Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures

Roberto Guanciale, Hamed Nemati, Christoph Baumann and Mads Dam
Department of Computer Science
KTH Royal Institute of Technology
Stockholm, Sweden
{robertog, hnnemati, cbaumann, mfd}@kth.se

***Abstract*—Caches pose a significant challenge to formal proofs of security for code executing on application processors, as the cache access pattern of security-critical services may leak secret information. This paper reveals a novel attack vector, exposing a low-noise cache storage channel that can be exploited by adapting well-known timing channel analysis techniques. The vector can also be used to attack various types of security-critical software such as hypervisors and application security monitors. The attack vector uses virtual aliases with mismatched memory attributes and self-modifying code to misconfigure the memory system, allowing an attacker to place incoherent copies of the same physical address into the caches and observe which addresses are stored in different levels of cache. We design and implement three different attacks using the new vector on trusted services and report on the discovery of an 128-bit key from an AES encryption service running in TrustZone on Raspberry Pi 2. Moreover, we subvert the integrity properties of an ARMv7 hypervisor that was formally verified against a cache-less model. We evaluate well-known countermeasures against the new attack vector and propose a verification methodology that allows to formally prove the effectiveness of defence mechanisms on the binary code of the trusted software.**

I. Introduction

Over the past decade huge strides have been made to realise the long-standing vision of formally verified execution platforms, including hypervisors [32], [33], separation kernels [18], [41], and microkernels [30]. Many of these platforms have been comprehensively verified, down to machine code [30] and Instruction Set Architecture (ISA) [18] levels, and provide unprecedented security and isolation guarantees.

Caches are mostly excluded from these analyses. The verification of both seL4 [29] and the Prosper kernels [18], [33] assume that caches are invisible and ignore timing channels. The CVM framework from the Verisoft project [4] treats caches only in the context of device management [24]. For the verification of user processes and the remaining part of the kernel, caches are invisible. Similarly, the Nova [45], [46] and CertiKOS [21] microvisors do not consider caches in their formal analysis.

How much of a problem is this? It is already well understood that caches are one of the key features of modern commodity processors that make a precise analysis of, e.g., timing and/or power consumption exceedingly difficult, and that this can be exploited to mount timing-based side channels, even for kernels that have been fully verified [13]. These channels, thus, must be counteracted by model-external means, e.g., by adapting scheduling intervals [44] or cache partitioning [40], [28].

The models, however, should preferably be sound with respect to the features that *are* reflected, such as basic memory reads and writes. Unfortunately, as we delve deeper into the Instruction Set Architecture we find that this expectation is not met: Certain configurations of the system enable an attacker to exploit caches to build storage channels. Some of these channels are especially dangerous since they can be used to compromise both confidentiality and integrity of the victim, thus breaking the formally verified properties of isolation.

The principle idea to achieve this, is to break coherency of the memory system by deliberately not following the programming guidelines of an ISA. In this report we focus on two programming faults in particular:

1) Accessing the same physical address through virtual aliases with mismatched cacheability attributes.
2) Executing self-modifying code without flushing the instruction cache.

Reference manuals for popular architectures (ARM, Power, x64) commonly warn that not following such guidelines may result in unpredictable behaviour. However, since the underlying hardware is deterministic, the actual behaviour of the system in these cases is quite predictable and can be reverse-engineered by an attacker.

The first fault results in an incoherent memory configuration where cacheable and uncacheable reads may see different values for the same physical address after a preceding write using either of the virtual aliases. Thus the attacker can discover whether the physical address is allocated in a corresponding cache line. For the second fault, jumping to an address that was previously written without flushing the instruction cache may result in the execution of the old instruction, since data and instruction caches are not synchronised automatically. By carefully selecting old and new instructions, as well as their addresses, the attacker can then deduce the status of a given instruction cache line.

Obtaining this knowledge, i.e., whether certain cache lines contain attacker data and instructions, is the basic principle behind the Prime+Probe flavor of access-driven timing channel attacks [47]. This type of attack can be adapted using the new attack vector. The main advantage of this approach

© 2016, Roberto Guanciale. Under license to IEEE.
DOI 10.1109/SP.2016.11

is that the cache storage channels presented here are both more stealthy, less noisy, and easier to measure than timing channels. Moreover, an incoherent data cache state can be used to subvert the integrity of trusted services that depend on untrusted inputs. Breaking the memory coherency for the inputs exposes vulnerabilities that enable a malicious agent to bypass security monitors and possibly to compromise the integrity of the trusted software.

The attacks sketched above have been experimentally validated in three realistic scenarios. We report on the implementation of a prototype that extracts a 128-bit key from an AES encryption service running in TrustZone on Raspberry Pi 2. We use the same platform to implement a process that extracts the exponent of a modular exponentiation procedure executed by another process. Moreover, implementing a cache-based attack we subverted the integrity properties of an ARMv7 hypervisor that was formally verified against a cacheless model. The scenarios are also used to evaluate several existing countermeasures against cache-based attacks as well as new ones that are targeted to the alias-driven attack vector.

Finally, we propose a methodology to repair the formal analysis of the trusted software, reusing existing means as much as possible. Specifically, we show (1) how a countermeasure helps restoring integrity of a previously formally verified software and (2) how to prove the absence of cache storage side channels. This last contribution includes the adaptation of an existing tool [6] to analyse the binary code of the trusted software.

II. Background

Natural preys of side-channel attacks are implementations of cryptographic algorithms, as demonstrated by early works of Kocher [31] and Page [36]. In cache-driven attacks, the adversary exploits the caches to acquire knowledge about the execution of a victim and uses this knowledge to infer the victim's internal variables. These attacks are usually classified in three groups, that differ by the means used by the attacker to gain knowledge. In "time-driven attacks" (e.g. [48]), the attacker, who is assumed to be able to trigger an encryption, measures (indirectly or directly) the execution time of the victim and uses this knowledge to estimate the number of cache misses and hits of the victim. In "trace-driven attacks" (e.g. [2], [36], [55]), the adversary has more capabilities: he can profile the cache activities during the execution of the victim and thus observe the cache effects of a particular operation performed by the victim. This highly frequent measurement can be possible due to the adversary being interleaved with the victim by the scheduler of the operating system or because the adversary executes on a separate core and monitors a shared cache. Finally, in "access-driven attacks" (e.g. [34], [47]), the attacker determines the cache indices modified by the victim. This knowledge is obtained indirectly, by observing cache side effects of victim's computation on the behaviour of the attacker.

In the literature, the majority of trace and access driven attacks use timing channels as the key attack vector. These vectors rely on time variations to load/store data and to fetch instructions in order to estimate the cache activities of the victim: the cache lines that are evicted, the cache misses, the cache hits, etc.

Storage channels, on the other hand, use system variables to carry information. The possible presence of these channels raises concerns, since they invalidate the results of formal verification. The attacker can use the storage channels without the support of an external measurement (e.g. current system time), so there is no external variable such as time or power consumption that can be manipulated by the victim to close the channel and whose accesses can alert the victim about malicious intents. Moreover, a storage channel can be less noisy than timing channels that are affected by scheduling, TLB misses, speculative execution, and power saving, for instance. Finally, storage channels can pose risk to the integrity of a system, since they can be used to bypass reference monitors and inject malicious data into trusted agents. Nevertheless, maybe due to the practical complexities in implementing these channels, few works in literature address cache-based storage channels.

One of the new attack vectors of this paper is based on mismatched cacheability attributes and has pitfalls other than enabling access-driven attacks. The vector opens up for Time Of Check To Time Of Use (TOCTTOU) like vulnerabilities. A trusted agent may check data stored in the cache that is not consistent with the data that is stored in the memory by a malicious software. If this data is later evicted from the cache, it can be subsequently substituted by the unchecked item placed in the main memory. This enables an attacker to bypass a reference monitor, possibly subverting the security property of formally verified software.

Watson [50] demonstrated this type of vulnerability for Linux system call wrappers. He uses concurrent memory accesses, using preemption to change the arguments to a system call in user memory after they were validated. Using non-cacheable aliases one could in the same way attack the Linux system calls that read from the caller's memory. A further victim of such attacks is represented by run time monitors. Software that dynamically loads untrusted modules often uses Software-based Fault Isolation (SFI) [49], [43] to isolate untrusted components from the trusted ones. If an online SFI verifier is used (e.g. because the loaded module is the output of a just-in-time compiler), then caches can be used to mislead the verifier to accept stale data. This enables malicious components to break the SFI assumptions and thus the desired isolation.

In this paper we focus on scenarios where the victim and the attacker are hosted on the same system. An instance of such scenarios consists of a malicious user process that attempts to compromise either another user process, a run-time monitor or the operating system itself. In a cloud environment, the attacker can be a (possibly compromised) complete operating system and the victim is either a colocated guest, a virtual machine introspector or the underlying hypervisor. Further instances of such scenario are systems that use specialised hardware to isolate security critical components from untrusted operating systems. For example, some ARM processors implement TrustZone [1]. This mechanism can be used to isolate and protect the system components that implement remote attestation, trusted anchoring or virtual private networks (VPN). In this case, the attacker is either a compromised operating system kernel or an untrusted user process threatening a TrustZone application.

III. The new attack vectors: cache storage channels

Even if it is highly desirable that the presence of caches is transparent to program behaviour, this is usually not the case unless the system configuration satisfies some architecture-specific constraints. Memory mapped devices provide a trivial example: If the address representing the output register of a memory mapped UART is cacheable, the output of a program is never visible on the serial cable, since the output characters are overwritten in the cache instead of being sent to the physical device. These behaviours, which occur due to misconfigurations of the system, can raise to security threats.

To better understand the mechanisms that constitute our attack vectors, we summarise common properties of modern architectures. The vast majority of general purpose systems use set-associative caches:

(i) Data is transferred between memory and cache in blocks of fixed size, called cache lines.
(ii) The memory addresses are logically partitioned into sets of lines that are congruent wrt. a set index; usually set index depends on either virtual addresses (then the cache is called virtually indexed) or physical addresses (then the cache is called physically indexed);
(iii) The cache contains a number of ways which can hold one corresponding line for every set index.
(iv) A cache line stores both the data, the corresponding physical memory location (the tag) and a dirty flag (which indicates if the line has been changed since it was read from memory).

Caches are used by processors to store frequently accessed information and thus to reduce the number of accesses to main memory. A processor can use separate instruction and data caches in a Harvard arrangement (e.g. the L1 cache in ARM Cortex A7) or unified caches (e.g. the L2 cache in ARM Cortex A7). Not all memory areas should be cached; for instance, accesses to addresses representing registers of memory mapped devices should always be directly sent to the main memory subsystem. For this reason, modern Memory Management Units (MMUs) allow to configure, via the page tables, the caching policy on a per-page basis, allowing a fine-grained control over if and how areas of memory are cached.

