On-chip Traffic Injection to Counteract Timing Side-Channel Attacks

Francisco Fuentes, Sergi Alcaide, Raimon Casanova, Jaume Abella

On-chip Traffic Injection to Counteract Timing Side-Channel Attacks

Francisco Fuentes[†,‡], Sergi Alcaide[†], Raimon Casanova[‡], Jaume Abella[†]

[†]Barcelona Supercomputing Center (BSC)
Barcelona, Spain

[‡]Microelectronic and Electronic Systems Department
Universitat Autònoma de Barcelona (UAB), Bellaterra, Spain

***Abstract*—Security has become a major concern in the last decade, specially with the increment of low-level attack vectors present in COTS MPSoCs. Safety-relevant systems are not an exception, and they are also exposed to security concerns. Side-channel attacks (SCAs) in general, and cache-based SCAs in particular, have gained prominent importance due to the proliferation of cache memories for increased performance. However, there are a plethora of such attacks and effective countermeasures are needed for all of those.**

This paper investigates the effectiveness of using hardware traffic injectors to counteract those attacks with the aim of assessing to what extent those can be effective. In particular, we consider the SafeTI, an open source traffic injector developed by us, and assess to what extent attack-specific traffic patterns can defeat Bernstein's SCA targeting an AES-128 encryption process in a space-relevant platform based on Frontgrade Gaisler's IPs.

***Index Terms*—Cyber security, MPSoC, side-channel attack, AES encryption**

I. Introduction

The increasing importance of security in all sorts of computing devices has pushed for the standardization and implementation of secure cryptographic ciphers (e.g., RSA) on modern machines. For instance, the growing RISC-V ecosystem has ratified in the past years two volumes of Instruction Set Architecture (ISA) extensions for the integration of inter-core cryptographic modules [8], [17], providing standardization for high-performance and secure encryption to the RISC-V community. However, these implementations include several components to be treated with security, such as cryptographic keys, intermediate cryptographic operations, etc, that may leak information that an attacker could use for malicious purposes. This is the specific case for side-channel attacks (SCAs), where an attacker without direct access to the desired data, for instance an encryption key, may be able to discover it through indirect methods such as temperature [15], electromagnetic radiation [11], power consumption [10] or timing [4], [14] analysis. Even modern Commercial-Off-The-Shelf (COTS) Multi-Processor System-on-Chip (MPSoC) platforms, using advanced Trusted Platform Modules (TPM2.0) following industry-adopted standard ISO/IEC 11889 [13], are vulnerable to such attacks [18].

This paper aims at assessing to what extent safety-critical platforms are vulnerable to those attacks and whether a programmable traffic injector could be used to counteract those attacks. In particular, we consider the open source platform SELENE [12], which is based on Frontgrade Gaisler 64-bit NOEL-V processor cores [3] and other Gaisler's IP, and whose main target is the space domain. We also consider a modified version of Bernstein's *cache-based timing attack on AES* [4]. Originally, this is a cache-based SCA against Advanced Encryption Standard (AES) [19] symmetric block cipher on a network environment, which we move to occur in an MPSoC (the SELENE platform).

To counteract the attack, we build on the SafeTI traffic injector [21], which we integrate into the SELENE platform. The SafeTI allows programming traffic patterns (i.e., read and write operations with varying parameters) that are injected into the specific interface where the SafeTI is integrated (e.g., the bus connecting the cores to the shared L2 cache). In particular, we focus on the injection of traffic patterns to evict some AES data from the second level (L2) cache of the core performing encryption tasks so that the SafeTI can provide, apart from support for performance validation during MPSoC design [9], security capabilities during operation.

The solution investigated in this paper uses the SafeTI traffic injector for evicting cached data at regular intervals. By enforcing the eviction of a specific AES table from L2 cache periodically, we are able to reduce the amount of information that Bernstein's attack can discover from a victim in a system without other countermeasures. This particular solution causes an encryption latency increase, between 1 and 14% in average depending on the protection level achieved, and since SafeTI is programmable, it can be adapted through software for other applications that may benefit from this solution.

The rest of the paper is organized as follows. Section II provides some background on Bernstein's attack and existing solutions. Section III introduces the framework used to conduct our case study, which includes the SELENE platform and the SafeTI traffic injector. Section IV provides a summary of the contributions made for this case study, being the tailoring of the Bernstein's attack for our study environment and SafeTI programming for timing SCA protection. Section V provides result data and explanations on the different SafeTI based protection vectors and Section VI provides a discussion of various subjects related to SafeTI based protection and its applicability. Finally, Section VII provides some final remarks and future work.

II. Background and State of the Art

Among the attack vectors based on collateral information leakage, timing attacks require special attention due to the high risk level they present on interconnected systems. An attacker with user permissions can perform a timing analysis on a specific task to extract secret information (e.g., cryptographic keys) remotely, without requiring physical access to the target device in comparison to other attacks. In this section, we first introduce the main characteristics of those timing SCAs in Section II-A, and then present the state of the art on protection methods in Section II-B.

A. Side-Channel Timing Attacks

Time based SCAs leverage the dependence between (a) the operation of secret data, where the term 'secret' refers to any un-encrypted data, key or information unknown by arbitrary users, with (b) the execution time, or operation latency, of the task using the secret data as an input. Time dependence is a by-product of the operation from two sources; (i) in-processor or accelerator execution of the algorithm (e.g., cryptographic encryption) and (ii) memory access latency. Leakage from both sources can be mitigated by designing time-constant algorithms, basing the logic operations on constant latency instructions with no latency-variant branches, while constraining memory allocation within the same cache level so all data access have an identical time cost. Full compliance with these statements limit the quality of the algorithm (e.g., encryption complexity), reduce the compatibility by targeting specific platform characteristics (e.g., cache capacity, instruction latency), and may hold back performance against using optimal operations (e.g., disable L1 cache). Moreover, some other SCAs would still be possible by, for instance, learning the cache sets or DRAM banks [16], [20] accessed by the protected algorithm. Thus, industry has opted for design policies and certification instead of a single air-tight solution, providing diversity in implementation with low risk of single point failure from a security standpoint.

Statistical analysis of the execution times has proven to be effective regardless of the leakage source. As example, this study uses Bernstein's attack as a base, whose source code is publicly available [4]. Operations with secrets are identifiable, due to non-constant-time operations and/or non-constant data access latencies, through a classification and correlation of the average encryption timings from two data samples, with a known (attacker) and unknown (victim) cryptographic keys respectively. In particular, the timing measurements are categorized by the value of the plaintext bytes being encrypted in byte segments, which for AES-128 makes a total of 4096 individual metrics (256 possible values for each of the 16 plaintext bytes), taking advantage from the internal byte-wise operation, a common characteristic among cryptographic ciphers. This segmentation allows the reduction of the total number of encryptions, or *sample size*, required by the attack to obtain a clear profile of the timings for each plaintext byte value. Furthermore, the samples obtained by measuring the duration of the cryptographic operation must be from a randomized input, referring to the plaintext in encryption or ciphertext in decryption, avoiding producing timing data with dependencies on the attack itself instead of the target operation.

