System-level non-interference for constant-time cryptography

Gilles Barthe[1], Gustavo Betarte[2], Juan Diego Campo[2], Carlos Luna[2], and David Pichardie[3]

[1] IMDEA Software, Madrid, Spain.
[2] InCo, Facultad de Ingeniería, Universidad de la República, Uruguay
[3] ENS Bretagne/INRIA, France

Abstract. Cache-based attacks are a class of side-channel attacks that are particularly effective in virtualized or cloud-based environments, where they have been used to recover secret keys from cryptographic implementations. One common approach to thwart cache-based attacks is to use *constant-time* implementations, i.e. which do not branch on secrets and do not perform memory accesses that depend on secrets. However, there is no rigorous proof that constant-time implementations are protected against concurrent cache-attacks in virtualization platforms with shared cache; moreover, many prominent implementations are not constant-time. An alternative approach is to rely on system-level mechanisms. One recent such mechanism is stealth memory, which provisions a small amount of private cache for programs to carry potentially leaking computations securely. Stealth memory induces a weak form of constant-time, called *S-constant-time*, which encompasses some widely used cryptographic implementations. However, there is no rigorous analysis of stealth memory and S-constant-time, and no tool support for checking if applications are S-constant-time.
We propose a new information-flow analysis that checks if an x86 application executes in constant-time, or in S-constant-time. Moreover, we prove that constant-time (resp. S-constant-time) programs do not leak confidential information through the cache to other operating systems executing concurrently on virtualization platforms (resp. platforms supporting stealth memory). The soundness proofs are based on new theorems of independent interest, including isolation theorems for virtualization platforms (resp. platforms supporting stealth memory), and proofs that constant-time implementations (resp. S-constant-time implementations) are non-interfering with respect to a strict information flow policy which disallows that control flow and memory accesses depend on secrets. We formalize our results using the Coq proof assistant and we demonstrate the effectiveness of our analyses on cryptographic implementations, including PolarSSL AES, DES and RC4, SHA256 and Salsa20.

Keywords: Non-interference, cache-based attacks, constant-time cryptography, stealth memory, Coq

1 Introduction

Cache-based attacks are side-channel attacks in which a malicious party is able to obtain confidential data through observing cache accesses of programs. They are particularly effective in cloud-based environments, where hardware support is virtualized and shared among tenants. In such settings, a malicious tenant can manage that an operating system under its control co-resides with the operating system which executes the program that the attacker targets. This allows the attacker to share the cache with its victim and to make fine-grained observations about its own cache hits and misses; using this knowledge, the attacker can then successfully retrieve confidential data of the program. Cache-based attacks are widely applicable, but are specially devastating against cryptographic implementations that form the security backbone of many Internet protocols (e.g. TLS) or wireless protocols (e.g. WPA2). Known targets of cache-based attacks include widely used implementations of AES, DES, ECDSA and RC4.

There are essentially two approaches for protecting oneself against cache-based attacks. The first approach is to build implementations that do not leak information through the cache. One common strategy is to make implementations *constant-time*[4], i.e. do not branch on secrets and do not perform memory accesses that

[4] The terminology is inherited from cryptography, where it is generally used for source level programs that do not branch on secrets and do not perform array accesses with indices that depend on secrets. Because the property

which depend on secrets. There exist constant-time implementations of many cryptographic algorithms, including AES, DES, RC4, SHA256, TEA, and Salsa20, and even RSA, as well as general techniques for turning implementations of cryptographic algorithms constant-time. However, and quite astonishingly, there is no rigorous proof that constant-time algorithms are protected to cache-based attacks when executed concurrently on virtualization platforms with shared cache. Moreover, many cryptographic implementations such as PolarSSL AES, DES, and RC4 make array accesses that depend on secret keys and are not constant-time.