In Sections III-A, III-B and III-C we present three new attack vectors that depends on misconfigurations of systems and caches. These attacks exploit the following behaviours:

- Mismatched cacheability attributes; if the data cache reports a hit on a memory location that is marked as non-cacheable, the cache might access the memory disregarding such hit. ARM calls this event "unexpected cache hit".
- Self-modifying code; even if the executable code is updated, the processor might execute the old version of it if this has been stored in the instruction cache.

```
A1) write(VA_c, 1)
A2) write(VA_nc, 0)
A3) call victim
A4) D = read(VA_nc)

V1) if secret
      access(VA3)
    else
      access(VA4)

V2) access(VA3+secret)
```

Fig. 1. Confidentiality threat due to data-cache (for write-back caches with inertia and lazy write)

```
A1) invalidate(VA_c)
A2) write(VA_nc, 0)
A3) D = read(VA_c)
A4) write(VA_nc, 1)
A5) call victim
A6) D = read(VA_c)
```

Fig. 2. Confidentiality threat due to data-cache (for write-through caches or caches that do not guarantee inertia or lazy write)

The attacks can be used to threaten both confidentiality and integrity of a target system. Moreover, two of them use new storage channels suitable to mount access driven attacks. This is particularly concerning, since so far only a noisy timing channel could be used to launch attacks of this kind, which makes real implementations difficult and slow. The security threats are particularly severe whenever the attacker is able to (directly or indirectly) produce the misconfigurations that enable the new attack vectors, as described in Section III-D.

A. Attacking confidentiality using data-caches

Here we show how an attacker can use mismatched cacheability attributes to mount access-driven cache attacks; i.e. measuring which data-cache lines are evicted by the execution of the victim.

We use the program in Figure 1 to demonstrate the attacker programming model. For simplicity, we assume that the cache is physically indexed, it has only one way and that it uses the write allocate/write back policy. We also assume that the attacker can access the virtual addresses va_c and va_{nc}, both pointing to the physical address pa; va_c is cacheable while va_{nc} is not. The attacker writes 1 and 0 into the virtual addresses va_c and va_{nc} respectively, then it invokes the victim. After the victim returns, the attacker reads back from the address va_{nc}.

Let idx be the line index corresponding to the address pa. Since va_c is cacheable, the instruction in $A1$ stores the value 1 in the cache line indexed by idx, the line is flagged as dirty and its tag is set to pa. When the instruction in $A2$ is executed, since va_{nc} is non-cacheable, the system ignores the

```
V1) D = access(VA_c)
A1) write(VA_nc, 1)
V2) D = access(VA_c)
V3) if not policy(D)
        reject
    [evict VA_c]
V4) use(VA_c)
```

Fig. 3. Integrity threat due to data-cache

```
A1) jmp A8
A2) write(&A8, {R0=1})
A3) call victim
A4) jmp A8
A5) D = R0
...
A8) R0=0
A9) return

V1) if secret
      jmp f1
    else
      jmp f2
```

Fig. 4. Confidentiality threat due to instruction-cache

“unexpected cache hit” and the value 0 is directly written into the memory, bypassing the cache. Now, the value stored in main memory after the execution of the victim depends on the behaviour of the victim itself; if the victim accesses at least one address whose line index is idx, then the dirty line is evicted and the value 1 is written back to the memory; otherwise the line is not evicted and the physical memory still contains the value 0 in pa. Since the address is non-cacheable, the value that is read from va_{nc} in $A4$ depends on the victim’s behaviour.

This mechanism enables the attacker to probe if the line index idx is evicted by the the victim. If the attacker has available a pair of aliases (cacheable and non-cacheable) for every cache line index, the attacker is able to measure the list of cache lines that are accessed by the victim, thus it can mount an access-driven cache attack. The programs V1 and V2 in Figure 1 exemplify two victims of such attack; in both cases the lines evicted by the programs depend on a confidential variable $secret$ and the access-driven cache attack can extract some bits of the secret variable.

Note that we assumed that the data cache (i) is “write-back”, (ii) has “inertia” and (iii) uses “lazy write”. That is, (i) writing is done in the cache and the write access to the memory is postponed, (ii) the cache evicts a line only when the corresponding space is needed to store new content, and (iii) a dirty cache line is written back only when it is evicted. This is not necessarily true; the cache can be write-through or it can (speculatively) write back and clean dirty lines when the memory bus is unused. Figure 2 presents an alternative attack whose success does not depend on this assumption. The attacker ($A1$-$A3$) stores the value 0 in the cache, by invalidating the corresponding line, writing 0 into the memory and reading back the value using the cacheable virtual address. Notice that after step $A3$ the cache line is clean, since the attacker used the non-cacheable virtual alias to write the value 0. Then, the attacker writes 1 into the memory, bypassing the cache. The value read in $A6$ using the cacheable address depends on the behaviour of the victim; if the victim accesses at least one address whose line index is idx, then the cache line for pa is evicted and the instruction in $A6$ fetches the value from the memory, yielding the value 1; otherwise the line is not evicted and the cache still contains the value 0 for pa.

B. Attacking Integrity Using Data-Caches

Mismatched cacheability attributes may also produce integrity threats, by enabling an attacker to modify critical data in an unauthorized or undetected manner. Figure 3 demonstrates an integrity attack. Again, we assume that the data-cache is direct-mapped, that it is physically indexed and that its write policy is write allocate/write back. For simplicity, we limit the discussion to the L1 caches. In our example, va_c and va_{nc} are virtual addresses pointing to the same memory location pa; va_c is the cacheable alias while va_{nc} is non-cacheable. Initially, the memory location pa contains the value 0 and the corresponding cache line is either invalid or the line has valid data but it is clean. In a sequential model where reads and writes are guaranteed to take place in program order and their effects are instantly visible to all system components, the program of Figure 3 has the following effects: V1) a victim accesses address va_c, reading 0; A1) the attacker writes 1 into pa using the virtual alias va_{nc}; V2) the victim accesses again va_c, this time reading 1; V3) if 1 does not respect a security policy, then the victim rejects it; otherwise V4) the victim uses 1 as the input for a security-relevant functionality.

On a real processor with a relaxed memory model the same system can behave differently, in particular: V1) using va_c, the victim reads initial value 0 from the memory at the location pa and fills the corresponding line in the cache; A1) the attacker use va_{nc} to write 1 directly into the memory, bypassing the cache; V2) the victim accesses again va_c, reading 0 from the cache; V3) the policy is evaluated based on 0; possibly, the cache line is evicted and, since it is not dirty, the memory is not affected; V4) the next time that the victim accesses pa it will read 1 and will use this value as input of the functionality, but 1 has not been checked against the policy. This enables an attacker to bypass a reference monitor, here represented by the check of the security policy, and to inject unchecked input as parameter of security critical functions.

C. Attacking Confidentiality Using Instruction Caches

Similar to data caches, instruction caches can be used to mount access-driven cache attacks; in this case the attacker probes the instruction cache to extract information about the victim execution path.

Our attack vector uses self-modifying code. The program in Figure 4 demonstrates the principles of the attack. We assume that the instruction cache is physically indexed and that it has only one way. We also assume that the attacker’s executable address space is cacheable and that the processor uses separate instruction and data caches.

Initially, the attacker's program contains a function at the address $A8$ that writes 0 into the register $R0$ and immediately returns. The attacker starts in $A1$, by invoking the function at $A8$. Let idx be the line index corresponding to the address of $A8$: Since the executable address space is cacheable, the execution of the function has the side effect of temporarily storing the instructions of the function into the instruction cache. Then ($A2$), the attacker modifies its own code, overwriting the instruction at $A8$ with an instruction that updates register $R0$ with the value 1. Since the processor uses separate instruction and data caches the new instruction is not written into the instruction cache. After that the victim completes the execution of its own code, the attacker ($A4$) re-executes the function at $A8$. The instruction executed by the second invocation of the function depends on the behaviour of the victim: if the execution path of the victim contains at least one address whose line index is idx then the attacker code is evicted, the second execution of the function fetches the new instruction from the memory and the register is updated with the value 1; otherwise, the attacker code is not evicted and the second execution of the function uses the old code updating the register with 0.

In practice, the attacker can probe if the line index idx is evicted by the victim. By repeating the probing phase for every cache line, the attacker can mount access-driven instruction cache attacks. The program V1 in Figure 4 exemplifies a victim of such attack, where the control flow of the victim (and thus the lines evicted by the program) depends on a confidential variable $secret$.

D. Scenarios

In this section we investigate practical applications and limits of the attack vectors. To simplify the presentation, we assumed one-way physically indexed caches. However, all attacks above can be straightforwardly applied to virtually indexed caches. Also, the examples can be extended to support multi-way caches if the way-allocation strategy of the cache does not depend on the addresses that are accessed: the attacker repeats the cache filling phase using several addresses that are all mapped to the same line index. The attack presented in Section III-C also assumes that the processor uses separate instruction and data caches. This is the case in most modern processors, since they usually use the "modified Harvard architecture". Modern x64 processors, however, implement a snooping mechanism that invalidates corresponding instruction cache lines automatically in case of self-modifying code ([26], Vol. 3, Sect. 11.6); in such a scenario the attack cannot succeed.

The critical assumptions of the attacks are the ability of building virtual aliases with mismatched cacheability attributes (for the attacks in Sections III-A and III-B) and the ability of self-modifying code (for the attack in Section III-C). These assumptions can be easily met if the attacker is a (possibly compromised) operating system and the victim is a colocated guest in a virtualized environment. In this case, the attacker is usually free to create several virtual aliases and to self-modify its own code. A similar scenario consists of systems that use specialised hardware to isolate security critical components (like SGX and TrustZone), where a malicious operating system shares the caches with trusted components. Notice also that in case of TrustZone and hardware assisted virtualization, the security software (e.g. the hypervisor) is not informed about the creation of setups that enable the attack vectors, since it usually does not interfere with the manipulation of the guest page tables.

In some cases it is possible to enable the attack vectors even if the attacker is executed in non-privileged mode. Some operating systems can allow user processes to reconfigure cacheability of their own virtual memory. The main reason of this functionality is to speed up some specialised computations that need to avoid polluting the cache with data that is accesses infrequently [39]. In this case two malicious programs can collude to build the aliases having mismatched attributes.

Since buffer overflows can be used to inject malicious code, modern operating systems enforce the executable space protection policy: a memory page can be either writable or executable, but it can not be both at the same time. However, to support just in time compilers, the operating systems allow user processes to change at run-time the permission of virtual memory pages, allowing to switch a writable page into an executable and vice versa (e.g. Linux provides the syscall "mprotect", which changes protection for a memory page of the calling process). Thus, the attack of Section III-C can still succeed if: (i) initially the page containing the function $A8$ is executable, (ii) the malicious process requests the operating system to switch the page as writable (i.e. between step $A1$ and $A2$) and (iii) the process requests the operating system to change back the page as executable before re-executing the function (i.e. between step $A2$ and $A4$). If the operating system does not invalidate the instruction cache whenever the permissions of memory pages are changed, the confidentiality threat can easily be exploited by a malicious process.

