Formal Analysis of Countermeasures against Cache Storage Side Channels

Hamed Nemati, Roberto Guanciale, Christoph Baumann and Mads Dam

ABSTRACT

Formal verification of systems-level software such as hypervisors and operating systems can enhance system trustworthiness. However, without taking low level features like caches into account the verification may become unsound. While this is a well-known fact wrt. timing leaks, few works have addressed latent cache storage side-channels. We present a verification methodology to analyse soundness of countermeasures used to neutralise cache storage channels. We apply the proposed methodology to existing countermeasures, showing that they allow to restore integrity and prove confidentiality of the system. We decompose the proof effort into verification conditions that allow for an easy adaption of our strategy to various software and hardware platforms. As case study, we extend the verification of an existing hypervisor whose integrity can be tampered using cache storage channels. We used the HOL4 theorem prover to validate our security analysis, applying the verification methodology to formal models of ARMv7 and ARMv8.

CCS CONCEPTS

• **Security and privacy** → Formal security models;Logic and verification;Information flow control;

KEYWORDS

Formal verification; Information Flow Security; Cache storage side-channel

1 INTRODUCTION

Formal verification of low-level software such as microkernels, hypervisors, and drivers has made big strides in recent years [3, 4, 16, 23, 28, 43, 50, 51]. We appear to be approaching the point where the promise of provably secure, practical system software is becoming a reality. However, existing verification use models that are far simpler than contemporary state-of-the-art hardware. Many features pose significant challenges: Memory models, pipelines, speculation, out-of-order execution, peripherals, and various coprocessors, for instance for system management. In a security context, caches are notorious. They have been known for years to give rise to timing side channels that are difficult to fully counteract [22]. Also, cache management is closely tied to memory management, which—since it governs memory mapping, access control, and cache configuration through page-tables residing in memory—is one of the most complex and security-critical components in the computer architecture flora.

Computer architects strive to hide this complexity from application programmers, but system software, device drivers, and high performance software for which tuning of cache usage is critical need explicit control over features like cacheability attributes. In virtualization scenarios, for instance, it is critical for performance to be able to delegate cache management authority for pages belonging to a guest OS to the guest itself. With such a delegated authority a guest is free to configure its share of the memory system as it wishes, including configurations that may break conventions normally expected for a well-behaved OS. For instance, a guest OS will usually be able to create memory aliases and to set cacheability attributes as it wishes. Put together, these capabilities can, however, give rise to memory incoherence, since the same physical address can now be pointed to by two virtual addresses, one to cache and one to memory. This opens up for cache storage attacks on both confidentiality and integrity, as was shown in [26]. Similarly to cache timing channels that use variations in execution time to discover hardware hidden state, storage channels use aliasing to profile cache activities and to attack system confidentiality. However, while the timing channels are external to models used for formal analysis and do not invalidate verification of integrity properties, storage channels simply make the models unsound: Using them for security analysis can lead to conclusions that are false.

This shows the need to develop verification frameworks for low-level system software that are able to adequately reflect the presence of caches. It is particularly desirable if this can be done in a manner that allows to reuse existing verification tools on simpler models that do not consider caches. This is the goal we set ourselves in this paper. We augment an existing cacheless model by adding a cache and accompanying cache management functionality in MMU and page-tables. We use this augmented model to derive proof obligations that can be imposed to ensure absence of both integrity and confidentiality attacks. This provides a verification framework that we use to analyse soundness of countermeasures. The countermeasures are formally modelled as new proof obligations that can be analysed on the cacheless model to ensure absence of vulnerabilities due to cache storage channels. Since these obligations can be verified using the cacheless model, existing tools [6, 12, 42] (mostly not available on a cache enabled model) can automate this task to a large extent. We then complete the paper by repairing the verification of an existing and vulnerable hypervisor [27], sketching how the derived proof obligations are discharged.

2 RELATED WORK

Formal Verification Existing work on formal verification do not takes into account cache storage channels. The verification of seL4 assumes a sequential memory model and leaves cache issues to be managed by model external means [33, 34]. Cock et al [14] examined the bandwidth of timing channels in seL4 and possible countermeasures including cache coloring. The verification of the Prosper kernel [16, 27] assumes that caches are invisible and correctly handled. Similarly Barthe et al. [9] ignores caches for the verification of an isolation property for an idealised hypervisor. Later, in [10] the authors extended the model to include an abstract

```
V1) D = access(va1)
A1) write(va2,1);
    free(va2)
V2) D = access(va1)
V3) if not policy(D)
       reject
    [evict va1]
V4) use(va1)
```

(a) Integrity

```
A1) write(va1, 1)
A2) write(va2, 0)
V1) if secret
       access(va3)
    else
       access(va4)

A3) D = access(va2)
```

(b) Confidentiality

Figure 1: Mismatched memory attribute threats

account of caches and verified that timing channels are neutralised by cache flushing. The CVM framework [4] treats caches only in the context of device management [29]. Similarly, the Nova [11, 43, 44] and CertiKOS [23–25] microvisors do not consider caches in their formal analysis. In a follow-up paper [15] the verification was extended to machine code level, using a sequential memory model and relying on the absence of address aliasing.

In scenarios such as OS virtualization, where untrusted software is allowed to configure cacheability of its own memory, all of the above systems can be vulnerable to cache storage channel attacks. For instance, these channels can be used to create illicit information flows among threads of seL4.

Timing Channels Timing attacks and countermeasures have been formally verified to varying degrees of detail in the literature. Almeida et al. [5] prove functional correctness and leakage security of MEE-CBC in presence of timing attackers. Barthe et al. [8] provide an abstract model of cache behaviour sufficient to replicate various timing-based exploits and countermeasures from the literature such as StealthMEM [32]. FlowTracker [41] detects leaks using an information flow analysis at compile time. Similarly, Ford et al. [23] uses information flow analysis based on explicit labeling to detect side-channels, and Vieira uses a deductive formal approach to guarantee that side-channel countermeasures are correctly deployed [47]. Other related work includes those adopting formal analysis to either check the rigour of countermeasures [18, 26, 45] or to examine bandwidth of side-channels [17, 35]. Zhou [52] proposed page access based solutions to mitigate the access-driven cache attacks and used model checking to show these countermeasure restore security. Illicit information flows due to caches can also be countered by masking timing fluctuations by noise injection [48] or by clock manipulation [30, 37, 46]. A extensive list of protective means for timing attacks is given in [21, 49].

By contrast, we tackle storage channels. These channels carry information through memory and, additionally to permit illicit information flows, can be used to compromise integrity. Storage channels have been used in [26] to show how cache management features could be used to attack both integrity and confidentiality of several types of application.

3 THREATS AND COUNTERMEASURES

The presence of caches and ability to configure cacheability of virtual alias enable the class of attacks called "alias-driven attacks" [26]. These attacks are based on building virtual aliases with mismatched cacheability attributes to break memory coherence; i.e, causing inconsistency between the values stored in a memory location and the corresponding cache line, without making the cache line dirty. We present here two examples to demonstrate how integrity and confidentiality can be attacked using this vector.

3.1 Integrity

Figure 1a demonstrates an integrity threat. Here, we assume the cache is direct-mapped, physically indexed and write-back. Also, both the attacker and victim are executed interleaved on a single core. Virtual addresses $va1$ and $va2$ are aliasing the same memory pa, $va1$ is cacheable and $va2$ is uncacheable. Initially, the memory pa contains the value 0 and the corresponding cache line is empty. In a sequential model reads and writes are executed in order and their effects are instantly visible: V1) a victim accesses $va1$, reading 0; A1) the attacker writes 1 into pa using $va2$ and releases the alias $va2$; V2) the victim accesses again $va1$, this time reading 1; V3) if 1 does not respect a security policy, then the victim rejects it; otherwise V4) the victim passes 1 to a security-critical functionality.

On a CPU with a weaker memory model the same code behaves differently: V1) using $va1$, the victim reads 0 from the memory and fills the cache; A1) the attacker uses $va2$ to directly write 1 in memory, bypasses the cache, and then frees the mapping; V2) the victim accesses again $va1$, reading 0 from the cache; V3) the security policy is evaluated based on 0; possibly, the cache line is evicted and, since it is not dirty, the memory is not affected; V4) next time the victim accesses pa it reads 1, but 1 is not the value that has been checked against the security policy. This permits the attacker to bypass the policy. [1]

Intuitive countermeasure against alias-driven attacks are to forbid the attacker from allocating cacheable aliases at all or to make cacheable its entire memory. A lightweight specialization of these approaches is "always cacheability": A fixed region of memory is made always cacheable and the victim rejects any input pointing outside this region. Coherency can also be achieved by flushing the entire cache before the victim accesses the attacker memory. Unfortunately, this countermeasure comes with severe performance penalties [26]. "Selective eviction" is a more efficient solution and consists in removing from the cache every location that is accessed by the victim and that has been previously accessed by the attacker. Alternatively, the victim can use mismatched cache attributes itself to detect memory incoherence and abort dangerous requests.

3.2 Confidentiality

Figure 1b shows a confidentiality threat. Both va_1 and va_2 point to the location pa and say idx is the cache line index of pa. All virtual addresses except va_2 are cacheable, and we assume that both pa and the physical address pointed by va_3 are allocated in the same cache line. The attacker writes A1) 1 in the cache, making the line dirty, and A2) 0 in the memory. From this point, the value stored in the memory after the execution of the victim depends on the victim behaviour; if the victim accesses at least one address (e.g. va_3) whose line index is idx, then the dirty line is evicted and 1 is written back to the memory; otherwise the line is not evicted

[1]Note that the attacker release its alias $va2$ before returning the control to the victim, making this attack different from the standard double mapping attacks.

and pa still contains 0. This allows the attacker to measure evicted lines and thus to launch an access-driven attack. In the following we summarise some of the countermeasures against cache storage channels presented in [26] and relevant to our work.

Similarly, to ensure that no secret can leak through the cache storage channel, one can forbid allocating cacheable aliases or always flush the cache after executing victim functionalities. An alternative is cache partitioning, where each process gets a dedicated part of the cache and there is no intersection between any two partitions. This makes impossible for the victim activities to affect by the attacker behaviour, thus preventing the attacker to infer information about victim internal variables. A further countermeasure is secret independent memory accesses, which aims at transforming the victim's code so that the victim accesses do not depend on secret.

Cache normalisation can also be used to close storage channels. In this approach, the cache is brought to a known state by reading a sequence of memory cells. This guarantees the subsequent secret dependent accesses only hit the normalised cache lines, preventing the attacker from observing access patterns of the victim.

4 HIGH-LEVEL SECURITY PROPERTIES

In this work we consider a trusted system software (the "kernel") that shares the system with an untrusted software (the "application"). Possible instances for the kernel include hypervisors, runtime monitors, low level operating system routines, and cryptographic services. The application is a software that requests services from the kernel and can be a user process or even a complete operating system. The hardware execution mode used by the application is less privileged than the mode use by the kernel. The application is potentially malicious and takes the role of the attacker here.

Some system resources are owned by the kernel and are called "critical", some other resources should not be disclosed to the application and are called "confidential". The kernel dynamically tracks memory ownership and provides mechanisms for secure ownership transfer. This enables the application to pass data to the kernel services, while avoiding expensive copy operations: the application prepares the input inside its own memory, the ownership of this memory is transferred to the kernel, and the corresponding kernel routine operates on the input in-place. Two instances of this are the direct-paging memory virtualization mechanism introduced by Xen [7] and runtime monitors that forbid self modifying code and prevent execution of unsigned code [13]. In these cases, predictability of the kernel behaviour must be ensured, regardless of any incoherent memory configuration created by the application.

In such a setting, a common approach to formalise security is via an integrity and a confidentiality property. We use $s \in \mathbb{S}$ to represent a state of the system and $\rightsquigarrow$ to denote a transition relation. The transition relation models the execution of one instruction by the application or the execution of a complete handler of the kernel. The integrity property ensures functional correctness (by showing that a state invariant I is preserved by all transitions) and that the critical resources can not be modified by the application (by showing a relation ψ):

Property 4.1 (Correctness). *For all s if $I(s)$ and $s \rightsquigarrow s'$ then $I(s')$ and $\psi(s, s')$*

The confidentiality property ensures that confidential resources are not leaked to the application and is expressed using standard non-interference. Let O be the application's observations (i.e. the resources that are not confidential) and let $\sim_O$ be observational equivalence (which requires states to have the same observations), then confidentiality is expressed by the following property

Property 4.2 (Confidentiality). *Let s_1, s_2 are initial states of the system such that $s_1 \sim_O s_2$. If $s_1 \rightsquigarrow^* s_1'$ then $\exists s_2'.\ s_2 \rightsquigarrow^* s_2'$ and $s_1' \sim_O s_2'$*

The ability of the application to configure cacheability of its resources can lead to incoherency, making formal program analysis on a cacheless model unsound. Nevertheless, directly verifying properties 4.1 and 4.2 using a complete cache-aware model is unfeasible for any software of meaningful size. Our goal is to show that the countermeasures can be used to restore coherency. We demonstrate that if the countermeasures are correctly implemented by the kernel then verification of the security properties on the cache-aware model can be soundly reduced to proof obligations on the cacheless model.

5 FORMALISATION

As basis for our study we define two models, a cacheless and a cache-aware model. The cacheless model represents a memory coherent single-core system where all caches are disabled. The cache-aware model is the same system augmented by a single level data cache.

5.1 Cacheless Model

The cacheless model is ARM-flavoured but general enough to apply to other architectures. A state $s = \langle reg, psrs, coreg, mem \rangle \in \mathbb{S}$ is a tuple of general-purpose registers reg (including program counter pc), control registers $psrs$, coprocessor state $coreg$, and memory mem. The core executes either in non-privileged mode U or privileged mode P, $Mode(s) \in \{U, P\}$. The control registers $psrs$ encode the execution mode and other execution parameters such as the arithmetic flags. The coprocessor state $coreg$ determines a range of system configuration parameters. The word addressable memory is represented by $mem : \mathbb{PA} \rightarrow \mathbb{B}^w$, where $\mathbb{B} = \{0, 1\}$, $\mathbb{PA}$ be the sets of physical addresses, and w is the word size.

The set $\mathbb{R}$ identifies all resources in the system, including registers, control registers, coprocessor states and physical memory locations (i.e. $\mathbb{PA} \subseteq \mathbb{R}$). We use $Cv : \mathbb{S} \times \mathbb{R} \rightarrow \mathbb{B}^*$ to represent the *core-view* of a resource, which looks up the resource and yields the corresponding value; e.g., for a physical address $pa \in \mathbb{PA}$, Cv returns the memory content in pa, $Cv(s, pa) = s.mem(pa)$.

