulated cache data which compares with other *KEP*s (including random KEPs). These results are archived in a report as a key detectable state ranking for each key execution point using the aforementioned score.

By examining the maximum score value for each *KEP*, this report indicates if a KEP is easily identifiable through an attack that could detect a specific *KDS* w.r.t a random execution point or other *KEP*s.

3.3.2 Ranking methodology

Our goal is to identify *KDS*es that are more likely to be triggered around a *KEP* and that are less likely to be triggered by other *KEP*s or randomly. If no such *KDS* can be found we can safely declare the associated *KEP* is not detectable by an attack. For that, we use a simple scoring system, that is computed along the execution, i.e. each time a new cache dump is collected on a *KEP*. A *KDS* may be triggered by other *KEP*s, therefore we have to account for other cache lines that might produce the same *KDS*. For this purpose, we use the score function (equation 1) to find the best *KDS* to detect a *KEP* class. Given $\mathbb{S}$ is the set of all *KEP*s: $\mathbb{S} = \{\spadesuit, \heartsuit\}$

Given $h_x(k)$ which is the number of times a *KDS* k is a key feature of cache dump corresponding to a *KEP* class x.

Given w_x which is the number of times a *KEP* of class x has been found and thus a cache dump has been made.

This score function presented in Eq. 1 has three components:

- $\frac{h_x(k)}{w_x}$: The *KDS* k present in cache dump during the *KEP* gets a positive score, normalized by the number of times the associated KEP has been triggered.
- $\frac{h_{⚄}(k)}{w_{⚄}}$: the *KDS* k present in random dumps negative score, normalized by the number of times this random dump has been done.
- $\sum_{s\in\mathbb{S}, s\neq x} \frac{h_s(k)}{w_x \times card(\mathbb{S})}$: is the conflict contribution: The *KDS* k found in other *KEP*s dump get a negative score and are normalized by their related *KEP*'s trigger count and by the number of *KEP*s' classes.

Overall the score function for a *KDS* k is given by:

$$\text{score}_x(k) = \frac{h_x(k)}{w_x} - \frac{h_{⚄}(k)}{w_{⚄}} - \sum_{s\in\mathbb{S}, s\neq x} \frac{h_s(k)}{w_x \times card(\mathbb{S})} \quad (1)$$

This score value is also generally indicative of how much a **class of KEPs** is identifiable using a cache timing attack. A negative score indicates that the **class of KEP** cannot be identified.

3.3.3 Example: Demo TA

We run *VictimScan* on the function in figure 7. Using, the two *KEP* defined. In this example, we have two classes of *KEP*s: ♥ and ♠, with each only containing one *KEP*. With this configuration, *VictimScan* produces the report in figure 8.

```
♥->max_hit:('0x212', 64)
  (1):0x210
     score:0.6666666666666667
     hit_count:64
     top_addr:
      1@128=S#0x40093400[S#0x30218400]:add + 76 in section .text
  (2):0x211
     score:0.6666666666666667
     hit_count:64
     top_addr:
      1@128=S#0x40093440[S#0x30218440]:add + 140 in section .text
♠->max_hit:('0x20e', 35)
  (1):0x58
     score:1.0
     hit_count:35
     top_addr:
      1@70=S#0x400fc600[S#0x30281600]:__ta_no_share_heap + 130992 in section .bss
  (2):0x210
     score:0.6666666666666667
     hit_count:35
     top_addr:
      1@70=S#0x40093400[S#0x30218400]:add + 76 in section .text
```

Figure 8: Typical report from *VictimScan*, showing the *KEP*s classes, and the associated *KDS* with their scores; ranked in decreasing order. Each *KDS* also specifies the corresponding address in the binary .e.g instructions from .text section, or variables in the heap.

VictimScan suggests the best *KDS* to attack. Here, we only displayed the top two for the two classes of *KEP*s. Thanks to *gem5* integration, we are able to trace the main source for *KDS*: attributing it to an address (virtual and physical) and, if possible, a code line. We added this information to *gem5* packets and stored it in the cache model in their associated cache line. In most situations, *VictimScan* also finds *KDS* that have hidden causes, like:

- Automatic translation table walking: Address sources are table addresses
- Prefetching: Sources are instructions outside the function
- Heap and stack addresses: Sources are typically in the function accesses around the *KEP*.

In the report in figure 8, *KDS*es correspond to cache lines present in cache dumps that we define in section 3.5 as *1hit KDS*.

3.4 Attack Monitoring

Using the same *KEP* descriptions, our simulation platform can be used with *gdb* to perform a cache timing attack and add the *KEP* trigger to the produced traces. This is produced by completely monitoring the attack scenario to fuse information from the victim, *KEP*s, and attack traces. This step is called attack monitoring which corresponds to step III shown in figure 6. The traces thus produced contain cache timing results and *KEP* labels which help in assessing if an attack can produce the intended results.

3.4.1 Example: Demo TA

We run *attack monitoring* (Step III) on the example function in figure 7 using the report in figure 8 to configure a *Prime+Probe* attack. As the *KDS* reported in the ranking corresponds to a cache line index, we can use them to configure a *Prime+Probe* attack to probe for victim access to this index. This attack produces traces that we can plot and annotate them with *KEP*s, as shown in figure 9. On the zoomed version below (figure 9),

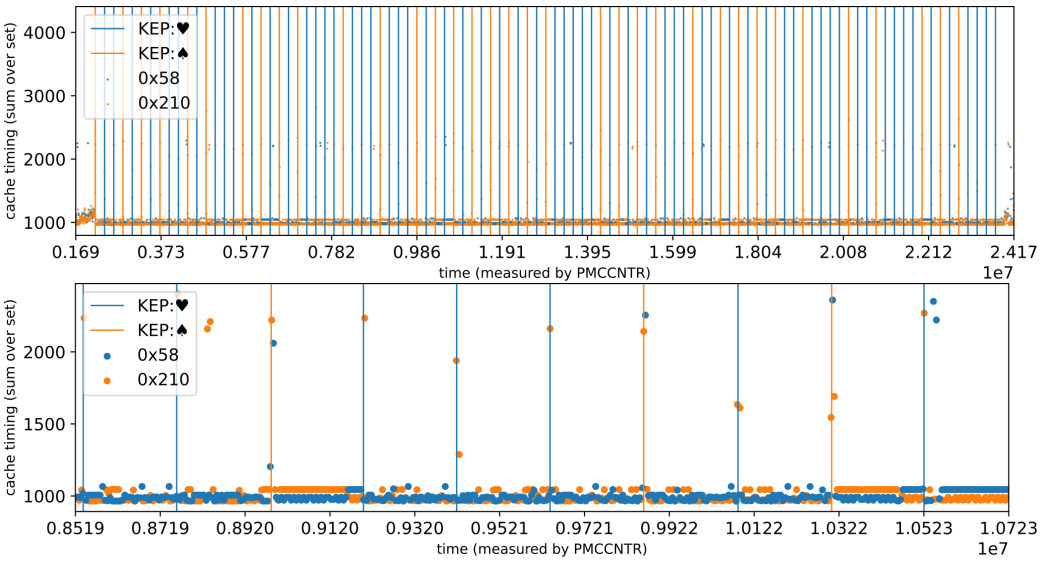

Figure 9: Cache timing traces for the simple example on figure 7, the bottom figure being the top zoomed. The X-axis is the time. The moments when execution reaches a *KEP*, are indicated with vertical lines. Cache-line timings are shown with colored dots, with their Y-value corresponding to the total access time of the prime set.

we can better see that *KEP*s are correlated with cache timings. We can thus isolate a signal pattern to re-identify *KEP*s using only the cache timings.

3.5 VictimScan policy: Detectable state definition

Given a set of *KEP*s corresponding to code segments, we need to find attack configurations capable of identifying such segments. To do so, we have to isolate key features in these cache dumps gathered around the *KEP*s. We call these key features, Key Detectable States (*KDS*). If the score (equation 1) of a *KDS* is high enough for a specific *KEP*, detecting this *KDS* is equivalent to detecting the *KEP*. These *KDS* definitions are therefore closely linked to attack types (like *Prime+Probe* or Flush+Reload) and describe the cache information observable by attacks of this type.