In Sections II and VII we provide a summary of existing literature on side channel attacks that use caches. In general, every attack (e.g. [2], [55], [34]) that is access-driven and that has been implemented by probing access times can be reimplemented using the new vectors. However, we stress that the new vectors have two distinguishing characteristics with respect to the time based ones: (i) the probing phase does not need the support of an external measurement, (ii) the vectors build a cache-based storage channel that has relatively low noise compared channels based on execution time which depend on many other factors than cache misses, e.g., TLB misses and branch mispredictions.

In fact, probing the cache state by measuring execution time requires the attacker to access the system time. If this resource is not directly accessible in the execution level of the attacker, the attacker needs to invoke a privileged function that can introduce delays and noise in the cache state (e.g. by causing the eviction from the data cache when accessing internal data-structures). For this reason, the authors of [55] disabled the timing virtualization of XEN (thus giving the attacker direct access to the system timer) to demonstrate a side channel attack. Finally, one of the storage channels presented here poses integrity threats clearly outside the scope of timing based attacks.

IV. Case Studies

To substantiate the importance of the new attack vectors, and the need to augment the verification methodology to properly take caches and cache attributes into account, we examine the attack vectors in practice. Three cases are presented: A malicious OS that extracts a secret AES key from a cryptoservice hosted in TrustZone, a malicious paravirtualized OS that subverts the memory protection of a hypervisor, and a user process that extracts the exponent of a modular exponentiation procedure executed by another process.

A. Extraction of AES keys

AES [17] is a widely used symmetric encryption scheme that uses a succession of rounds, where four operations (SubBytes, ShiftRows, MixColumn and AddRoundKey) are iteratively applied to temporary results. For every round i, the algorithm derives the sub key K_i from an initial key k. For AES-128 it is possible to derive k from any sub key K_i.

Traditionally, efficient AES software takes advantage of precomputed SBox tables to reach a high performance and compensate the lack of native support to low-level finite field operations. The fact that disclosing access patterns to the SBoxes can make AES software insecure is well known in literature (e.g. [51], [47], [2]). The existing implementations of these attacks probe the data cache using time channels, here we demonstrate that such attacks can be replicated using the storage channel described in Section III-A. With this aim, we implement the attack described in [34].

The attack exploits a common implementation pattern. The last round of AES is slightly different from the others since the MixColumn operation is skipped. For this reason, implementations often use four SBox tables T_0, T_1, T_2, T_3 of 1KB for all the rounds except the last one, whereas a dedicated T_4 is used. Let c be the resulting cipher-text, n be the total number of rounds and x_i be the intermediate output of the round i. The last AES round computes the cipher-text as follows:

$$c = K_n \oplus \mathit{ShiftRows}(\mathit{SubBytes}(x_{n-1}))$$

Instead of computing $\mathit{ShiftRows}(\mathit{SubBytes}(x_{n-1}))$, the implementation accesses the precomputed table T_4 according to an index that depends on x_{n-1}. Let $b[j]$ denote the j-th byte of b and $[T_4\ \mathit{output}]$ be one of the actual accesses to T_4, then

$$c[j] = K_n[j] \oplus [T_4\ \mathit{output}]\,.$$

Therefore, it is straightforward to compute K_n knowing the cipher-text and the entry yielded by the access to T_4:

$$K_n[j] = c[j] \oplus [T_4\ \mathit{output}]$$

Thus the challenge is to identify the exact $[T_4\ \mathit{output}]$ for a given byte j. We use the “non-elimination” method described in [34]. Let L be a log of encryptions, consisting of a set of pairs (c_l, e_l). Here, c_l is the resulting cipher-text and e_l is the set of cache lines accessed by the AES implementation. We define $L_{j,v}$ to be the subset of L such that the byte j of the cipher-text is v:

$$L_{j,v} = \{(c_l, e_l) \in L \text{ such that } c_l[j] = v\}$$

Since $c[j] = K_n[j] \oplus [T_4\ \mathit{output}]$ and the key is constant, if the j-th byte of two cipher-texts have the same value then the accesses to T_4 for such cipher-text must contain at least one common entry. Namely, the cache line accessed by the implementation while computing $c[j] = K_n[j] \oplus [T_4\ \mathit{output}]$ is (together with some false positives) in the non-empty set

$$E_{j,v} = \bigcap_{(c_l,e_l)\in L_{j,v}} e_l$$

Let $T_4^{j,v}$ be the set of distinct bytes of T_4 that can be allocated in the cache lines $E_{j,v}$. Let v, v' be two different values recorded in the log for the byte j. We know that exist $t_4^{i,v} \in T_4^{j,v}$ and $t_4^{i,v'} \in T_4^{j,v'}$ such that $v = K_n[j] \oplus t_4^{i,v}$ and $v' = K_n[j] \oplus t_4^{i,v'}$. Thus

$$v \oplus v' = t_4^{j,v} \oplus t_4^{j,v'}$$

This is used to iteratively shrink the sets $T_4^{j,v}$ and $T_4^{j,v'}$ by removing the pairs that do not satisfy the equation. The attacker repeats this process until for a byte value v the set $T_4^{j,v}$ contains a single value; then the byte j of key is recovered using $K_n[j] = v \oplus t_4^{j,v}$. Notice that the complete process can be repeated for every byte without gathering further logs and that the attacker does not need to know the plain-texts used to produce the cipher-texts.

We implemented the attack on a Raspberry Pi 2 [42], because this platform is equipped with a widely used CPU (ARM Cortex A7) and allows to use the TrustZone extensions. The system starts in TrustZone and executes the bootloader of our minimal TrustZone operating system. This installs a secure service that allows an untrusted kernel to encrypt blocks (e.g. to deliver packets over a VPN) using a secret key. This key is intended to be confidential and should not be leaked to the untrusted software. The trusted service is implemented using an existing AES library for embedded devices [53], that is relatively easy to deploy in the resource constrained environment of TrustZone. However, several other implementations (including OpenSSL [35]) expose the same weakness due to the use of precomputed SBoxes. The boot code terminates by exiting TrustZone and activating the untrusted kernel. This operating system is not able to directly access the TrustZone memory but can invoke the secure service by executing Secure Monitor Calls (SMC).

In this setting, the attacker (the untrusted kernel), which is executed as privileged software outside TrustZone, is free to manipulate its own page tables (which are different from the ones used by the TrustZone service). Moreover, the attacker can invalidate and clean cache lines, but may not use debugging instructions to directly inspect the state of the caches.

The attacker uses the algorithm presented in Figure 2, however several considerations must be taken into account to make the attack practical. The attacker repeats the filling and probing phases for each possible line index (128) and way (4) of the data-cache. In practice, since the cache eviction strategy is pseudo random, the filling phase is also repeated several times, until the L1 cache is completely filled with the probing data (i.e. for every pair of virtual addresses used, accessing to the two addresses yield different values).

On Raspberry Pi 2, the presence of a unified L2 cache can obstruct the probing phase: even if a cache line is evicted from the L1 cache by the victim, the system can temporarily store the line into the L2 cache, thus making the probing phase yield false negatives. It is in general possible to extend the attack to deal with L2 caches (by repeating the filling and probing phases for every line index and way of the L2 cache subsystem), however, in Raspberry Pi 2 the L2 cache is shared between the CPU and the GPU, introducing a considerable amount of noise in the measurements. For this reason we always flushes the L2 cache between the step $A5$ and $A6$ of the attack. We stress that this operation can be done by the privileged software outside TrustZone without requiring any support by TrustZone itself.

To make the demonstrator realistic, we allow the TrustZone service to cache its own stack, heap, and static data. This pollutes the data extracted by the probing phase of the attack: it can now yield false positives due to access of the victim to such memory areas. The key extraction algorithm can handle such false positives, but we decide to filter them out to speed up the analysis phase. For this reason, the attacker first identifies the cache lines that are frequently evicted independently of the resulting cipher-text (e.g. lines where the victim stack is probably allocated) and removes them from the sets $E_{j,v}$. As common, the AES implementation defines the SBox tables as consecutive arrays. Since they all consists of 1 KB of data, the cache lines where different SBoxes are allocated are non-overlapping, helping the attacker in the task of reducing the sets $E_{j,v}$ to contain a single line belonging to the table T_4 and of filtering out all evictions that are due the previous rounds of AES.

For practical reasons we implemented the filling and probing phase online, while we implemented the key extraction algorithm as a offline Python program that analyses the logs saved by the online phase. The complete online phase (including the set-up of the page tables) consists of 552 lines of C, while the Python programs consists of 152 lines of code. The online attacker generates a stream of random 128 bits plain-texts and requests to the TrustZone service their encryption. Thus, the frequency of the attacker's measurements isolates one AES encryption of one block per measurement. Moreover, even if the attacker knows the input plain-texts, they are not used in the offline phase. We repeated the attack for several randomly generated keys and in the worst case, the offline phase recovered the complete 128-bit key after 850 encryption in less than one second.

B. Violating Spatial Isolation in a Hypervisor

A hypervisor is a low-level execution platform controlling accesses to system resources and is used to provide isolated partitions on a shared hardware. The partitions are used to execute software with unknown degree of trustworthiness. Each partition has access to its own resources and cannot encroach on protected parts of the system, like the memory used by the hypervisor or the other partitions. Here we demonstrate that a malicious operating system (guest) running on a hypervisor can gain illicit access to protected resources using the mechanism described in Section III-B.

As basis for our study we use a hypervisor [33] that has been formally verified previously with respect to a cache-less model. The hypervisor runs on an ARMv7 Cortex-A8 processor [16], where both L1 and L2 caches are enabled. On ARMv7 the address translation depends on the page tables stored in the memory. Entries of the page tables encode a virtual-to-physical mapping for a memory page as well as access permissions and cacheability setting. On Cortex-A8 the MMU consults the data cache before accessing the main memory whenever a page table descriptor must be fetched.

The architecture is paravirtualized by the hypervisor for several guests. Only the hypervisor is executing in privileged mode, while the guests are executed in non-privileged mode and need to invoke hypervisor functionality to alter the critical resources of the system, like page tables.

A peculiarity of the hypervisor (and others [7]) that makes it particularly relevant for our purpose is the use of so-called direct paging [33]. Direct paging enables a guest to manage its own memory space with assistance of the hypervisor. Direct paging allows the guest to allocate the page tables inside its own memory and to directly manipulate them while the tables are not in active use by the MMU. Then, the guest uses dedicated hypervisor calls to effectuate and monitor the transition of page tables between passive and active state. The hypervisor provides a number of system calls that support the allocation, deallocation, linking, and activation of guest page tables. These calls need to read the content of page tables that are located in guest memory and ensure that the proposed MMU setup does not introduce any illicit access grant. Thus the hypervisor acts as a reference monitor of the page tables.

As described in Section III-B, on a Cortex-A8 processor sequential consistency is not guaranteed if the same memory location is accessed by virtual aliases with mismatched cacheability attributes. This opens up for vulnerabilities. The hypervisor may check a page table by fetching its content from the cache. However, if the content of the page table in the cache is clean and different from what has been placed by the attacker in the main memory and the page table is later evicted from the cache, the MMU will use a configuration that is different from what has been validated by the hypervisor.