All software activities are restricted by a hardware monitor. The monitor configuration can depend on coprocessor states (e.g. control registers for a TrustZone memory controller) and on regions of memory (e.g. page-tables for a Memory Management Unit (MMU)). We use the predicate $Mon(s, r, m, acc) \in \mathbb{B}$ to represent the hardware monitor, which holds if in the state s the access $acc \in \{wt, rd\}$ (for write and read respectively) to the resource $r \in \mathbb{R}$ is granted for the execution mode $m \in \{U, P\}$. In ARM the hardware monitor consists of a static and a dynamic part. The static access control is defined by the processor architecture, which prevents non-privileged modifications to coprocessor states and control registers and whose model is trivial. The dynamic part is defined by the MMU and controls memory accesses. We use $Mmu(s, va, m, acc) \in (\mathbb{PA} \times \mathbb{B}) \cup \{\bot\}$ to model the memory management unit. This function yields for a virtual address va the translation and the cacheability attribute if the access permission is granted and $\bot$ otherwise. Therefore, $Mon(s, pa, m, acc)$ is defined as $\exists va.\ Mmu(s, va, m, acc) = (pa, -)$. Further, $MD : \mathbb{S} \to \mathbb{R}$ is the function determining resources (i.e. the coprocessor registers, the current master page-table, the linked page-tables) which affect the monitor's behaviour.

The behaviour of the system is defined by an LTS $\to_m \subseteq \mathbb{S} \times \mathbb{S}$, where $m \in \{U, P\}$ and if $s \to_m s'$ then $Mode(s) = m$. Each transition represents the execution of a single instruction. When needed, we let $s \to_m s'\ [dop]$ denote that the instruction executed has dop effects on the data memory subsystem, where dop can be $wt(R)$, $rd(R)$, or $cl(R)$ to represent update, read and cache cleaning of resources $R \subseteq \mathbb{R}$. Finally, we use $s_0 \rightsquigarrow s_n$ to represent the weak transition relation that holds if there is a finite execution $s_0 \to \cdots \to s_n$ such that $Mode(s_n) = U$ and $Mode(s_j) \neq U$ for $0 < j < n$ (i.e. the weak transition hides internal states of the kernel).

5.2 Cache-Aware Model

We model a single-core processor with single level unified cache. A state $\bar{s} \in \bar{\mathbb{S}}$ has all the components of the cacheless model together with the cache, $\bar{s} = \langle reg, psrs, coreg, mem, cache \rangle$. The function Mmu and transition relation $\to_m \subseteq \bar{\mathbb{S}} \times \bar{\mathbb{S}}$ are extended to take into account caches. Other definitions of the previous subsection are extended trivially. In the following we use $c\text{-}hit(\bar{s}, pa)$ to denote a cache hit for address pa, $c\text{-}dirty(\bar{s}, pa)$ to identify dirtiness of the corresponding cache-line (i.e. if the value of pa has been modified in cache but not written back in the memory), and $c\text{-}cnt(\bar{s}, pa)$ to obtain value for pa stored in the cache.

Both the kernel and the hardware monitor (i.e. MMU) have the same view of the memory. In fact, the kernel always uses cacheable virtual addresses and the MMU always consults first the cache when it fetches a page table descriptor[2]. This allows us to define the core-view for memory resources $pa \in \mathbb{PA}$ in presence of caches:

$$Cv(\bar{s}, pa) = \begin{cases} c\text{-}cnt(\bar{s}, pa) & : \ c\text{-}hit(\bar{s}, pa) \\ \bar{s}.mem(pa) & : \ \text{otherwise} \end{cases}$$

5.3 Security Properties

Since the application is untrusted we assume its code is unknown and that it can break its memory coherence at will. Therefore, the behaviour of the application in the cacheless and the cache-aware models can differ significantly. In particular, memory incoherence caused by mismatched cacheability may lead a control variable of the application to have different values in these two models, causing the application to have different control flows. This makes the long-established method of verification by refinement [38] not feasible for analysing behaviour of the attacker.

To accomplish our security analysis we identify a subset of resources that are *critical*: resources for which integrity must be preserved and on which critical functionality depends. The security type of registers, control and coprocessor registers is statically assigned. The security type of memory locations, however, can dynamically change due to transfer of memory ownership; i.e. the type of these resources depends on the state of the system. The function $CR : (\bar{\mathbb{S}} \cup \mathbb{S}) \to 2^{\mathbb{R}}$ retrieves the subset of resources that are critical. Function CR usually depends on a subset of resources, for instance the internal kernel data-structures used to store the security type of memory resources. The definition of function CR must be based on the *core-view* to make it usable for both the cacheless and cache-aware models. We also define (for both cacheless and cache-aware state) $Ext_{\mathrm{cr}}(s) = \{(a, Cv(s, a)) \mid a \in CR(s)\}$, this is the function that extracts the value of all critical resources. Finally, the set of resources that are confidential is statically defined and identified by $CO \subseteq \mathbb{R}$. This region of memory includes all internal kernel data-structures whose value can depend on secret information.

Property 4.1 requires to introduce a system invariant $\bar{I}$ that is software dependent and defined per kernel. The invariant specifies: (i) the shape of a sound page-table (e.g. to prohibit the non-privileged writable accesses to the kernel memory and the page-tables themselves), (ii) properties that permits the kernel to work properly (e.g. the kernel stack pointer and its data structures are correctly configured), (iii) functional properties specific for the selected countermeasure, and (iv) cache related properties that allow to restore coherency. A corresponding invariant I for the cacheless model is derived from $\bar{I}$ by excluding properties that constrain caches. Property 4.1 is formally demonstrated by two theorems: one constraining the behaviour of the application and one showing functional correctness of the kernel. Let $ex\text{-}entry(\bar{s})$ be a predicate identifying the state of the system immediately after switching to the kernel (i.e. when an exception handler is executed the mode is privileged and the program counter points to the exception table). Theorem 5.1 enforces that the execution of the application in the cache enabled setting cannot affect the critical resources.

Theorem 5.1 (Application-integrity). *For all $\bar{s}$, if $\bar{I}(\bar{s})$ and $\bar{s} \to_U \bar{s}'$ then $\bar{I}(\bar{s}')$, $Ext_{\mathrm{cr}}(\bar{s}) = Ext_{\mathrm{cr}}(\bar{s}')$, and if $Mode(\bar{s}') \neq U$ then $ex\text{-}entry(\bar{s}')$*

While characteristics of the application prevents establishing refinement for non-privileged transitions, for the kernel we show that the use of proper countermeasures enables transferring the security properties from the cacheless to the cache-aware model. This demands proving that the two models behave equivalently for kernel transitions. We prove the behavioural equivalence by showing refinement between two models using forward simulation. We define the simulation relation $\mathcal{R}_{sim}$ (guaranteeing equality of critical resources) and show that both the invariant and the relation are preserved by privileged transitions:

Theorem 5.2 (Kernel-integrity). *For all $\bar{s}_1$ and s_1 such that $\bar{I}(\bar{s}_1)$, $\bar{s}_1\ \mathcal{R}_{sim}\ s_1$, and $ex\text{-}entry(\bar{s}_1)$ if $\bar{s}_1 \rightsquigarrow \bar{s}_2$ then $\exists s_2.\ s_1 \rightsquigarrow s_2$, $\bar{s}_2\ \mathcal{R}_{sim}\ s_2$ and $\bar{I}(\bar{s}_2)$*

Applying a similar methodology, in Section 7 we prove the confidentiality property (i.e. Theorem 5.3) in presence of caches. Here, we use bisimulation (equality of the application's observations) as unwinding condition:

Theorem 5.3 (Confidentiality). *For all $\bar{s}_1$ and $\bar{s}_2$ such that $\bar{I}(\bar{s}_1)$, $\bar{I}(\bar{s}_2)$, and $\bar{s}_1 \sim_O \bar{s}_2$, if $\bar{s}_1 \rightsquigarrow \bar{s}_1'$ then $\exists \bar{s}_2'.\ \bar{s}_2 \rightsquigarrow \bar{s}_2'$ and $\bar{s}_1' \sim_O \bar{s}_2'$ as well as $\bar{I}(\bar{s}_1')$, $\bar{I}(\bar{s}_2')$.*

[2]This is for instance the case of ARM Cortex-A53 and ARM Cortex-A8.

6 INTEGRITY

Our strategy to demonstrate correctness of integrity countermeasures consists of two steps. We first decompose the proof of Theorems 5.1 and 5.2 and show that the integrity properties are met if a set of proof conditions are satisfied. The goal of this step is to provide a theoretical framework that permits to analyse soundness of a countermeasure without the need of dealing with the complex transition relation of the cache-aware model. Then, we demonstrate correctness of two countermeasures of Section 3.1, namely always cacheability and selective eviction, by showing that if they are correctly implemented by the kernel then verification of the integrity properties can be soundly reduced to analysing properties of the kernel in the cacheless model.

6.1 Integrity: Application Level (Theorem 5.1)

To formalise the proof we introduce the auxiliary definitions of *coherency*, *derivability* and *safety*.

Definition (Coherency). *In $\bar{s} \in \bar{\mathbb{S}}$ a set of memory resources $R \subseteq \mathbb{PA}$ is coherent ($Coh(\bar{s}, R)$), if for all $pa \in R$, such that pa hits the cache and its value differs from the memory (i.e. $\bar{s}.mem(pa) \neq c\text{-}cnt(\bar{s}, pa)$), the corresponding cache line is dirty ($c\text{-}dirty(\bar{s}, pa)$).*

Coherency of the critical resources is essential to prove integrity. In fact, for an incoherent resource the *core-view* can be changed indirectly without an explicit memory write, i.e., through evicting the clean cache-line corresponding to the resource which has different values in the cache and memory. Moreover, in some cores, (e.g. ARMv7/v8) the MMU looks first into the caches when it fetches a descriptor. Then if the page-tables are coherent, a cache eviction cannot indirectly affect the behaviour of the MMU.

To allow the analysis of specific countermeasures to abstract from the complex cache-aware transition system, we introduce the notion of *derivability*. This is a relation overapproximating the effects over the memory and cache for instructions executed in non-privileged mode. Derivability is an architectural property and it is independent of the software executing.

Definition (Derivability). *We say $\bar{s}'$ is derivable from $\bar{s}$ in non-privileged mode (denoted as $\bar{s} \rhd_U \bar{s}'$) if $\bar{s}.coreg = \bar{s}'.coreg$ and for every $pa \in \mathbb{PA}$ one of the following properties holds:*

$D_\emptyset(\bar{s}, \bar{s}', pa)$: Independently of the access rights for the address pa, the corresponding memory can be changed due to an eviction of a dirty line and subsequent write-back of the cached value into the memory. Moreover, the cache can always change due to an eviction.

$D_{rd}(\bar{s}, \bar{s}', pa)$: If non-privileged mode can read the address pa, the cache state can change through a fill operation which loads the cache with the value of pa in the memory.

$D_{wt}(\bar{s}, \bar{s}', pa)$: If non-privileged mode can write the address pa, it can either write directly into the cache, making it dirty, or bypass it, by using an uncacheable alias.

Figure 2 reports the formal definition of these predicates.

Definition (Safety). *A state $\bar{s}$ is safe, $safe(\bar{s})$, if for every state $\bar{s}'$, resource r, mode m and access request acc if $\bar{s} \rhd_U \bar{s}'$ then $Mon(\bar{s}, r, m, acc) = Mon(\bar{s}', r, m, acc)$.*

$$
\begin{aligned}
&D_\emptyset(\bar{s}, \bar{s}', pa) \overset{\text{def}}{\equiv} M'(pa) \neq M(pa) \Rightarrow (c\text{-}dirty(\bar{s}, pa) \wedge M'(pa) = c\text{-}cnt(\bar{s}, pa)) \\
&\quad \wedge W(\bar{s}', pa) \neq W(\bar{s}, pa) \Rightarrow \\
&\qquad\qquad (\neg c\text{-}hit(\bar{s}', pa) \wedge (c\text{-}dirty(\bar{s}, pa) \Rightarrow M'(pa) = c\text{-}cnt(\bar{s}, pa)))
\end{aligned}
$$

$$
\begin{aligned}
&D_{rd}(\bar{s}, \bar{s}', pa) \overset{\text{def}}{\equiv} (Mon(\bar{s}, pa, U, rd)) \\
&\quad \wedge M'(pa) = M(pa) \wedge (W(\bar{s}', pa) \neq W(\bar{s}, pa) \Rightarrow \\
&\qquad\qquad (c\text{-}cnt(\bar{s}', pa) = M(pa) \wedge \neg c\text{-}hit(\bar{s}, pa)))
\end{aligned}
$$

$$
\begin{aligned}
&D_{wt}(\bar{s}, \bar{s}', pa) \overset{\text{def}}{\equiv} Mon(\bar{s}, pa, U, wt) \\
&\quad \wedge (W(\bar{s}', pa) \neq W(\bar{s}, pa) \Rightarrow c\text{-}dirty(\bar{s}', pa)) \\
&\quad \wedge (M'(pa) \neq M(pa) \Rightarrow \\
&\qquad\qquad (\neg c\text{-}dirty(\bar{s}', pa) \Rightarrow \exists va. Mmu(\bar{s}, va, U, wt) = (pa, 0)))
\end{aligned}
$$

Figure 2: Derivability. Here $M = \bar{s}.mem$, $M' = \bar{s}'.mem$, and $W(\bar{s}, pa) = \langle c\text{-}hit(\bar{s}, pa), c\text{-}dirty(\bar{s}, pa), c\text{-}cnt(\bar{s}, pa) \rangle$ denotes the cache-line corresponding to *pa* in $\bar{s}.cache$.

A state is *safe* if non-privileged executions cannot affect the hardware monitor, i.e. only the kernel can change page-tables.

To decompose proof of Theorem 5.1, the invariant $\bar{I}$ must be split in three parts: a functional part $\bar{I}_{fun}$ which only depends on the *core-view* for the critical resources, an invariant $\bar{I}_{coh}$ which only depends on coherency of the critical resources, and an optional countermeasure-specific invariant $\bar{I}_{cm}$ which depends on coherency of non-critical memory resources such as resources in an always-cacheable region.