These *KDS* definitions are implemented in our *VictimScan* model as a component called *VictimScan policies*, visible on figure 10. As our *VictimScan* script running in *gdb* accesses

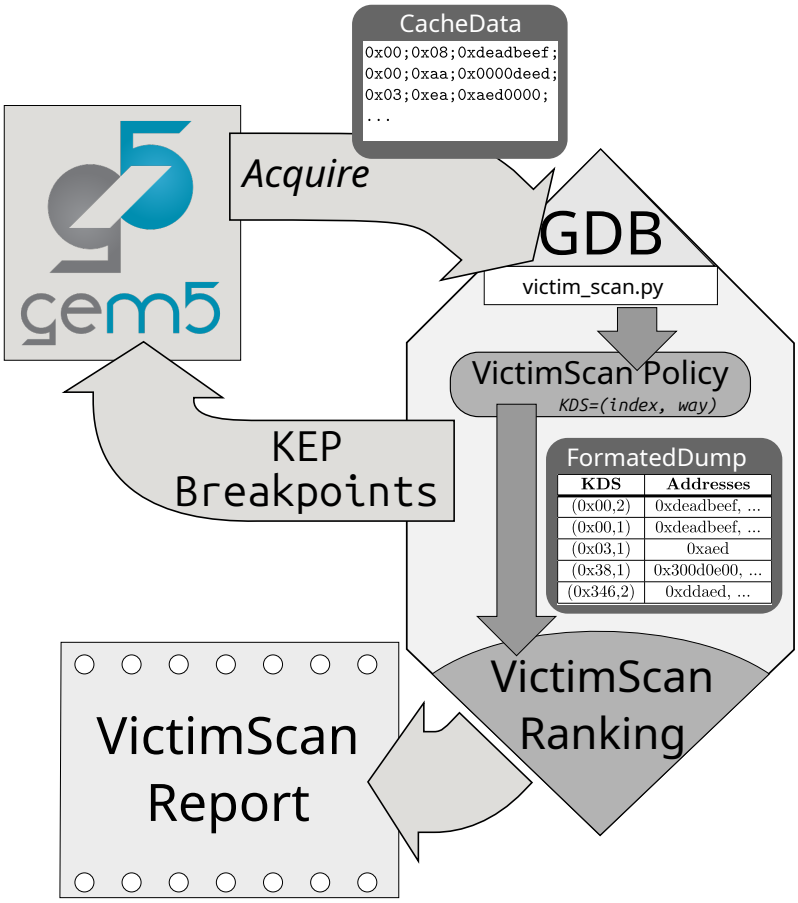

Figure 10: From cache Dump on *KEP*s to *VictimScan* ranking algorithm: *VictimScan policies* formats the cache dumps to highlight what states are observable in the cache using a *Prime+Probe* Attack.

raw cache data from *gem5*, these dumps are formatted to be exploitable by the ranking algorithm. These policies define how cache dumps taken on a *KEP*, are formatted to only present observable state to the ranking algorithm. The ranking algorithm is completely oblivious to the *KDS* definitions and ranks them with no regard to any relation between two *KDS*es besides equality.

As mentioned in section 3.3.2, it tries to rank the set of *KDS*es. For each class of *KEP*, the highest ranked *KDS* can be used to detect it reliably.

3.5.1 Key Detectable States model

Let $\mathbb{D}$ be the cache dump set. An element d of the cache dump set $\mathbb{D}$ which represents the state of a specific way w from an cache set with index i occupied by a line which corresponds to the address a is a 3-tuple as follows:

$$d = (i, w, a) \in \mathbb{D} \tag{2}$$

Each dump produced, D_u, is a set of $d \in \mathbb{D}$ which we call $\mathbb{U}$, the set of cache dumps, such that $D_u \in \mathbb{U}$. Therefore:

$$\forall D_u \in \mathbb{U}, D_u = \{d_1, d_2, d_3, ..., d_n\} \text{ with } (d_1, d_2, d_3, ..., d_n) \in \mathbb{D}^n \tag{3}$$

A *VictimScan policy* x is thus a function f_x of $\mathbb{U}$ to a set of elements from a simpler set called $\mathbb{K}_x$, the key detectable state set.

$$f_x : D_u \in \mathbb{U} \mapsto \{k_1, k_2, ..., k_n\} \text{ with } (k_1, k_2, ..., k_m) \in \mathbb{K}_x^m \tag{4}$$

Each element of $k \in \mathbb{K}_x$ can be distinguished using an attack $A_k \in \mathbb{A}_x$, with $\mathbb{A}_x$ being the set of attacks that can be configured to detect an element of $\mathbb{K}_x$. This attack A_k produces traces along the execution of the victim, influenced by the shared cache state. For a given point of execution p, we can define the result of the attack:

$$A_k : p \mapsto \mathbb{T}_A \tag{5}$$

$\mathbb{T}_A$ is the A attack output space, such as $A_k(p) \in \mathbb{T}_A$ is the output of the attack for a point p. An attack trace is therefore a list of execution points $\{p_1, p_2, ...\}$, and attack result $\{A_k(p_1), A_k(p_2), ...\}$. In that regard, key execution points ($p_{\spadesuit 1}$) are specific points in the execution that can be organized into classes that the attacker wants to distinguish using the output of the attack ($A_k(p_{\spadesuit 1})$).

KDS Property: Given two *KEP*s, $p_\spadesuit$ from KEP class ♠ and $p_\heartsuit$ from KEP class ♥ that each produced a dump, $D_{u\spadesuit}$ and $D_{u\heartsuit}$, $A_k \in \mathbb{A}_x$ is equivalent to the following:

$$\forall k' \in \mathbb{K}_x, k' \in f_x(D_{u\spadesuit}) \text{ and } k' \notin f_x(D_{u\heartsuit}) \implies A_{k'}(p_\spadesuit) \neq A_{k'}(p_\heartsuit). \tag{6}$$

if the property 6 is true for an attack A_k, and therefore $A_k \in \mathbb{A}_x$, it means that it can be used to detect *KEP*s using *KDS*es k from the image of their dump through the policy x.

In this configuration, the *VictimScan* ranking proposes a set of attack configuration (A_k), one for each *KEP class* which can be used in tandem to detect and distinguish *KEP*s. To represent our cache timing attacks, we define the following *VictimScan policies*.

3.5.2 VictimScan policy: 1hit

1hit is the simplest *VictimScan policy* that we used in figure 8 and figure 9. With this policy, *KDS*es are only made of non-empty cache indices (0x1, 0x23, ...) with no regard for the number of occupied ways or set occupancy. Their associated $f_{1\text{hit}}$ function is as follows:

$$\begin{aligned} &f_{1\text{hit}} : D_u \in \mathbb{U} \mapsto \{k, ...\} \\ &f_{1\text{hit}}(D_u) = \{(i) | \exists (w, a), (i, w, a) \in D_u\} \end{aligned} \tag{7}$$

The attacks $A_{(i)}^{1\text{hit}}$ for this policy are attacks which can distinguish between *hit* and *miss* for a specific set with index i.

3.5.3 VictimScan policy: nhit

nhit is the second *VictimScan policy* which takes into account set occupancy. For each cache dump, the *KDS*es, that it produces are composed of: a cache index and the number of occupied ways for this index ((0x1,1),(0x23,4), ...). Given $\mathbb{O}_i(D_u) = \{w | \exists a, (i, w, a) \in D_u\}$, their associated f_{nhit} function is as follows:

$$\begin{aligned} &f_{\text{nhit}} : D_u \in \mathbb{U} \mapsto \{k, ...\} \\ &f_{\text{nhit}}(D_u) = \{(i, card(\mathbb{O}_i(D_u))) | card(\mathbb{O}_i(D_u)) > 1\} \end{aligned} \tag{8}$$

The attacks $A_{(i,o)}^{\text{nhit}}$ for this policy are attacks which can distinguish between different occupancies o (the number of ways filled) for a set of index i.

3.5.4 VictimScan policy: nhit_inclusive

This is a derived *VictimScan* policy from the *nhit* policy. For each cache dump, its *KDS*es have the same definition as *nhit*. However, for each *nhit KDS*es, additional *KDS*es are added for included occupancies: For (0x23,4), (0x23,3), (0x23,2), and(0x23,1) are also emitted. Their associated $f_{\text{nhit\_inclusive}}$ function is as follows:

$$\begin{aligned} &f_{\text{nhit\_inclusive}} : D_u \in \mathbb{U} \mapsto \{k, ...\} \\ &f_{\text{nhit\_inclusive}}(D_u) = \{(i, o_{th}) | (i, w) \in f_{\text{nhit}}(D_u), o_{th} \in [1, w]\} \end{aligned} \tag{9}$$

The attacks $A^{\text{nhit\_inclusive}}_{(i,o_{th})}$ can be seen as a variation on $A^{\text{nhit}}_{(i,o)}$. This means that an attack $A^{\text{nhit\_inclusive}}_{(i,o_{th})}$ which has a number of filled ways for set index i of at least an occupancy threshold o_{th}, can be defined using a sum of $A^{\text{nhit}}_{(i,w)}$, with assoc the cache associativity:

$$A^{\text{nhit\_inclusive}}_{(i,o_{th})} = \sum_{w=o_{th}}^{\text{assoc}-1} A^{\text{nhit}}_{(i,w)} \tag{10}$$

3.5.5 Additional consideration for secure line labeling

VictimScan policies can also be configured to exclude unsecure lines when monitoring a *secure OS* operation. In that case, they produce a new *noise KEP* dump entry (that we noted ⚄). This is called *REJECT_UNSECURE*. This means that for each dump produced on *KEP*, the produced *KDS*es are separated between the real *KEP* (♠, ♥, ...) contribution (secure cache lines), and the noise (⚄) contribution (unsecure cache lines). This setting improves *VictimScan KDS* detection performance by depreciating *KDS* that are linked with unsecure access.

4 Vulnerability model for AutoLock

To attack a system with an AutoLock cache, we must select an attack capable of detecting changes in cache states despite the presence of AutoLock. Given our focus on an application running in a secure enclave, we have opted for *Prime+Probe* attacks. However, since AutoLock hinders the effectiveness of *Prime+Probe* attacks [GRLZ+17], we need to thoroughly examine the cache states that can still be detected by such attacks to determine the appropriate *KDS* definition. Indeed, with the correct attack and *VictimScan policy*, we can apply the methodology described in section 3 to AutoLock.

4.1 *Prime+Probe* model and implementation

For *Prime+Probe*: the attacker targets a specific cache index which corresponds to specific information about the cryptographic process. The attack proceeds as follows:

1 **Allocating**: We allocate our prime set. A prime set is composed of data for which the address has been chosen to have the same cache index as the victim address. There is an entry in the prime set for each associative way in the cache. (i.e. if the cache is 16-way associative, we need 16-entries prime set). If there are multiple victim addresses, each needs its own prime set.

2 **Priming** We access our prime set for the victim address we want to attack. This way our prime set will fill each possible alias for this victim address in the cache. Thus the victim address will be forced to evict one of the entries to be cached.