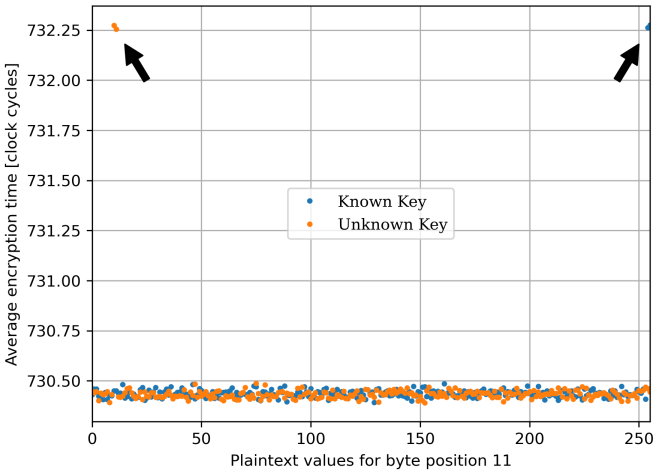

Fig. 1. Average encryption timings, in clock cycles, with plaintext byte position 11 values listed by the x-axis. Data obtained from modified Bernstein's SCA with a sample size of 2^{27} encryptions for each known and unknown keys. Data values used by the SCA correlation are marked by arrows.

As a practical example, Figure 1 shows the plaintext encryption timings of a SCA in an unprotected system. The SCA uses a sample size of 2^{27} encryptions for each key, for specifically the byte position 11. In detail, Bernstein's SCA correlation is a simple observation step, leaving aside standard error calculations, where the most distant timings from the average, around 732.25 clock cycles in the example, are taken as usable data. The idea is that these plaintext values, 10 and 11 for the unknown key and 254 and 255 for the known key marked in the figure, incur in an equivalent systematic latency overhead during the cryptographic execution. Hence, they can be correlated. Since the cipher base operation is the logic *exclusive or* $\oplus$, the attacker is able of producing a list of candidates for the key to discover by operating with the same operation between the known key and the usable plaintext values.

The actual equation used is $K'_b \oplus P'_b = K_b \oplus P_b$, where K_b is the key byte value, P_b is a plaintext byte value, b the byte position, and the $'$ apostrophe indicates to be from unknown key byte value or plaintext. Note that the exclusive or neutral element in $\oplus$ is 0 ($x \oplus 0 = x$), therefore, using a zero for the known key simplifies the operation to $K'_b|_{K=zero} = P'_b \oplus P_b$. Hence, all experiments presented in this paper use a zero known key. Following the figure example, the candidates the SCA produces for the unknown key byte position 11 are found as $10 \oplus 254 = 244$, $10 \oplus 255 = 245$, $11 \oplus 254 = 245$ and $11 \oplus 255 = 244$, finding 7 out of 8 bits from the actual unknown key byte position 11, whose value is 244. Note that each byte position has its own timing profile. Therefore, the same operations are made for each byte position.

SCA sample sizes of a low number of encryptions will provide a noisy cloud of timings, with high dispersion and difficult correlation. Thus, for the attack to succeed, it is crucial to work with large sample sizes, so the values with particular higher or lower average encryption times move far away from the average, reducing the number of dots to be correlated. Modifications made to Bernstein's attack for our study are listed in Section IV-A, while further practical details are explored in Section V-B.

B. State of the Art

Invulnerability against timing SCAs is challenging to achieve since several factors and components interact in non-obvious ways. Solutions to prevent timing SCAs can be categorized into two branches: (i) implementing time-constant cryptographic operations, and (ii) uncorrelating the access latency during operation.

Time-constant operations executed by processors require compliance with design policies for cryptographic security, such as RISC-V ISA extensions [8], [17], which have already been applied in a practical implementation [24]. Accelerator and discrete co-processor solutions, such as TPMs, are also included within this time-constant category, where the security and performance trade-off is apparent due to lacking full time-constant compliance for some products [18]. Software solutions also tend to focus on the operation latency, such as compiler optimizations [23] for avoiding branch prediction and instruction cache attacks that may present non-constant timings, and hence, a side-channel leak. Even if the cipher algorithm was designed to comply with constant-time requirements, fitting within the lowest cache layer, an SCA could still occur by forcing specific cache evictions between the timed cryptographic operations, exposing secret data through cache misses, hence leaking side-channel information. This paper aims at counteracting SCA by focusing on the data access latency.

Uncorrelation of the operating data and access latency, to avoid timing pattern identification of secrets, can be achieved by modifying the replacement policy of the data with custom cache implementations [22], [25], forcing SCA data samples to diverge due to timing diversity, making them uncorrelated, hence, protecting secret data. In detail, the address allocation of the application are encoded with a randomized seed, using the resulting encoding for allocating the data in the cache sets. In order to make the protection effective, it is necessary to ensure the seed is randomized periodically, not to let the attacker learn. Such re-randomization needs to be performed by the Operating System (OS), by the user program or, periodically in an automated manner. One of the key characteristics of these solutions is that the amount of timing data that can be used for correlation purposes by the attacker is proportional to the time a given seed is used for both keys without re-randomization. Therefore, an attack can only be successful if the time elapsed between seed updates is long enough to collect a sufficiently large sample to learn from.

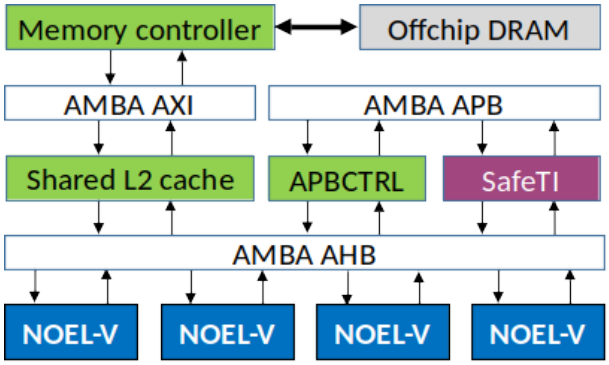

Fig. 2. SELENE platform schematic including SafeTI integration.