Figure 5 illustrates how a guest can use the aliasing of the physical memory to bypass the validation needed to create a new page table. Hereafter we assume that the guest and the hypervisor use two different virtual addresses to point to the same memory location. Initially, the hypervisor (1) is induced to load a valid page table in the cache. This can be done by writing a valid page table, requesting the hypervisor to verify and allocate it and then requesting the hypervisor to deallocate the table. Then, the guest (2) stores an invalid page table in the same memory location. If the guest uses a non-cacheable virtual alias, the guest write (3) is directly applied to the memory bypassing the cache. The guest (4) requests the hypervisor to validate and allocate this memory area, so that it can later be used as page table for the MMU. At this point, the hypervisor is in charge of verifying that the memory area contains a valid page table and of revoking any direct access of the guest to this memory. In this way, a validated page table can be later used securely by the MMU. Since the hypervisor (4) accesses the same physical location through the cache, it can potentially validate stale data, for example the ones fetched during the step (1). At a later point in time, the validated data is evicted from the cache. This data is not written back to the memory since the hypervisor has only checked the page table content and thus the corresponding cache lines are clean. Finally, the MMU (5) uses the invalid page table and its settings become untrusted.

Note that this attack is different from existing "double mapping" attacks. In double-mapping attacks the same physical memory is mapped "simultaneously" to multiple virtual memory addresses used by different agents; the attack occurs when the untrusted agent owns the writable alias, thus being able to directly modify the memory accessed by the trusted one. Here, the attacker exploits the fact that the same physical memory is first allocated to the untrusted agent and then re-allocated to the trusted one. After that the ownership is transferred (after step A1), the untrusted agent has no mapping to this memory area. However, if the cache contains stale data the trusted agent may be compromised. Moreover, the attack does not depend on misconfiguration of the TLBs; the hypervisor is programmed to completely clean the TLBs whenever the MMU is reconfigured.

We implemented a malicious guest that managed to bypass the hypervisor validation using the above mechanism. The untrusted data, that is used as configuration of the MMU, is used to obtain writable access to the master page table of the hypervisor. This enables the attacker to reconfigure its own access rights to all memory pages and thus to completely take over the system.

Not all hypervisors are subject to this kind of vulnerability. For example, if a hypervisor uses shadow paging, then guest pages are copied into the hypervisor's own memory where they are transformed into so-called shadow page tables. The guest has no access to this memory area and the hypervisor always copies cached data (if present), so the attack described above cannot be replicated. On the other hand, the adversary can still attack secure services hosted by the hypervisor, for example a virtual machine introspector. In [12] the hypervisor is used to implement a run-time monitor to protect an untrusted guest from its internal threats. The monitor is deployed in a separate partition to isolate it from the untrusted guest.

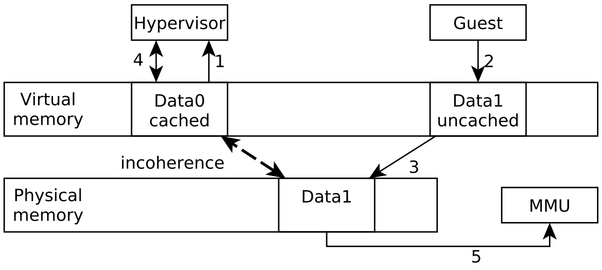

Fig. 5. Compromising integrity of a direct paging mechanism using incoherent memory. The MMU is configured to use a page table that was not validated by the hypervisor.

```
y := 1
for i = m down to 1
    y = Square(y)
    y = ModReduce(y, N)
    if e_i == 1
       y = Mult(y,x)
       y = ModReduce(y, N)
```

Fig. 6. Square and multiply algorithm

The policy enforced by the monitor is executable space protection: each page in the memory can be either writable or executable but not both at the same time. The monitor, via the hypervisor, intercepts all changes to the executable codes. This allows to use standard signature checking to prevent code injection. Each time the guest operating system tries to execute an application, the monitor checks if the binary of the application has a valid signature. In case the signature is valid, the monitor requests the hypervisor to make executable the physical pages that contain the binary code. The security of this system depends on the fact that the adversary cannot directly modify a validated executable due to executable space protection. However, if a memory block of the application code is accessed using virtual aliases with mismatched cacheability attributes, the untrusted guest can easily mislead the monitor to validate wrong data and execute unsigned code.

C. Extraction of exponent from a modular exponentiation procedure

The square and multiply algorithm of Figure 6 is often used to compute the modular exponentiation $x^e \bmod N$, where $e_m \ldots e_1$ are the bits of the binary representation of e. This algorithm has been exploited in access-driven attacks, since the sequence of function calls directly leaks e, which corresponds to the private key in several decryption algorithms. Here we demonstrate that an attacker that is interleaved with a victim can infer e using the storage channel described in Section III-C.

The attack was implemented on Raspberry Pi 2. We build a setting where a malicious process (e.g. a just in time compiler) can self-modify its own code. Moreover, we implement a scheduler that allows the attacker to be scheduled after every loop of the victim.[1]

The attacker uses the vector presented in Figure 4, repeating the filling and probing phases for every way of the instruction cache and for every line index where the code of the functions `Mult` and `ModReduce` can be mapped. Due to the separate instruction and data L1 caches, the presence of the L2 cache does not interfere with the probing phase. However, we must ensure that the instruction overwritten in the step ($A2$) does not sit in the L1 data-cache when the step ($A4$) is executed. Since user processes cannot directly invalidate or clean cache lines, we satisfy this requirement by adding a further step ($A3.b$). This step writes several addresses whose line indices in the L1 data-cache are the same of the address $\&A8$, thus forcing the eviction from the L1 data-cache of the line that has contains the instruction stored at $\&A8$.

We repeated the attack for several randomly generated values of e and in each case the attacker correctly identified the execution path of the victim. This accuracy is due to the simple environment (no other process is scheduled except the victim and the attacker) and the lack of noise that is typical in attacks that use time channels.

V. Countermeasures

Literature on access-based timing channel attacks suggests a number of well-known countermeasures. Specifically, for attacks on the confidentiality of AES encryption, a rather comprehensive list of protective means is provided in [51]. Some of the approaches are specific to AES, e.g., using registers instead of memory or dedicated hardware instructions for the SBox table look-up. Others are specific to the timing attack vector, e.g., reducing the accuracy of timing information available to the attacker. Still, there are well-known solutions addressing the presence of caches in general, thus they are suitable to defend against attacks built on the cache storage channel described in this paper.

In what follows we identify such known general countermeasures (Sections V-A and V-C.1-5) and propose new ones that are specific to the attack vector using uncacheable aliases (Sections V-B, V-C.6, and V-D). In addition it is examined which countermeasures are suitable to protect against the integrity threat posed by incoherent aliases in the memory system and propose a fix for the hypervisor example.

Different countermeasures are evaluated by implementing them for the AES and hypervisor scenarios introduced in the previous section and analysing their performance. The corresponding benchmark results are shown in Tables I and II. Since our main focus is on verifying systems in the presence of caches, for each group of countermeasures we also sketch how a correctness proof would be conducted. Naturally, such proofs require a suitable model of the memory system including instruction and data caches.

It should be emphasised that the verification of the countermeasures is meant to be performed separately from the verification of the overall system which is usually assuming a much simpler memory model for feasibility. The goal is to show that the countermeasures neutralise the cache storage channels and re-establish a coherent memory model. The necessary program verification conditions from such a proof can then be incorporated into the overall verification methodology, supporting its soundness.

A. Disabling Cacheability

The simplest way to eliminate the cache side channel is to block an attacker from using the caches altogether. In a virtualization platform, like an operating system or a hypervisor, this can be achieved by enforcing the memory allocated to untrusted guest partitions to be uncacheable. Consequently, cache-driven attacks on confidentiality and integrity of a system are no longer possible. Unfortunately, this countermeasure comes at great performance costs, potentially slowing down a system by several orders of magnitude. On the other hand, a proof of the correctness of the approach is straight-forward. Since the attacker cannot access the caches, they are effectively invisible to him. The threat model can then be specified using a coherent memory semantics that is a sound abstraction of a system model where caches are only used by trusted code.

B. Enforcing Memory Coherency

Given the dramatic slowdown expected for a virtualization platform, it seems out of the question to completely deny the use of caches to untrusted guests. Nevertheless, the idea of enforcing that guest processes cannot break memory coherency through uncacheable aliases still seems appealing.

1) Always Cacheable Guest Memory

When making all guest memory uncacheable is prohibitively expensive, an intuitive alternative could be to just make all guest memory cacheable. Indeed, if guests are user processes in an operating system this can be easily implemented by adapting the page table setup for user processes accordingly, i.e., enforcing cacheability for all user pages. Then user processes cannot create uncacheable aliases to measure cache contents and start cache-based time-of-check-to-time-of-use attacks on their host operating system.

However, for hypervisors, where guests are whole operating systems, the approach has several drawbacks. First of all, operating systems are usually controlling memory mapped I/O devices which should be operated through uncacheable memory accesses. If a hypervisor would make all memory accesses of a guest OS cacheable, the OS will not be able to properly control I/O devices and probably not work correctly. Thus, making all untrusted guest memory cacheable only works for (rather useless) operating systems that do not control I/O devices. Furthermore, there are cases when a guest can optimise its performance by making seldomly used pages uncacheable [39].

2) $C \oplus U$ Policy

Instead of making all guest pages cacheable, a hypervisor could make sure that at all times a given physical page can either be accessed in cacheable or uncacheable mode

[1] Forcing the scheduler of a general purpose OS to grant such high frequency of measurements is out of the scope of this paper. The interested reader can refer to [34], [55].