3 **Probing** Measuring access time to all the lines in our prime set to check if one was evicted by our victim. If an entry is evicted, probing it causes a *miss* resulting in a longer access time. On the contrary, if an entry is still in the cache, it is a cache *hit*.

Since **Probing** also fills the cache with the prime set like **Priming** would do. We do not need to prime after probing as long as we are only doing that. We need to be especially careful with our *Prime+Probe* implementation due to AutoLock. Therefore, we closely studied the implementation mentioned in [TOS10] and [LYG+15].

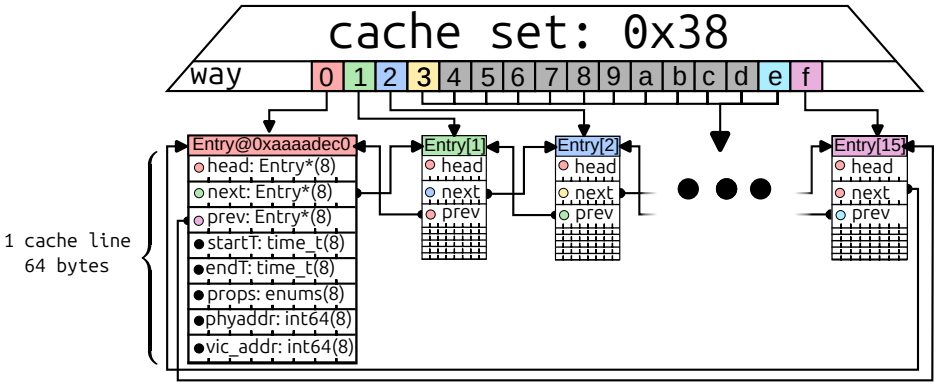

Figure 11: Our prime set uses a doubly-linked list. Its elements are allocated in such a way that they have all the same cache index (0x38). If enough are allocated they fill all the possible ways for their index.

This leads us to use a doubly-linked list data structure for our prime set (see figure 11). From each entry, we can automatically go to the previous or the next. With that, we can go through the set in any direction. We can use the *head* entry as our signal that we have reached the end. We can also easily swap entries to randomize the prime-set order. With this structure, we can probe and prime the set as we are going through it. After the first priming, we only probe the set, each probing serving as priming for the following one. Each entry of the prime set fully uses its line of cache to store all the necessary properties. Each result point is stored in a single cache line, to ensure minimum noise.

4.2 Prime set self-eviction

Already mention in [LYG+15], under the name *thrashing*, self-eviction happens when probing a prime-set entry evicts another prime-set entry. It can cause an entry to be wrongfully considered as been evicted by the victim. Figure 12 presents a situation in which *Prime+Probe* causes self-eviction. After priming, a victim evicts some element from the prime set (0 and 1). The figure then presents how the different ways of probing interact with self-eviction:

- **Forward**: Probing is always done in the same direction. If a *miss* is encountered, the following element will be bumped out of the cache due to self-eviction, and therefore all the following entries will be *misses*. On figure 12, the V-lines from the victim are not evicted by the prime set because they were accessed the most recently. Instead, the prime set evicts all its lines until it reaches the end to finally evict the V-lines.

- **Reverse** We change direction each time we finish probing. Thus elements, are never bumped and we can observe the sensibility of each element. On figure 12, no additional entries are evicted because the entry accessed is always the next one that would be evicted. And thus the V-lines are accessed last.

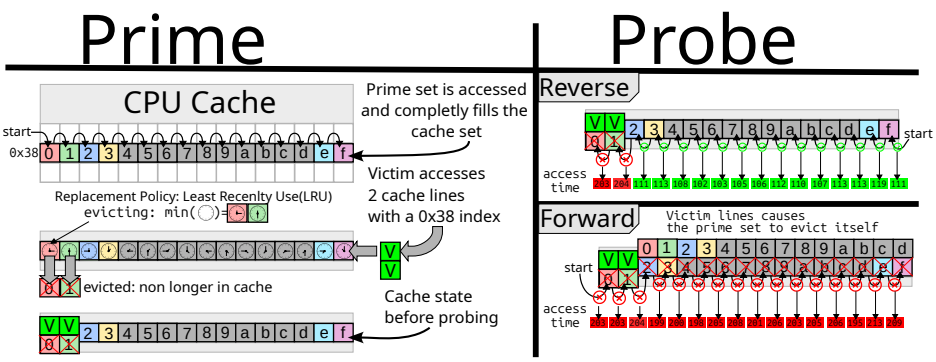

Figure 12: How direction of probing control self-eviction: after a victim accessed two lines, different probe directions produce different results.

Depending on the victim we want to observe, we can choose between the two directions of probing.

4.3 Link between *VictimScan* policies and *Prime+Probe* direction

Since the direction of probing reacts differently to the victim evicting elements of the prime set, they are represented by different *VictimScan policies*. *Prime+Probe* is configured using a prime set index i, which corresponds to a *1hit KDS* (k_i) or the first element of *nhit KDS* ($k_{(i,o)}$). Indeed, the output of *Prime+Probe* for each execution point p is the access time for each element of the prime set. Given $assoc$ the last-level cache associativity, we have:

$$A_{k_i}^{\text{Prime+Probe}}(p) = \underbrace{\{t_0, t_1, .., t_{\text{assoc}-1}\}}_{\text{assoc}} \tag{11}$$

Making abstraction of noise, we can propose a model for these expected timing results depending on the probing direction.

When probing forward, due to self-eviction, all the access timing for the prime set will have the same value, either t_{hit} or t_{miss}. Thus *Prime+Probe* forward can output only two possible value T_{miss} or T_{hit}

$$T_{miss} = \underbrace{\{t_{miss}, t_{miss}, .., t_{miss}\}}_{\text{assoc}} \qquad T_{hit} = \underbrace{\{t_{hit}, t_{hit}, .., t_{hit}\}}_{\text{assoc}} \tag{12}$$

This means that when probing forward the *Prime+Probe* attack can only reliably detect between a set being empty and being filled with one or more entries. This behavior links the *Prime+Probe* attack to the *1hit VictimScan policy*. In that context, *KDS*es are made of only a single index. And if we take two points of execution p_1 and p_2 whose dumps only differ by a single cache line in an otherwise empty cache set with index i. We have $k_i \in f_{1\text{hit}}(\text{Dump}_{p_1})$ and $k_i \notin f_{1\text{hit}}(\text{Dump}_{p_2})$ and our attack $A_{k_i}^{\text{Prime+Probe-forward}}$ produce the following results:

$$\begin{aligned} A_{k_i}^{\text{Prime+Probe-forward}}(p_1) &= T_{miss} \\ A_{k_i}^{\text{Prime+Probe-forward}}(p_2) &= T_{hit} \end{aligned} \tag{13}$$

Therefore, we have $A_k^{\text{Prime+Probe-forward}} \in \mathbb{A}_{1\text{hit}}$. This property is still valid when using the sum of timing over the prime set. In that, case we have $\sum T_{miss} = \text{assoc} \times t_{miss}$ and $\sum T_{hit} = \text{assoc} \times t_{hit}$. Therefore, we can plot only the sum of the timing without losing information.

On the other hand, for *Prime+Probe* in reverse, timing values can differ between entries in the set. Each entry can be a *hit* or a *miss*. However, due to the LRU (Least Recently

Used) cache replacement policy and the direction of probing, the victim program evicts elements of the prime set in order, from the least recently probed to the most recently probed. This results in all entries after the first *miss* being *misses* because the prime set is evicted from the extremity where the last probe started. Consequently, the number of prime entries evicted is directly linked to the number of occupied cache ways o by the victim for their associated index. Thus we can define:

$$T_{\textit{hit-miss}}(o) = \{\underbrace{t_{hit}, .., t_{hit}}_{\text{assoc}-o}, \underbrace{t_{miss}, .., t_{miss}}_{o}\} \tag{14}$$

For $o \in [0, \text{assoc} - 1[$, $T_{\textit{hit-miss}}(o)$ represents all possible outputs for the *Prime+Probe* reverse attack ($A_{k_i}^{\text{Prime+Probe-reverse}}$). Each of these outputs is linked with a number of occupied way o for the cache index i which was used to allocate the prime set. The attack has therefore a different output for each occupancy of the cache set. This behavior links the *Prime+Probe* reverse attack to the *nhit VictimScan policy*. In that context, *KDS* made of index i and occupancy o correspond to the attack $A_{k_i}^{\text{Prime+Probe-reverse}}$ outputting $T_{\textit{hit-miss}}(o)$.

Given two execution points, p_1 and p_2, whose dumps differ by only a single cache line in cache set i. In p_2, this cache line occupies an additional way o, assuming that all ways from 0 to $o-1$ are already filled. We have: $k_{(i,o)} \in f_{\text{nhit}}(\text{Dump}(p_1))$ and $k_{(i,o)} \notin f_{\text{nhit}}(\text{Dump}(p_2))$ and our attack $A_{k_{(i,o)}}^{\text{Prime+Probe-reverse}} = A_{k_i}^{\text{Prime+Probe-reverse}}$ produces the following results:

$$\begin{aligned} A_{k_{(i,o)}}^{\text{Prime+Probe-reverse}}(p_1) &= A_{k_i}^{\text{Prime+Probe-reverse}}(p_1) = T_{\textit{hit-miss}}(o) \\ A_{k_{(i,o)}}^{\text{Prime+Probe-reverse}}(p_2) &= A_{k_i}^{\text{Prime+Probe-reverse}}(p_2) = T_{\textit{hit-miss}}(o-1) \end{aligned} \tag{15}$$

Therefore, we have $A_k^{\text{Prime+Probe-reverse}} \in \mathbb{A}_{\text{nhit}}$. This property is still valid when using the sum of timing over the prime set. In that, case we have:

$$\begin{aligned} \sum T_{\textit{hit-miss}}(o) &= o \times t_{miss} + (\text{assoc} - o) \times t_{hit} \\ &= (t_{miss} - t_{hit}) \times o + (\text{assoc} \times t_{hit}) \end{aligned} \tag{16}$$

Therefore, if we use as an attack trace *the sum of the prime set timing values*, there will be a distinct trace point value for each $T_{\textit{hit-miss}}(o)$.