In our case, we investigate the effectiveness of a less intrusive hardware solution based on the integration and software programming of the SafeTI traffic injector for protecting against timing SCAs. While the SafeTI is not particularly suited for security purposes, it is a flexible and programmable component that could be leveraged for multiple functions, such as providing protection against multiple attacks, as well as for platform testing.

III. Case Study Framework

This section introduces the platform used as research vehicle for our work, as well as the SafeTI traffic injector used to counteract Bernstein's SCA.

A. SELENE MPSoC Platform

The SELENE platform considered in this work [12] has been released fully integrated and as an open-source platform usable on FPGA [7]. It is based on Frontgrade Gaisler's technology including its 64-bit NOEL-V processor cores [3], as well as the Advanced Microcontroller Bus Architecture (AMBA) Advanced Peripheral Bus (APB) and Advanced High-performance Bus (AHB) interconnects, and L1 and L2 caches, which are integrated from Gaisler's GRLIB IP [1].

For this case study, we have integrated a SafeTI module targeting the shared L2 cache level in a SELENE instance with 4 NOEL-V cores (see Figure 2). The 4-way 32 B line 512 KBs L2 cache includes a pseudo-Least Recently Used (pLRU) replacement policy and keeps coherence within the core cluster. The L2 cache is also connected to the off-chip DRAM through an AMBA Advanced eXtensible Interface (AXI) and a memory controller. Each NOEL-V core integrates two individual L1 caches, for instruction (IL1) and data (DL1), of 4 ways and 16 KBs each, implementing a LRU replacement policy, and write-through policy with bus-snooping and an equal cache line size of 32 B to maintain coherence with the L2 cache. Both pLRU and LRU replacement policies offer vulnerabilities in front of SCAs due to their systematic eviction patterns, which attackers can leverage to alter the latency of data allocation during the cryptographic task.

B. SafeTI Traffic Injector

The SafeTI is an open-source hardware component, created in our research group, devised as a flexible, portable and programmable traffic injector [5], developed in VHDL. It is AMBA AHB compatible, and we foresee making it also AMBA AXI compatible in the near future. The SafeTI is programmed through its integrated AMBA APB interface using 32-bit descriptors, which are stored within the internal descriptor buffer, made user-friendly through the public drivers along with the component designs.

Traffic injection is effectively limited by the throughput capable to be generated at the target interconnect. However, if such traffic is injected by software means through processor cores, it is further limited by the transaction size allowed by cores (either a double-word or a cache line) and controlled indirectly by inducing specific hardware behavior with a sequence of software operations. Conversely, the SafeTI can inject precisely any traffic pattern, programmed as a set of descriptors, that the target interconnect accepts, including varying size data requests (e.g., from 1 byte up to 512 MBs), with/without burst mode, read/write, etc., and even introduce specific cycle-accurate delays between traffic injections. Hence, the SafeTI offers the flexibility and controllability needed for our work.

Compared with some previous solutions, SafeTI programmability permits tailoring it for different applications, compatible with virtually any cryptographic algorithm or vulnerable process to timing SCAs under certain conditions (e.g., shared L2 cache access). Integrability is supported by being a standalone module, making it suitable for other platforms as long as it is included with a compatible interface for targeting the desired interconnect. Further discussion is provided in Section VI.

IV. Attack Characteristics and Counteracting Approach

Our realizations in this paper include the adaptation of Bernstein's attack for a bare-metal execution on the SELENE MPSoC platform, explained in Section IV-A, and the preparation of injection patterns for programming the SafeTI traffic injector to counteract the attack, presented in Section IV-B.

A. Tailoring of Bernstein's Attack

The original implementation [4] allows an attacker to discover part of the AES-128 encryption key stored in another computer server by timing plaintext encryptions with a known key and the unknown victim's key. We modified the attack source code to adapt it for the following evaluation environment: (i) an MPSoC case study on bare-metal without a network connection, meaning that all data and required resources for the encryption are loaded in main system's memory prior to the start of the attack; (ii) AES encryption implemented using the OpenSSL 3.1.2 low-level API [6]; and (iii) programmation of the SafeTI injection patterns through calls within the OpenSSL library.

The original source code of the attack is split into individual programs for commodity and presentation, which we packed into a single program and extended to apply an incremental sample size, producing timing data and unknown key candidates every power of 2 encryptions for each key. For instance, whenever we indicate a SCA sample size of 2^{21} encryptions, we refer to a pair of timing data samples, one for each key, of 2^{21} encryptions each. Compiling data following this method allows us to study the evolution of the attack at every step, displaying the SCA sample size needed to retrieve some information about the unknown key (i.e., what key values can be discarded) and how much information is discovered with increasing attack samples, which reduces the size of a brute force attack to explore all remaining combinations (i.e., those that the attack could not disregard).

These modifications have been made to evaluate the protection in the most favorable environment for the attacker. In detail, we reduce the sources of noise that could challenge the effectiveness of the attack by (1) constraining the attack to occur in the cache hierarchy of the cores without any other external interference (e.g., due to peripheral activity) and (2) avoiding the simultaneous execution of any other software within the MPSoC. This includes shrinking the timing to exclusively the encryption operation, data access latencies inclusive, cutting off any algorithm related to networking. Regarding the encryption AES cipher from the OpenSSL library, we selected specifically the 128 bits Electronic CodeBook (ECB) mode due to being the simplest one to attack.

B. SafeTI Injection Patterns

SafeTI countermeasure potential against SCAs has a great dependence on the physical location of the module within the hardware platform, being the best location as close to the processor running the vulnerable operations as possible. However, in this paper we wanted to consider a realistic implementation where the SafeTI does not have access to the DL1 cache, limiting protection capabilities, but instead it has access to the shared L2 cache, expanding protection to any of the processor cores from the MPSoC as Figure 2 shows.

Note that, in order to maximize the protection range, it is required to design an injection pattern tailored for the target environment. For instance, given that the SafeTI is unable to access the DL1 cache, and the DL1 is non-inclusive with the L2 cache in our case study, the data utilized by the vulnerable operation cannot be pre-cached or evicted by the SafeTI at the lowest cache level. Instead, SafeTI protection must be completely based on L2 cache evictions and rely on the vulnerable data to not be fully allocated within the DL1 cache. That way, SafeTI traffic generation may influence the timings of the operation by evicting essential data from the L2 cache, producing eventual L2 misses with increased latency.