TABLE I
HYPERVISOR MICRO AND APPLICATION BENCHMARKS

LMbench micro benchmark	Native	Hyp	ACPT	SelFl	Flush
null syscall	0.41	1.75	1.76	1.77	1.76
read	0.84	2.19	2.20	2.20	2.38
write	0.74	2.09	2.10	2.15	2.22
stat	3.22	5.61	5.50	5.89	5.92
fstat	1.19	2.53	2.55	2.56	2.65
open/close	6.73	14.50	14.42	14.86	14.71
select(10)	1.86	3.29	3.30	3.33	3.42
sig handler install	0.85	2.87	2.89	2.92	2.95
sig handler overhead	4.43	14.45	14.48	15.11	14.91
protection fault	2.66	3.73	3.83	3.91	3.70
pipe	21.83	48.78	47.79	47.62	692.91
fork+exit	1978	5106	5126	6148	38787
fork+execve	2068	5249	5248	6285	39029
pagefaults	3.76	11.21	11.12	21.55	332.82

Application benchmark	Native	Hyp	ACPT	SelFl	Flush
tar (500K)	70	70	70	70	190
tar (1M)	120	120	120	120	250
tar (2M)	230	210	200	210	370
dd (10M)	90	140	140	160	990
dd (20M)	190	260	260	570	1960
dd (40M)	330	500	450	600	3830
jpg2gif(5KB)	60	60	60	60	130
jpg2gif(250KB)	920	810	820	830	1230
jpg2gif(750KB)	930	870	870	880	1270
jpg2bmp(5KB)	40	40	40	40	110
jpg2bmp(250KB)	1350	1340	1340	1350	1720
jpg2bmp(750KB)	1440	1420	1420	1430	1790
jpegtrans(270’, 5KB)	10	10	10	10	80
jpegtrans(270’, 250KB)	220	240	240	250	880
jpegtrans(270’, 750KB)	380	400	400	420	1050
bmp2tiff(90 KB)	10	10	10	10	60
bmp2tiff(800 KB)	20	20	20	20	80
ppm2tiff(100 KB)	10	10	10	10	70
ppm2tiff(250 KB)	10	10	10	20	80
ppm2tiff(1.3 MB)	20	30	30	30	90
tif2rgb(200 KB)	10	20	20	20	120
tif2rgb(800 KB)	40	40	40	50	270
tif2rgb(1.200 MB)	130	160	160	180	730
sox(aif2wav 100KB)	20	20	20	30	140
sox(aif2wav 500KB)	40	60	60	60	180
sox(aif2wav 800KB)	60	100	100	110	220

LMbench micro benchmarks [μs] and application benchmarks [ms] for the Linux kernel v2.6.34 running natively on BeagleBone Black, paravirtualized on the hypervisor without protection against the integrity threat (Hyp), with always cacheable page tables (ACPT), with selective flushing (SelFl), and with full cache flushes on entry (Flush).

($C \oplus U$ policy). To this end it would need to monitor the page table setup of the guests and forbid them to define both cacheable and uncacheable aliases of the same physical address. Then guests may set up uncacheable virtual pages only if no cacheable alias exists for the targeted physical page. Moreover, the hypervisor has to flush a cacheable page from the caches when it becomes uncacheable, in order to remove stale copies of the page that might be abused to set up an alias-driven cache attack. In this way, the hypervisor would enforce memory coherency for the guest memory by making sure that no content from uncacheable guest pages is ever cached and for cacheable pages cache entries may only differ from main memory if they are dirty.

A Trust-zone cryptoservice that intends to prevent a malicious OS to use memory incoherency to measure the Trustzone accesses to the cache can use TZ-RKP [5] and extend its run-time checks to force the OS to respect the $C \oplus U$ policy.

3) Second-Stage MMU

Still, for both the static and the dynamic case, the $C \oplus U$ policy may be expensive to implement for fully virtualizing hypervisors that rely on a second stage of address translation. For example, the ARMv8 architecture provides a second stage MMU that is controlled by the hypervisor, while the first stage MMU is controlled by the guests. Intermediate physical addresses provided by the guests are then remapped through the second stage to the actual physical address space. The mechanism allows also to control the cacheability of the intermediate addresses, but it can only enforce non-cacheability. In order to enforce cacheability, the hypervisor would need to enforce it on the first stage of translation by intercepting the page table setup of its guests, which creates an undesirable performance overhead and undermines the idea of having two independently operated stages of address translation.

4) $W \oplus X$ Policy

Unfortunately, enforcing cacheability of memory accesses does not protect against the instruction-cache-based confidentiality threat described earlier. In order to prevent an attacker from storing incoherent copies for the same instruction address in the memory system, the hypervisor would also need to prohibit self-modifying code for the guests, i.e., ensure that all guest pages are either writable or executable ($W \oplus X$ policy). Since operating systems regularly use self-modification, e.g., when installing kernel updates or swapping in pages, the association of pages to the executable or writable attribute is dynamic as well and must be monitored by the hypervisor. It also needs to flush instruction caches when an executable page becomes writable.

Overall, the solutions presented above seem to be more suitable for paravirtualizing hypervisors, that are invoked by the guests explicitly to configure their virtual memory. Adding the required changes to the corresponding MMU virtualization functionality seems straightforward. In fact, for the paravirtualizing hypervisor presented in this paper a tamper-proof security monitor has been implemented and formally verified, which enforces executable space protection on guest memory and checks code signatures in order to protect the guests from malicious code injection [12].

5) Always Cacheable Page Tables

To protect the hypervisor against the integrity threat a lightweight specialization of the $C \oplus U$ policy introduced above was implemented. It is based on the observation that uncacheable aliases can only subvert the integrity of the hypervisor if they are constructed for the inputs of its MMU virtualization functions. Thus the hypervisor needs only to enforce the $C \oplus U$ policy, and consequently memory coherency, on its inputs. While this can be achieved by flushing the caches appropriately (see Section V-C), a more efficient approach is to allocate the page tables of the guests in regions that are always cacheable. These regions of physical memory are fixed for each guest and the hypervisor only validates a page table for the guest if it is allocated in this area. In all virtual addresses mapping to the area are forced to be cacheable. Obviously, also the guest system needs to be

TABLE II
AES ENCRYPTION BENCHMARKS

AES encryption	5 000 000 × 16B		10 000 × 8KB	
	Time	Throughput	Time	Throughput
Original SBoxes	23s	3.317 MB/s	13s	6.010 MB/s
Compact Last SBox	24s	3.179 MB/s	16s	4.883 MB/s
Scrambled Last SBox	30s	2.543 MB/s	20s	3.901 MB/s
Uncached Last SBox	36s	2.119 MB/s	26s	3.005 MB/s
Scrambled All SBoxes	132s	0.578 MB/s	125s	0.625 MB/s
Uncached All SBoxes	152s	0.502 MB/s	145s	0.539 MB/s

AES encryption on Raspberry Pi 2 of one block (128 bits = 16 Bytes) and 512 blocks for different SBox layouts.

adapted to support the new requirement on the allocation of page tables. However, given a guest system that was already prepared to run on the original hypervisor, the remaining additional changes should be straight-forward. For instance, the adaptation of the hypervisor example required changes to roughly 35 LoC in the paravirtualized Linux kernel and an addition of 45 LoC to the hypervisor for the necessary checks,

The performance of the hypervisor with always cacheable page tables (ACPT) can be observed in Table I. Compared to the original hypervisor there are basically no performance penalties. In some cases the new version even outperforms the original hypervisor, due to the ensured cacheability of page tables. It turns out that in the evaluated Linux kernel, page tables are not always allocated in cacheable memory areas. The correctness of the approach is discussed in detail in Section VI. The main verification condition to be discharged in a formal proof of integrity is that the hypervisor always works on coherent memory, hence any correctness proof based on a coherent model also holds in a more detailed model with caches.

C. Repelling Alias-Driven Attacks

The countermeasures treated so far were aimed at restricting the behaviour of the attacker to prevent him from harvesting information from the cache channel or break memory coherency in an attack on integrity. A different angle to the problem lies in focusing on the trusted victim process and ways it can protect itself against an unrestricted attacker that is allowed to break memory coherency of its memory and run alias-driven cache attacks. The main idea to protect integrity against such attacks is to (re)establish coherency for all memory touched by the trusted process. For confidentiality, the idea is to adapt the code of the victim in a way that its execution leaks no additional information to the attacker through the cache channel. Interestingly, many of the techniques described below are suitable for both purposes, neutralizing undesirable side effects of using the caches.

1) Complete Cache Flush

One of the traditional means to tackle cache side channels is to flush all instruction and data caches before executing trusted code. In this way, all aliases in the cache are either written back to memory (in case they are dirty) or simply removed from the cache (in case they are clean). Any kind of priming of the caches by the attacker becomes ineffective since all his cache entries are evicted by the trusted process, foiling any subsequent probing attempts using addresses with mismatched cacheability. Similarly, all input data the victim reads from the attacker's memory are obtained from coherent main memory due to the flush, thus thwarting alias-driven attacks on integrity.

A possible correctness proof that flushing all caches eliminates the information side channel would rely on the assertion that, after the execution of the trusted service, an attacker will always make the same observation using mismatched aliases, i.e., that all incoherent lines were evicted from the cache. Thus he cannot infer any additional knowledge from the cache storage channel. Note, that here it suffices to flush the caches before returning to the attacker, but to protect against the integrity threat, data caches need to be flushed before any input data from the attacker is read.

For performance evaluation the flushing approach was implemented in the AES and hypervisor examples. At each call of an AES encryption or hypervisor function, all data and instruction caches are flushed completely. Naturally this introduces an overhead for the execution of legitimate guest code due to an increased cache miss rate after calls to trusted processes. At the same time the trusted process gets slowed down for the same reason, if normally some of its data and instructions were still allocated in the caches from a previous call. Additionally the flushing itself is often expensive, e.g., for ARM processors the corresponding code has to traverse all cache lines in all ways and levels of cache to flush them individually. That all these overheads can add up to a sizeable delay of even one order of magnitude is clearly demonstrated by the benchmarks given in Tables II and I.

2) Cache Normalization

Instead of flushing, the victim can eliminate the cache information side channel by reading a sequence of memory cells so that the cache is brought into a known state. For instruction caches the same can be achieved by executing a sequence of jumps that are allocated at a set of memory locations mapping to the cache lines to be evicted. In the context of timing channels this process is called normalization. If subsequent memory accesses only hit the normalized cache lines, the attacker cannot observe the memory access pattern of the victim, because the victim always evicts the same lines. However the correctness of this approach strongly depends on the hardware platform used and the replacement policy of its caches. In case several memory accesses map to the same cache line the normalization process may in theory evict lines that were loaded previously. Therefore, in the verification a detailed cache model is needed to show that all memory accesses of the trusted service hit the cache ways touched during normalization.

3) Selective Eviction

The normalization method shows that cache side effects can be neutralized without evicting the whole cache. In fact, it is enough to focus on selected cache lines that are critical for integrity or confidentiality. For example, the integrity threat on the hypervisor can be eliminated by evicting the cache lines corresponding to the page table provided by the attacker.

The flushing or normalization establishes memory coherency for the hypervisor's inputs, thus making sure it validates the right data. The method of selective flushing was implemented for the hypervisor scenario and benchmark results in Table I show, as one would expect, that it is more efficient than flushing the whole cache, but still slower than our specialized ACPT solution.

To ensure confidentiality in the AES example it suffices to evict the cache lines occupied by the SBoxes. Since the incoherent entries placed in the same cache lines are removed by the victim using flushing or normalization, the attacker subsequently cannot measure key-dependent data accesses to these cache lines. For the modular exponentiation example the same technique can be used, evicting only the lines in the instruction cache where the code of the functions `Mult` and `ModReduce` is mapped.

The correctness of selective eviction of lines for confidentiality depends on the fact that accesses to other lines do not leak secret information through the cache side channel, e.g., for the AES encryption algorithm lines that are not mapped to an SBox are accessed in every computation, independent of the value of the secret key. Clearly, this kind of trace property needs to be added as a verification condition on the code of the trusted service. Then the classic confidentiality property can be established, that observations of the attacker are the same in two computations where only the initial values of the secret are different (non-infiltration [23]).