A second, more permissive approach, is to allow implementations that are not constant-time, but to deploy system-level countermeasures that prevent an attacker from drawing useful observations from the cache. Some these mechanisms are transparent to applications, but sacrifice performance: instances include flushing the cache at each context switch [46] or randomizing its layout [48]. Other mechanisms are not transparent, and must be used correctly, either via APIs or via compilers that enforce their correct usage. One lightweight such mechanism is stealth memory [29, 32]; in contrast to many of its competitors, stealth memory can be implemented in software, does not require any specific hardware and does not incur a significant performance overhead. Informally, stealth memory enforces a locking mechanism on a small set of cache lines, called stealth cache lines, saves them into (protected) memory and restores them upon context switches, thereby ensuring that entries stored in stealth cache lines are never evicted, and do not leak information. From an abstract perspective, memory accesses to stealth addresses, i.e addresses that map to stealth cache lines, become "hidden" and have no visible effect. Thus, applications can perform memory accesses that depend on secrets without revealing confidential information, provided these accesses are done on stealth addresses. This induces a relaxation of constant-time, which we call S-constant-time: an implementation is S-constant-time if it does not branch on secrets and only memory accesses to stealth addresses may depend on secrets. Although early work on stealth memory suggests that several prominent cryptographic implementations meet the requirements of S-constant-time, this class has not been considered formally before, and in particular, there is no rigorous security analysis of S-constant-time algorithms, and no mechanism to ensure that assembly code makes a correct usage of stealth addresses.

Our contributions We undertake a rigorous study of constant-time and S-constant-time implementations. We prove that such implementations are protected against cache-based attacks in virtualized platforms where their supporting operating system executes concurrently with other, potentially malicious, operating systems. Moreover, we provide support for deploying constant-time or S-constant time applications, in the form of type-based enforcement mechanisms on x86 implementations; the mechanisms are integrated into CompCert, a realistic verified compiler for C [36]. Finally, we experimentally validate our approach on a set of prominent cryptographic implementations. To achieve these goals, we make the following contributions:

1. We define an analysis for checking if x86 applications are constant-time. Our analysis is based on a type system that simultaneously tracks aliasing and information flow. For convenience, we package our analysis as a certifying compiler for CompCert. Our certifying compiler takes as input a C program whose confidential data is tagged with an annotation High, and transforms the program into annotated x86 assembly code, which can be checked for constant-time.

2. We provide the first formal proof that constant-time programs are protected against cache-based attacks in virtualization platforms. The proof contemplates a very strong threat model with a malicious operating system that controls the scheduler, executes concurrently with the operating system on which the victim application runs, and can observe how the shape of the cache evolves throughout execution.

3. As a first key step in the proof, we prove that constant-time programs is non-interfering with respect to an information flow policy which mandates that the control flow and the sequence of memory accesses during program execution do not depend on secrets. The policy is captured using an operational semantics of x86 programs where transitions are labelled with their read and write effects;

intends to characterize the behavior of program executions on concrete architectures, rather than in abstract operational models, we focus on low-level languages, and on a variant of constant-time expressed in terms of addresses (which consist of base addresses plus offsets) instead of arrays.

4. As a second key step in the proof, we prove isolation between operating systems in virtualization platforms. The proof is based on a model of virtualization that accounts for virtual addresses, physical and machine addresses, memory mappings, page tables, TLBs, and cache, and provides an operational semantics for a representative set of actions, including reads and writes, allocation and deallocation, context and mode switching, and hypercalls. The isolation theorem states that an adversary cannot distinguish between two execution traces of the platform in which the victim operating system performs two sequences of actions that have the same visible effects.

5. We extend our analysis and formal proofs to S-constant-time. As a significant contribution of the extension, we obtain the first rigorous security analysis of stealth memory.

6. We formalize our results in the Coq proof assistant (over 50,000 lines of Coq). The formalization is based on the first formal model of stealth memory. The model is a significant development in itself (over 10,000 lines of Coq) and is of independent interest;

7. We successfully evaluate the effectiveness of our framework on several cryptographic implementations, including AES, DES, and RC4 from the PolarSSL library, and SHA256, Salsa20. Figure 11 provides a summary of results.