Considering the L2 cache implementation characteristics, i.e., pLRU replacement policy and 4-way set associative, it is compulsory, in order to ensure data eviction, to execute at least 4 traffic accesses to each of the cache sets used by the target data (see Figure 3). Therefore, for every targeted

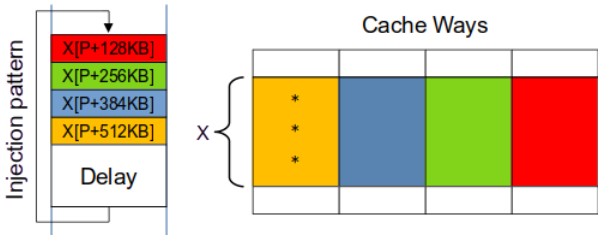

Fig. 3. Traffic generated by SafeTI injection to evict a specific data array, allocated at the cache lines marked with asterisks, from the L2 cache by filling all sets with arbitrary data using a stride of 128 KBs, the L2 cache way size. The arrow out from the end to the start of the injection pattern indicates an execution loop, which is always present in all injection patterns used.

block for eviction, SafeTI injection pattern includes 4 read descriptors with an address stride of 128 KB, the L2 cache way size, targeting the same cache sets as the target data. In detail, if the data desired to evict is allocated starting from address *P*, the first descriptor is programmed to start the access at address *P+S·128 KBs*, the second *P+(S+1)·128 KBs*, the third *P+(S+2)·128 KBs*, and the fourth *P+(S+3)·128 KBs*, where *S* is any integer but -1, -2, -3 and -4 to avoid accessing the eviction target.

In addition, the descriptors include an *access size* field, allowing us to study protection capabilities with varied eviction size with bursty accesses. In Figure 3, the different strided accesses (with starting addresses separated by 128 KBs across accesses) are shown with different colors, namely red, green, blue and yellow. The amount of data fetched by each access is set identical for all accesses matching the amount of cache lines to be evicted from the L2 cache.

The data selected for eviction is data accessed by the vulnerable operation recurrently at a fixed memory address for each execution. This ensures that evicting such data produces a latency overhead generated by the L2 eviction that propagates to the SCA's timing profile. In this paper, we have used the 4 encryption tables from the AES library as eviction target, from *Te0* to *Te3*, whose size is 1 KB each, being the address of all of them set at compile time. The access of these tables by the ECB cipher depends on both the key and plaintext being encrypted, making several accesses to different segments of the tables for every encryption operation.

Finally, a constant (too frequent) data eviction may be detrimental for the operation latency and/or the protection. Therefore, a *stand-by time*, in clock cycles, is added between target evictions to adjust the eviction rate frequency. To sustain the protection, SafeTI is programmed during initialization and configured in *QUEUE* mode, where the injection pattern is iterated until disabled. In summary, this paper explores a SafeTI L2 eviction based protection against Bernstein's SCA testing a wide range of injection rate frequencies, varying also the target (a fraction of a table, a full table, or several tables), and considering increasingly large sample sizes for the attack.

TABLE I
NUMBER OF BITS DISCOVERED BY THE SCA AGAINST ALTERNATIVE CACHE COMPOSITIONS

Enabled caches		SCA sample size (encryptions)				
DL1	L2	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}
✓	✓	2.9	14.3	33.3	70.6	72.2
	✓	0.0	0.0	0.0	0.0	0.0
✓		3.7	14.9	28.1	89.3	92.0
		41.8	64.2	72.6	80.0	80.0

V. EMPIRICAL ASSESSMENT

A. Evaluation Framework

The experiments and data presented in this paper have been produced from software executions on a bare-metal synthesis of the MPSoC SELENE hardware platform on the Xilinx Virtex UltraScale+ VCU118 FPGA-based evaluation kit [26], operating at a frequency of 100 MHz. Software programs executed on the cores have been written in C and compiled with Frontgrade Gaisler AB's NCC GCC Bare-metal toolchain version 1.0.4 on a Linux system with an O2 optimization level for a RISC-V target. Programs are loaded into memory using the FPGA debug software GRMON3 [2] with the main core set with a specific pointer during the platform booting, matching the program compilation pointer. All secondary cores are left disabled to provide a noise-free environment for the experiments. The program software includes the modified Bernstein's timing SCA targeting AES-128 ECB cryptographic cipher. Specifically, the attack targets low-level encryption operations of randomized plaintext.

The evaluation presented in this paper is divided per each type of countermeasure tested as follows:

- Disabling platform caches (Section V-B).
- Evicting *Te* tables by segments in every time interval (Section V-C).
- Evicting a single *Te* table every time interval (Section V-D).
- Evicting a combination of *Te* tables every time interval or alternatively in every time interval (Section V-E).

B. Cache Disabling as Countermeasure

Obtaining a base reference of the timing SCA effectiveness on this case study environment is imperative in order to provide a contrasted view of the protection achieved by the SafeTI in the following sections. Therefore, this section presents and reasons about the SCA capabilities in four different environments where DL1 and L2 caches can be enabled or disabled, namely, when both are enabled, when only DL1 is enabled, when only L2 is enabled, and when both are disabled.

Table I shows the equivalent number of bits found by the attacker (out of the total 128 bits of the key) for several sample sizes of the attack in our evaluation platform. These values must be read as follows: if the attacker discovers X bits of the key, it would need a brute force attack exploring 2^{128-X} key values. Note that, in practice, the attacker does not discover specific key bits but discards byte values for different parts

of the key. However, we represent results as the number of key bits that would need to be discovered to match the same cost of a brute force attack to facilitate understanding. For instance, given a key of 16 bits, hence consisting of 2 bytes, if the attack narrows down the value of the key to 37 out of 256 values for one byte and 59 for the other, the subsequent brute force attack would require exploring $37x59 = 2,183$ key combinations (instead of 2^{16}), which we express as having to find 11.1 bits (i.e. $log_2(2183)$), or equivalently, as having found 4.9 bits.

Starting from the base cache composition of DL1 and L2 caches enabled, the SCA results correspond with the information presented in previous sections. The attacker is capable of discovering more bits of the unknown key by increasing the sample size of the attack, since that way, it is capable of reducing the number of plaintext values highlighted by a higher or lower average encryption times compared to other plaintext value timing averages.

Disabling the DL1 cache but maintaining the L2 cache enabled shows that no information has been discovered by the timing SCA, therefore, we learn 2 things. First, all memory accesses from the AES-128 CBS algorithm have an equal operation latency in this setup. This evidence proves data access latency is the exclusive leakage source from the cryptographic operations being timed, at least on the SELENE platform. Second, not finding timing differences between plaintext values, even when increasing the SCA sample size, indicates all data accessed by the SCA fits within the 512 KBs of the L2 cache.