Proof Obligation 6.1. *For all $\bar{s}$, $\bar{I}(\bar{s}) = \bar{I}_{fun}(\bar{s}) \wedge \bar{I}_{coh}(\bar{s}) \wedge \bar{I}_{cm}(\bar{s})$ and:*

1. *for all $\bar{s}'$ if $Ext_{\text{cr}}(\bar{s}) = Ext_{\text{cr}}(\bar{s}')$ then $\bar{I}_{fun}(\bar{s}) = \bar{I}_{fun}(\bar{s}')$;*
2. *for all $\bar{s}'$ if $Coh(\bar{s}, CR(\bar{s}))$, $Coh(\bar{s}', CR(\bar{s}'))$, and $Ext_{\text{cr}}(\bar{s}) = Ext_{\text{cr}}(\bar{s}')$ then $\bar{I}_{coh}(\bar{s}) = \bar{I}_{coh}(\bar{s}')$;*
3. *for all $\bar{s}'$ if $\bar{I}(\bar{s})$ and $\bar{s} \rhd_U \bar{s}'$ then $\bar{I}_{cm}(\bar{s}')$.*

Also, the function *CR* must be correctly defined: i.e. resources affecting the set of critical resources are critical themselves.

Proof Obligation 6.2. *For all $\bar{s}, \bar{s}'$ if $Ext_{\text{cr}}(\bar{s}) = Ext_{\text{cr}}(\bar{s}')$ then $CR(\bar{s}) = CR(\bar{s}')$*

Safety is essential to prove integrity: if a state is not safe, the application can potentially elevate its permissions by changing configurations of the hardware monitor and get access to resources beyond its rights.

Lemma 6.3. *If $\bar{I}(\bar{s})$ then $safe(\bar{s})$*

Proof of Lemma 6.3 depends on the formal model of the hardware monitor and guarantees provided by the invariant. Using the invariant $\bar{I}$, we identify three main proof obligations that are needed to prove this lemma.

(1) The functional part of the invariant must guarantee that the resources that control the hardware monitor are considered critical.

Proof Obligation 6.4. *If $\bar{I}_{fun}(\bar{s})$ then $MD(\bar{s}) \subseteq CR(\bar{s})$*

(2) The application should not be allowed to directly affect the critical resources. This means there is no address writable in non-privileged mode that points to a critical resource.

Proof Obligation 6.5. *If $\bar{I}_{fun}(\bar{s})$ and $r \in CR(\bar{s})$ then $\neg Mon(\bar{s}, r, U, wt)$*

(3) Finally, to prevent the application from indirectly affecting the hardware monitor, e.g. by line eviction, the invariant must ensure coherency of critical resources.

Proof Obligation 6.6. *If $\bar{I}_{coh}(\bar{s})$ then $Coh(\bar{s}, CR(\bar{s}))$*

We overapproximate the reachable states in non-privileged mode to prove properties that do not depend on the software platform or countermeasure. This eliminates the need for revalidating properties of the instruction set (i.e. properties that must be verified for every possible instruction) in every new software scenario.

Lemma 6.7. *For all $\bar{s}$ such that $safe(\bar{s})$ and $Coh(\bar{s}, MD(\bar{s}))$, if $\bar{s} \rightarrow_U \bar{s}'$ then* [3]

(1) $\bar{s} \rhd_U \bar{s}'$, i.e. non-privileged transitions from safe states can only lead into derivable states

(2) if $Mode(\bar{s}') \neq U$ then $ex\text{-}entry(\bar{s}')$, i.e. the mode can only change by entering an exception handler

Corollary 6.8 shows that derivability can be used as sound overapproximation of the behaviour of non-privileged transitions if the invariant holds.

Corollary 6.8. *For all $\bar{s}$ if $\bar{I}(\bar{s})$ and $\bar{s} \rightarrow_U \bar{s}'$ then $\bar{s} \rhd_U \bar{s}'$*

Proof. The statement directly follows by Lemma 6.7.1 which is enabled by Lemma 6.3, Obligation 6.4, and Obligation 6.6. ■

We now proceed to show that the hardware monitor enforces access policy correctly; i.e. the application transitions cannot modify critical resources.

Lemma 6.9. *For all $\bar{s}, \bar{s}'$ such that $\bar{I}(\bar{s})$ if $\bar{s} \rhd_U \bar{s}'$ then $Ext_{\mathrm{cr}}(\bar{s}) = Ext_{\mathrm{cr}}(\bar{s}')$*

Proof. Since $\bar{I}(\bar{s})$ holds, the hardware monitor prohibits writable accesses of the application to critical resources (Obligation 6.5) and $safe(\bar{s})$ holds (Lemma 6.3). Also, derivability shows that the application can directly change only resources that are writable according to the monitor. Thus, the application cannot directly update $CR(\bar{s})$. Beside, the invariant guarantees coherency of critical resources in $\bar{s}$ (Obligation 6.6). This prevents indirect modification of these resources. ■

To complete the proof of Theorem 5.1 we additionally need to show that coherency of critical resources (Lemma 6.10) and invariant (Lemma 6.11) are preserved by non-privileged transitions.

Lemma 6.10. *For all $\bar{s}$ if $\bar{I}(\bar{s})$ and $\bar{s} \rhd_U \bar{s}'$ then $Coh(\bar{s}', CR(\bar{s}'))$*

Proof. The proof depends on Obligation 6.6 and Obligation 6.5: coherency can be invalidated only through non-cacheable writes, which are not possible since aliases to critical resources that are writable by non-privileged mode are forbidden. ■

Lemma 6.11. *For all $\bar{s}$ and $\bar{s}'$ if $\bar{I}(\bar{s})$ and $\bar{s} \rightarrow_U \bar{s}'$ then $\bar{I}_{fun}(\bar{s}')$*

Proof. To show that non-privileged transitions preserve the invariant we use Corollary 6.8, Lemma 6.9, and Obligation 6.1.1. ■

Finally, Lemma 6.7.2, Corollary 6.8, Lemma 6.9, Lemma 6.10, Lemma 6.11, and Obligation 6.1 imply Theorem 5.1, completing the proof of integrity for non-privileged transitions.

[3] In [31] the authors proved a similar theorem for the HOL4 ARMv7 model provided by Anthony Fox et. al. [20].

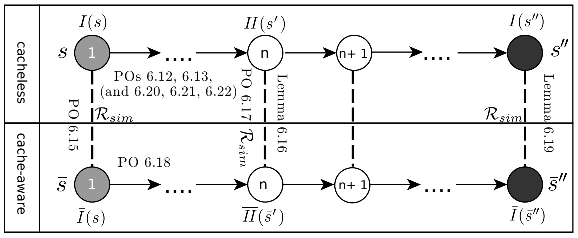

Figure 3: Verification of kernel integrity

6.2 Integrity: Kernel Level (Theorem 5.2)

The proof of functional kernel correctness (Theorem 5.2) is done by: (i) showing that the kernel preserves the invariant I in the cacheless model (Obligation 6.12); (ii) requiring a set of general proof obligations (e.g. the kernel does not jump outside its address space) that must be verified at the cacheless level (Obligation 6.13), (iii) demonstrating that the simulation relation permits transferring the invariant from the cache-aware model to the cacheless one (Obligation 6.15), (iv) verifying a refinement between the cacheless model and the cache-aware model (Lemma 6.16) assuming correctness of the countermeasure (Obligation 1), and finally (v) proving that the refinement theorem allow to transfer the invariant from the cacheless model to the cache-aware model (Lemma 6.19). Figure 3 indicates our approach to prove kernel integrity.

Our goal is to lift the code verification to the cacheless model, so that existing tools can be used to discharge proof obligations. The first proof obligation requires to show that the kernel is functionally correct when there is no cache:

Proof Obligation 6.12. *For all s such that $I(s)$ and $ex\text{-}entry(s)$ if $s \rightsquigarrow s'$ then $I(s')$.*

To enable the verification of simulation, it must be guaranteed that the kernel's behaviour is predictable: (6.13.1) the kernel cannot execute instructions outside the critical region (this property is usually verified by extracting the control flow graph of the kernel and demonstrating that it is contained in a static region of the memory), and (6.13.2) the kernel does not change page-tables entries that maps the kernel virtual memory (here this region is assumed to be static and identified by $\mathbf{K_{vm}}$) which must be accessed only using cacheable aliases.

Proof Obligation 6.13. *For all s such that $I(s)$ and $ex\text{-}entry(s)$ if $s \rightarrow_P^* s'$ then*

(1) if $Mmu(s', s'.reg.pc, P, acc) = (pa, -)$ then $pa \in CR(s)$

(2) for every $va \in \mathbf{K_{vm}}$ and acc if $Mmu(s, va, P, acc) = (pa, c)$ then $Mmu(s', va, P, acc) = (pa, c)$ and $c = 1$

In order to establish a refinement, we introduce the *memory-view* Mv of the cache-aware model. This function models what can be observed in the memory after a cache line eviction and is used to handle dynamic growth of the critical resources, which are potentially not coherent when a kernel handler starts.

$$Mv(\bar{s}, r) \overset{\text{def}}{\equiv} \begin{cases} c\text{-}cnt(\bar{s}, pa) & : r \in \mathbb{PA} \wedge c\text{-}dirty(\bar{s}, pa) \\ \bar{s}.mem(pa) & : r \in \mathbb{PA} \wedge \neg c\text{-}dirty(\bar{s}, pa) \\ Cv(\bar{s}, r) & : \text{otherwise} \end{cases}$$

Note that Mv only differs from Cv in memory resources that are cached, clean, and have different values stored in the cache and memory, i.e., incoherent memory resources. In particular, we can prove the following lemma about coherency:

Lemma 6.14. *Given a memory resource $pa \in \mathbb{PA}$ and cache-aware state $\bar{s}$ then $Coh(\bar{s}, \{pa\}) \Leftrightarrow (Cv(\bar{s}, pa) = Mv(\bar{s}, pa))$, i.e., memory resources are coherent, iff both views agree on them.*

We define the simulation relation between the cacheless and cache-aware models using the memory-view: $\bar{s}\ \mathcal{R}_{sim}\ s \equiv \forall r \in \mathbb{R}.\ Cv(s, r) = Mv(\bar{s}, r)$. The functional invariant of the cache-aware model and the invariant of the cacheless model must be defined analogously, that is the two invariants are equivalent if the critical resources are coherent, thus ensuring that functional properties can be transfer between the two models.

Proof Obligation 6.15. *For all $\bar{s}$ and s, such that $\bar{I}_{coh}(\bar{s})$ and $\bar{s}\ \mathcal{R}_{sim}\ s$, holds $\bar{I}_{fun}(\bar{s}) \Leftrightarrow I(s)$*

A common problem of verifying low-level software is coupling the invariant with every possible internal states of the kernel. This is a major concern here, since the set of critical resources changes dynamically and can be stale while the kernel is executing. We solve this problem by defining a *internal invariant* $\overline{II}(\bar{s}, \bar{s}')$ which allows us to define properties of the state $\bar{s}'$ in relation with the initial state of the kernel handler $\bar{s}$. This invariant $\overline{II}$ (similarly II for the cacheless model) includes: (i) $\bar{I}(\bar{s})$ holds, (ii) the program counter in $\bar{s}'$ points to the kernel memory, (iii) coherency of resources critical in the initial state (i.e. $Coh(\bar{s}', CR(\bar{s}))$) (iv) all virtual addresses in $\mathbf{K_{vm}}$ are cacheable and their mapping in $\bar{s}$ and $\bar{s}'$ is unchanged, and (v) additional requirements that are countermeasure specific and will be described later.

Lemma 6.16. *For all $\bar{s}$ and s such that $\bar{I}(\bar{s})$, ex-entry($\bar{s}$), and $\bar{s}\ \mathcal{R}_{sim}\ s$, if $\bar{s} \rightarrow_P^n \bar{s}'$ and $s \rightarrow_P^n s'$ then $\bar{s}'\ \mathcal{R}_{sim}\ s'$, $\overline{II}(\bar{s}, \bar{s}')$, and $II(s, s')$*

Proof. By induction on the execution length. The base case is trivial, since no step is taken. For the inductive case we first show that the instruction executed is the same in both the models: $\mathcal{R}_{sim}$ guarantees that the two states have the same program counter, Obligation 6.13.1 ensures that the program counter points to the kernel memory contained in the critical resources, property (iii) of the invariant guarantees that this memory region is coherent, and Lemma 6.14 shows that the fetched instruction is the same.

Proving that the executed instruction preserves kernel virtual memory mapping is trivial, since Obligation 6.13.2 ensures that the kernel does not change its own memory layout and that it only uses cacheable aliases. Showing that the resources accessed by the instruction have the same value (thus guaranteeing $\mathcal{R}_{sim}$ is preserved) in the cache-aware and cacheless states depends on demonstrating their coherency. This is countermeasure specific and is guaranteed by proof Obligation 6.17.1. Similarly, showing the internal invariant is maintained by privileged transitions depends on the specific counter that is in place (Obligation 6.17.2). ■

Proof Obligation 6.17. *For all states $\bar{s}$ and $\bar{s}'$ that satisfy the refinement (i.e. $\bar{I}(\bar{s})$, ex-entry($\bar{s}$), and $\bar{s}\ \mathcal{R}_{sim}\ s$), after any number n of instructions of the kernel that preserve the refinement and the internal invariants ($\bar{s} \rightarrow_P^n \bar{s}'$, $s \rightarrow_P^n s'$, $\bar{s}'\ \mathcal{R}_{sim}\ s'$, $\overline{II}(\bar{s}, \bar{s}')$, and $II(s, s')$)*

1. *if the execution of the n + 1-th instruction in the cacheless model accesses resources R ($s' \rightarrow_P s''$ [dop] and either dop = rd(R) or dop = wt(R)) then $Coh(\bar{s}', R)$*
2. *the execution of the n + 1-th instruction in the cacheless and cache-aware models ($s' \rightarrow_P s''$ and $\bar{s}' \rightarrow_P \bar{s}''$) preserves the internal invariants ($II(s, s'')$ and $\overline{II}(\bar{s}, \bar{s}'')$)*

Additionally, the internal invariant must ensure the countermeasure specific requirements of coherency for all internal states of the kernel.

Proof Obligation 6.18. *For all $\bar{s}$ and $\bar{s}'$ if $\overline{II}(\bar{s}, \bar{s}')$ then $\bar{I}_{cm}(\bar{s}')$*

Finally, we show that the invariant $\bar{I}$ holds in a cache-aware state when the control is returned to non-privileged mode, i.e. when the invariant is re-established in the cacheless model.