2 Setting

Our first step is to define static analyses for enforcing constant-time (and variants) on x86 programs. Our analysis is built on top of CompCert [36], a formally verified, optimizing C compiler that generates reasonably efficient assembly code for x86 platforms (as well as PowerPC and ARM). In addition to being a significant achievement on its own, CompCert provides an excellent platform for developing verified static analyses. We take specific advantage of two features of CompCert: i. its memory model, which achieves a subtle and effective compromise between exposure to machine-level representation of memory and tractability of formal proofs, and is ideal for reasoning about properties that relate to sequences of memory accesses; ii. its sophisticated compilation chain, which involves over 15 passes, and about 10 intermediate languages, which are judiciously chosen to provide compact representations on which program analyses can be verified.

Our goal is to implement static analyses for checking whether programs perform conditional jumps or memory accesses that depend on secrets, and to derive strong semantical guarantees for the class of programs accepted by one of our analyses. In order to obtain meaningful results, it is important that our analyses are performed on intermediate representations towards the end of the compilation chain, rather than source C programs; indeed, some compilation passes in the compiler middle-end (typically at RTL level) may typically modify and reorder memory accesses and hence a constant-time C program could well be transformed into a non constant-time x86 program, or vice-versa. Therefore, we settle on defining our analysis on one of the final intermediate forms. A natural representation for reasoning about sequences of memory accesses is Mach, the last-but-final intermediate language in the compilation chain. The Mach language is used after passes that may introduce new memory accesses (such as register allocation, branch tunneling and layout of the activation records for procedure calls), and immediately before generation of assembly code. Hence the sequence of memory accesses at Mach and assembly levels coincide. Moreover, Mach has a compact syntax, which is important to reduce proof effort. On the other hand, the Mach language does not enjoy a control flow graph representation, which is a drawback for performing static analyses. We therefore adopt a minor variant of Mach, which we call MachIR, that retains the same instruction set as Mach but makes explicit the successor(s) of each instruction. MachIR is an idoneous representation for building verified static analyses about sequences of memory accesses of programs.

Syntax A MachIR program p is represented by a (partial) map of program nodes to instructions, i.e. as an element of $\mathbb{N} \rightharpoonup \mathbb{I}$. Each instruction carries its successor(s) node(s) explicitly. The most basic instructions are $\mathsf{op}(op, \boldsymbol{r}, r, n)$ (register r is assigned the result of the operation op on arguments $\boldsymbol{r}$; next node is n) that manipulates registers and $\mathsf{goto}(n)$ (unconditional jump to node n) and $\mathsf{cond}(c, \boldsymbol{r}, n_{then}, n_{else})$ (conditional jump; next node is n_{then} or n_{else} depending on the boolean value that is obtained by evaluating condition c on arguments $\boldsymbol{r}$) that offer basic control flow instructions. Memory is manipulated trough load

$$
\begin{array}{lll}
\mathbb{N} \ni & n & \text{CFG nodes} \\
\mathbb{R} \ni & r & \text{register names} \\
\mathbb{S} \ni & S & \text{global variable names} \\
\mathbb{A} \ni addr ::= & & \\
& \mid \mathsf{based}(\mathcal{S}) & \text{based addressing} \\
& \mid \mathsf{stack}(\delta) & \text{stack position} \\
& \mid \mathsf{indexed} & \text{indexed addressing} \\
\\
\mathbb{O} \ni & op ::= & \\
& \mid \mathsf{addrof}(addr) & \text{symbol address} \\
& \mid \mathsf{move} & \text{register move} \\
& \mid \mathsf{arith}(a) & \text{arithmetic operation} \\
\\
\mathbb{I} \ni instr ::= & & \\
& \mid \mathsf{op}(op, \boldsymbol{r}, r, n) & \text{register operation} \\
& \mid \mathsf{load}_\varsigma(addr, \boldsymbol{r}, r, n) & \text{memory load} \\
& \mid \mathsf{store}_\varsigma(addr, \boldsymbol{r}, r, n) & \text{memory store} \\
& \mid \mathsf{goto}(n) & \text{static jump} \\
& \mid \mathsf{cond}(c, \boldsymbol{r}, n_{then}, n_{else}) & \text{conditional static jump}
\end{array}
$$