4) Secret-Independent Memory Accesses

The last method of eliminating the cache information side channel is a special case of this approach. It aims to transform the victim's code such that it produces a memory access trace that is completely independent of the secret, both for data accesses and instruction fetches. Consequently, there is no need to modify the cache state set up by the attacker, it will be transformed in the same way even for different secret values, given the trusted service receives the same input parameters and all hidden states in the service or the cache model are part of the secret information.

As an example we have implemented a modification of AES suggested in [51], where a 1KB SBox look-up table is scrambled in such a way that a look-up needs to touch all cache lines occupied by the SBox. In our implementation on Raspberry Pi 2 each L1 cache line consists of 64 Bytes, hence a 32bit entry is spread over 16 lines where each line contains two bits of the entry. While the decision which 2 bits from every line are used is depending on the secret AES key, the attacker only observes that the encryption touches the 16 cache lines occupied by the SBox, hence the key is not leaked.

Naturally the look-up becomes more expensive now because a high number of bitfield and shift operations is required to reconstruct the original table entry. For a single look-up, a single memory access is substituted by 16 memory accesses, 32 shifts, 16 additions and 32 bitfield operations. The resulting overhead is roughly 50% if only the last box is scrambled (see Table II). This is sufficient if all SBoxes are mapped to the same cache lines and the attacker cannot interrupt the trusted service, probing the intermediate cache state. Scrambling all SBoxes seems prohibitively expensive though, slowing the encryption down by an order of magnitude. However, since the number of operations depends on the number of lines used to store the SBox, if the system has bigger cache lines the countermeasure becomes cheaper.

5) Reducing the Channel Bandwidth

Finally for the AES example there is a countermeasure that does not completely eliminate the cache side channel, but makes it harder for the attacker to derive the secret key. The idea described in [51] is to use a more compact SBox that can be allocated on less lines, undoing an optimization in wolfSSL for the last round of AES. There the look-up only needs to retrieve one byte instead four, still the implementation word-aligns these bytes to avoid bit masking and shifting. By byte-aligning the entries again, the table shrinks by a factor of four, taking up four lines instead of 16 on Raspberry Pi 2. Since the attacker can distinguish less entries by the cache line they are allocated on, the channel leaks less information. This theory is confirmed in practice where retrieving the AES key required about eight times as many encryptions compared to the original one. At the same time, the added complexity resulted in a performance delay of roughly 23% (see Table II).

6) Detecting memory incoherency

A reference monitor (e.g. the hypervisor) can counter the integrity threat by preventing the invocation of the critical functions (e.g. the MMU virtualization functions) if memory incoherency is detected. The monitor can itself use mismatched cache attributes to detect incoherency as follows. For every address that is used as the input of a critical function, the monitor checks if reading the location using the cacheable and non-cacheable aliases yield the same result. If the two reads differs, then memory incoherency is detected and the monitor rejects the request, otherwise then request is processed.

D. Hardware based countermeasures

The cache-based storage channels rely on misbehaviour of the system due to misconfigurations. For this reason, the hardware could directly take care of them. The vector based on mismatched cacheability attributes can be easily made ineffective if the processor does not ignore unexpected cache hits. For example, if a physical address is written using a non-cacheable alias, the processor can invalidate every line having the corresponding tag. Virtually indexed caches are usually equipped with similar mechanisms to guarantee that there can not be aliases inside the cache itself.

Hardware inhibition of the vector that uses the instruction cache can be achieved using a snooping mechanism that invalidates instruction cache lines whenever self-modification is detected, similar to what happens in x64 processors. In architectures that perform weakly ordered memory accesses and aggressive speculative execution, implementing such a mechanism can become quite complex and make the out-of-order execution logic more expensive. There is also a potential slow-down due to misspeculation when instructions are fetched before they are overwritten.

Overall, the presented countermeasures show that a trusted service can be efficiently secured against alias-driven cache attacks if two properties are ensured: (1) for integrity, the trusted service may only accesses coherent memory (2) for confidentiality, the cache must be transformed in a way such that the attacker cannot observe memory accesses depending on secrets. In next section, a verification methodology presented that aims to prove these properties for the code of the trusted service.

VI. Verification Methodology

The attacks presented in Section IV demonstrate that the presence of caches can make a trustworthy, i.e. formally verified, program vulnerable to both confidentiality and security threats. These vulnerabilities depend on the fact that for some resources (i.e. some physical addresses of the memory) the actual system behaves differently from what is predicted by the formal model: we refer to this misbehaviour as "loss of sequential consistency".

As basis for the study we assume a low level program (e.g. a hypervisor, a separation kernel, a security monitor, or a TrustZone crypto-service) running on a commodity CPU such as the ARMv7 Cortex A7 of Raspberry Pi 2. We refer to the trusted program as "the kernel". The kernel shares the system with an untrusted application, henceforth "the application". We assume that the kernel has been subject to a pervasive formal verification that established its functional correctness and isolation properties using a model that reflects the ARMv7 ISA specification to some level of granularity. For instance for both seL4 and the Prosper kernel the processor model is based on Anthony Fox's cacheless L3 model of ARMv7 [2].

We identify two special classes of system resources (read: Memory locations):

- Critical resources: These are the resources whose integrity must be protected, but which the application needs access to for its correct operation.
- Confidential resources: These are the resources that should be read protected against the application.

There may in addition be resources that are both critical and confidential. We call those *internal* resources. Examples of critical resources are the page tables of a hypervisor, the executable code of the untrusted software in a run-time monitor, and in general the resources used by the invariants needed for the verification of functional correctness. Confidential (internal) resources can be cryptographic keys, internal kernel data structures, or the memory of a guest colocated with the application.

The goal is to repair the formal analysis of the kernel, reusing as much as possible of the prior analysis. In particular, our goals are:

1) To demonstrate that critical and internal resources cannot be directly affected by the application and that for these resources the actual system behaves according to the formal specification (i.e. that sequential consistency is preserved and the integrity attacks described in Section III-B cannot succeed).
2) To guarantee that no side channel is present due to caches, i.e. that the real system exposes all and only the channels that are present in the formal functional specification that have been used to verify the kernel using the formal model.

A. Repairing the Integrity Verification

For simplicity, we assume that the kernel accesses all resources using cacheable virtual addresses. To preserve integrity we must ensure two properties:

- That an address belonging to a critical resource cannot be directly or indirectly modified by the application.
- Sequential consistency of the kernel.

The latter property is equivalent to guaranteeing that what is observed in presence of caches is exactly what is predicted by the ISA specification.

The verification depends on a system invariant that must be preserved by all executions: For every address that belongs to the critical and internal resources, if there is a cache hit and the corresponding cache line differs from the main memory then the cache line must be dirty. The mechanism used to establish this invariant depends on the specific countermeasure used. It is obvious that if the caches are disabled (Section V-A) the invariant holds, since the caches are always empty. In the case of "Always Cacheable Memory" (Section V-B) the invariant is preserved because no non-cacheable alias is used to access these resources: the content of the cache can differ from the content of the memory only due to a memory update that changed the cache, thus the corresponding cache line is dirty. Similar arguments apply to the $C \oplus U$ Policy, taking into account that the cache is cleaned whenever a resource type switch from cacheable (C) to uncacheable (U) and vice versa.

More complex reasoning is necessary for other countermeasures, where the attacker can build uncacheable aliases in its own memory. In this case we know that the system is configured so that the application cannot write the critical resources, since otherwise the integrity property cannot be established for the formal model in the first place. Thus, if the cache contains critical or internal data different from main memory it must have been written there by the kernel that only uses cacheable memory only, hence the line is dirty as well.

To show that a physical address pa belonging to a critical resource cannot not be directly or indirectly modified by the application we proceed as follows. By the assumed formal verification, the application has no direct writable access to pa, otherwise the integrity property would not have been established at the ISA level. Then, the untrusted application can not directly update pa neither in the cache nor in the memory. The mechanism that can be used to indirectly update the view of the kernel of the address pa consists in evicting a cache line that has a value for pa different from the one

[2] In case of Prosper, augmented with a detailed model of the MMU [33].

stored in the memory and that is not dirty. However, this case is prevented by the new invariant.

Proving that sequential consistency of the kernel is preserved is trivial: The kernel always uses cacheable addresses so it is unable to break the new invariant: a memory write always updates the cache line if there is a cache hit.

B. Repairing the Confidentiality Verification

Section III demonstrates the capabilities of the attacker: Additionally to the resources that can be accessed in the formal model (registers, memory locations access to which is granted by the MMU configuration, etc) the attacker is able to measure which cache lines are evicted. Then the attacker can (indirectly) observe all the resources that can affect the eviction. Identifying this set of resources is critical to identify the constraints that must be satisfied by the trusted kernel. For this reason, approximating this set (e.g. by making the entire cache observable) can strongly reduce the freedom of the trusted code. A more refined (still conservative) analysis considers observable by the attacker the cache line tag[3] and whether a cache line is empty (cache line emptiness). Then to guarantee confidentiality it is necessary to ensure that, while the application is executing, the cache line tag and emptiness never depend on the confidential resources. We stress that this is a sufficient condition to guarantee that no additional information is leaked due to presence of caches with respect to the formal model

Showing that the condition is met by execution of the application is trivial. By the assumed formal verification we already know that the application has no direct read access (e.g. through a virtual memory mapping) to confidential resources. On the other hand, the kernel is able to access these resources, for example to perform encryption. The goal is to show that the caches do not introduce any channel that has not been taken into account at the level of the formal model. Due to the overapproximation described above, this task is reduced to a "cache-state non-interference property", i.e. showing that if an arbitrary functionality of the kernel is executed then the cache line emptiness and the line tags in the final state do not depend on confidential data.

The analysis of this last verification condition depends on the countermeasure used by the kernel. If the kernel always terminates with caches empty, then the non-interference property trivially holds, since a constant value can not carry any sensible information. This is the case if the kernel always flushes the caches before exiting, never use cacheable aliases (for both program counter and memory accesses) or the caches are completely disabled.

In other cases (e.g. "Secret-Independent Memory Accesses" and "Selective Eviction") the verification condition is further decomposed to two tasks:

1) Showing that starting from two states that have the same cache states, if two programs access at the same time the same memory locations then the final states have the same cache states.
2) Showing that the sequence of memory accesses performed by the kernel only depends on values that are not confidential.

The first property is purely architectural and thus independent of the kernel. Hereafter we summarise the reasoning for a system with a single level of caches, with separated instruction and data caches and whose caches are physically indexed and physically tagged (e.g. the L1 memory subsystem of ARMv7 CPUs). We use s_1, s_1', s_2, s_2' to range over machine states and $s_1 \rightarrow s_1'$ to represent the execution of a single instruction. From an execution $s_1 \rightarrow s_2 \cdots \rightarrow s_n$ we define two projections: $\pi_I(s_1 \rightarrow s_2 \cdots \rightarrow s_n)$ is the list of encountered program counters and $\pi_D(s_1 \rightarrow s_2 \cdots \rightarrow s_n)$ is the list of executed memory operations (type of operation and physical address). We define $\mathcal{P}$ as the biggest relation such that if $s_1 \; \mathcal{P} \; s_2$ then for both data and instruction cache

- a line in the cache of s_1 is empty if and only if the same line is empty in s_2, and
- the caches of s_1 and s_2 have the same tags for every line.