Lemma 6.19. *For all $\bar{s}$, $\bar{s}'$, and s' if $I(s')$, $\bar{s}'\ \mathcal{R}_{sim}\ s'$, and $\overline{II}(\bar{s}, \bar{s}')$ then $\bar{I}(\bar{s}')$ holds.*

Proof. The three parts of invariant $\bar{I}(\bar{s}')$ are demonstrated independently. Obligation 6.18 establishes $\bar{I}_{cm}(\bar{s}')$. Property (iii) of $\overline{II}(\bar{s}, \bar{s}')$ guarantees $\bar{I}_{coh}(\bar{s}')$. Finally, Obligation 6.15 demonstrates $\bar{I}_{fun}(\bar{s})$. ■

6.3 Correctness of countermeasures

Next, we turn to show that selected countermeasures for the integrity attacks prevent usage of cache to violate the integrity property. Thus, we show that the countermeasures help to discharge the coherency related proof obligations reducing verification of integrity to analysing properties of the kernel code using the cacheless model.

Always Cacheability. We use $M_{ac} \subseteq \mathbb{PA}$ to statically identify the region of physical memory that should be always accessed using cacheable aliases. The verification that the always cacheability countermeasure is in place can be done by discharging the following proof obligations at the cacheless level: (6.20.1) the hardware monitor does not permit uncacheable accesses to M_{ac}, (6.20.2.a) the kernel never allocates critical resources outside M_{ac}, thus restricting the application to use M_{ac} to communicate with the kernel, and (6.20.2.b) the kernel accesses only resources in M_{ac}:

Proof Obligation 6.20. *For all s such that $I(s)$*

1. *for every va, m and acc if $Mmu(s, va, m, acc) = (pa, c)$ and $pa \in M_{ac}$ then $c = 1$,*
2. *if $s \rightarrow_P^* s'$ then*
   1. *$CR(s') \subseteq M_{ac}$ and*
   2. *if $s' \rightarrow_P s''$ [dop] and R are the resources in dop then $R \subseteq M_{ac}$*

These three properties, together with Obligation 6.15, enable us to prove that the resources accessed by the instructions executed in the privileged mode have the same value in the cache-aware and cacheless states (Obligation 6.17.1). In a similar vein, we instantiate part (v) of the internal invariant as $Coh(\bar{s}', M_{ac})$, and then we use Obligations 6.13 and 6.20.2.a to show that the internal invariant is preserved by kernel steps (Obligation 6.17.2). To discharge coherency related proof obligations for non-privileged mode, we set $\bar{I}_{coh}$ and $\bar{I}_{cm}$ to be $Coh(\bar{s}, M_{ac} \cap CR(\bar{s}))$ and $Coh(\bar{s}, M_{ac} \setminus CR(\bar{s}))$ respectively. This makes the proof of Obligations 6.1.2, 6.6, and 6.18 trivial.

We use Lemma 6.3 to guarantee that derivability preserves page-tables and, thus, cacheability of the resources in M_{ac}, and we use Lemma 6.9 and Obligation 6.2 to demonstrate that derivability preserves *CR*. Finally, Obligation 6.20.1 enforces that all aliases to M_{ac} are cacheable, demonstrating Obligations 6.1.3.

Selective Eviction. This approach requires to selectively flush the lines that correspond to the memory locations that become critical when the kernel acquire ownership of a region of memory. To verify that the kernel correctly implements this countermeasure we need to track evicted lines, by adding to the cacheless model a history variable h.

$$\frac{s \rightarrow_P s' \ [cl(R)]}{(s,h) \rightarrow_P (s', h \cup R) \ [cl(R)]} \qquad \frac{s \rightarrow_P s' \ [dop] \wedge dop \neq cl(R)}{(s,h) \rightarrow_P (s', h) \ [dop]}$$

All the kernel accesses must be restricted to resources that are either critical or have been previously cleaned (i.e. are in h).

Proof Obligation 6.21. *For all s such that $I(s)$ if $(s, \emptyset) \rightarrow_P^* (s', h')$, $(s', h') \rightarrow_P (s'', h'')$ $[dop]$, and either $dop = rd(R)$ or $dop = wt(R)$ then $R \subseteq CR(s) \cup h'$*

Moreover, it must be guaranteed that the set of critical resources always remains coherent.

Proof Obligation 6.22. *For all s such that $I(s)$ and ex-entry(s) if $(s, \emptyset) \rightsquigarrow (s', h')$ then $CR(s') \subseteq CR(s) \cup h'$*

This ensures that the kernel accesses only coherent resources and allows to establish Obligation 6.17.1. To discharge Obligation 6.17.2, we first define part (v) of the internal invariant as $Coh(\bar{s}', CR(\bar{s}) \cup h')$ and then use Obligations 6.13 and 6.21 to discharge it. To discharge proof obligations of non-privileged mode in the cache-aware model, for state $\bar{s}$ the invariant must ensure $\bar{I}_{coh}(\bar{s}) = Coh(\bar{s}, CR(\bar{s}))$ and $\bar{I}_{cm}(\bar{s}) = true$. This makes the proof of Obligations 6.1.2, 6.6 and 6.18 trivial. Part (v) of the internal invariant and Obligation 6.22 ensure that $\bar{I}_{coh}$ is established when the kernel returns to the application.

7 CONFIDENTIALITY

This section presents the proof of the confidentiality property. The proof relies on the cache behaviour, hence we briefly present cache model.

7.1 Generic Data Cache Model

We have formally defined a generic model which fits a number of common processor data-cache implementations. The intuition behind is that most data-caches are direct mapped or set-associative caches, sharing a similar structure: (i) Memory is partitioned into sets of lines which are congruent wrt. to a set index, (ii) data-caches contain a number of ways which can hold one corresponding line for every set index, being uniquely identified by a tag, (iii) writes can make lines dirty, i.e., potentially different from the associated data in memory, (iv) there is a small set of common cache operations, e.g., filling the cache with a line from memory, (v) an eviction policy controls the allocation of ways for new lines, and the eviction of old lines if the cache is full.

In addition to the cache contents, partitioned into line sets, we keep history $H \in \mathbb{A}^*$ of internal cache actions performed for each line set. An action $a \in \mathbb{A}$ can be (i) a read or write access to a present line, (ii) a line eviction, or (iii) a line fill. All actions also specify the tag of the corresponding line.

As each line set can only store limited amounts of entries, eviction policy $evict?(H, t)$ returns the tag of the line to be replaced at a line fill for a given tag t, or $\perp$ if eviction is not required. Evicted dirty lines are then written back into the memory.

We assume here that the eviction policy is only depending on the action history of a line set and the tag of the line to be filled in. This is trivially the case for direct-mapped caches, where each line set at most contains one entry and no choice of eviction targets is possible, and for purely random replacements strategies, where the cache state and history is completely ignored. Also more sophisticated eviction policies, like LRU or pseudo-LRU, usually only depend on preceding operations on the same line set.

Another observation for these replacement strategies is that they only depend on finite subsequences of the history. For instance,LRU only depends on the past actions on the entries currently present in a given set and after a cache flush subsequent line fills and evictions are not influenced by operations from before the flush. We formalize this notion as the *filtered history* on which a given eviction policy depends, computed from the full history by policy-specific filter $\varphi : \mathbb{A}^* \rightarrow \mathbb{A}^*$. The idea is that the eviction policy makes the same decisions for filtered and unfiltered histories.

Assumption 7.1. *If φ is a policy-specific filter then for all $H \in \mathbb{A}^*$ and tag t holds $evict?(H, t) = evict?(\varphi(H), t)$.*

We use this property in combination with our countermeasures in the confidentiality proof, in order to make cache states indistinguishable for the attacker. A complete formalization of the cache state and semantics is omitted here as these details are irrelevant for the general proof strategy outlined below (see Appendix for the detailed model).

7.2 Observations in the Cache-Aware Model

The kernel and untrusted application share and control parts of the memory and the caches. Our goal is to ensure that through these channels the application cannot infer anything about the confidential resources of the kernel. We fix the confidential resources of the kernel as a static set $CO \subset \mathbb{R}$ and demand that they cannot be directly accessed by the application (Obligation 7.3).

The observations of the application are over-approximated by the set $O = \{r \mid r \notin CO\}$. Note, that some of the observable resources may be kernel resources that are not directly accessible by the application, but affect its behaviour (e.g. page tables). Two cacheless states s_1 and s_2 are considered observationally equivalent, if all resources in O have the same core-view. Formally we define $s_1 \sim_O s_2 \equiv \forall r \in O.\ Cv(s_1, r) = Cv(s_2, r)$.

In the cache-aware model, naturally, also the cache has to be taken into account. Specifically, for addresses readable by the attacker, both the corresponding cache line and underlying main memory content can be extracted using uncacheable aliases. For all other cache lines, we overapproximate the observable state, assuming that the attacker can infer whether a tag is present in the cache (*tag state*[4]), measure whether an entry is dirty, and derive

[4]Tags of kernel accesses can be identified when a line set contains both a line of the attacker and the kernel, and subsequent secret-dependent accesses may either cause a hit on that line or a miss which evicts the attacker's line.

the filtered history[5] of cache actions on all line sets. For caches and memories C_1, C_2, M_1, and M_2 we denote observational equivalence wrt. a set $A \subseteq \mathbb{PA}$ by $(C_1, M_1) \approx_A (C_2, M_2)$. The relation holds if:

(1) the memories agree for all $pa \in A$, i.e., $M_1(pa) = M_2(pa)$,
(2) the line sets with any index i in C_1, C_2:
   (a) agree on the tags of their entries (same tag state),
   (b) agree on the dirtiness of their entries
   (c) agree on the contents of those entries that have tags pointing to addresses in A, and
   (d) agree on their filtered history ("$\varphi(H_1(i)) = \varphi(H_2(i))$").

Notice that $\approx_A$ implies core-view equivalence for any address in A.

Now we distinguish observable resources which are always coherent, from potentially incoherent non-critical memory resources $NC \subseteq \mathbb{PA} \cap O$. Intuitively, this set contains all observable addresses that the application may access through uncacheable aliases. For coherent observable resources we introduce relation

$$\bar{s}_1 \sim_{coh} \bar{s}_2 \overset{\text{def}}{\equiv} \forall r \in O \setminus NC.\ Cv(\bar{s}_1, r) = Cv(\bar{s}_2, r)\,,$$

and define observational equivalence for the cache-aware model:

$$\begin{aligned} \bar{s}_1 \sim_O \bar{s}_2 \ \overset{\text{def}}{\equiv}\ & \bar{s}_1 \sim_{coh} \bar{s}_2 \\ \wedge\ & (\bar{s}_1.cache, \bar{s}_1.mem) \approx_{NC} (\bar{s}_2.cache, \bar{s}_2.mem)\,. \end{aligned}$$

Note that we are overloading notation here and that $\sim_{coh}$ and $\sim_O$ are equivalence relations. Allowing to apply relation $\sim_{coh}$ also to cacheless states, we get the following corollary.

Corollary 7.2. *For all s_1, s_2, if $s_1 \sim_O s_2$ then $s_1 \sim_{coh} s_2$.*

Confidentiality (Theorem 5.3) is demonstrated by showing that relation $\sim_O$ is a bisimulation (i.e. it is preserved for pairs of computations starting in observationally equivalent states). Below, we prove this property separately for of application and kernel steps.

7.3 Confidentiality: Application Level

As relation $\sim_O$ is countermeasure-independent, verification for application's transition can be shown once and for all for a given hardware platform, assuming the kernel invariant guarantees several properties for the hardware configuration. First, the hardware monitor must ensure that confidential resources are only readable in privileged mode.

Proof Obligation 7.3. *For all $\bar{s}$ such that $\bar{I}(\bar{s})$ if $r \in CO$ then $\neg Mon(\bar{s}, r, U, rd)$.*

Secondly, the invariant needs to guarantee that the hardware monitor data is always observable. This implies that in observationally equivalent states the same access permissions are in place.

Proof Obligation 7.4. *For all $\bar{s}$, if $\bar{I}(\bar{s})$ then $MD(\bar{s}) \subset O$.*

In addition, the monitor never allows the application to access coherent resources through uncacheable aliases.

Proof Obligation 7.5. *For all $\bar{s}$ such that $\bar{I}(\bar{s})$ for every va, acc if $Mmu(\bar{s}, va, U, acc) = (pa, c)$ and $pa \in \mathbb{PA} \setminus NC$ then $c = 1$.*

By this property it is then easy to derive the coherency of resources outside of *NC*, assuming that the hardware starts in a coherent memory configuration and that the kernel never makes memory incoherent itself (Obligation 6.13.2).

Proof Obligation 7.6. *For all $\bar{s}$, if $\bar{I}(\bar{s})$ then $Coh(\bar{s}, \mathbb{PA} \setminus NC)$.*

Finally, it has to be shown that non-privileged cache operations preserve the equivalence of coherent and incoherent resources.

Lemma 7.7. *Given a set of potentially incoherent addresses $A \subset O \cap \mathbb{PA}$ and cache-aware states $\bar{s}_1$ and $\bar{s}_2$ such that*

- *$Coh(\bar{s}_1, (O \cap \mathbb{PA}) \setminus A)$ and $Coh(\bar{s}_2, (O \cap \mathbb{PA}) \setminus A)$,*
- *$\forall pa \in (O \cap \mathbb{PA}) \setminus A.\ Cv(\bar{s}_1, pa) = Cv(\bar{s}_2, pa)$, and*
- *$(\bar{s}_1.cache, \bar{s}_1.mem) \approx_A (\bar{s}_2.cache, \bar{s}_2.mem)$,*

if $\bar{s}_1 \rightarrow_U \bar{s}_1'$ [dop] and $\bar{s}_2 \rightarrow_U \bar{s}_2'$ [dop], and dop is cacheable, then

- *$Coh(\bar{s}_1', (O \cap \mathbb{PA}) \setminus A)$ and $Coh(\bar{s}_2', (O \cap \mathbb{PA}) \setminus A)$,*
- *$\forall pa \in (O \cap \mathbb{PA}) \setminus A.\ Cv(\bar{s}_1', pa) = Cv(\bar{s}_2', pa)$, and*
- *$(\bar{s}_1'.cache, \bar{s}_1'.mem) \approx_A (\bar{s}_2'.cache, \bar{s}_2'.mem)$.*

This lemma captures three essential arguments about the underlying hardware: (i) on coherent memory, observational equivalence is preserved wrt. the core-view and thus independent of whether the data is currently cached, (ii) cacheable accesses cannot break the coherency of these resources, and (iii) on potentially incoherent memory (addresses A), observational equivalence is preserved if cache and memory are equivalent for the corresponding lines, i.e., they have the same tag states, contents, and filtered history.