The *nhit_inclusive* policy, is similar and shares the same *KDS* definitions than the *nhit* policy. Indeed we have $\mathbb{A}_{nhit} \subset \mathbb{A}_{nhit\_inclusive}$. In that regard, *nhit_inclusive* mostly differs on what attack from $\mathbb{A}_{nhit\_inclusive}$ is searching for. Whereas attacks from $\mathbb{A}_{nhit}$ search for exact $T_{\textit{hit-miss}}(o)$ values associated with the *KDS* (i, o), attacks from $\mathbb{A}_{nhit\_inclusive}$ search for $T_{\textit{hit-miss}}(w)$ higher with w higher than a certain o_{th} associated with the *KDS* (i, o_{th}). For *Prime+Probe* reverse, this can be computed as a $\sum T_{\textit{hit-miss}}(o_{th})$ threshold value for the acquired $\sum T_{\textit{hit-miss}}(w)$. In that case a $\sum T_{\textit{hit-miss}}(w) \geq \sum T_{\textit{hit-miss}}(o_{th})$ is our signal for the *KDS* (i, o_{th}).

We sum up the link between, policies, and *Prime+Probe* direction in the table 3. It also contains the trace points we use and how it is linked with the *KDS* we want to detect. In this table, we also give the signal we are searching for to detect a *KDS*, although in real measures, we would have to account for the noise.

4.4 *Prime+Probe* interaction with AutoLock

As explained in section 2.4, AutoLock prevents the eviction of lines already present in L1. This mechanism will prevent some elements of the prime set from being *hit* because they can never evict the victim cache lines. This means that certain attack output values become less likely because some element of the prime set entry may be forced to be *misses*.

Table 3: Correspondence between attack policy and *Prime+Probe*

Policy	***KDS***	**Attack**	**Output**	**Trace (Σ)**	**Signal**
1hit	(i)	*Prime+Probe* forward	T_{hit} or T_{miss}	$\sum T_{hit}$ or $\sum T_{miss}$	$\sum T = \sum T_{miss}$
nhit	(i, o)	*Prime+Probe* reverse	$T_{hit\text{-}miss}(w)$ $w \in [0, \text{assoc}[$	$\sum T_{hit\text{-}miss}(w)$ $w \in [0, \text{assoc}[$	$\sum T_{hit\text{-}miss}(w) = \sum T_{hit\text{-}miss}(o)$
nhit_inclusive	(i, o_{th})	*Prime+Probe* reverse	$T_{hit\text{-}miss}(w)$ $w \in [0, \text{assoc}[$	$\sum T_{hit\text{-}miss}(w)$ $w \in [0, \text{assoc}[$	$\sum T_{hit\text{-}miss}(w) \geq \sum T_{hit\text{-}miss}(o_{th})$

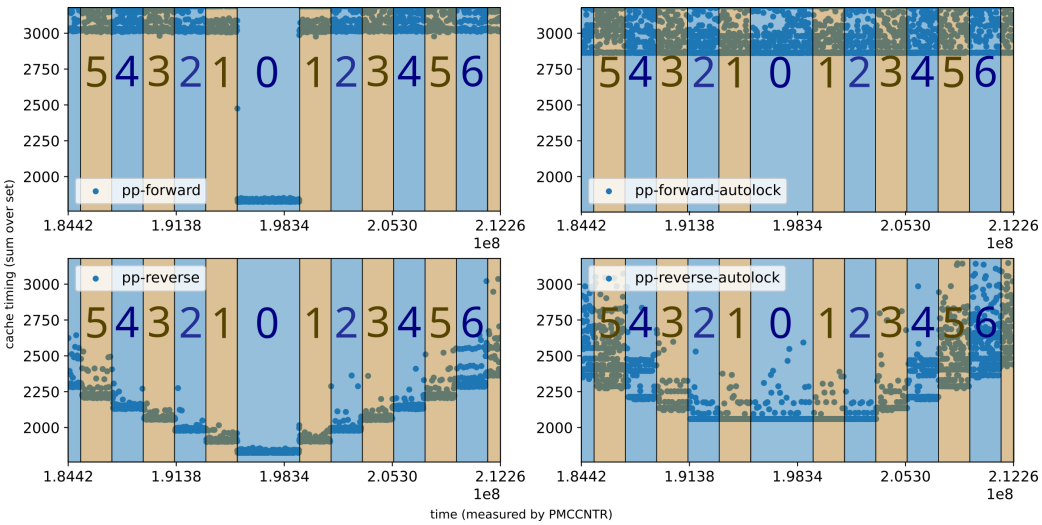

Figure 13: How *Prime+Probe* interacts with AutoLock: above are *Prime+Probe* forward and below are *Prime+Probe* reverse. On the left, without AutoLock, and on the right, with AutoLock. The victim uses cache occupancy, indicated as colored rectangles, to send a stair signal clearly visible on *pp-reverse*.

We propose the figure 13 to visualize, the different output values for *Prime+Probe* forward and reverse. On the bottom left, which corresponds to *Prime+Probe* reverse traces, each stair level corresponds to a $\sum T_{hit\text{-}miss}(o)$ (with o from 0 to 7) associated with a cache occupancy o. The same victim behavior produces the trace on the top left when using *Prime+Probe* forward, with only two value $\sum T_{hit}$, the lowest, and $\sum T_{miss}$, the highest.

When we enable AutoLock, we have the two plots on the right of figure 13. On them, because of AutoLock, the T_{hit} output and $T_{hit\text{-}miss}(o)$ for $o \leq 2$ are no longer distinguishable. This is caused by the 2-way cache L1. Consequently, AutoLock hides some T_{hit} outputs from *Prime+Probe* forward. For this reason, to still be able to use *Prime+Probe* despite AutoLock, we have to use *Prime+Probe* reverse.

4.5 Interaction with pseudo-LRU implementation

Real hardware platforms implement variations of the LRU cache replacement policy, called pseudo-LRU. They approximate LRU behavior without having to rely on exact time stamps to choose the Least Recently Used cache set entry. These variations still statistically behave like LRU. However, they do impact *Prime+Probe* by unexpectedly evicting prime set entries. This is visible in outputs, with entries being swapped and no longer being split

between *miss* and *hit* (like in equation 14).

$$\{\underbrace{t_{hit}, .., t_{hit}}_{\text{assoc}-o}, \underbrace{t_{miss}, .., t_{miss}}_{o}\} \rightarrow \{t_{hit}, t_{miss}, t_{miss}, t_{hit}, t_{miss}, t_{hit}, ..., t_{hit}\}$$

This effect can be mitigated by using the sum of individual traces because it does not change with set entries permutations.

On figure 13, we see that $\sum T_{hit\text{-}miss}(o)$ acts as a minimum threshold for real measures. This minimum threshold is not affected by pseudo-LRU cache replacement policies. However, pseudo replacement policy can also sometimes evict more entries than expected, due to entry permutation causing self-eviction. In that case, the measured timing is higher. This means that, in this situation, the lower threshold is statistically more accurate than searching for an exact value (w.r.t noise). For this reason, we prefer *nhit_inclusive*, *VictimScan* policy, which uses a minimum threshold as a signal.

With these and our *gem5* model for ARM-secure platform, including the AutoLock cache replacement policy, we are able to analyze the security of sensitive applications, in a way that could be then used on real hardware.

5 Attacking mbedTLS on OPTEE

To sign a hash, OP-TEE uses the function `rsa_exptmod` in Libtomcrypt. To sign, `rsa_exptmod` will use the private exponent which should be kept secret. A key can be provided to `rsa_exptmod` without specifying RSA-CRT parameters. In that case `rsa_exptmod` uses a simple *bignum* exponentiation provided by *libmbedtls* after blinding the base. In this context, `rsa_exptmod` computes:

$$\text{sign} = \text{hash}^{D} mod\ N \tag{17}$$

With D the private exponent, and N the modulus (with $N = p \times q$, p and q are two primes).

5.1 RSA with OP-TEE: mbedTLS exponentiation

The exponentiation function in *mbedtls*, `mbedtls_mpi_exp_mod`, uses the sliding-window to compute the bignum exponentiation [MvOV01]. In our case, the exponent is the private exponent (D).