The complementary case where the DL1 cache is enabled but the L2 cache is disabled denotes slightly more susceptibility to the SCA than the base cache configuration. In this case, DL1 cache misses, instead of hitting in L2 cache, need to access main memory, whose latency is higher than that of L2 cache hits. Hence, those timings that were discrepant in the setup with both caches enabled become even more discrepant when the L2 cache is disabled. Therefore, larger differences provide easier correlation, which in turn provides a smaller list of unknown key candidates, or what is equivalent to, a higher discovery rate.

Last but not least, the disabled DL1 and L2 caches configuration shows to be the most susceptible to the attack from all cache combinations with small samples. Initially, one could expect this case to behave similarly to the prior case with disabled DL1 and enabled L2 caches, given that all data accessed by the timed operations fits within the external DRAM. However, these off-chip components introduce data and access latency dependencies [16], [20], leaking side-channel information whereas the previous case where DL1 is disabled and L2 enabled provides homogeneous latency for all L2 cache hits. In detail, there is an access latency difference when accessing depending on the data accessed due to bank and rank access patterns, generating a data-dependent timing profile due to specific plaintext encryption values requiring extra latency for their accesses than others.

From a countermeasure perspective, the disabled DL1 and enabled L2 cache composition is a strong contender as a protection solution against Bernstein's timing attack. However, reducing cache levels results in higher data access latencies, increasing the average encryption time, from the 730 clock cycles of the baseline case (both caches enabled) to 4,763 clock cycles, a considerable 552% overhead. Moreover, it could be argued that a smart attacker may be able to re-enable the attack by evicting parts of the data used by the cryptographic operation to highlight the use of specific data, hence enable correlations in the timing behavior of the cipher algorithm since the accesses latency would depend on the plaintext.

C. Partial Table Eviction with SafeTI

The first countermeasure method using SafeTI for evicting data from the L2 cache consists of evicting AES Te tables, block by block, but only evicting one segment in every time period. For instance, if a table occupies 1 KB of cache space, and it is divided into 4 blocks of 256 B each, SafeTI evicts bytes 0-255 in period P, 256-511 in period $P + 1$, 512-767 in $P + 2$, 768-1023 in $P + 3$, 0-255 again in $P + 4$, and so on and so forth.

Due to SafeTI's limited descriptor buffer, the injection patterns tested have been constrained to the following 4 different cases: Te0 eviction by 64 B blocks; Te0, Te1 and Te2 eviction by 128 B blocks; all Te tables eviction by 256 B blocks; and all Te tables eviction by 512 B blocks. Between each evicted block, the injection pattern includes a stand-by time in clock cycles, which we refer as *Delay*, that is constant for each experiment so that evictions are homogeneously distributed over time.

Regardless of the Delay or table/s eviction granularity, experimental results show that this protection method is ineffective at counteracting the SCA. The Delay values tested are in the range between 10^3 and 10^5 clock cycles. These values allow full tables to be evicted at the same frequency as the best cases for subsequent experiments where tables are evicted at once instead of block by block. Results show a similar discovery rate of the key by the attack among all 4 protection cases.

Furthermore, the SCA is slightly more successful with this approach than for the base SCA without protection. This negative effect (i.e., the protection helps the attack rather than counteracting it) relates to the fact that the data accessed from the cryptographic operation depends on the plaintext. Hence, evicting single blocks of the Te tables only highlights such plaintext values that access the recently evicted cache lines. Therefore, the attacker learns faster and injection patterns evicting full tables at once are expected to cure this anomaly as analyzed next.

D. Table Eviction with SafeTI

The injection pattern for the experiment in this section is analogous to that of the previous section, but with the evicted block matching the table size of 1 KB. Hence, in every period the target Te table is evicted. Then, SafeTI stands by for Delay clock cycles before looping again.

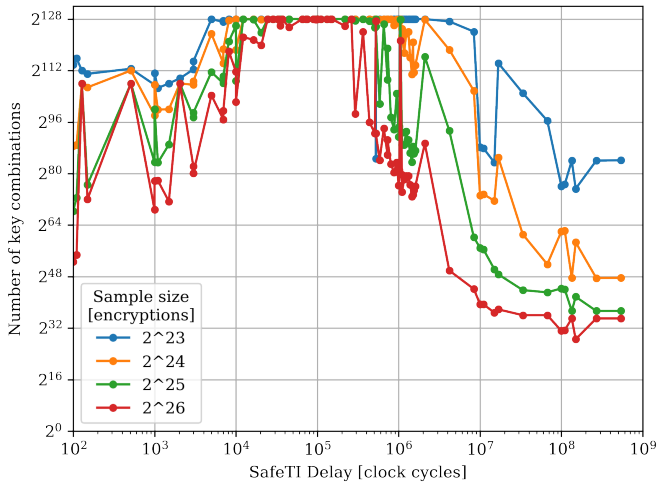

Fig. 4. Remaining key combinations from several SCAs with single table eviction protection, including different Delay values, from 2^{23} to 2^{26} encryption sample sizes.

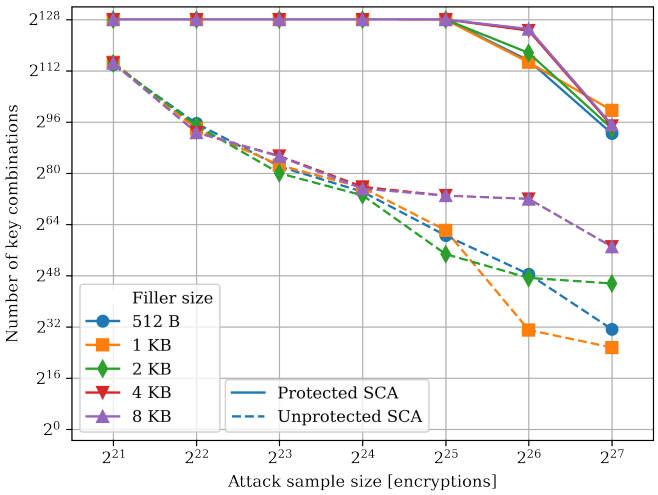

Fig. 5. Remaining key combinations from unprotected and protected SCAs with Te3 table eviction and optimal Delay of $2x10^5$ clock cycles for different sample sizes, and varying filler sizes shifting the compile address alignment of useful data.

Figure 4 shows the result of the SCA while a full Te table eviction protection is in place (for Te0 in particular), for different sample sizes for the SCA (between 2^{23} and 2^{26} sample sizes), varying the Delay between full-table evictions. We reach the following observations:

- Results are noisy due to minor modifications in the source code, presenting an intrinsic variability in the execution time measurements and SCA results (shown in Figure 5) as we discuss next.
- There is a (central) range of Delay values for which the SCA is unable to learn anything about the key so that the number of potential key combinations to explore by brute force remains at 2^{128}. However, as the sample size increases, such Delay range narrows down. If the sample is large enough, as we show in later experiments, the range becomes null and the SCA starts learning about the secret key regardless of the Delay value. Still, there is always a particular delay minimizing the amount of information learnt by the SCA.