These properties inherently depend on the specific cache architecture, in particular on the eviction policy and its history filter. For any two equal filtered histories the eviction policy selects the same entry to evict (Assumption 7.1), therefore corresponding cache states stay observationally equivalent. Moreover, they rely on the verification conditions that evictions do not change the core-view of a coherent memory resource and that cache line fills for coherent addresses read the same values from main memory.

We have formally verified Lemma 7.7 for an instantiation of our cache model that uses an LRU replacement policy (see Appendix).

Based on the proof obligations lined out above we can now prove confidentiality for steps of the application

Lemma 7.8. *For all $\bar{s}_1, \bar{s}_2, \bar{s}_1'$ where $\bar{I}(\bar{s}_1)$ and $\bar{I}(\bar{s}_2)$ hold, if $\bar{s}_1 \sim_O \bar{s}_2$ and $\bar{s}_1 \rightarrow_U \bar{s}_1'$, then $\exists \bar{s}_2'.\ \bar{s}_2 \rightarrow_U \bar{s}_2'$ and $\bar{s}_1' \sim_O \bar{s}_2'$.*

Proof Lemma 7.8. We perform a case split over all possible hardware steps in non-privileged mode. Observational equivalence on cache, memory, and program counterprovide that the same instruction is fetched and executed in both steps[6]. For hardware transitions that do not access memory, we conclude as in the proof on the cacheless model (Obligation 7.9, similar theorems were proved in [39]).

In case of memory instructions, since general purpose registers are equivalent, the same addresses are computed for the operation. Obligations 7.3 and 7.4 yield that the same access permissions are in place in $\bar{s}_1$ and $\bar{s}_2$ such that the application can only directly read observationally equivalent memory resources. Then we distinguish cacheable and uncacheable accesses to an address *pa*.

[5]The history of kernel cache operations may be leaked in a similar way as the tag state, affecting subsequent evictions in the application, however only as far as the filter for the eviction policy allows it.

[6]We do not model instruction caches in the scope of this work, but assume a unified data and instruction cache. In fact, instruction caches allow further storage-channel-based attacks [26] that would need to be addressed at this point in the proof.

In the latter case, by Obligation 7.5 and we know that $pa \in NC$, and we obtain $\bar{s}_1.mem(pa) = \bar{s}_2.mem(pa)$ from $\bar{s}_1 \sim_O \bar{s}_2$. Now, since the access bypasses the caches, they are unchanged. Moreover, the memory accesses in both states yield the same result and the equivalence of cache and memory follows trivially.

In case of cacheable accesses we know by Obligation 7.6 that resources $(O \cap \mathbb{PA}) \setminus NC$ are coherent and we apply Lemma 7.7 to deduce the observational equivalence of cache and memory for addresses in $(O \cap \mathbb{PA}) \setminus NC$ and NC. As also register resources are updated with the same values, we conclude $\bar{s}_1' \sim_O \bar{s}_2'$. ■

7.4 Confidentiality: Kernel Level

Kernel level confidentiality ensures that the execution of kernel steps does not leak confidential information to the application. Specifically, this entails showing that at the end of the kernel execution observational equivalence of resources in O is (re)established. First, the kernel should not leak confidential resources in absence of caches:

Proof Obligation 7.9. *For all s_1, s_2, s_1' such that $I(s_1)$, $I(s_2)$ ex-entry(s_1), ex-entry(s_2), and $s_1 \sim_O s_2$ if $s_1 \rightsquigarrow s_1'$ then $\exists s_2'$. $s_2 \rightsquigarrow s_2'$ and $s_1' \sim_O s_2'$.*

The goal is now to show that the chosen countermeasure against information leakage through the caches allows transferring the confidentiality property to the cache-aware model. Formally, this countermeasure is represented by a two-state property $CM(s, s')$ on the cacheless and $\overline{CM}(\bar{s}, \bar{s}')$ on the cache-aware model. Here the first argument is the starting state of the kernel execution, while the second argument is some arbitrary state that is reached from there by a privileged computation. Property $CM(s, s')$ should only cover functional properties of the countermeasure that can be verified on the cacheless model as part of the kernel's internal invariant.

Proof Obligation 7.10. *For all s, s' with $s \rightarrow_P^* s'$ and ex-entry(s), if $II(s, s')$ then $CM(s, s')$*

The countermeasure property on the cache-aware model, on the other hand, extends CM with conditions on the cache state that prevent information leakage to the application. We demand that it can be established through the bisimulation between cacheless and cache-aware model for a given countermeasure.

Proof Obligation 7.11. *For all $s, s', \bar{s}, \bar{s}'$ where $s \rightarrow_P^* s', \bar{s} \rightarrow_P^* \bar{s}'$, ex-entry($s$), $\bar{s}\ \mathcal{R}_{sim}\ s$, and $\bar{s}'\ \mathcal{R}_{sim}\ s'$, if $II(s, s')$ then $\overline{CM}(\bar{s}, \bar{s}')$*

Hereafter, to enable transferring non-interference properties from the cacheless model to the cache-aware model we assume that the transition relations $\rightarrow_P$ are total functions for both models. As we want to reuse Lemma 6.16 in the following proofs, we require a number of properties of the simulation relation and the invariants.

Lemma 7.12. *For all $\bar{s}_1, \bar{s}_2, s_1, s_2$, such that $\bar{I}(\bar{s}_1)$, $\bar{I}(\bar{s}_2)$, $\bar{I}(s_1)$, and $\bar{I}(s_2)$ as well as $\bar{s}_1\ \mathcal{R}_{sim}\ s_1$ and $\bar{s}_2\ \mathcal{R}_{sim}\ s_2$:*

(1) $\bar{s}_1 \sim_O \bar{s}_2 \Rightarrow s_1 \sim_O s_2$ (2) $\bar{s}_1 \sim_{coh} \bar{s}_2 \Leftrightarrow s_1 \sim_{coh} s_2$

The properties follow directly from the definition of $\mathcal{R}_{sim}$, the coherency of resources in $O \setminus NC$ (Obligation 7.6), and Lemma 6.14.

In addition we require the following technical condition on the relation of the cacheless and cache-aware invariant.

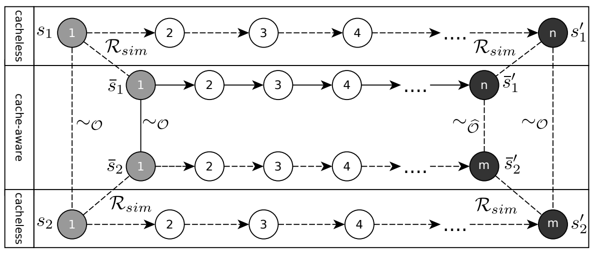

Figure 4: Transferring confidentiality property. Dashed lines indicate proof obligations.

Proof Obligation 7.13. *For each cache-aware state $\bar{s}$ with $\bar{I}(\bar{s})$, there exists a cacheless state s such that $\bar{s}\ \mathcal{R}_{sim}\ s$ and $I(s)$.*

Now we can use Lemma 6.16 to transfer the equivalence of coherent resources after complete kernel executions.

Lemma 7.14. *For all $\bar{s}_1$, $\bar{s}_1'$, $\bar{s}_2$ such that $\bar{I}(\bar{s}_1)$, $\bar{I}(\bar{s}_2)$, ex-entry($\bar{s}_1$), ex-entry($\bar{s}_2$), and $\bar{s}_1 \rightsquigarrow \bar{s}_1'$, if $\bar{s}_1 \sim_O \bar{s}_2$ then $\exists \bar{s}_2'$. $\bar{s}_2 \rightsquigarrow \bar{s}_2'$ and $\bar{s}_1' \sim_{coh} \bar{s}_2'$.*

Proof. We follow the classical approach of mapping both initial cache-aware states to simulating ideal states by Obligation 7.13, transferring observational equivalence by Lemma 7.12.1, and using Lemma 6.16 to obtain a simulating computation $s_1 \rightsquigarrow s_1'$ for $\bar{s}_1 \rightsquigarrow \bar{s}_1'$ on the cacheless level (cf. Fig. 4). Obligation 7.9 yields an observationally equivalent cacheless computation $s_2 \rightsquigarrow s_2'$ starting in the state simulating $\bar{s}_2$. Applying Lemma 6.16 again results in a corresponding cache-aware computation $\bar{s}_2 \rightsquigarrow \bar{s}_2'$. We transfer equivalence of the coherent resources in the final states using Corollary 7.2 and Lemma 7.12.2. ■

It remains to be shown that the kernel execution also maintains observational equivalence for the potentially incoherent memory resources. This property depends on the specific countermeasure used, but can be proven once and for all for a given hardware platform and cache architecture.

Proof Obligation 7.15. *For all $\bar{s}_1$, $\bar{s}_1'$, $\bar{s}_2$, and $\bar{s}_2'$ with $\bar{s}_1 \rightsquigarrow \bar{s}_1'$ and $\bar{s}_2 \rightsquigarrow \bar{s}_2'$ as well as $\bar{I}(\bar{s}_1)$, $\bar{I}(\bar{s}_2)$, ex-entry($\bar{s}_1$), ex-entry($\bar{s}_2$), $\overline{CM}(\bar{s}_1, \bar{s}_1')$ and $\overline{CM}(\bar{s}_2, \bar{s}_2')$, if $\bar{s}_1 \sim_O \bar{s}_2$ then*

$$(\bar{s}_1'.cache, \bar{s}_1'.mem) \approx_{NC} (\bar{s}_2'.cache, \bar{s}_2'.mem).$$

Finally, we combine above results to prove that the confidentiality of the system is preserved on the cache-aware level.

Proof Theorem 5.3. Performing an induction on the computation length, we distinguish application and complete kernel steps. In the first case we apply Lemma 7.8. The invariants on both states are preserved by Lemmas 6.3, 6.7.1, and 6.11. For kernel computations, we first step into non-privileged states $\bar{s}_1''$, $\bar{s}_2''$ deducing $\bar{I}(\cdot)$ and ex-entry($\cdot$) for both states by Theorem 5.1, as well as $\bar{s}_1'' \sim_O \bar{s}_2''$ by Lemma 7.8. By Lemma 7.14 we show the existence of a cache-aware computation $\bar{s}_2'' \rightsquigarrow \bar{s}_2'$ that preserves the equivalence of coherent resources. Using Lemma 6.16 we obtain a corresponding cacheless computation. By Obligation 7.11 we get $\overline{CM}(\bar{s}_1'', \bar{s}_1')$ and $\overline{CM}(\bar{s}_2'', \bar{s}_2')$. Invariants $\bar{I}(\bar{s}_1')$ and $\bar{I}(\bar{s}_2')$ hold due to Obligation 6.12 and Lemma 6.19. We conclude by Obligation 7.15. ■

The core requirements that enable this result are Proof Obligations 7.10, 7.11, and 7.15. Below we discuss how to discharge them for two common countermeasures.

7.5 Correctness of Countermeasures

For a given cache leakage countermeasure, one needs to define predicates CM, $\overline{CM}$ and discharge the related proof obligations. Here we describe the countermeasures of secret-independent code and cache flushing that are well-known to work against cache timing channels. However, in the presence of uncacheable aliases they are not sufficient to prevent information leakage. In particular, if a secret-dependent value in memory is overshadowed by secret-independent dirty cache entry for the same address, the application can still extract the memory value by an uncacheable read.

To tackle this issue we require that all countermeasures sanitise writes of the kernel to addresses in NC by a cache cleaning operation. Then if the core-views of NC are equivalent in two related states, so are the memories. A sufficient verification condition on the cacheless model is imposed as follows.

Proof Obligation 7.16. *For all s, s' such that $CM(s, s')$ and $s \rightsquigarrow s'$ performs $dop_1 \cdots dop_n$ with $dop_i = wt(R_i)$ and $pa \in R_i$, if $pa \in NC$ then there is a $j > i$ such that $dop_j = cl(R_j)$ and $pa \in R_j$.*

This condition is augmented with countermeasure-specific proof obligations that guarantee the observational equivalence of caches.

7.5.1 Secret-Independent Code. The idea behind this countermeasure is that no information can leak through the data caches, if the kernel accesses the same memory addresses during all computations from observationally equivalent entry states, i.e., the memory accesses do not depend on the confidential resources. We approximate this functional requirement by predicate $CM(s, s')$:

Proof Obligation 7.17. *If a complete kernel execution $s \rightsquigarrow s'$ performs $dop_1 \cdots dop_n$ on resources $R_1 \cdots R_n$, then for all s_2, s_2' with $s \sim_O s_2$ and $s_2 \rightsquigarrow s_2'$ operations $dop_1' \cdots dop_n'$ on $R_1' \cdots R_n'$ are performed such that addresses in R_i and R_i' map to the same set of tags.*

Note that this allows to access different addresses depending on a secret, as long as both addresses have the same tag. Such trace properties can be discharged by a relational analysis at the binary level [6]. On the cache-aware level, the property is refined by:

$$\overline{CM}(\bar{s}, \bar{s}') \overset{\text{def}}{\equiv} \forall \bar{s}_2, \bar{s}_2'.\ \bar{s} \rightsquigarrow \bar{s}' \wedge \bar{s} \approx_{NC} \bar{s}_2 \wedge \bar{s}_2 \rightsquigarrow \bar{s}_2' \Rightarrow \bar{s}' \approx_{NC} \bar{s}_2'$$

The refinement (Obligation 7.11) is proven similar to Lemma 7.14 using the Lemma 6.16 and the determinism of the hardware when applying the confidentiality of the system in the cacheless model (Lemma 7.9) as well as Obligation 7.16. Then, proving Obligation 7.15 is straight-forward.