This algorithm exploits a window (wbits) to accumulate multiple bits of the key (n_i) together and then uses them to do the exponentiation using a pre-computed value (A^{wbits}). When a leading 1 is found, the following *wsize* bits are accumulated in *wbits*. The associated precomputed value is then multiplied with X: $X \leftarrow X \times A^{wbits}$. Zeros outside of the accumulation phase are skipped by just squaring X. This algorithm implementation (taken from OP-TEE 3.21) is presented in figure 14. The following function is from the implementation in figure 14:

```
mpi_select( &WW, W, (size_t) 1 << wsize, wbits );
```

This is the *multiplier obfuscation* mentioned by [KSHZ23]. It ensures that accessing the precomputed window using the accumulated window bits is time-constant. However, this algorithm is known to leak some information about the key. Refs. [UH23] and [BBG+17] suggest to detect *Montgomery multiplication call* and categorizing them: *square*([𝕊]) or *multiply* ([𝕄]). These are the KEPs that we will use to extract the partial keys, which can be then used to reconstruct the key as mentioned by [UH23]. The series of *square*([𝕊]) and *multiply*([𝕄]) can then be used to extract a partial key (figure 15).

```
1  int mbedtls_mpi_exp_mod( mbedtls_mpi *X, const mbedtls_mpi *A,
2          const mbedtls_mpi *E, const mbedtls_mpi *N ){
3    /* Preparing W :
4    W[I]= X^I */
5    state=1;wsize=6 nbits=0;
6    int i=Skip_leading_zeros(E,X);
7    while( 1 ){
8      if(is_Finished(i))
9          break;
10     ei = (E[i]) & 1;//E[i] is i-th bit of E
11     if( ei == 0 && state == 1 ) {
12         /*X=X * X*/
13 [S]     mbedtls_mpi_montmul( X, X, N, mm, &T );                    Square
14         continue;
15     }
16     state = 2; nbits++;
17     wbits |= ( ei << ( wsize - nbits ) );//
18     //
19     if( nbits == wsize ){                                          Window section
20       /* X = X^wsize R^-1 mod N*/
21       for( i = 0; i < wsize; i++ )//
22         mbedtls_mpi_montmul( X, X, N, mm, &T );                    Square
23       /* X = X * W[wbits] R^-1 mod N */
24       mpi_select( &WW, W, (size_t) 1 << wsize, wbits );
25 [M]   mbedtls_mpi_montmul(X,&WW,N,mm,&T);                         Multiply
26       state=1; nbits = 0; wbits = 0;//
27     }
28   }
29   /* process the remaining bits */
30   return( ret );
31 }
```

Figure 14: Implementation of the sliding window algorithm. We call the section in red the window section. [S] and [M] corresponds to the two KEPs used to analyze this function.

5.2 Attacker Model

Our attacker model assumes kernel-level privilege. This attacker needs kernel privilege for two reasons:

- To use the PMCCNTR counter for timing measurement, which requires kernel privilege.
- To deduce physical addresses from virtual addresses.

The victim is a *trusted application*(TA) running within the *secure world*. We make no assumptions about core affinity but we can control the Linux applications' affinity using `taskset` (both client and attack). Other cache-related parameters are deduced through simulation.

5.3 Experimental Platform

To represent, a more practical scenario, we chose the RockPi4 [rad20] board based on *RK3399* [Roc21] SoC. The *RK3399* has been utilized in consumer devices like Chromebooks

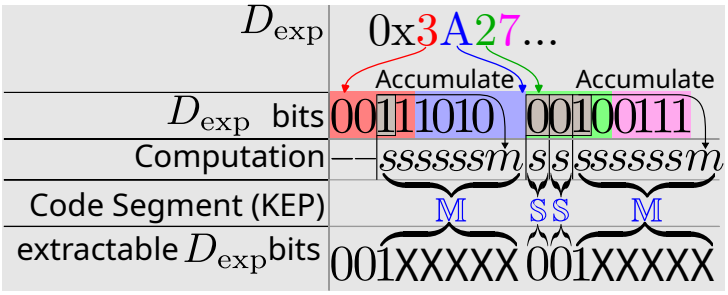

Figure 15: S are *square* operation and M are *multiply* operation.

and tablets, with its variants continuing to be used today. OP-TEE is compatible with *RK3399* and proposes a build configuration to build an OP-TEE-compatible workload. The RockPi4 variant we use is the RockPi4 C plus which uses the *RK3399*-T which are identical except for CPU clock and voltages.

5.3.1 *RK3399* Software Environment

The key practical issue in reproducing the software environment lies in the extraction of the *RK3399 bootrom*. The *RK3399* contains a simple *bootrom* integrated in the SoC and map at address `0xffff0000`. This integrated *bootrom* can load, the next booting step from multiple sources (SPI,eMMC, SD-Card,etc.). Using the integrated eFUSE, it is possible to force the loaded *bootrom* to be signed with a key contained in the same fuse. We used a standard boot scenario for the *RK3399*, with all the bootloader steps loaded in the same SD-Card.

This integrated *bootrom* was extracted using a modified *U-boot TPL* [Eng23] from real hardware, and the *eFUSE* device is modeled with our *PyDevices* method (pl. see sec. 2.3). More detailed description of the standard embedded Linux software stack with *trusted firmware* and *optee* can be found in Annexe A.

5.3.2 *RK3399* Hardware Environment

The key practical issues are to use the exact same security features in simulation, and modeling peripheral devices. The standard components can be modeled using already existing *SimObject* (RAM, CPU, GIC, etc.) in *gem5*.

Peripheral Devices: Thanks to our *PyDevices* method, and the rather complete TRM [Roc21] manual for the *RK3399*, it is possible to model the peripheral devices. This is essential to boot our Rockpi4 workload in *gem5*. We use the *RK3399* memory map described in the TRM, to instantiate all the memory-mapped devices. We then use their register definitions when they are described in the TRM (cf. figure 16). As device registers are presented in tables, we can automatically generate dummy devices that report to *gdb* any register modification using our *gdb* API.

When this definition is not available, we can use a dummy device to fill the memory space and notifies *gdb* when it is accessed. This workload is made of two elements: our SD-card image that we created and the integrated *bootrom* included in the *RK3399*. We then designed new config files, to start reproducing the *RK3399* platform in *gem5* by integrating both already existing *SimObject*s (RAM, CPU, GIC, etc.) and new *PyDevices*-implemented objects(UART, fuse, etc.).

These incomplete descriptions integrate perfectly in a reverse-engineering software like Ghidra[Roh19]. In Ghidra, our *gdb* messages based on dummy device implementation guide us on device usage throughout the undocumented *bootrom*. We can then only implement

Name	Offset	Size	Reset Value	Description
EFUSE_CTRL	0x0000	W	0x00000000	efuse control register
EFUSE_DOUT	0x0004	W	0x00000000	efuse data out register
EFUSE_RF	0x0008	W	0x00000000	efuse redundancy bit used indicator register
EFUSE_JTAG_PASS	0x0010	W	0x0cf7680a	Jtag password
EFUSE_STROBE _FINISH_CTRL	0x0014	W	0x00009003	efuse strobe finish control register

*Notes: Size: **B**- Byte (8 bits) access, **HW**- Half WORD (16 bits) access, **W**- WORD (32 bits) access*

Figure 16: Extract from the *RK3399* TRM: register description for the efuse

the device and functionalities we need based on *bootrom* disassembly and TRM manual. This is how, with *PyDevices*, we are able to reverse engineer the *RK3399 bootrom* and implement only the necessary devices used in our scenarios.

Security Features: The *RK3399* has multiple devices reserved for a *secure OS* running in TrustZone. As mentioned before it contains, a secure eFUSE (efuse1). They are used by the integrated *bootrom* to verify the first boot-loaded stage, in our scenario from the SD-Card. The *RK3399* also features a programmable access controller that can protect memory and devices to ensure that they are only accessible from the secure world (EL1S or EL3). It is also configured by OP-TEE to create a secure 32MB partition in the DRAM memory. OP-TEE secure OS and TAs reside in this memory region which can not be accessed from the Rich OS.

Considering, the *RK3399-T* we decided to run our attack and victim in the A72 core complex. Each program is running in a different CPU and sharing an AutoLock-enabled ([GRLZ+17]) 1MB 16-way set-associative L2 cache with 64-byte cache lines.

5.4 Experimental Setup

Our attack scenarios have to integrate with *VictimScan* and still be able to run on the real platform. Indeed, we want the same script and disk image to describe an attack scenario which:

- Runs an attack on the RockPi4 without any intervention.
- Can be configured in the simulated environment through the *gdb* interface, changing attack and victim arguments and potentially disabling them.

To instrument and configure our attack scenario when it is run in the simulation, an additional dummy device is added to the simulated *RK3399* model. This device, absent from the real platform, is checked to enable `m5`: the *gem5* in-simulation command line utility. To it, we added a `m5_env` function to export environment variables to a shell environment in the simulation. Through `m5`, command-line arguments can be passed from *gdb* to the attack running in the simulation. `m5` also transfers the attack output directly to the host machine, outside the simulation. With this method, we can configure the simulation without modifying our disk image. If `m5` is not enabled, default parameters are used to configure the attack scenario and the attack traces are written to the disk.

5.5 Using *VictimScan* to search for weaknesses

With our instrumented scenario, we can use *VictimScan* (see section 3) to search for potential weaknesses that we can leverage for an attack against the RSA TA. To prepare for specific Pseudo-LRU in the *RK3399* about which we have no details, we use *nhit_inclusive* as our *VictimScan policy*.

5.5.1 Finding good *KEP*s against *AutoLock*

```
Score ranking:
>>>>M->max_hit:(('0x38', 1), 146)
(1):('0x38', 1)
    score:0.9944598337950139
    hit_count:146
    top_addr:
    1@146=S#0x300d0e00:data + 39160 in section .bss
    2@0=0x53a60e00:UKN
>>>>S->max_hit:(('0x346', 1), 66)
(1):('0x346', 1)
    score:0.4925373134328358
    hit_count:66
    top_addr:
    1@66=S#0x300cd180:data + 23672 in section .bss
    2@66=NoMMU;:maybe_tag_buf + 40 in section .text
    3@66=S#0x300dd180:__heap1_start + 41112 in section .heap1
```

Figure 17: First *VictimScan* report for the RSA TA with *KEP*s from figure 14, using the *nhit_inclusive policy*.