During our experiments, we noticed that small variations in the code created significant variations in the results for a given Delay and sample size, and concluded that the particular cache alignment of the data has an impact on the results in absolute terms, yet trends hold. This is illustrated in Figure 6, which represents two sets from a 4-way cache, where there is data in static addresses allocated during compile time, and data in dynamic addresses allocated during runtime (e.g., in the stack frame), both marked with *s* and *d* suffixes respectively. The data is ordered from most recently used *A* to least recently used *E*. Focusing on the first case without a filler size (i), the set 0 caches sA (spanning across two cache lines), dC, dD and dE data lines, but once a filler size is applied to displace dynamic data by one set at (ii), dE is no longer able to fit within the cache, illustrating why some pointer displacements are able to leak more information than others.

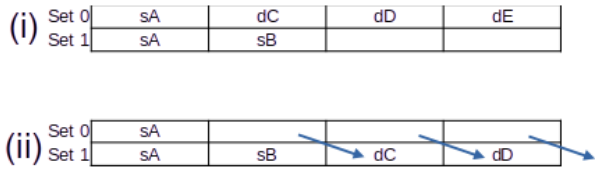

Fig. 6. Associative-set mapping diagram from a 4-way cache in two instances of (i) no filler size and (ii) filler size with one set of displacement. Data is named after being statically *s* or dynamically *d* allocated, from most used *A* to least *E*.

Figure 5 shows both the protected (straight lines) and unprotected attacks (dashed lines) for several compilations of the same program (cipher and attack) but with different *filler sizes* (between 512 B and 8 KB), which is an unrelated data array used to shift the cryptographic operation pointers for each experiment. The figure shows the diverse results in the unprotected case, ranging between 2^{25} and 2^{57} unknown key candidates, with a sample size of 2^{27} encryptions. In the protected case, variability is drastically decreased, partly because few key combinations are filtered out by the attack. Note that, whenever the filler size is a multiple of the DL1 way size (4 KBs), such as 4 and 8 KBs, results remain the same, confirming the DL1 set-mapping influence over the SCA. In any case, no array for shifting pointers has been used for the remaining experiments in this paper.

As shown before in Figure 4, the degree of protection achieved depends on the Delay value, or eviction period. Such evictions aim at generating arbitrary noise able to remove any correlation that could be used by an attacker. If performed with the right periodicity, evicting a Te table from the L2 cache causes L2 misses, and hence, access latency increases (and so execution time increases) arbitrarily and with enough magnitude to surpass the execution time variability caused by the underlying access patterns that the attacker is trying to

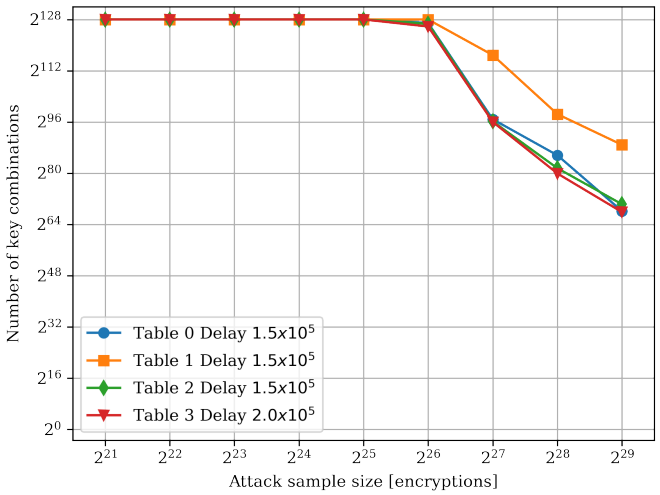

Fig. 7. Remaining key combinations from protected SCAs with Te0, Te1, Te2 or Te3 table eviction and optimal Delay, in order to maximize protection, for each table at different sample sizes.

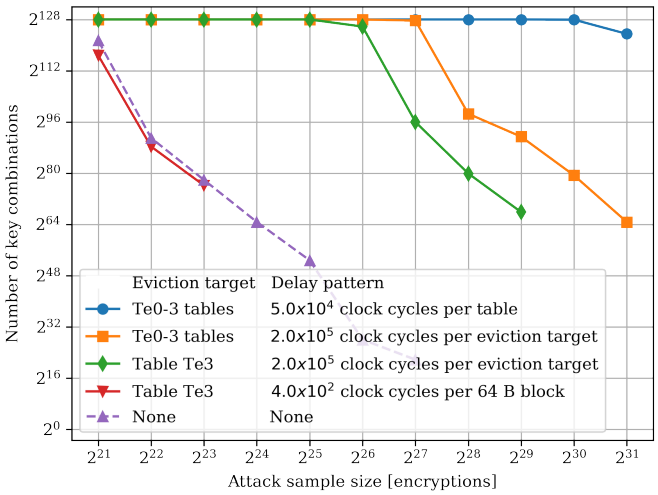

Fig. 8. Remaining key combinations from unprotected and protected SCAs using all 4 methods presented in this paper at different sample sizes.

learn. For instance, a DL1 cache set may contain a Te table line or not depending if it has been recently used. Given that the plaintext encrypted in the recent past has determined, along with the pLRU replacement policy of the DL1 cache, what lines of the Te tables are stored in DL1, Te lines retrieved from L2 are, to some extent, arbitrary. Hence, when those accesses experience higher latencies due to L2 misses caused by SafeTI evictions is, therefore, highly arbitrary. This makes execution times be apparently random because the level of noise introduced is high enough and, apparently, uncorrelated with the key. However, if the eviction period is too small, DL1 misses also miss in L2 highly systematically, which makes overall execution time increase, but noise be low. Similarly, if the eviction period is too high, meaning that evictions only occur seldom, the protection effect SafeTI has on the SCA is very limited.

Finding the optimal Delay for the protection is challenging, due to a dependence with collateral data being evicted from the same sets where the target Te is cached. This makes, in fact, that the optimal Delay varies across Te tables, as shown in Figure 7. Therefore, the only method available to optimize the Delay and choose the value that maximizes the sample size needed by the attacker is through empirical testing. As shown in the figure, the degree of protection achieved across the different tables, even for near-optimal Delay periods, may also vary. For instance, Te1 periodic eviction provides slightly higher protection than that achieved by evicting other tables due to the interactions with other data of the cipher program. Yet, these results also depend on the program pointer shift as shown before in Figure 5.