The predicate $\mathcal{P}$ is preserved by executions $s_1 \rightarrow \ldots$ and $s_2 \rightarrow \ldots$ if the corresponding projections are *cache safe*: (i) the instruction tag and index of $\pi_I(s_1 \rightarrow \ldots)[i]$ is equal to the instruction tag and index of $\pi_I(s_2 \rightarrow \ldots)[i]$ (ii) if $\pi_D(s_1 \rightarrow \ldots)[i]$ is a read (write) then $\pi_D(s_2 \rightarrow \ldots)[i]$ is a read (write) (iii) the cache line tag and index of the address in $\pi_D(s_1 \rightarrow \ldots)[i]$ is equal to the cache line tag and index of the address in $\pi_D(s_2 \rightarrow \ldots)[i]$

Consider the example in Figure 1, where va_3 and va_4 are different addresses. In our current setting this is secure only if va_3 and va_4 share the same data cache index and tag (but they could point to different positions within a line). Similarly, the example in Figure 4 is secure only if the addresses of both targets of the conditional branch have the same instruction cache index and tag. Notice that these conditions are less restrictive than the ones imposed by the program counter security model. Moreover, these restrictions dot not forbid completely data-dependent look-up tables. For example, the scrambled implementation of AES presented In Section V-C satisfies the rules that we identified even if it uses data-dependent look-up tables.

In practice, to show that the trusted code satisfies the cache safety policy, we rely on a relational observation equivalence and we use existing tools for relational verification that support trace based observations. In our experiments we adapted the tool presented in [6]. The tool executes two analyses of the code. The first analysis handles the instruction cache: we make every instruction observable and we require that the matched instructions have the same set index and tag for the program counter. The second analysis handles the data cache: we make every memory access an observation and we require that the matched memory accesses use the same set index and tag

[3] On direct mapped caches, we can disregard the line tag, because they contain only one way for each line. In order to observe the tags of addresses accessed by the kernel, the attacker requires at least two ways per cache line: one that contains an address accessible by the kernel and one that the attacker can prime in order to measure whether the first line has been accessed.

(originally the tool considered observable only memory writes and required that the matched memory writes access the same address and store the same value). Note that the computation of set index and tag are platform-dependent, thus when porting the same verified code to a processor, whose caches use a different method for indexing lines, the code might not be cache safe anymore. To demonstrate the feasibility of our approach we applied the tool to one functionality of the hypervisor described in Section IV-B, which is implemented by 60 lines of assembly and whose analysis required 183 seconds.

VII. Related work

Kocher [31] and Kelsey et al. [27] were the first to demonstrate cache-based side-channels. They showed that these channels contain enough information to enable an attacker to extract the secret key of cryptographic algorithms. Later, Page formally studied cache side-channels and showed how one can use them to attack naïve implementations of the DES cryptosystem [36]. Among the existing cache attacks, the trace-driven and access-driven attacks are the most closely related to this paper since they can be reproduced using the vectors presented in Section III.

In trace-driven attacks [36] an adversary profiles the cache activities while the victim is executed. Acıiçmez showed a trace-driven cache attack on the first two rounds of AES [2], which has been later improved and extended by X. Zhao [56] to compromise a CLEFIA block cipher. A similar result is reported in [9]. In an access-driven, or Prime+Probe, attack the adversary can determine the cache sets modified by the victim. In several papers this technique is used to compromise real cryptographic algorithms like RSA [37], [25] and AES [22], [34], [47].

Due to the security concerns related to cache channels, research on the security implications of shared caches has so far been focusing on padding [54] and mitigation [3] techniques to address timing channels. Notably, Godfrey and Zulkernine have proposed efficient host-based solutions to close timing channels through selective flushing and cache partitioning [20]. In the StealthMem approach [28] each guest is given exclusive access to a small portion of the shared cache for its security critical computations. By ensuring that this stealth memory is always allocated in the cache, no timing differences are observable to an attacker.

In literature, few works investigated cache based storage channels. In fact, all implementations of the above attacks use timing channels as the attack vector. Brumley [11] recently conjectured the existence of a storage channel that can be implemented using cache debug functionality on some ARM embedded microprocessors. However, the ARM technical specification [15] explicitly states that such debug instructions can be executed only by privileged software in TrustZone, making practically impossible for an attacker to access them with the exception of a faulty hardware implementation.

The attack based on mismatched cacheability attributes opens up for TOCTTOU like vulnerabilities. Watson [50] demonstrated this vulnerability for Linux system call wrappers. A similar approach is used in [10] to invalidate security guarantees, attestation of a platform's software, provided by a Trusted Platform Module (TPM). TPM takes integrity measurements only before software is loaded into the memory, and it assumes that once the software is loaded it remains unchanged. However, this assumption is not met if the attacker can indirectly change the software before is used.

Cache-related architectural problems have been exploited before to bypass memory protection. In [52], [19] the authors use a weakness of some Intel x86 implementations to bypass SMRAM protection and execute malicious code in System Management Mode (SMM). The attack relies on the fact that the SMRAM protection is implemented by the memory controller, which is external to the CPU cache. A malicious operating system first marks the SMRAM memory region as cacheable and write-back, then it writes to the physical addresses of the SMRAM. Since the cache is unaware of the SMRAM configuration, the writes are cached and do not raise exceptions. When the execution is transferred to SMM, the CPU fetches the instructions from the poisoned cache. While this work shows similarities to the integrity threat posed by cache storage channels, the above attack is specific to certain Intel implementations and targets only the highest security level of x86. On ARM, the cache keeps track which lines have been filled due to accesses performed by TrustZone SW. The TrustZone SW can configure via its page tables the memory regions that are considered "secure" (e.g. where its code and internal data structure are stored). A TrustZone access to a secure memory location can hit a cache line only if it belongs to TrustZone.

The attack vectors for data caches presented in this paper abuse undefined behaviour in the ISA specification (i.e., accessing the same memory address with different cacheability types) and deterministic behaviour of the underlying hardware (i.e., that non-cacheable accesses completely bypass the data caches and unexpected cache hits are ignored). While we focused on an ARMv7 processor here, there is a strong suspicion that other architectures exhibit similar behaviour. In fact, in experiments we succeeded to replicate the behaviour of the memory subsystem on an ARMv8 processor (Cortex-A53), i.e., uncacheable accesses do not hit valid entries in the data cache. For Intel x64, the reference manual states that memory type aliases using the page tables and page attribute table (PAT) "may lead to undefined operations that can result in a system failure" ([26], Vol. 3, 11.12.4). It is also explicitly stated that the accesses using the (non-cacheable) WC memory type may not check the caches. Hence, a similar behaviour as on ARM processors should be expected. On the other hand, some Intel processors provide a self-snooping mechanism to support changing the cacheability type of pages without requiring cache flushes. It seems to be similar in effect as the hardware countermeasure suggested in Section V-D. In the Power ISA manual ([38], 5.8.2), memory types are assumed to be unique for all aliases of a given address. Nevertheless this is a software condition that is not enforced by the architecture.

When changing the storage control bits in page table entries the programmer is required to flush the caches. This also hints to the point that no hardware mechanisms are mandated to handle unexpected cache hits.

Recently, several works successfully verified low level execution platforms that provide trustworthy mechanisms to isolate commodity software. In this context caches are mostly excluded from the analysis. An exception is the work by Barthe et al. [8] that provide an abstract model of cache behaviour sufficient to replicate various timing-based exploits and countermeasures from the literature such as STEALTHMEM.

The verification of seL4 assumes that caches are correctly handled [29] and ignores timing channels. The bandwidth of timing channels in seL4 and possible countermeasures were examined by Cock et al [13]. While storage based channels have not been addressed, integrity of the kernel seems in practice to be preserved by the fact that system call arguments are passed through registers only.

The VerisoftXT project targeted the verification of Microsoft Hyper V and a semantic stack was devised to underpin the code verification with the VCC tool [14]. Guests are modelled as full x64 machines where caches cannot be made transparent if the same address is accessed in cacheable and uncacheable mode, however no implications on security have been discussed. Since the hypervisor uses a shadow page algorithm, where guest translations are concatenated with a secure host translation, the integrity properties do not seem to be jeopardised by any actions of the guest.

Similarly the Nova [45], [46] and CertiKOS [21] microvisors do not consider caches in their formal analysis, but they use hardware which supports a second level address translation which is controlled by the host and cannot be affected by the guest. Nevertheless the CertiKOS system keeps a partition management software in a separate partition that can be contacted by other guests via IPC to request access to resources. This IPC interface is clearly a possible target for attacks using uncacheable aliases.

In any case all of the aforementioned systems seem to be vulnerable to cache storage channel information leakage, assuming they allow the guest systems to set up uncacheable memory mappings. In order to be sound, any proof of information flow properties then needs to take the caches into account. In this paper we show for the first time how to conduct such a non-interference proof that treats also possible data cache storage channels.

VIII. CONCLUDING REMARKS

We presented novel cache based attack vectors that use storage channels and we demonstrated their usage to threaten integrity and confidentiality of real software. To the best of our knowledge, it is the first time that cache-based storage channels are demonstrated on commodity hardware.

The new attack vectors partially invalidate the results of formal verification performed at the ISA level. In fact, using storage-channels, the adversary can extract information without accessing variables that are external to the ISA specification. This is not the case for timing attacks and power consumption attacks. For this reason it is important to provide methodologies to fix the existing verification efforts. We show that for some of the existing countermeasures this task can be reduced to checking relational observation equivalence. To make this analysis practical, we adapted an existing tool [6] to check the conditions that are sufficient to prevent information leakage due to the new cache-channels. In general, the additional checks in the code verification need to be complemented by a correctness proof of a given countermeasure on a suitable cache model. In particular it has to be shown that memory coherency for the verified code is preserved by the countermeasure and that an attacker cannot observe sensitive information even if it can create non-cacheable aliases.

The attack presented in Section III-B raises particular concerns, since it poses integrity threats that cannot be carried out using timing channels. The possible victims of such an attack are systems where the ownership of memory is transferred from the untrusted agent to the trusted one and where the trusted agent checks the content of this memory before using it as parameter of a critical function. After that the ownership is transferred, if the cache is not clean, the trusted agent may validate stale input while the critical function uses different data. The practice of transferring ownership between security domains is usually employed to reduce memory copies and is used for example by hypervisors that use direct paging, run-time monitors that inspect executable code to prevent execution of malware, as well as reference monitors that inspect the content of IP packets or validate requests for device drivers.