7.5.2 Cache Flushing. A simple way for the kernel to conceal its secret-dependent operations is to flush the cache before returning to the application. The functional requirement $CM(s, s')$ implies:

Proof Obligation 7.18. *For any kernel computation $s \rightsquigarrow s'$ performing data operations $dop_1 \cdots dop_n$:*

(1) there exists a contiguous subsequence $dop_i \cdots dop_j$ of clean operations on all cache lines,
(2) operations $dop_1 \cdots dop_j$ do not write resources in NC,
(3) operations $dop_{j+1} \cdots dop_n$ accessed address tags and written values for NC do not depend on confidential resources,

Condition (3) is formalised and proven like the secret-independent code countermeasure discussed above. Condition (1) can be verified by binary code analysis, checking that the expected sequence of clean operations is eventually executed. We identify the resulting state by $fl(s)$. Condition (2) is not strictly necessary, but it reduces the overall verification effort. Then we demand by $\overline{CM}(\bar{s}, \bar{s}')$:

Definition. *For all $\bar{s}''$ such that $\bar{s} \rightarrow_P^* \bar{s}'' \rightarrow_P^* \bar{s}'$:*

(1) if $fl(\bar{s}'')$ or $\bar{s}''$ is a preceding state in the computation, then for all $pa \in NC$ it holds that $Mv(\bar{s}'', pa) = Mv(\bar{s}, pa)$,
(2) if $fl(\bar{s}'')$, all cache lines and their filtered histories are empty,
(3) if $fl(\bar{s}'')$ and $\bar{s}'' \rightsquigarrow \bar{s}'$, then for all computations $\bar{s}_2 \rightsquigarrow \bar{s}_2'$ with $\bar{s}'' \approx_{NC} \bar{s}_2$ we have $\bar{s}' \approx_{NC} \bar{s}_2'$.

Here, Condition (1) holds as resources NC are not written and are affected only by cache evictions which preserve the memory-view. Condition (2) follow directly from the cache-flush semantics. Condition (3) is discharged using Lemma 6.16 and Obligation 7.18.3. In the proof of Obligation 7.15 we establish memory equivalence between intermediate states $\bar{s}_1''$ and $\bar{s}_2''$ where $fl(\bar{s}_1'')$ and $\bar{s}_2''$ using Condition (1). By Condition (2) we obtain $\bar{s}_1'' \approx_{NC} \bar{s}_2''$ and conclude by Condition (3).

8 CASE STUDY

As a case study we use a hypervisor capable of hosting a Linux guest that has been formally verified previously on a cacheless model [16, 27] and vulnerable to attacks based on cache storage channel [26].

A hypervisor is a system software which controls access to resources and can be used to created isolated partitions on a shared hardware. The hypervisor paravirtualizes the platform for several guests. Only the hypervisor executes in privileged mode, while guests entirely run in non-privileged mode and need to invoke hypervisor functionalities to modify critical resources, such as page-tables. The hypervisor uses direct paging [36] to virtualize the memory subsystem. Using this approach a guest prepares a page-table in its own memory, which after validation, is used by the hypervisor to configure the MMU, without requiring memory copy operations. For validated page-tables the hypervisor ensures that the guest has no writable accesses to the page-tables, thus ensuring that the guest cannot change the MMU configuration. This mechanism makes the hypervisor particularly relevant for our purpose since the critical resources change dynamically and ownership transfer is used for communication.

To efficiently implement direct paging, the hypervisor keeps a type (either *page-table* or *data*) and a reference counter for each physical memory page. The counter tracks (i) for a data-page the number of virtual aliases that enable non-privileged writable accesses to this page, and (ii) for a page-table the number time the page is used as page-table. The intuition is that the hypervisor can change type of a page (e.g., when it allocates or frees a page-table) only if the corresponding reference counter is zero.

The security of this system can be subverted if page-tables are accessed using virtual aliases with mismatched cacheability attributes: The application can potentially mislead the hypervisor to validate stale data and to make a non-validated page a page-table. The hypervisor must counter this threat by preventing incoherent memory from being a page-table. Here, we fix the vulnerability forcing the guest to create page-tables only inside an always cacheable region of the memory.

We instantiate the general concepts of Section 5.1 for the case study hypervisor. The hypervisor uses a static region of physical memory HM to store its stack, data structures and code. This region includes the data structure used to keep the reference counter and type for memory pages. Let $T(s, pa) = pt$ represent that the hypervisor data-structure types the page containing the address pa as *page-table*, then the critical resources are $CR(s) = HM \cup \{pa.\ T(s, pa) = pt\}$.

The state invariant $\bar{I}$ guarantees: (i) soundness of the reference counter, (ii) that the state of the system is well typed; i.e., the MMU uses only memory pages that are typed *page-table*, and page-tables forbid non-privileged accesses to pages outside HM or not typed data. Since the hypervisor uses always cacheability, the invariant also requires (iii) that $HM \subseteq M_{ac}$, if $T(s, pa) = pt$ then $pa \in M_{ac}$, coherency of M_{ac}, and that all aliases to M_{ac} are cacheable.

Obligation 6.1.1 is demonstrated by showing that the functional part of the invariant only depends on page-tables and the internal data-structures, which are critical and contained in HM. Obligations 6.1.2 and 6.1.3 are demonstrated by correctness of always cacheability. Obligation 6.2 trivially holds, since type of memory pages only depends on the internal data structure of the hypervisor, which is always considered critical. Property (ii) of the invariant guarantees Obligation 6.4, since the MMU can use only memory blocks that are typed page-table, which are considered critical. The same property ensures that all critical resources are not writable by the application, thus guaranteeing Obligation 6.5. Moreover, Obligations 6.6 and 1 are guaranteed by soundness of the countermeasure. Obligation 6.15 is proved by showing that the functional invariants for the two models are defined analogously using the core-view. Finally, property (iii) guarantees countermeasure specific Obligations 6.20.1 and 6.20.2.a.

It remains to verify obligations on the hypervisor code. This can be done using a binary analysis tool, since the obligations are defined solely on the cacheless model: (Obligation 6.12) the hypervisor handlers are correct, preserving the functional invariant, (Obligation 6.13) the control flow graph of the hypervisor is correct and the hypervisor never changes its own memory mapping, (Obligation 6.20.2.b) all memory accesses of the hypervisor are in the always cacheable region.

9 IMPLEMENTATION

We used the HOL4 interactive theorem prover [2] to validate our security analysis. We focused the validation on Theorems 5.1 and 5.2, since the integrity threats posed by storage channels cannot be countered by model external means (e.g. information leakage can be neutralised by introducing noise in the cache state).

Following the proof strategy of Section 6, the proof has been divided in three layers: an abstract verification establishing lemmas that are platforms and countermeasures independent, the verification of soundness of countermeasures, and a part that is platform specific. For the latter, we instantiated the models for both ARMv7 and ARMv8, using extensions of the models of Anthony Fox [1, 19] that include the cache model of Section A.1. The formal specification used in our analysis consists of roughly 2500 LOC of HOL4 and the complete proof consists of 10000 LOC.

For the case study we augmented the existing hypervisor [16, 27] with the always cacheability countermeasure. This entailed some engineering effort to adapt the memory allocator of the Linux kernel to allocate page-tables inside M_{ac}. The adaptation required changes to 45 LoC in the hypervisor and an addition of 35 LoC in the paravirtualized Linux kernel and imposes a negligible performance overhead. The formal model of the hypervisor has been modified to include the additional checks performed by the hypervisor to prevent allocation of page-tables outside M_{ac} and to forbid uncacheable aliases to M_{ac}. Similarly, we extended the functional invariant with the new properties guaranteed by the adopted countermeasure. The model of the hypervisor has been used to show that the new design preserves the functional invariant. The invariant and the formal model of ARMv7 processor have been used to validate the proof obligations that do not require binary code analysis.

We did not analyse the binary code of the hypervisor. However, we believe that the proof of the corresponding proof obligations can be automated to a large extent using binary analysis tools (e.g. [27]) or using refinement techniques (e.g. [40]).

Finally, we validated the analysis of Section 7.3, instantiating the verification strategy for the formal model of ARMv7 processor and the flushing countermeasure.

10 CONCLUSION

We presented an approach to verity countermeasures for cache storage channels. We identified the conditions that must be met by a security mechanism to neutralise the attack vector and we verified correctness of some of the existing techniques to counter both integrity and confidentiality attacks.

The countermeasures are formally modelled as new proof obligations that can be imposed on the cacheless model to ensure absence of vulnerability due to cache storage channels. The result of this analysis are theorems in Section 5.3. They demonstrate that a software satisfying a set of proof obligations (i.e. correctly implementing the countermeasure) is not vulnerable due to cache storage channels. Since these proof obligations can be verified using a memory coherent setting, existing verification tools can be used to analyse the target software.

While this paper focuses on storage channels built on data-caches, the countermeasures and techniques can be used to counter leakage through timing channels. Also, similar countermeasures can prevent confidentiality attacks using memory incoherence induced by instruction caches.

Note that the security analysis requires trustworthy models of hardware, which are needed to verify platform dependent proof obligations. Some of these properties (e.g. Assumption 7.1) require extensive tests to demonstrate that corner cases are correctly handled by models. For example, while the conventional wisdom is that flushing caches can close side-channels, a new study [22] showed flushing does not sanitise caches thoroughly and leaves some channels active, e.g. instruction cache attack vectors.

There are several open questions concerning side-channels due to similar shared low-level hardware features such as second-level cashes, TLBs, instruction caches, and branch prediction units, which undermine the soundness of formal verification. This is an unsatisfactory situation since formal proofs are costly and should pay off by giving reliable guarantees. Moreover, the complexity of contemporary hardware is such that a verification approach allowing reuse of models and proofs as new hardware features are added is essential for formal verification in this space to be economically sustainable.

REFERENCES

[1] *ARMv8 model.* Available from: http://www.cl.cam.ac.uk/~acjf3/l3/isa-models.tar.bz2. Accessed: 2017-05-19.

[2] *HOL4.* Available from: http://hol.sourceforge.net/. Accessed: 2017-05-19.

[3] *seL4 Project.* Available from: http://sel4.systems/. Accessed: 2017-04-21.

[4] Eyad Alkassar, Mark A. Hillebrand, Dirk Leinenbach, Norbert Schirmer, Artem Starostin, and Alexandra Tsyban. 2009. Balancing the Load. *J. Autom. Reasoning* 42, 2-4 (2009), 389–454. https://doi.org/10.1007/s10817-009-9123-z

[5] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. 2016. Verifiable Side-Channel Security of Cryptographic Implementations: Constant-Time MEE-CBC. In *Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers.* 163–184. https://doi.org/10.1007/978-3-662-52993-5_9

[6] Musard Balliu, Mads Dam, and Roberto Guanciale. 2014. Automating Information Flow Analysis of Low Level Code. In *Proceedings of the Conference on Computer and Communications Security (CCS'14).* ACM, 1080–1091. https://doi.org/10.1145/2660267.2660322

[7] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. *ACM Operating Systems Review* 37, 5 (2003), 164–177.

[8] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Jesús Mauricio Chimento, and Carlos Luna. 2014. Formally verified implementation of an idealized model of virtualization. In *Proceedings of the 19th International Conference on Types for Proofs and Programs (TYPES'13).* 45–63.

[9] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, and Carlos Luna. 2011. Formally verifying isolation and availability in an idealized model of virtualization. In *International Symposium on Formal Methods.* Springer, 231–245.

[10] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, and Carlos Luna. 2012. Cache-Leakage Resilient OS Isolation in an Idealized Model of Virtualization. In *Proc. CSF'12.* IEEE Computer Society, Washington, DC, USA, 186–197. https://doi.org/10.1109/CSF.2012.17

[11] Hanno Becker, Juan Manuel Crespo, Jacek Galowicz, Ulrich Hensel, Yoichi Hirai, César Kunz, Keiko Nakata, Jorge Luis Sacchini, Hendrik Tews, and Thomas Tuerk. 2016. *Combining Mechanized Proofs and Model-Based Testing in the Formal Analysis of a Hypervisor.* Springer International Publishing, Cham, 69–84. https://doi.org/10.1007/978-3-319-48989-6_5

[12] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *CAV.* 463–469.

[13] Hind Chfouka, Hamed Nemati, Roberto Guanciale, Mads Dam, and Patrik Ekdahl. 2015. Trustworthy prevention of code injection in linux on embedded devices. In *Proceedings of the 20th European Symposium on Research in Computer Security.* Springer, 90–107.

[14] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. 2014. The Last Mile: An Empirical Study of Timing Channels on seL4. In *Proceedings of the Conference on Computer and Communications Security (CCS'14).* ACM, 570–581. https://doi.org/10.1145/2660267.2660294

[15] David Costanzo, Zhong Shao, and Ronghui Gu. 2016. End-to-end Verification of Information-flow Security for C and Assembly Programs. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16).* ACM, New York, NY, USA, 648–664. https://doi.org/10.1145/2908080.2908100

[16] Mads Dam, Roberto Guanciale, Narges Khakpour, Hamed Nemati, and Oliver Schwarz. 2013. 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).* ACM, 223–234.

[17] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2013. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *Proceedings of the 22Nd USENIX Conference on Security (SEC'13).* USENIX Association, Berkeley, CA, USA, 431–446. http://dl.acm.org/citation.cfm?id=2534766.2534804

[18] Goran Doychev and Boris Köpf. 2016. Rigorous Analysis of Software Countermeasures against Cache Attacks. *CoRR* abs/1603.02187 (2016). http://arxiv.org/abs/1603.02187

[19] Anthony Fox and Magnus O. Myreen. 2010. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. In *Proceedings of the First International Conference on Interactive Theorem Proving (ITP'10).* Springer-Verlag, Berlin, Heidelberg, 243–258. https://doi.org/10.1007/978-3-642-14052-5_18

[20] Anthony C. J. Fox and Magnus O. Myreen. 2010. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. In *Proc. ITP'10.* 243–258.

[21] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2016. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *Journal of Cryptographic Engineering* - (dec 2016), 1–27. https://doi.org/10.1007/s13389-016-0141-6

[22] Q. Ge, Y. Yarom, and G. Heiser. 2016. Do Hardware Cache Flushing Operations Actually Meet Our Expectations? *ArXiv e-prints* (Dec. 2016).

[23] Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao, and David Costanzo. 2011. CertiKOS: a certified kernel for secure cloud computing. In *Proceedings of the Second Asia-Pacific Workshop on Systems (APSys'11).* ACM, 3.

[24] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan Newman Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep specifications and certified abstraction layers. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 595–608.