We use the [𝕊] and [𝕄] on figure 14 as our *KEP*s. They produce the following *VictimScan* result on figure 17. It proposes the following *nhit_inclusive KDS*es:

- for [𝕄]: $(0x38, 1)$ with score 0.994459.
- for [𝕊]: $(0x346, 1)$ with score 0.49253731.

From the report on figure 17, we see that the [𝕊]*KEP* can not be accurately detected using these *KEP*s: with a score of less than 0.5, [𝕊] can not be distinguished from [𝕄], (pl. see section 3.3.2) or is obscured by other sources of cache activity.

Despite the low score, we can use *attack monitoring* (pl.see sec. 3.4) to generate traces to have a better understanding of what the attacker sees. *Attack monitoring* uses our two *KDS* to configure the *Prime+Probe* reverse attack. We expect it to detect [𝕄], but not [𝕊]. The results of *attack monitoring* are in figure 18. It shows how the cache timings

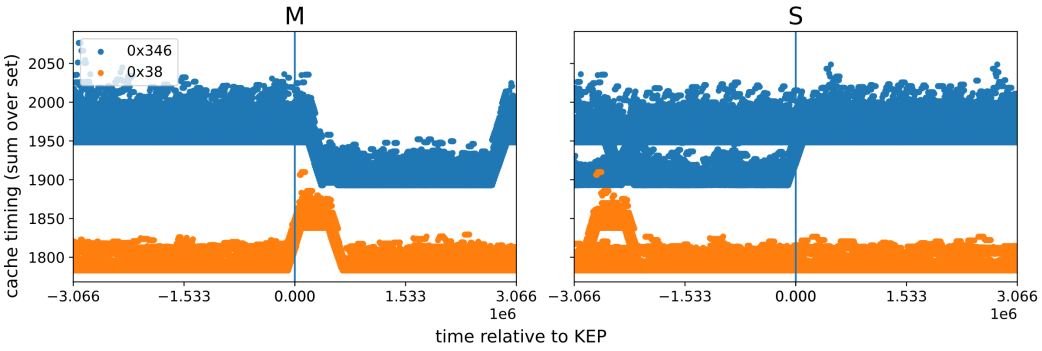

Figure 18: Zoomed in timing traces plotted relatively to *KEP*s. Two lines are used in order to distinguish between the two *KEP*s ([𝕊] and [𝕄]). We see a clear pattern around [𝕄] but not around [𝕊]. Thus we cannot detect [𝕊].

behave in the vicinity of each *KEP*. On figure 18, we see that the timing associated with

[𝕊]-*KDS* (0x346) does not stay low when leaving its associated section. It seems because of *AutoLock* an attacker cannot detect a single *squaring operation* and can only detect the start of a series of [𝕊]*KEP*s.

We propose to use different *KEP*s to better account for how the victim behaves, and thus have better scores if our assumptions are correct:

- We place the [𝕄] KEP as a code section around lines 24 to 25 of the function (see figure 14).
- We set up the [𝕊] KEP such that only the first squaring section in a row is registered.

With these *KEP*s, we can detect:

- When we enter the multiply phase and leave the multiply phase.
- When we are in a squaring phase.

```
Score ranking:
>>>>M->max_hit:(('0x38', 1), 148)
(1):('0x38', 1)
  score:1.0
  hit_count:148
  top_addr:
    1@148=S#0x300d0e00[S#0x300d0e00]:data + 39160 in section .bss
>>>>S->max_hit:(('0x346', 1), 66)
(1):('0x346', 3)
  score:1.0
  hit_count:66
  top_addr:
    1@66=NoMMU;[S#0x3008d180]:maybe_tag_buf + 40 in section .text
    2@66=S#0x300cd180[S#0x300cd180]:data + 23672 in section .bss
    3@66=S#0x300dd180[S#0x300dd180]:__heap1_start + 41112 in section .heap1
```

Figure 19: *VictimScan* report for the RSA TA with *KEP*s from figure 14 redefined, using the *nhit_inclusive policy*.

These improved *KEP*s produce the report on figure 19. It proposes the following *nhit_inclusive*, *KDS*es:

- for [𝕄]: $(0x38, 1)$ with score 1.0.
- for [𝕊]: $(0x346, 3)$ with score 1.0.

Both have a 1.0 score which guarantees that they can be detected and distinguished. However, with these points, we can only detect the end of windows. If we are out of a window section we must be in a squaring section of the exponentiation function. We can still try to reconstruct the [𝕊] information by using the fact that the multiply window contains seven ($wsize + 1$) (pl. see fig. 14) `mbedtls_mpi_montmul` which all take the same time as they are done with the same modulo. We call this *window measurement*: we compare the time difference between two of our [𝕄]-points, points that we detected using the [𝕄]-*KDS* and our [𝕊]-*KDS*. We know that among these time differences, there is at least one which contains the window section and no extra squaring phase ([𝕊]). We know that because the first operation cannot be a [𝕊] and is necessarily a window.

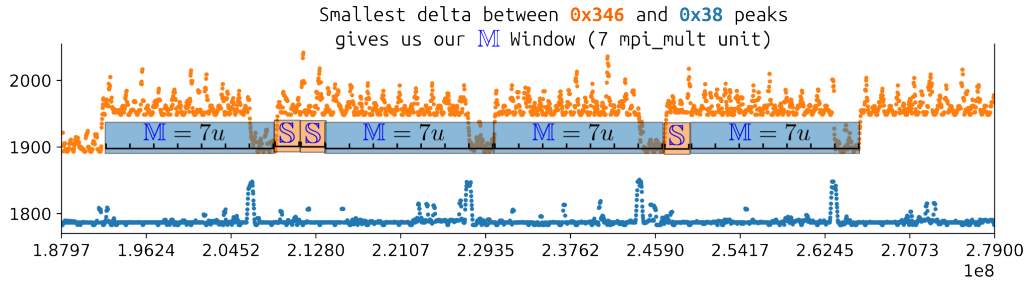

Figure 20: *Window measurement*: we can measure the time between peaks in `0x38` or `0x346`. With this measurement by comparing them with the minimum difference between 2 of these peaks, we can design a system of units to reconstruct the series of [𝕊]and [𝕄].

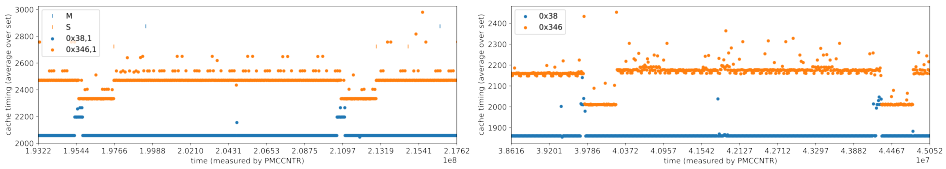

Figure 21: comparison between simulation and real hardware: centered around a similar pattern.

5.5.2 Attack monitoring and real hardware results

We ran the same attack scenario on the RockPi4 and on the simulated platform. This scenario is run multiple times on the real hardware to compare different runs. We observe that the real platform attack results are similar to the simulation in certain instances (see figure 21) and different in others(comparing top traces and bottom traces from figure 22) inside a single trace. When we compared different traces acquired from the real platform, we saw that different parts of the traces were identical to the simulation while the others were just noise. This difference is likely due to the simulation using perfect LRU while the real platform likely uses Tree-Pseudo LRU. We confirmed this assumption

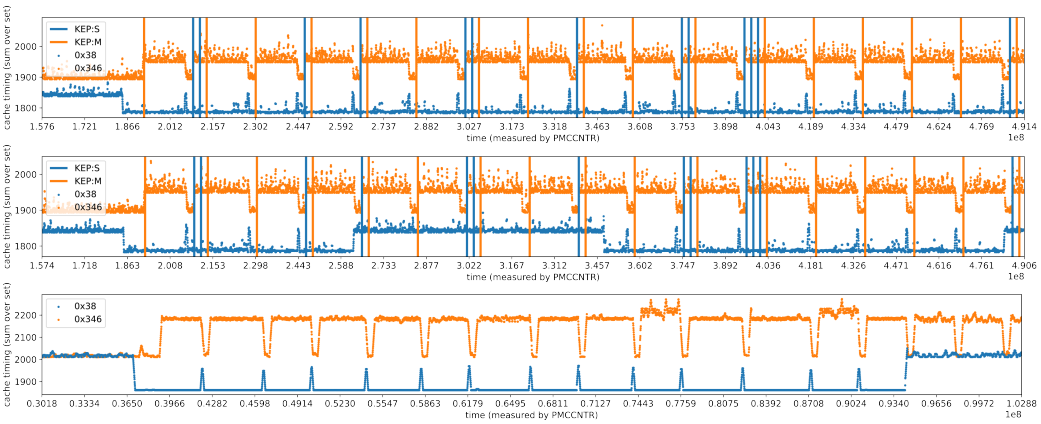

Figure 22: We compared traces between (from top to bottom): Simulation using LRU, Simulation using Tree-Pseudo LRU and the real platform. We can see that there is the same behavior during which a prime set gets "stuck" in an occupied state between simulated Tree LRU and the real platform.

by configuring our simulation to use *Tree-LRU* with *AutoLock* (middle trace on figure 22).