Fig. 1. Instruction set

($\mathsf{load}_\varsigma(addr, \boldsymbol{r}, r, n)$: register r receives the content of the memory at an address that is computed with addressing mode $addr$ and arguments $\boldsymbol{r}$; next node is n) and store ($\mathsf{store}_\varsigma(addr, \boldsymbol{r}, r, n)$: the content of the register r is stored in memory at an address that is computed with addressing mode $addr$ and arguments $\boldsymbol{r}$; next node is n) operations. ς describes the type of memory chunk that is accessed (of size 1, 2 or 4 bytes). Addressing $\mathsf{based}(\mathcal{S})$ (resp. $\mathsf{stack}(\delta)$) directly denotes the address of a global symbol (resp. of the stack memory block). Pointer arithmetic is performed through addressing mode $\mathsf{indexed}$. Additional instructions are used to access the activation record of a procedure call, and to perform the call. Figure 1 gives an excerpt of the language instruction set.

Semantics Values are either numeric values $\mathsf{Vnum}(i)$ or pointer values $\mathsf{Vptr}(b, \delta)$ with b a memory block name and δ a block offset. We let $\&\mathsf{SP}$ denote the memory block that stores the stack. A state (n, ρ, μ) is composed of the current CFG node n, the register bank $\rho \in \mathbb{R} \to \mathbf{Val}$ and the memory $\mu \in \mathbf{Mem}$, where **Mem** is the CompCert type of memories.

The operational semantics is given by judgments of the form:

$$s \xrightarrow{a} s'$$

The semantics is implicitly parameterized by a program p. Informally, the judgment above says that executing the program p with state s leads to a state s', and has visible effect a, where a is either a read effect $\mathsf{read}\ x$ (with x an address), or a write effect $\mathsf{write}\ x$, or the null effect $\emptyset$. Note that effects model the addresses that are read and written, but not their value. Figure 2 presents selected rules of the semantics. Note that an instruction like $\mathsf{store}_4(\mathsf{stack}(\delta), [], r, n')$ will assign the four stack positions δ, $\delta + 1$, $\delta + 2$ and $\delta + 3$.

3 A Type System for Constant-Time

This section introduces a type-based information flow analysis that checks whether a MachIR program is constant-time, i.e. its control flow and its sequence of memory accesses do not depend on secrets. To track how dependencies evolve during execution, the information flow analysis must be able to predict the set of memory accesses that each instruction will perform at runtime. However, instructions such as $\mathsf{store}_\varsigma(\mathsf{indexed}, [r_1; r_2], r, n')$ do not carry this information. The standard solution to recover this information is

$$\frac{p[n] = \mathsf{op}(op, \boldsymbol{r}, r, n')}{(n, \rho, \mu) \xrightarrow{\emptyset} (n', \rho[r \mapsto [\![op]\!](\rho, \boldsymbol{r})], \mu)}$$

$$\frac{p[n] = \mathsf{load}_\varsigma(addr, \boldsymbol{r}, r, n') \qquad [\![addr]\!](\rho, \boldsymbol{r}) = v_{\text{addr}} \qquad \mu[v_{\text{addr}}]_\varsigma = v}{(n, \rho, \mu) \xrightarrow{\texttt{read}\ v_{\text{addr}}} (n', \rho[r \mapsto v], \mu)}$$

$$\frac{p[n] = \mathsf{store}_\varsigma(addr, \boldsymbol{r}, r, n') \qquad [\![addr]\!](\rho, \boldsymbol{r}) = v_{\text{addr}} \qquad store(\mu, \varsigma, v_{\text{addr}}, \rho(r)) = \mu'}{(n, \rho, \mu) \xrightarrow{\texttt{write}\ v_{\text{addr}}} (n', \rho, \mu')}$$