There are several issues we leave out as future work. We did not provide a mechanism to check the security of some of the countermeasures like Cache Normalisation and we did not apply the methodology that we described to a complete software. Moreover, the channels that we identified probably do not cover all the existing storage side channels. Branch prediction, TLBs, sharebility attributes are all architectural details that, if misconfigured, can lead to behaviours that are inconsistent with the ISA specification. If the adversary is capable of configuring these resources, like in virtualized environments, it is important to identify under which conditions the trusted software preserves its security properties.

From a practical point of view, we focused our experiments on exploiting the L1 cache. For example, to extract the secret key of the AES service on Raspberry Pi 2 we have been forced to flush and clean the L2 cache. The reason is that on this platform the L2 cache is shared with the GPU and we have little to no knowledge about the memory accesses it performs. On the other hand, shared L2 caches open to the experimentation with concurrent attacks, where the attacker can use a shader executed on the GPU. Similarly, here we only treated cache channels on a single processor core. Nevertheless the same channels can be built in a multi-core settings using the shared caches (e.g. L2 on Raspberry Pi 2). The new vectors can then be used to replicate known timing attacks on shared caches (e.g. [25]).

Acknowledgment

The authors would like to thank Didrik Lundberg for supporting the development of the Raspberry Pi 2 prototypes. Work partially supported by the PROSPER framework grant from the Swedish Foundation for Strategic Research, by by the Swedish Governmental Agency for Innovation Systems under grant 2014-00702, and by the CERCES project funded by the Swedish Civil Contingencies Agency.

References

[1] ARM TrustZone. http://www.arm.com/products/processors/technologies/trustzone.php.

[2] O. Acıiçmez and c. K. Koç. Trace-driven cache attacks on AES (short paper). In *Proceedings of the 8th International Conference on Information and Communications Security*, ICICS'06, pages 112–121. Springer-Verlag, 2006.

[3] J. Agat. Transforming out timing leaks. In *Proceedings of the 27th Symposium on Principles of Programming Languages*, POPL '00, pages 40–53. ACM, 2000.

[4] E. Alkassar, M. A. Hillebrand, D. Leinenbach, N. Schirmer, A. Starostin, and A. Tsyban. Balancing the load. *J. Autom. Reasoning*, 42(2-4):389–454, 2009.

[5] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In *Proceedings of the Conference on Computer and Communications Security*, CCS'14, pages 90–102. ACM, 2014.

[6] M. Balliu, M. Dam, and R. Guanciale. Automating information flow analysis of low level code. In *Proceedings of the Conference on Computer and Communications Security*, CCS'14, pages 1080–1091. ACM, 2014.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *ACM Operating Systems Review*, 37(5):164–177, 2003.

[8] G. Barthe, G. Betarte, J. D. Campo, J. M. Chimento, and C. Luna. Formally verified implementation of an idealized model of virtualization. In *Proceedings of the 19th International Conference on Types for Proofs and Programs*, TYPES'13, pages 45–63, 2014.

[9] G. Bertoni, V. Zaccaria, L. Breveglieri, M. Monchiero, and G. Palermo. AES power attack based on induced cache miss and countermeasure. In *Proceedings of the International Conference on Information Technology: Coding and Computing*, ITCC'05, pages 586–591. IEEE Computer Society, 2005.

[10] S. Bratus, N. D'Cunha, E. Sparks, and S. W. Smith. TOCTOU, traps, and trusted computing. In *Proceedings of the 1st International Conference on Trusted Computing and Trust in Information Technologies: Trusted Computing - Challenges and Applications*, Trust'08, pages 14–32. Springer-Verlag, 2008.

[11] B. Brumley. Cache storage attacks. In *Topics in Cryptology CT-RSA*, pages 22–34. 2015.

[12] H. Chfouka, H. Nemati, R. Guanciale, M. Dam, and P. Ekdahl. Trustworthy prevention of code injection in linux on embedded devices. In *Proceedings of the 20th European Symposium on Research in Computer Security*, ESORICS'15, pages 90–107. Springer, 2015.

[13] D. Cock, Q. Ge, T. Murray, and G. Heiser. The Last Mile: An Empirical Study of Timing Channels on seL4. In *Proceedings of the Conference on Computer and Communications Security*, CCS'14, pages 570–581. ACM, 2014.

[14] E. Cohen, W. Paul, and S. Schmaltz. Theory of multi core hypervisor verification. In *39th International Conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM'2013, pages 1–27. Springer, 2013.

[15] Cortex-A7 mpcore processors. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.cortexa.cortexa7.

[16] Cortex-A8 processors. http://infocenter.arm.com/help/topic/com.arm.doc.subset.cortexa.a8.

[17] J. Daemen and V. Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[18] M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz. Formal verification of information flow security for a simple ARM-based separation kernel. In *Proceedings of the Conference on Computer and Communications Security*, CCS'13, pages 223–234. ACM, 2013.

[19] L. Duflot, O. Levillain, B. Morin, and O. Grumelard. Getting into the SMRAM: SMM reloaded. *CanSecWest*, 2009.

[20] M. M. Godfrey and M. Zulkernine. Preventing cache-based side-channel attacks in a cloud environment. *IEEE T. Cloud Computing*, 2(4):395–408, 2014.

[21] L. Gu, A. Vaynberg, B. Ford, Z. Shao, and D. Costanzo. CertiKOS: a certified kernel for secure cloud computing. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys'11, page 3. ACM, 2011.

[22] D. Gullasch, E. Bangerter, and S. Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *Proceedings of the Symposium on Security and Privacy*, SP'11, pages 490–505. IEEE Computer Society, 2011.

[23] C. Heitmeyer, M. Archer, E. Leonard, and J. McLean. Applying formal methods to a certifiably secure software system. *IEEE Trans. Softw. Eng.*, 34(1):82–98, Jan. 2008.

[24] M. A. Hillebrand, T. I. der Rieden, and W. J. Paul. Dealing with i/o devices in the context of pervasive system verification. In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors*, ICCD'05, pages 309–316. IEEE, 2005.

[25] M. S. Inci, B. Gülmezoglu, G. I. Apecechea, T. Eisenbarth, and B. Sunar. Seriously, get off my cloud! cross-vm RSA key recovery in a public cloud. *IACR Cryptology ePrint Archive*, 2015:898, 2015.

[26] Intel 64 and IA-32 Architectures Software Developer's Manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[27] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Side channel cryptanalysis of product ciphers. *J. Comput. Secur.*, 8(2,3):141–158, Aug. 2000.

[28] T. Kim, M. Peinado, and G. Mainar-Ruiz. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *USENIX*, pages 189–204, 2012.

[29] G. Klein, J. Andronick, K. Elphinstone, T. C. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.*, 32(1):2, 2014.

[30] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd Symposium on Operating Systems Principles*, SOSP'09, pages 207–220. ACM, 2009.

[31] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 104–113. Springer-Verlag, 1996.

[32] D. Leinenbach and T. Santen. Verifying the microsoft hyper-v hypervisor with VCC. In *Proceedings of the 2Nd World Congress on Formal Methods*, FM '09, pages 806–809. Springer-Verlag, 2009.

[33] H. Nemati, R. Guanciale, and M. Dam. Trustworthy virtualization of the armv7 memory subsystem. In *Proceedings of the 41st International Conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM'15, pages 578–589. Springer, 2015.

[34] M. Neve and J.-P. Seifert. Advances on access-driven cache attacks on AES. In *Proceedings of the 13th International Conference on Selected Areas in Cryptography*, SAC'06, pages 147–162. Springer-Verlag, 2007.

[35] OpenSSL. https://www.openssl.org.

[36] D. Page. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptology ePrint Archive*, 2002:169, 2002.

[37] C. Percival. Cache missing for fun and profit. *BSDCan*, 2005.

[38] Power ISA version 2.07. https://www.power.org/wp-content/uploads/2013/05/PowerISA_V2.07_PUBLIC.pdf.

[39] N. Qu, X. Gou, and X. Cheng. Using uncacheable memory to improve unity linux performance. In *Proceedings of the 6th Annual Workshop on the Interaction between Operating Systems and Computer Architecture*, pages 27–32, 2005.

[40] H. Raj, R. Nathuji, A. Singh, and P. England. Resource management for isolation enhanced cloud services. In *Proceedings of the Workshop on Cloud Computing Security*, CCSW '09, pages 77–84. ACM, 2009.

[41] R. Richards. Modeling and security analysis of a commercial real-time operating system kernel. In D. S. Hardin, editor, *Design and Verification*

of Microprocessor Systems for High-Assurance Applications, pages 301–322. Springer US, 2010.

[42] Raspberry Pi 2 Model B. https://www.raspberrypi.org/products/raspberry-pi-2-model-b/.

[43] S. M. Silver. Implementation and analysis of software based fault isolation. Technical Report PCS-TR96-287, Dartmouth College, 1996.

[44] D. Stefan, P. Buiras, E. Yang, A. Levy, D. Terei, A. Russo, and D. Mazires. Eliminating cache-based timing attacks with instruction-based scheduling. In *Proceedings of the 18th European Symposium on Research in Computer Security*, ESORICS'13, pages 718–735. Springer, 2013.

[45] U. Steinberg and B. Kauer. NOVA: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 209–222. ACM, 2010.

[46] H. Tews, M. Völp, and T. Weber. Formal memory models for the verification of low-level operating-system code. *J. Autom. Reasoning*, 42(2-4):189–227, 2009.

[47] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on AES, and countermeasures. *J. Cryptol.*, 23(2):37–71, Jan. 2010.

[48] Y. Tsunoo, T. Saito, T. Suzaki, and M. Shigeri. Cryptanalysis of DES implemented on computers with cache. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems*, CHES'03, LNCS, pages 62–76. Springer, 2003.

[49] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *SIGOPS Oper. Syst. Rev.*, 27(5):203–216, Dec. 1993.

[50] R. N. M. Watson. Exploiting concurrency vulnerabilities in system call wrappers. In *Proceedings of the First USENIX Workshop on Offensive Technologies*, WOOT'07, pages 2:1–2:8. USENIX Association, 2007.

[51] M. Weiß, B. Heinz, and F. Stumpf. Proceedings of the 16th international conference on financial cryptography and data security. FC'2012, pages 314–328. Springer, 2012.

[52] R. Wojtczuk and J. Rutkowska. Attacking SMM memory via intel cpu cache poisoning. *Invisible Things Lab*, 2009.

[53] wolfSSL: Embedded SSL Library. https://www.wolfssl.com/wolfSSL/Home.html.

[54] D. Zhang, A. Askarov, and A. C. Myers. Language-based control and mitigation of timing channels. In *Proceedings of the 33rd Conference on Programming Language Design and Implementation*, PLDI '12, pages 99–110. ACM, 2012.

[55] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the Conference on Computer and Communications Security*, CCS '12, pages 305–316. ACM, 2012.

[56] X.-J. Zhao and T. Wang. Improved cache trace attack on AES and CLEFIA by considering cache miss and S-box misalignment. *IACR Cryptology ePrint Archive*, 2010:56, 2010.