The transient "stuck" timings appear in both simulation with *Tree-LRU* traces (middle trace in figure 22) and real platform traces (bottom trace in figure 22). However, in detail, simulation *Tree-LRU* and real platform Pseudo-LRU still behave differently either because of randomness or small differences between model and reality. We can remedy this issue by cumulating multiple real traces to reproduce the information contained in simulation traces. However, if we choose to do that the short burst caused by set `0x38` which used to detect [𝕄] will likely be lost by the averaging. For this reason, we solely rely on the `0x346` set to recover the window start points that correspond to our [𝕄] *KEP*s. This is possible because the [𝕊] *KDS* was chosen to distinguish [𝕊] from [𝕄]. We call these points associated with the [𝕄] code segment: [𝕄]-points. With them, we can perform the window measurement on the real traces.

5.6 Simulation performances for RockChip platform

Table 4 shows the software versions used and the run-times for the different steps of the analysis process for our demo TA (figure 7) and the RSA TA (figure 14). We also included the boot time which is only done once to generate a checkpoint used for subsequent runs. Table 4 also contains real performances of our RSA TA scenario on the real platform.

Table 4: Simulation Configuration and Run-time. Times measured by *gem5*. When using *gdb*, simpler CPU models are used outside of the region of interest. We run our examples on a *Intel(R) Xeon(R) Gold 6128* with 256GB of DDR4. Runtimes are in second

Configurations	
gem5	version 21.2
Software Stack	optee-3.21.0 (based on Linux v6.2-rc3)
	U-boot : v2020.07-rc3
	ARM Trusted Firmware-A v2.7
Runtimes	
Simulation on *gem5*	
Boot (only needed once)	2360.58s
Demo TA	1212.52s
Demo TA + Attack	2359.49s
Step II: Demo TA *VictimScan* (*gdb*)	1809.89s
Step III: Demo TA Attack monitoring (*gdb*)	2680.65s
Step II: RSA TA *VictimScan* (*gdb*)	10603.99s
Step III: RSA TA Attack monitoring (*gdb*)	10006.69s
Real platform (*RK3399*-T on RockPi4)	
Attack+RSA TA	1.080240s
Export to SD-card	5.394986s

5.7 Extracting a key from real traces

After accumulating 50 traces, we fused them and then filtered them using a gaussian filter. To recover our [𝕄]-points, from this traces we used a peak detection algorithm, each

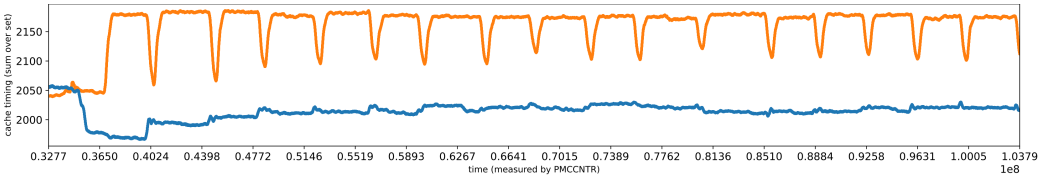

Figure 23: Accumulation of 50 real traces. They have been fused and filtered with a Gaussian filter

[𝕄]-points corresponding to a peak in the `0x346` filtered trace. We automatically tune this algorithm knowing roughly the number of peaks in a trace: For a 1024 bits-private key, as our `mbedtls_mpi_exp_mod` function uses a 6 bit window, they cannot be more than $1024/6 \cong 171$ peaks. We facilitate this process using the `0x38` to determine a region of interest when it has a lower value.

Using this algorithm on the traces on figure 23: we have the result on figure 24. From the peaks in this figure, we can retrieve the [𝕄]-points. With the [𝕄]-Points retrieved, we can now carry on *window measurement*:

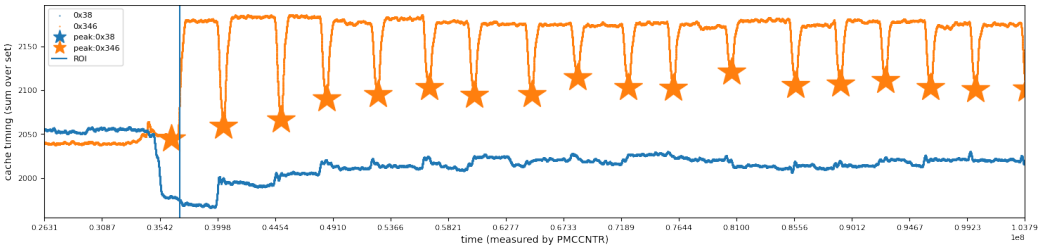

Figure 24: Peaks detected from figure 23

- We determine the time length of a single window using the smallest difference between [𝕄]-points.
- With this single window, we find the length of a single Montgomery multiplication (`mbedtls_mpi_montmul`).
- We then try filling each time difference between [𝕄]-points with one window and then as much multiplication([𝕊]) as needed.
- We also treat specifically the difference between the last [𝕄]-point and the end of the region-of-interest to find the trailing multiplication.
- For each multiplication, we count a "[𝕊]"(possibly none) that will then be followed by a [𝕄].
- This makes our [𝕊][𝕄]-series which we showed can be used to recover a partial key (figure 15). [𝕄] indicates 1 followed by 5 Xs (window_size − 1), [𝕊] each indicates single 0 and trailing [𝕊]s (at the end of the exponent) each indicates single a X (either a 1 or a 0)

As the *sliding window exponentiation* skipped zeros at the start of the exponent, we know that all missing bits are leading zeros. We also know that D, the private exponent, cannot be even, therefore the last bit is necessary a 1 ($ED = 1\ mod(p-1)(q-1)$ implies that D is necessary odd because E is odd.). We can overlay all this process on the traces which gives us the full figure: figure 25. In this figure, we can see the partially reconstructed key overlaid above its trace. We performed the same operation for other keys, and compared how many bits we recovered using our *window measurement* method (see table 5).

keys	[𝕊]	[𝕄]	total bits in [𝕊][𝕄]	known bits
key 1	145	146	1021	294
key 2	164	143	1022	310
key 3	134	148	1022	284

Table 5: Example of D reconstruction using [𝕊][𝕄]-series

With our partial key recovered, [UH23, HS09, MH20] and [KSHZ23] suggest that we could use *Branch and Prune* to go further and reconstruct the key.

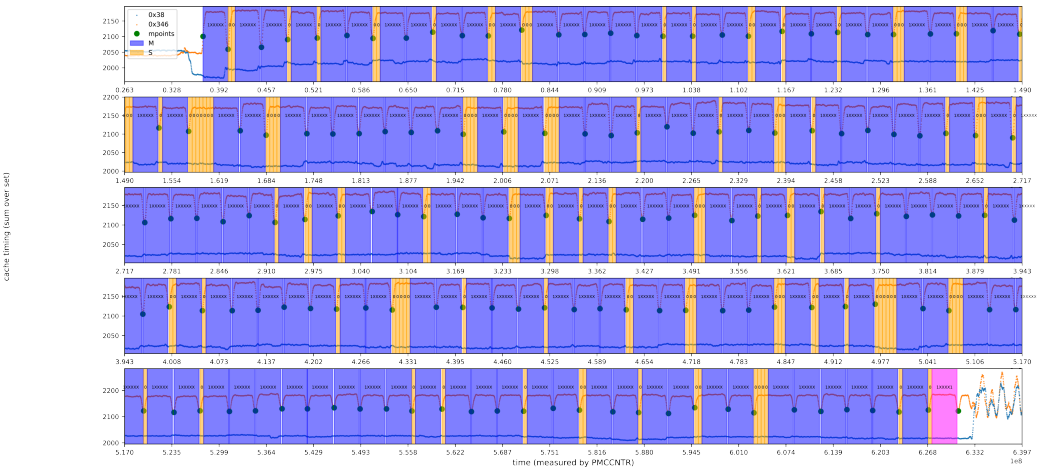

Figure 25: Attack on the real platform against the RSA TA: 1 and 0 are bits from the private key that we identified using the [𝕊][𝕄]-series, the X corresponds to bits that we do not know and that can be either a 1 or a 0

6 Conclusion & Future Work

AutoLock is a great feature in ARM processors, which with a simple trick enhances the performance and makes cache timing attacks considerably difficult. However, we show that it is still possible to attack TrustZone applications albeit with more complex attacks. Furthermore, we have made very few assumptions about the attacker. It can run from any core, and it requires kernel privilege only to use precise timers. In the future, it could be possible to implement a timer in a parallel thread and let go of these privileges. Thus *trusted application* designs need to be more cautious in the future.

We presented a vulnerability analysis method taking into account cache set occupancy and replacement policies. This is necessary to detect information leakage in presence of *AutoLock*. The existing cache analysis tools do not take into account such level of detail and thus are not suitable for bypassing *AutoLock*. We also introduce a new methodology of attack where a large part of the attack development happens inside the simulator, thanks to *gem5*. The key is to use the same unmodified binary in both cases and model the peripheral devices correctly in the simulator. This simulation platform can then be used for hypothesis testing about the real board. It is a real boon for reverse engineering. In the future, we can also devise counter-measures that can be tested on the simulator itself.

Last but not least, these simulation-based attacks really improve the platform-independent reputability of attacks which is often lacking within the hardware security community.

References

[And16] Android. https://source.android.com/docs/security/features/trusty/, 2016.

[Ass24] Buildroot Association. Buildroot making embedded linux easy, 2024.

[BBG+17] Daniel J Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. Sliding right into disaster: Left-to-right sliding windows leak. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 555–576. Springer, 2017.

[Bel05] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, page 46. Califor-nia, USA, 2005.

[CD16] Victor Costan and Srinivas Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016.

[Eng23] DENX Software Engineering. U-boot. https://www.denx.de/wiki/U-Boot, 2023. Accessed: 2023-16-01.