Fig. 2. Mach IR semantics (excerpts)

$$\frac{p(n) = \mathsf{op}(op, \boldsymbol{r}, r, n')}{X_h \vdash n : \tau \Rightarrow \tau[r \mapsto \tau(\boldsymbol{r})]}$$

$$\frac{p(n) = \mathsf{load}_\varsigma(addr, \boldsymbol{r}, r, n') \qquad PointsTo(n, addr, \boldsymbol{r}) = \mathsf{Symb}(\mathcal{S}) \qquad \tau(\boldsymbol{r}) = \mathsf{Low}}{X_h \vdash n : \tau \Rightarrow \tau[r \mapsto X_h(\mathcal{S})]}$$

$$\frac{p(n) = \mathsf{load}_\varsigma(addr, \boldsymbol{r}, r, n') \qquad PointsTo(n, addr, \boldsymbol{r}) = \mathsf{Stack}(\delta)}{X_h \vdash n : \tau \Rightarrow \tau[r \mapsto \tau(\delta) \sqcup \cdots \sqcup \tau(\delta + \varsigma - 1)]}$$

$$\frac{p(n) = \mathsf{store}_\varsigma(addr, \boldsymbol{r}, r, n') \qquad PointsTo(n, addr, \boldsymbol{r}) = \mathsf{Symb}(\mathcal{S}) \qquad \tau(\boldsymbol{r}) = \mathsf{Low} \qquad \tau(r) \sqsubseteq X_h(\mathcal{S})}{X_h \vdash n : \tau \Rightarrow \tau}$$

$$\frac{p(n) = \mathsf{store}_\varsigma(addr, \boldsymbol{r}, r, n') \qquad PointsTo(n, addr, \boldsymbol{r}) = \mathsf{Stack}(\delta)}{X_h \vdash n : \tau \Rightarrow \tau[\delta \mapsto \tau(r), \ldots, \delta + \varsigma - 1 \mapsto \tau(r)]}$$

$$\frac{p(n) = \mathsf{goto}(n')}{X_h \vdash n : \tau \Rightarrow \tau}$$

Fig. 3. Information flow type system for constant-time

to let the information flow analysis use the results of another static analysis that performs these computations. There are several possible choices that achieve different trade-offs between expressiveness, precision, and simplicity. We opt for a conventional points-to [7] analysis. A similar analysis has already been formalized for the CompCert toolchain [41], but it targets a different language (RTL) and makes a different trade-off between efficiency and precision; we use our own formalization here.

Alias (points-to) type system The definition of the alias type system is given in Appendix A. For the purpose of understanding the rest of the paper, it is sufficient to know that the type system computes statically the points-to information $PointsTo(n, addr, \boldsymbol{r})$ at every node n for a memory access with an addressing mode $addr$ and arguments $\boldsymbol{r}$. Hence, if node n contains an instruction $\mathsf{load}_\varsigma(addr, \boldsymbol{r}, r, n')$ or $\mathsf{store}_\varsigma(addr, \boldsymbol{r}, r, n')$, we have a prediction, at compile time, of the targeted memory address. In this context, a so-called points-to information is one of the following:

– $\mathsf{Symb}(\mathcal{S})$, which represents pointer values $\mathsf{Vptr}(b, \delta)$ such that b is equal to the memory address $\&\mathcal{S}$ of the global variable $\mathcal{S}$;

– $\mathsf{Stack}(\delta)$, which represents the pointer value $\mathsf{Vptr}(\&\mathsf{SP}, \delta)$.

For example, if a store instruction $\mathsf{store}_\varsigma(\mathsf{indexed}, [r_1; r_2], r, n')$ is performed at node n when r_1 contains $\mathsf{Vptr}(\&\mathcal{S}, 8)$ and r_2 contains the integer 16, the points-to static analysis may safely predict $PointsTo(n, addr, \boldsymbol{r}) = \mathsf{Symb}(\mathcal{S})$, because the accessed pointer is $\mathsf{Vptr}(\&\mathcal{S}, 24)$.

Information flow type system Next, we define an information flow type system for constant-time. As usual, we consider a lattice of security levels $\mathbb{L} = \{\mathsf{Low}, \mathsf{High}\}$ with $\mathsf{Low} \sqsubseteq \mathsf{High}$. Initially, the user declares a set $X_h^0 \subseteq \mathbb{S}$ of high variables.