[25] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16).* USENIX Association, Berkeley, CA, USA, 653–669. http://dl.acm.org/citation.cfm?id=3026877.3026928

[26] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. 2016. Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016.* 38–55. https://doi.org/10.1109/SP.2016.11

[27] Roberto Guanciale, Hamed Nemati, Mads Dam, and Christoph Baumann. 2016. Provably secure memory isolation for Linux on ARM. *Journal of Computer Security* 24, 6 (2016), 793–837. https://doi.org/10.3233/JCS-160558

[28] Constance L. Heitmeyer, Myla Archer, Elizabeth I. Leonard, and John McLean. 2006. Formal specification and verification of data separation in a separation kernel for an embedded system *(CCS).* 346–355. https://doi.org/10.1145/1180405.1180448

[29] Mark A Hillebrand, Thomas In der Rieden, and Wolfgang J Paul. 2005. 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).* IEEE, 309–316.

[30] Wei-Ming Hu. 1992. Reducing Timing Channels with Fuzzy Time. *J. Comput. Secur.* 1, 3-4 (May 1992), 233–254. http://dl.acm.org/citation.cfm?id=2699806.2699810

[31] Narges Khakpour, Oliver Schwarz, and Mads Dam. 2013. Machine assisted proof of ARMv7 instruction level isolation properties. In *Certified Programs and Proofs.* Springer, 276–291.

[32] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. 2012. STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud. In *USENIX.* 189–204. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/kim

[33] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.* 32, 1 (2014), 2. https://doi.org/10.1145/2560537

[34] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP'09).* ACM, 207–220. https://doi.org/10.1145/1629575.1629596

[35] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. 2012. Automatic Quantification of Cache Side-channels. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV'12).* Springer-Verlag, Berlin, Heidelberg, 564–580. https://doi.org/10.1007/978-3-642-31424-7_40

[36] Hamed Nemati, Roberto Guanciale, and Mads Dam. 2015. Trustworthy Virtualization of the ARMv7 Memory Subsystem. In *SOFSEM.* 578–589. https://doi.org/10.1007/978-3-662-46078-8_48

[37] PROSPER 2015. Reduce resolution of performance.now to prevent timing attacks. https://bugs.chromium.org/p/chromium/issues/detail?id=506723/. (2015). https://bugs.chromium.org/p/chromium/issues/detail?id=506723/

[38] Willem-Paul de Roever and Kai Engelhardt. 2008. *Data Refinement: Model-Oriented Proof Methods and Their Comparison* (1st ed.). Cambridge University Press, New York, NY, USA.

[39] Oliver Schwarz and Mads Dam. 2016. Automatic Derivation of Platform Noninterference Properties. In *International Conference on Software Engineering and Formal Methods.* Springer International Publishing, 27–44.

[40] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. 2013. Translation Validation for a Verified OS Kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 471–482. https://doi.org/10.1145/2491956.2462183

[41] Bruno R Silva, Diego Aranha, and Fernando MQ Pereira. December 2015. Uma Técnica de Análise Estática para Detecç ao de Canais Laterais Baseados em Tempo. In *In Brazilian Symposium on Information and Computational Systems Security*. FlorianÃşpolis, SC, BR, 16–29.

[42] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS '08)*. Springer-Verlag, Berlin, Heidelberg, 1–25. https://doi.org/10.1007/978-3-540-89862-7_1

[43] Udo Steinberg and Bernhard Kauer. 2010. NOVA: A Microhypervisor-based Secure Virtualization Architecture. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*. ACM, 209–222. https://doi.org/10.1145/1755913.1755935

[44] Hendrik Tews, Marcus Völp, and Tjark Weber. 2009. Formal Memory Models for the Verification of Low-Level Operating-System Code. *J. Autom. Reasoning* 42, 2-4 (2009), 189–227. https://doi.org/10.1007/s10817-009-9122-0

[45] Mohit Tiwari, Jason K. Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. 2011. Crafting a Usable Microkernel, Processor, and I/O System with Strict and Provable Information Flow Security. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. ACM, New York, NY, USA, 189–200. https://doi.org/10.1145/2000064.2000087

[46] Bhanu C. Vattikonda, Sambit Das, and Hovav Shacham. 2011. Eliminating Fine Grained Timers in Xen. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop (CCSW '11)*. ACM, New York, NY, USA, 41–46. https://doi.org/10.1145/2046660.2046671

[47] Bárbara Vieira. 2012. *Formal verification of cryptographic software implementations*. Ph.D. Dissertation. Universidade do Minho, Portugal. https://repositorium.sdum.uminho.pt/bitstream/1822/20770/1/B%C3%A1rbara%20Isabel%20de%20Sousa%20Vieira.pdf

[48] Zhenghong Wang and Ruby B. Lee. 2007. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*. ACM, New York, NY, USA, 494–505. https://doi.org/10.1145/1250662.1250723

[49] Michael Weiß, Benedikt Heinz, and Frederic Stumpf. 2012. A Cache Timing Attack on AES in Virtualization Environments *(Proceedings of the 16th International Conference on Financial Cryptography and Data Security",)*, Angelos D. Keromytis (Ed.). Springer, 314–328, isbn= 978–3–642–32946–3,.

[50] Matthew M. Wilding, David A. Greve, Raymond J. Richards, and David S. Hardin. 2010. Formal Verification of Partition Management for the AAMP7G Microprocessor. In *Design and Verification of Microprocessor Systems for High-Assurance Applications*. 175–191. https://doi.org/10.1007/978-1-4419-1539-9_6

[51] Lu Zhao, Guodong Li, Bjorn De Sutter, and John Regehr. 2011. ARMor: Fully verified software fault isolation. In *EMSOFT*. 289–298.

[52] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. 2016. A Software Approach to Defeating Side Channels in Last-Level Caches. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 871–882. https://doi.org/10.1145/2976749.2978324

$$SL(C,i) \equiv C(i).slice, \quad W(C,i,t) \equiv (SL(C,i))(t), \quad H(C,i) \equiv C(i).hist,$$
$$c\text{-}cnt(C,pa) \equiv W(C,i,t).D(wi), \quad c\text{-}hit(C,pa) \equiv W(C,i,t) \neq \bot,$$
$$c\text{-}dirty(C,pa) \equiv c\text{-}hit(C,pa) \wedge W(C,i,t).d$$

Figure 5: Abbreviations; for $pa \in \mathbb{PA}$ mapped by $va \in \mathbb{VA}$ and where i, t and wi are the corresponding set index, tag and word index, and $C \in \mathbb{C}$.

$$\begin{array}{lll}
touch & : & \mathbb{SL} \times \mathbb{A}^* \times \mathbb{B}^T \times \mathbb{B}^L \times (\mathbb{B}^w \cup \{\bot\}) \rightarrow \mathbb{SL} \times \mathbb{A}^* \\
lfill & : & \mathbb{SL} \times \mathbb{A}^* \times \mathbb{M} \times \mathbb{PA} \rightarrow \mathbb{SL} \times \mathbb{A}^* \\
evict & : & \mathbb{SL} \times \mathbb{A}^* \times \mathbb{B}^T \rightarrow \mathbb{SL} \times \mathbb{A}^* \\
wriba & : & \mathbb{L} \times \mathbb{M} \rightarrow \mathbb{M} \qquad\qquad evict? \;\; : \;\; \mathbb{A}^* \times \mathbb{B}^T \rightarrow \mathbb{B}^T \cup \bot
\end{array}$$

Figure 6: Internal operations of the cache. The fifth input of *touch* is either $\bot$ for read accesses or the value $v' \in \mathbb{B}^w$ being written to the cache line.

A APPENDIX

A.1 Generic Data Cache Model

Below with give the definition of our generic cache model which is the basis for discharging Assumptions 7.1, 7.7 and Proof Obligations 7.11, 7.15.

Our cache model does not fix the cache size, the number of ways, the format of set indices, lines, tags, or the eviction policy. Moreover, cache lines can be indexed according to physical or virtual addresses, but are physically tagged, and our model allows both write-back and write-through caching.

Let n and m are the bitsize of virtual and physical addresses, then $\mathbb{VA} = \mathbb{B}^{n-\alpha}$ and $\mathbb{PA} = \mathbb{B}^{m-\alpha}$ are the sets of word-aligned addresses where 2^α is the word size in number of bytes (e.g. in a 64-bit architecture $\alpha = 3$). Our cache has 2^N sets, its lines are of size 2^L words and we use functions $si : \mathbb{VA} \times \mathbb{PA} \rightarrow \mathbb{B}^N$, $tag : \mathbb{PA} \rightarrow \mathbb{B}^T$, and $widx : \mathbb{VA} \times \mathbb{PA} \rightarrow \mathbb{B}^L$ to compute the set indices, tags and word indices of the cache. We have $T = m - (N + L)$ for physically-indexed and $T = m - L$ for virtually-indexed caches.

We define a *cache slice* as a mapping from a tag to a cache line, $\mathbb{SL} = \mathbb{B}^T \rightarrow \mathbb{L} \cup \{\bot\}$; $\bot$ if no line is cached for the given tag. A line $ln \in \mathbb{L}$ is a pair $(ln.D, ln.d) \in \mathbb{D} \times \mathbb{B}$, $\mathbb{D}$ is the mapping from word-indices to data, and d indicates the line dirtiness.

Then, a cache $C \in \mathbb{C}$ maps a set index i to a slice $C(i).slice \in \mathbb{SL}$ and a history of actions performed $C(i).hist \in \mathbb{A}^*$ on that slice, i.e., $\mathbb{C} = \mathbb{B}^N \rightarrow \mathbb{SL} \times \mathbb{A}^*$. The history records internal cache actions of type $\mathbb{A} = \{\text{touch}_{\text{r|w}}\, t, \text{evict}\, t, \text{lfill}\, t\}$, where $t \in \mathbb{B}^T$ denotes the tag associated with the action.

Here, $\text{touch}_{\text{r|w}}\, t$ denotes a read or write access to a line tagged with t, lfill t occurs when a line for tag t is loaded from memory and placed in the cache. Similarly, evict t represents the eviction of a line with tag t. To simplify formalization of properties, we define a number of abbreviations in Figure 5. Note that in case of virtual indexing, we assume an unaliased cache, i.e., each physical address is cached in at most one cache slice.

The semantics of internal cache actions on a cache slice and history are given by the corresponding functions in Figure 6. If tag t hits the cache in the line set i, then *touch* and *evict* update the cache $C \in \mathbb{C}$ as follows, where $D'_i = W(C,i,t).D[wi \mapsto v']$ is the

$$\begin{array}{ll}
\mathit{fill} & : \mathbb{C}\times\mathbb{M} \rightarrow \mathbb{VA}\times\mathbb{PA} \rightarrow \mathbb{C}\times\mathbb{M} \\
\mathit{read} & : \mathbb{C}\times\mathbb{M}\times\mathbb{VA}\times\mathbb{PA} \rightarrow \mathbb{C}\times\mathbb{M}\times\mathbb{B}^{w} \\
\mathit{write} & : \mathbb{C}\times\mathbb{M}\times\mathbb{VA}\times\mathbb{PA}\times\mathbb{B}^{w} \rightarrow \mathbb{C}\times\mathbb{M} \\
\mathit{invba} & : \mathbb{C}\times\mathbb{M}\times\mathbb{VA}\times\mathbb{PA} \rightarrow \mathbb{C}\times\mathbb{M} \\
\mathit{clnba} & : \mathbb{C}\times\mathbb{M}\times\mathbb{VA}\times\mathbb{PA} \rightarrow \mathbb{C}\times\mathbb{M}
\end{array}$$

Figure 7: Core accessible interface. Functions *clnba* and *invba* clean and invalidate cache lines for given virtual and physical addresses; *clnba* only resets the dirty bit and writes back dirty lines, while *invba* also evicts the line. Function *fill* is used to pre-load lines into the cache.

resulting line of a write operation at word index wi with value v':

$$(SL(C',i),\ H(C',i)) := \begin{cases} (SL(C,i) & , H(C,i)@(\mathrm{touch_r}\ t)) : \mathrm{touch_r}\ t \\ (SL(C,i)[t \mapsto (D'_i, 1)] & , H(C,i)@(\mathrm{touch_w}\ t)) : \mathrm{touch_w}\ t \\ (SL(C,i)[t \mapsto \bot] & , H(C,i)@(\mathrm{evict}\ t)) : \mathrm{evict}\ t \end{cases}$$

For cache misses on a physical address pa with tag t, function *lfill* loads memory content $v = mem[pa + L \cdot 2^{\alpha} - 1 : pa]$ and places it as a clean line into the cache: $(SL(C',i), H(C',i)) := (SL(C,i)[t \mapsto (\lambda i.v[i], 0)],\ H(C,i)@(\mathrm{lfill}\ t))$.

As the cache can only store limited amounts of lines, eviction policy $evict?(H,t)$ returns the tag of the line to be replaced at a line fill for tag t, or $\bot$ if eviction is not required. Evicted dirty lines are then written back into the memory using function *wriba*. As explained in the main text, the eviction policy is only depending on a finite subset of the history represented by filter function $\varphi : \mathbb{A}^* \rightarrow \mathbb{A}^*$ (Assumption 7.1).

The definitions above provide the minimal building blocks to define a detailed cached memory system that responds to reads, writes, and eviction requests from the core. Below we give an example.

A.2 Operational Write-back Cache Semantics

With the help of functions in Fig. 6 and abbreviations of Fig. 5. we give semantics to a write-back cache with LRU replacement strategy. Figure 7 lists the interface available to the core to control the cache. When one of these functionalities is called, the cache uses the internal actions to update its state according to the requested operation. In what follows we set $t = tag(va, pa)$ and $i = si(va, pa)$.