[fERtS] Toyohashi OPen Platform for Embedded Real-time Systems. Toppers tee. http://www.toppers.jp/en/safeg.html.

[GMWM16] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings 13*, pages 279–299. Springer, 2016.

[GRLZ+17] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. AutoLock: Why cache attacks on ARM are harder than you think. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1075–1091, Vancouver, BC, August 2017. USENIX Association.

[GSM15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive Last-Level caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912, Washington, D.C., August 2015. USENIX Association.

[GVR+23] Antoine Geimer, Mathéo Vergnolle, Frédéric Recoules, Lesly-Ann Daniel, Sébastien Bardin, and Clémentine Maurice. A systematic evaluation of automated tools for side-channel vulnerabilities detection in cryptographic libraries. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 1690–1704. ACM, 2023.

[HS09] Nadia Heninger and Hovav Shacham. Reconstructing rsa private keys from random key bits. In *Annual International Cryptology Conference*, pages 1–17. Springer, 2009.

[KHF+19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019.

[KHSZ21] Zili Kou, Wenjian He, Sharad Sinha, and Wei Zhang. Load-step: A precise trustzone execution control framework for exploring new side-channel attacks like flush+evict. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 979–984, 2021.

[KSHZ22] Zili Kou, Sharad Sinha, Wenjian He, and Wei Zhang. Attack directories on ARM big.little processors. In Tulika Mitra, Evangeline F. Y. Young, and Jinjun Xiong, editors, *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2022, San Diego, California, USA, 30 October 2022 - 3 November 2022*, pages 62:1–62:9. ACM, 2022.

[KSHZ23] Zili KOU, Sharad Sinha, Wenjian HE, and Wei ZHANG. Cache side-channel attacks and defenses of the sliding window algorithm in tees. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2023.

[lea21] GlobalPlatform leadership. Global platform. https://globalplatform.org/, 2021.

[LGS+16] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, Austin, TX, August 2016. USENIX Association.

[Lin23] Linaro. Trusted-firmware a. https://www.trustedfirmware.org/projects/tf-a/, 2023. Accessed: 2023-16-01.

[Lin24] Linaro. Mbed tls, 2024.

[LKS+20] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the*, EuroSys '20, New York, NY, USA, 2020. ACM.

[LPAA+20] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. The gem5 simulator: Version 20.0+, 2020.

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. In *USENIX Security*, 2018.

[LW18] Ben Lapid and Avishai Wool. Navigating the samsung trustzone and cache-attacks on the keymaster trustlet. In *European Symposium on Research in Computer Security*, pages 175–196. Springer, 2018.

[LYG+15] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, May 2015.

[MH20] Gabrielle De Micheli and Nadia Heninger. Recovering cryptographic keys from partial information, by example. Cryptology ePrint Archive, Paper 2020/1506, 2020. https://eprint.iacr.org/2020/1506.

[MvOV01] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001.

[NMB+16] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. Trustzone explained: Architectural features and use cases. In *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, pages 445–451. IEEE, 2016.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06, page 1–20, Berlin, Heidelberg, 2006. Springer-Verlag.

[Per05] Colin Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, 2005.

[rad20] radxa. Rock pi 4 - the next generation rpi, 2020.

[Roc21] Rockchip. Rockchip rk3399 technical reference manual, 2021.

[Roh19] Roman Rohleder. Hands-on ghidra - a tutorial about the software reverse engineering framework. In *Proceedings of the 3rd ACM Workshop on Software Protection*, SPRO'19, page 77–78, New York, NY, USA, 2019. Association for Computing Machinery.

[Rya19] Keegan Ryan. Hardware-backed heist: Extracting ecdsa keys from qualcomm's trustzone. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 181–194, New York, NY, USA, 2019. Association for Computing Machinery.

[Sam15] Samsung. https://docs.samsungknox.com/admin/whitepaper/kpe/samsung-knox.htm, 2015.

[TOS10] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23:37–71, 2010.

[UH23] Rei Ueno and Naofumi Homma. How secure is exponent-blinded rsa–crt with sliding window exponentiation? *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 241–269, 2023.

[WGSW18] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. Eliminating timing side-channel leaks using program repair. In Frank Tip and Eric Bodden, editors, *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 15–26. ACM, 2018.

[WWL+17] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. {CacheD}: Identifying {Cache-Based} timing channels in production software. In *26th USENIX security symposium (USENIX security 17)*, pages 235–252, 2017.

[YF14] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 719–732, Berkeley, CA, USA, 2014. USENIX Association.

[YL20] Heedong Yang and Manhee Lee. Demystifying arm trustzone tee client api using op-tee. In *The 9th International Conference on Smart Media and Applications*, SMA 2020, page 325–328, New York, NY, USA, 2020. Association for Computing Machinery.

[ZSS+16] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y. Thomas Hou. Truspy: Cache side-channel information leakage from the secure world on ARM devices. *IACR Cryptol. ePrint Arch.*, page 980, 2016.

[ZXZ16] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. Return-oriented flush-reload side channels on ARM and their implications for android devices. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 858–870. ACM, 2016.

A Reproducing RK3399 environment in simulation

The RK3399 contains a simple BootROM integrated in the SoC and map at address `0xffff0000`. This integrated BOOTROM can load, the next booting step from multiple sources (SPI,eMMC, SD-Card,etc.). Using the integrated eFUSE, it is possible to force the loaded Bootrom to be signed with a key contained in the same fuse. We used a standard boot scenario for the RK3399, with all the bootloader steps loaded in the same SD-Card. They are in order of execution (see figure 26):

1 A two stages U-boot bootloader (TPL, SPL): in charge of initializing the DRAM and loading more complex bootloader stages from the SD-card. The RK3399 features a simple SRAM used when the DRAM is not configured.

2 A three stages OP-TEE Bootrom which contains: A secure monitor based on *TrustedFirmware-A BL31*, OP-TEE secure OS and a U-Boot bootloader to deploy the Linux kernel.

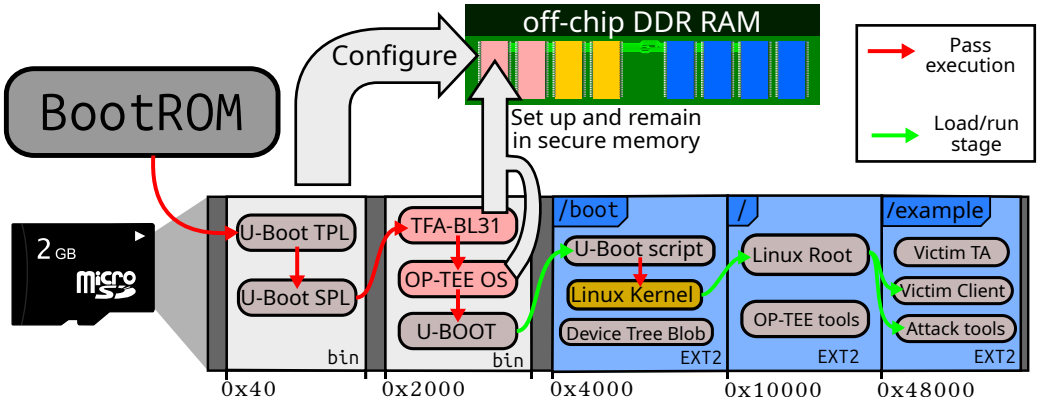

Figure 26: U-Boot assembled RK3399 boot process.

The last U-Boot stage finds an OP-TEE-enabled Linux kernel in an EXT2 `/boot` partition directly in the SD-Card. In the `/boot` partition, U-Boot also finds a Device Tree Blob (DTB) which it provides to the Linux kernel. This device tree blob has been modded to remove unused devices to simplify the platform and accelerate boot times.

On the same SD-Card, are also featured a `/root` partition and a `/example` partition. The `/root` contains a *buildroot*-made Busybox distribution [Ass24]. It features the necessary OP-TEE library, tools, and daemon to load and run Trusted Applications. Because this partition is set to read-only, we use an `/example` partition to store our Trusted applications and demonstration software (Client application and attack tools).

All of these elements are built into a single disk image which is then written to an SD-card. Loaded in the SD-card slot of the RockPi4 Board, all the communication with the RK3399 is done using UART2 accessible through GPIO pins on the board.

To reproduce the workload in a simulation environment, we relied on the original disk image used for the SD-card and an extracted RK3399 BOOTROM. This integrated BOOTROM was extracted using a modified U-boot TPL.

A.1 RK3399 Architecture

The RK3399 uses the ARM BIG.little architecture (see Table 6) By combining the TRM manual and ARM documentation of the Cortex-A53 and Cortex-A72 CPU clusters, we are able to reconstruct the RK3399-T CPU and cache topology in gem5.

Table 6: RK3399-T: CPU and cache information gathered from ARM and Rockchip TRM documentation.

RK3399-T	
Cortex-A53	4 CPUs at 1GHz
	In-order CPUs: -Armv8-A ISA including NEON and Crypto ext.
	Split L1 cache: -instruction: 32kB L1(4-way) -data: 32kB L1(4-way) -Replacement policy: pseudo-random
	L2 cache: -512kB (16-way) -Cache coherency: exclusive -Replacement policy: pseudo-least-recently-used
Cortex-A72	2 CPUs at 1.5 GHz
	Out-of-order: -Armv8-A ISA including NEON and Crypto ext. -Variable-length pipeline & Dynamic Branch Prediction
	Split L1 cache: -Instruction: 48kB (3-way) -Data: 32kB (2-way)
	L2 cache: -1MB (16-way) -Cache coherency: inclusive (AutoLock) -Replacement policy: pseudo-least-recently-used