Overall, the single table eviction protection, once adjusted with the optimal delay, maintains zero side-data leak up to an attack sample size no lower than 2^{25} with an average encryption time of 741 clock cycles, x32 times the attack sample size at the cost of 1% increase in average encryption latency when compared with the unprotected SCA.

E. Multiple Table Eviction with SafeTI

This section extends the injection patterns, considering cases where all tables are evicted rather than focusing on one of them. The goal of evicting all tables rather than the very same one systematically is introducing higher entropy, and hence, further challenging the attack.

Figure 8 shows a summary of all SCA cases presented in this paper, focusing on the most favorable setups identified in each case, including scenarios where all tables are evicted. In particular, the configurations evaluated are as follows:

- Unprotected SCA (dashed purple line).
- Te3 eviction by 64 B blocks every 400 clock cycles (red line).
- Full Te3 table eviction every $2x10^5$ clock cycles (green line).
- All Te tables evicted simultaneously every $2x10^5$ clock cycles (orange line).
- Same as previous one, but instead of evicting all Te tables at once, we evict them in an interleaved manner so that one Te table is evicted every $5x10^4$ clock cycles (blue line), which matches the eviction frequency of the previous case where all tables are evicted every $2x10^5$ clock cycles.

Focusing on the patterns where we evict all tables, the orange line withstands full protection up to an attack sample size of 2^{27}, with a 4% average latency overhead, whereas the blue line keeps full protection up to an attack of 2^{30} encryptions, with a 12% average latency overhead. Although one could expect the encryption latency overhead to be proportional to the number of L2 misses induced, and thus to the L2 cache evictions performed by the traffic injector, this is not the case between orange and blue lines, which correspond to an equivalent eviction rate, and thus, should cause a similar performance overhead. However, the case of individual table evictions at a higher frequency (blue line) turns out to perform

the evictions clashing with the execution of the encryption function more often, and hence causing higher L2 cache access interference, and increasing encryption latency.

All Delay values present in the figure have been empirically optimized, with the exception of the last case where we evict all tables in an interleaved manner, where the 2^{31} encryptions experiment duration has been 29 hours in our FPGA, making it unreasonable any further increase in the sample size.

VI. Discussion

This section provides light and additional considerations for several topics related to the proposed mechanism, namely, alternative cache compositions in Section VI-A, comparison with related work in Section VI-B, considerations and requirements of the protection in Section VI-C, and protection capacity against alternative SCA sources in Section VI-D.

A. Alternative cache compositions

The case study setup includes a write-through DL1, hence propagating all write operations to the L2 cache. Our eviction patterns evict all data in specific L2 cache sets (i.e. those sets where the target Te is mapped). Hence, write operations to data in those sets also experience L2 misses due to SafeTI's evictions. If DL1 implemented a write-back policy instead, those other write operations would not be affected by our evictions if they hit in DL1. Hence, the effect of the evictions would be lower and we would expect to need a higher eviction frequency to compensate this effect or, alternatively, being able to plug the SafeTI in a way that it can evict data from DL1 rather than from L2 cache only.

Using a random replacement policy in the L2 cache would challenge to some extent the generation of eviction patterns with the SafeTI to evict full tables, which would only be evicted probabilistically. However, noise introduced would be more random, which would play against the attack.

The SELENE platform used in this work implements 2 levels of cache. Adding further cache levels is expected to be innocuous since all data fits in L2, and hence, it would also do in L3, which would provide analogous behavior to that of the DRAM memory in the current setup.

B. Related work comparison

Being SafeTI a hardware component, the closest solution for a fair comparison would be the caches implementing custom placement policies [22], [25] in order to uncorrelate cryptographic operation input data with its data access latency. These solutions offer a higher protection grade (full protection, indeed) than our protection based on SafeTI, with negligible execution time impact. However, they are intrusive with the original cache components, which would require a new implementation, and verification and validation processes for each affected cache component, hence challenging portability and increasing costs. Our solution, instead, offers a different tradeoff by providing some relevant protection and needing only additional validation of the SafeTI integration, since existing cores and caches remain unaltered.

In terms of resource usage, the SafeTI implementation represents a 1% LUTs and 5% registers with respect the whole platform, or 3.2% LUTs and 12.6% registers with respect to one of the NOEL-V cores, which we consider to be low by supporting all cores.

C. Considerations and requirements of the protection

Eviction patterns caused by the SafeTI are systematic since they repeat specific actions at specific time intervals. This could, theoretically, be leveraged by the attacker to speed up its learning process and decrease the size of the sample needed to retrieve information from the secret key. However, it is unclear how this could be done given that the impact of the evictions caused by the SafeTI vary depending on the plaintext encrypted, as discussed before. Moreover, it would not be difficult extending SafeTI to make Delay across evictions be random while preserving average eviction frequency to further challenge any attack.

Overall, we do not find practical methods where SafeTI-based protection could be defeated other than increasing the attack sample size, or disabling the SafeTI altogether.

All experiments presented show protection capabilities on a SCA targeting AES-128 ECB cryptographic cipher for a specific unknown victim key on encryption operations. In principle, the base of the protection builds exclusively on the SCA profiling dynamic and the encryption timings. Therefore, the protection is agnostic to the key being protected and the calibration may be kept for symmetrical operations such as decryption, offering an equivalent protection level. Tailoring for other vulnerable operations may be achieved through calibration of the inter-eviction Delay time, which in theory makes it capable of tailoring virtually to any vulnerable operation under the following specific requirements:

- The initial address and size of a cryptographic resource, such as the Te table(s), is required to be known during SafeTI injection pattern programming in order to evict such data during operation.
- Our protection method requires the SafeTI be able to reach a cache memory where the protected process (e.g., the encryption function) performs a relevant number of cache hits since, otherwise, SafeTI evictions would be ineffective.

D. Defense capacity against alternative SCA sources

The focus on this case study has been timing attacks, but other attack vectors exist as mentioned at the introduction, such as power, electromagnetic, temperature analysis, among others. A defining characteristic of SCAs is that, due to being a collateral data analysis, they build on some non-functional metrics from where to infer information about secret keys. We believe that SafeTI patterns can be used in many cases to induce additional activity or alter the activity of the unprotected system in a way that attack vectors other than timing can also be counteracted. Yet, how to tailor SafeTI patterns in each such case is beyond the scope of this work.