Function $\mathit{fill}(C, M, va, pa)$ loads the cache $C \in \mathbb{C}$ by invoking *lfill*. However, if the cache is full the eviction policy determines the line to evict to make space. Using *wriba*, the evicted line is then written back in the memory $M \in \mathbb{M}$. We denote this conditional eviction by $alloc(C, M, va, pa)$, which is defined as:

$$\begin{cases} (C[i \mapsto evict(SL(C,i), H(C,i), t')], wriba(W(C,i,t'), M)) : evict?(H(C,i),t) = t' \\ (C, M) : evict?(H(C,i),t) = \bot \end{cases}$$

We save the result of this function as the pair $(\bar{C}, \bar{M})$. Moreover, if an alias for the filled line is present in another cache slice, i.e., a line with the same tag, that line has to be evicted as well. We define this condition as follows:

$$alias?(C,t,i,i') \equiv \exists va', pa'.\ i' = si(va',pa') \wedge i \neq i' \wedge W(C, si(va',pa'), t)\ \neq \bot$$

Then alias detection and eviction $alias(C, M, va, pa)$ is defined as:

$$\begin{cases} (C[i' \mapsto evict(SL(C,i'), H(C,i'), t)], wriba(W(C,i',t), M)) & : alias?(C,t,i,i') \\ (C, M) & : \text{otherwise} \end{cases}$$

The result of this function applied to $(\bar{C}, \bar{M})$ is saved as $(\hat{C}, \hat{M})$. The combination of these actions with a line fill is denoted by $\mathit{fillwb}(C, M, va, pa)$ and defined below.

$$\begin{cases} (\hat{C}[i \mapsto \mathit{lfill}(SL(\hat{C},i), H(\hat{C},i), M, pa)], \hat{M}) & : \neg c\text{-}hit(C, pa) \\ (C, M) & : \text{otherwise} \end{cases}$$

Thus, $(\tilde{C}, \tilde{M}) = \mathit{fillwb}(C, M, pa, va)$. Now the definition of reading, writing, flushing, and cleaning the cache is straightforward, for $x = C, M, va, pa$ we have:

$$\begin{array}{lcl}
read(x) & = & (\tilde{C}[i \mapsto touch(SL(\tilde{C},i), H(\tilde{C},i), t, i, \bot), \tilde{M}, c\text{-}cnt(\tilde{C}, pa)) \\
write(x, v) & = & (\tilde{C}[i \mapsto touch(SL(\tilde{C},i), H(\tilde{C},i), t, i, v), \tilde{M}) \\
invba(x) & = & (C[i \mapsto evict(SL(C,i), H(C,i), t)], wriba(W(C,i,t), M)) \\
clnba(x) & = & (C[i \mapsto SL(C,i)[t \mapsto W(C,i,t)[d \mapsto 0]]], wriba(W(C,i,t), M))
\end{array}$$

Other cache functionalities can be defined similarly. It remains to instantiate the eviction policy and its filter φ. We choose the Least Recently Used (LRU) policy, which always replaces the element that was not used for the longest time if there is no space. In a k set associative cache we model LRU as a decision queue $q \in \mathbb{Q}$ of size k. This queue maintains an order on how the tags in a cache set are accessed. In this ordering the queue's front is the one that has been touched most recently and its back points to the tag to be replaced next upon a miss (or an empty way).

We assume two functions to manipulate queue content: (1) $push : \mathbb{Q} \times \mathbb{B}^T \rightarrow \mathbb{Q}$ adjusts the queue to make room for the coming tag, inserting the tag at the front of the queue, and (2) $pop : \mathbb{Q} \times \mathbb{B}^T \rightarrow \mathbb{Q}$ removes the input tag and shifts all elements to the front. Additionally $back : \mathbb{Q} \rightarrow \mathbb{B}^T \cup \bot$ returns the back ($k - 1th$) element in the queue or $\bot$ if there is still space. We construct the queue recursively from the history h with the function $Cons : h \rightarrow \mathbb{Q}$:

$$\begin{array}{lcl}
Cons(\varepsilon) & = & \emptyset \\
Cons(h@a) & = & \begin{cases} pop_t(Cons(h)) & : a = \mathrm{evict}\ t \\ push_t(Cons(h)) & : a = \mathrm{lfill}\ t \\ push_t(pop_t(Cons(h))) & : a = \mathrm{touch_{r|w}}\ t \end{cases}
\end{array}$$

Then the eviction policy *evict?* is defined by:

$$evict?_{lru}(h, t) := back(Cons(h))$$

Proposition A.1. *On each cache slice, the LRU replacement strategy depends only on the action history for the (at most k) present tags, since their lines were last filled into the cache.*

We capture this part of the action history through the filter function $\varphi_{lru} : \mathbb{A}^* \rightarrow \mathbb{A}^*$. Its definition depends on a number of helper functions. First we introduce the last action $last(h, t)$ on a tag t in history h and the set of tags $T_h \subset \mathbb{B}^T$ that are currently present in a slice according to history information. For action $a \in \mathbb{A}$, $tag(a)$ returns the associated tag.

$$last(\varepsilon, t) \ = \ \bot \qquad last(h@a, t) \ = \ \begin{cases} a & : tag(a) = t \\ last(h,t) & : \text{otherwise} \end{cases}$$

$$T_h \ = \ \{t \mid last(h,t) \notin \{\bot, \mathrm{evict}\ t\}\}$$

For a set of tags $T \subset \mathbb{B}^T$ we define the LRU filter includes the least recent fills and subsequent touches to tags in the set, but leaves out evictions and actions on irrevelant tags.

$$\varphi_{lru}(\varepsilon, T) \quad = \quad \varepsilon \qquad\qquad \varphi_{lru}(h, \emptyset) \quad = \quad \varepsilon$$

$$\varphi_{lru}(h@a, T) \quad = \quad \begin{cases} \varphi_{lru}(h, T)@a & : \ \exists t.\ a = \mathrm{touch}_{\mathrm{r|w}}\ t \wedge t \in T \\ \varphi_{lru}(h, T \setminus \{t\})@a & : \ \exists t.\ a = \mathrm{lfill}\ t \wedge t \in T \\ \varphi_{lru}(h, T) & : \ \text{otherwise} \end{cases}$$

Then we set $\varphi_{lru}(h) := \varphi_{lru}(h, T_h)$.

A.3 Proof of Assumption 7.1

Proving the claim

$$evict?(h, t) = evict?(\varphi_{lru}(h), t)$$

boils down to the following property.

Lemma A.2. *For all $h \in \mathbb{A}^*$, $Cons(h) = Cons(\varphi_{lru}(h))$.*

It is proven by induction on the length of h, with one additional induction hypothesis:

$$\forall t \in T_h.\ pop_t(Cons(h), t) = Cons(\varphi_{lru}(h, T_h \setminus \{t\}))$$

We need two invariants of the cache semantics for histories $h' = h@a$: (1) if a is an eviction or a touch on tag t, then $t \in T_h$, and (2) if a is a line fill of tag t then $t \notin T_h$. Moreover, we use idempotence and commutativity properties of applying $\varphi_{lru}(h, T)$ several times.

$$\begin{aligned} \varphi_{lru}(h, T) \quad &= \quad \varphi_{lru}(\varphi_{lru}(h, T), T) \\ \varphi_{lru}(\varphi_{lru}(h, T), T') \quad &= \quad \varphi_{lru}(\varphi_{lru}(h, T'), T) \end{aligned}$$

With these arguments the detailed proof becomes a straightforward exercise. We also introduce the following lemmas for later use.

Lemma A.3. *For two histories h_1, h_2 and two tag sets T and $T' \subseteq T$, if $\varphi_{lru}(h_1, T) = \varphi_{lru}(h_2, T)$ then $\varphi_{lru}(h_1, T') = \varphi_{lru}(h_2, T')$.*

We prove this property by induction on the length of the filtered histories using the definition of $\varphi_{lru}(h, T)$.

Lemma A.4. *All cache actions preserve the following invariant on slices i of cache C:*

$$T_{H(C,i)} = \{t \mid W(C, i, t) \neq \bot\}$$

This follows directly from the semantics of the cache actions.

A.4 Proof of Assumption 7.7

Before we conduct the proof we first give a formal definition of observational equivalence of cache and memory wrt. a set of addresses A. Let $TG_A = \{tag(a) \mid a \in A\}$ and $SI_A = \{si(a) \mid a \in A\}$ denote the sets of tags and set indices corresponding to addresses in A, then:

$$\begin{aligned} (C_1, M_1) \approx_A (C_2, M_2) \ \overset{\text{def}}{\equiv} \ & \\ & \forall pa \in A.\ M_1(pa) = M_2(pa) \\ & \wedge \forall i, t.\ W(C_1, i, t) = \bot \Leftrightarrow W(C_2, i, t) = \bot \\ & \wedge \forall i, t.\ W(C_1, i, t).d = W(C_2, i, t).d \\ & \wedge \forall i \in SI_A, t \in TG_A. \\ & \qquad W(C_1, i, t) \neq \bot \Rightarrow W(C_1, i, t) = W(C_2, i, t) \\ & \wedge \forall i.\ \varphi(H(C_1, i)) = \varphi(H(C_2, i)) \end{aligned}$$

Now let $C_1 = \bar{s}_1.cache$, $C_2 = \bar{s}_2.cache$, $M_1 = \bar{s}_1.mem$, $M_2 = \bar{s}_2.mem$, and similar for the primed states, then it needs to be shown:

$$\begin{aligned} & \forall \bar{s}_1, \bar{s}_2, \bar{s}_1', \bar{s}_2', dop. \\ & \qquad Coh(\bar{s}_1, (O \cap \mathbb{PA}) \setminus A) \\ & \quad \wedge \quad Coh(\bar{s}_2, (O \cap \mathbb{PA}) \setminus A) \\ & \quad \wedge \quad \forall pa \in (O \cap \mathbb{PA}) \setminus A.\ Cv(\bar{s}_1, pa) = Cv(\bar{s}_2, pa) \\ & \quad \wedge \quad (C_1, M_1) \approx_A (C_2, M_2) \\ & \quad \wedge \quad \bar{s}_1 \rightarrow_U \bar{s}_1'\ [dop] \\ & \quad \wedge \quad \bar{s}_2 \rightarrow_U \bar{s}_2'\ [dop] \\ & \Longrightarrow \\ & \qquad Coh(\bar{s}_1', (O \cap \mathbb{PA}) \setminus A) \\ & \quad \wedge \quad Coh(\bar{s}_2', (O \cap \mathbb{PA}) \setminus A) \\ & \quad \wedge \quad \forall pa \in (O \cap \mathbb{PA}) \setminus A.\ Cv(\bar{s}_1', pa) = Cv(\bar{s}_1', pa) \\ & \quad \wedge \quad (C_1', M_1') \approx_A (C_2', M_2') \end{aligned}$$

In general, cache operations cannot make coherent resources incoherent, thus we can focus on the last two claims. All memory instructions are broken down into a sequence of internal cache actions so it suffices to make a case split on the possible cache actions $a \in \mathbb{A}$. We outline the general proof strategy for each case below.

$a = \mathrm{touch}_{\mathrm{r}}\ t$ — A read hit does not change contents of cache and memory at all. We only need to consider the changes to the action history of the affected slice i. By definition of φ_{lru} we have:

$$\begin{aligned} \varphi_{lru}(H(C_1', i)) \quad &= \quad \varphi_{lru}(H(C_1, i)@a) \\ &= \quad \varphi_{lru}(H(C_1, i))@a \\ &= \quad \varphi_{lru}(H(C_2, i))@a \\ &= \quad \varphi_{lru}(H(C_2, i)@a) \\ &= \quad \varphi_{lru}(H(C_2', i)) \end{aligned}$$

$a = \mathrm{touch}_{\mathrm{w}}\ t$ — The case of write hits is analoguous to the read case, with the exception that the data content and dirty bit may change. Nevertheless the written line is present in both caches with the same contents and the dirty bit becomes 1 in both states after the write operation. Since the same value is written, we can also show the claim that the data content for tags $t \in TG_A$ are equal.

$a = \mathrm{lfill}\ t$ — A line fill leaves the memory and dirty bits unchanged and since we have the same tag states, the line fill occurs in both caches.

For tags $t \notin TG_A$ that belong to coherent addresses we know that the core-view stays unchanged because a line is only fetched if it was not present in the cache before and the memory content that was visible in the core-view of the pre-state is loaded into the cache to be visible in the core-view of the post-state.

For $t \in TG_A$, relation $\approx_A$ guarantees the equivalence of the memory contents directly for addresses A and again the same line is filled into the cache.

In both cases, the tag states stay equivalent because the same tag is added into the cache slice. Concerning the history of the cache slice, we get from the definition of φ_{lru} with $h_1 = H(C_1, i)$ and $h_2 = H(C_2, i)$:

$$\begin{array}{rcl}
\varphi_{lru}(H(C'_1, i)) & = & \varphi_{lru}(h_1@a) \\
& = & \varphi_{lru}(h_1@a, T_{h_1@a}) \\
& = & \varphi_{lru}(h_1@a, T_{h_1} \cup \{a\}) \\
& = & \varphi_{lru}(h_1, T_{h_1})@a \\
& = & \varphi_{lru}(h_1)@a \\
& = & \varphi_{lru}(h_2)@a \\
& = & \varphi_{lru}(h_2, T_{h_2})@a \\
& = & \varphi_{lru}(h_2@a, T_{h_2} \cup \{a\}) \\
& = & \varphi_{lru}(h_2@a, T_{h_2@a}) \\
& = & \varphi_{lru}(h_2@a) \\
& = & \varphi_{lru}(H(C'_2, i))
\end{array}$$

$a = \text{evict}\ t$ — For coherent resources evictions do not change the core-view, as any line that is evicted was either dirty before and thus written back to memory, maintaining its addresses' core-view, or it was clean but coherent with the corresponding memory content that becomes visible in the core-view after the eviction. If a confidential line is evicted there is nothing more to show.

For tags in a line i tag states and filtered histories are equal. By Assumption 7.1 the eviction policy yields the same result in both states, thus if a line is evicted it is done so in both caches and these lines have the same tag.

For evicted coherent lines or confidential lines we argue as above. For lines belonging to the set A we know that they have the same contents, so if they are dirty, memory changes in the same way. In case they are clean, memories stay unchanged and are still equivalent.

In all cases the tag state is manipulated in the same way, as the same tags are evicted, thus they stay equal. The filtered histories for line i are still the same by definition of φ_{lru} and the equality of tag states.

$$\begin{array}{rcll}
\varphi_{lru}(H(C'_1, i) & = & \varphi_{lru}(h_1@a) & \\
& = & \varphi_{lru}(h_1@a, T_{h_1@a}) & \\
& = & \varphi_{lru}(h_1@a, T_{h_1} \setminus \{a\}) & \\
& = & \varphi_{lru}(h_1, T_{h_1} \setminus \{a\}) & \\
& = & \varphi_{lru}(h_1, T_{h_2} \setminus \{a\}) & (\mathit{Lemma\ A}.4) \\
& = & \varphi_{lru}(h_2, T_{h_2} \setminus \{a\}) & (\mathit{Lemma\ A}.3) \\
& = & \varphi_{lru}(h_2@a, T_{h_2} \setminus \{a\}) & \\
& = & \varphi_{lru}(h_2@a, T_{h_2@a}) & \\
& = & \varphi_{lru}(h_2@a) & \\
& = & \varphi_{lru}(H(C'_2, i)) &
\end{array}$$

This concludes the proof of Assumption 7.7.