VII. Conclusions and Future Work

Security concerns become increasingly significant in safety-relevant platforms. In this paper, we explore the effectiveness of Bernstein's SCA in a space-relevant platform and show how it rapidly discovers encryption key information by exploiting cache latencies. We propose using a programmable traffic injector as a lowly intrusive and adaptable countermeasure and show that it is highly effective and causes very low performance degradation for some configurations, but starts losing efficacy as the sample size of the attack grows. Therefore, we consider this solution is particularly appropriate to be used in conjunction with other defense mechanisms that may take advantage or require the attacker to be staggered in order to provide full protection against timing SCA. This would be the case of, for instance, software solutions whose latency may be substantially higher than that of a hardware mechanism as the one proposed in this work.

The solution proposed in this paper aims at emphasizing the feasibility to use a traffic injector to counteract SCAs, and how it can be easily programmed to challenge the ability of the attacker to learn. However, underlying patterns to be learnt by the attacker still exist and, with a sufficiently large sample, eventually emerge and are learnt. Part of our ongoing research consists of devising approaches to inject traffic with the aim of, rather than adding noise, making emerge fake information so that the attacker is completely fooled and, instead of learning more or less information, it simply learns false information, which would completely defeat the attack.

Acknowledgement

This work is part of the project (ISOLDE), funded by MICIU/AEI/10.13039/501100011033 and the European Union NextGenerationEU/PRTR under grant PCI2023-143372, and the European Union's Horizon Europe Programme under project KDT Joint Undertaking (JU) under grant agreement No 101112274. This work has also been partially supported by the Spanish Ministry of Science and Innovation under grant PID2019-107255GB-C21 funded by MICIU/AEI/10.13039/501100011033.

References

[1] Frontgrade Gaisler AB. Frontgrade Gaisler AB SoC GRLIB IP library. Retrieved January 16, 2024 from https://www.gaisler.com/index.php/products/ipcores/soclibrary.

[2] Frontgrade Gaisler AB. GRMON3 FPGA debugger software product page. Retrieved January 16, 2024 from https://www.gaisler.com/index.php/products/debug-tools/grmon3.

[3] Frontgrade Gaisler AB. NOEL-V processor webpage. Retrieved Novemeber 17, 2023 from https://www.gaisler.com/index.php/products/processors/noel-v.

[4] Daniel J. Bernstein. Cache-timing attacks on aes. 2005. Retrieved Novemeber 17, 2024 from https://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[5] Barcelona Supercomputing Center. Traffic injector SafeTI open IP repository. Retrieved January 18, 2023 from https://github.com/bsc-loca/SafeTI.

[6] OpenSSL Community. Openssl 3.1.2 library repository. Retrieved Novemeber 17, 2023 from https://github.com/openssl/openssl/releases/tag/openssl-3.1.2.

[7] Universitat Politècnica de València. SELENE platform open source repository. Retrieved January 16, 2024 from https://gitlab.com/selene-riscv-platform/selene-hardware.

[8] Ken Dockser, Allen Baum, Barna Ibrahim, Barry Spinney, Ben Marshall, Derek Atkins, Markku-Juhani O. Saarinen, Nicolas Brunie, and Richard Newell. RISC-V Cryptography Extensions Volume II: Vector Instructions. Version v1.0.0, 05 October 2023.

[9] F. Fuentes et al. SafeTI traffic injector enhancement for effective interference testing in critical real-time systems, 2023. Retrieved Novemeber 17, 2023 from https://doi.org/10.48550/arXiv.2308.11528.

[10] Hasindu Gamaarachchi and Harsha Ganegoda. Power analysis based side channel attack, 2018. Retrieved January 10, 2024 from https://doi.org/10.48550/arXiv.1801.00932.

[11] Nilupulee A. Gunathilake, Ahmed Al-Dubai, William J. Buchanan, and Owen Lo. Electromagnetic side-channel attack resilience against present lightweight block cipher, 2021. Retrieved January 10, 2024 from https://doi.org/10.48550/arXiv.2112.12232.

[12] C. Hernàndez et al. SELENE: Self-monitored dependable platform for high-performance safety-critical systems. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 370–377, 2020.

[13] International Standards Organization. *ISO/IEC 11889. Trusted platform module library*, 2015.

[14] Elmira Karimi, Zhen Hang Jiang, Yunsi Fei, and David Kaeli. A timing side-channel attack on a mobile gpu. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 67–74, 2018.

[15] Taehun Kim and Youngjoo Shin. ThermalBleed: A practical thermal side-channel attack. *IEEE Access*, 10:25718–25731, 2022.

[16] Myoung Jin Lee and Kun Woo Park. A mechanism for dependence of refresh time on data pattern in dram. *IEEE Electron Device Letters*, 31(2):168–170, 2010.

[17] Ben Marshall, Alexander Zeh, Andy Glew, Barry Spinney, Daniel Page, Derek Atkins, Ken Dockser, Markku-Juhani O. Saarinen, Nathan Menhorn, L Peter Deutsch, Richard Newell, and Claire Wolf. RISC-V Cryptography Extensions Volume I: Scalar & Entropy Source Instructions. Version v1.0.1, ratified on 18'th Feb, 2022.

[18] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL: TPM meets timing and lattice attacks. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2057–2073. USENIX Association, August 2020.

[19] National Institute of Standards and Technology. *Advanced Encryption Standard (AES)*, May 2023.

[20] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for Cross-CPU attacks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 565–581, Austin, TX, August 2016. USENIX Association.

[21] Oriol Sala, Sergi Alcaide, Guillem Cabo, Francisco Bas, Ruben Lorenzo, Pedro Benedicte, David Trilla, Guillermo Gil, Fabio Mazzocchetti, and Jaume Abella. Safeti: a hardware traffic injector for mpsoc functional and timing validation. In *2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 1–7, 2021.

[22] David Trilla Rodríguez, Carles Hernández Luz, Jaume Abella Ferrer, and Francisco Javier Cazorla Almeida. *Cache side-channel attacks and time-predictability in high-performance critical real-time systems*. Association for Computing Machinery (ACM), Jun 2018.

[23] Jeroen Van Cleemput, Bjorn De Sutter, and Koen De Bosschere. Adaptive compiler strategies for mitigating timing side channel attacks. *IEEE Transactions on Dependable and Secure Computing*, 17(1):35–49, 2020.

[24] Aleksander Waage. Secure implementation of a RISC-V AES accelerator, 2022. Retrieved January 10, 2024 from https://hdl.handle.net/11250/3023096.

[25] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. *SIGARCH Comput. Archit. News*, 35(2):494–505, jun 2007.

[26] Xilinx Virtex UltraScale+ FPGA. VCU117 evaluation kit. Retrieved January 16, 2024 from https://www.xilinx.com/products/boards-and-kits/vcu118.html.