Programs are assigned types (X_h, T), where $X_h \in \mathbb{S} \to \mathbb{L}$ is a global type, and $T \in \mathbb{N} \to (\mathbb{N} + \mathbb{R}) \to \mathbb{L}$ is a mapping from program nodes to local types. X_h is a flow-insensitive global type which assigns a security

level $X_h(\mathcal{S})$ for every global variable $\mathcal{S} \in \mathbb{S}$. T is a flow-sensitive local type which assigns for every offset $\delta \in \mathbb{N}$ the security level $T[n](\delta)$ of the stack cell at address $\mathsf{Vptr}(\&\mathsf{SP}, \delta)$ and node n, and for every register $r \in \mathbb{R}$ its security level $T[n](r)$ at node n. Formally, the type system manipulates judgments of the form:

$$X_h \vdash n : \tau_1 \Rightarrow \tau_2$$

where X_h is a global type, n is a node, and τ_1 and τ_2 are local types, i.e. $\tau_1, \tau_2 \in (\mathbb{N} + \mathbb{R}) \to \mathbb{L}$. The type system enforces a set of constraints on X_h^0, X_h and T. Typing rules are given in Figure 3; we note $\tau(\boldsymbol{r})$ for $\bigsqcup_{r \in \boldsymbol{r}} \tau(r)$.

The rule for $\mathsf{op}(op, \boldsymbol{r}, r, n')$ simply updates the security level of r with the supremum of the security levels of $\boldsymbol{r}$.

There are two rules for $\mathsf{load}_\varsigma(addr, \boldsymbol{r}, r, n')$. The first one considers the case where the value is loaded from a global variable $\mathcal{S}$. In this case, the typing rule requires that all registers are low, i.e. $\tau(\boldsymbol{r}) = \mathsf{Low}$, as we want to forbid memory accesses that depend on a secret. The security level of the register r is updated with the security level $X_h(\mathcal{S})$ of the variable. The second rule considers the case where the value is loaded from a stack position at offset δ. In this case, our type system conservatively requires that the memory access is constant (and statically predicted by the alias type system). In this case, no information is leaked. Note that the security level of the register r is set to the maximum of $\tau(\delta), \ldots, \tau(\delta + \varsigma - 1)$. Indeed, the security level of $\tau(\delta)$ models the level of the 8-bits value at position δ; if the load is performed with a memory chunk of size strictly bigger than 1, several 8-bits value will be accessed. Our type system takes care of this subtlety.

The two typing rules for store are similar to the rules for load. If the store is performed on a global variable, we again require $\tau(\boldsymbol{r}) = \mathsf{Low}$ to make sure the dereferenced pointer does not leak secrets. The constraint $\tau(r) \subseteq X_h(\mathcal{S})$ propagates the security level of the stored value. For a store on a stack offset, we again make sure to consider enough stack offsets by considering the memory chunk of the instruction.

Definition 1 (Constant-time programs). *A program p is constant-time with respect to a set of variables X_h^0, written $X_h^0 \vdash p$, if there exists (X_h, T) such that for every $\mathcal{S} \in X_h^0$, $X_h(\mathcal{S}) = \mathsf{High}$ and for all nodes n and all its successors n', there exists τ such that*

$$X_h \vdash n : T(n) \Rightarrow \tau \quad \wedge \quad \tau \sqsubseteq T(n')$$

where $\sqsubseteq$ is the natural lifting of $\sqsubseteq$ from $\mathbb{L}$ to types.

We automatically infer X_h and T using Kildall's algorithm.

4 Soundness of Constant-Time Type System

We capture the soundness of the static analyses with respect to two distinct non-interference properties. The first property is cast relative to the operational semantics of MachIR (or equivalently x86) programs, and capture a passive and non-concurrent attacker. This property is similar to non-interference results as they arise in the literature on language-based security, and serves as a key step towards the second property. The latter is cast relative to the operational semantics of a virtualization platform, and captures an active and adaptive adversary. For the sake of readability, this section defines the security policies, relate them informally to existing threat models, and provide informal soundness statements. Formalization details are deferred to Section 6 and to the appendices.

4.1 Language-level security

Our first soundness result establishes a non-interference property based on the semantics of MachIR programs. We assume given for every pair (X_h, τ) consisting of a global type and a local type an equivalence relation $\sim_{X_h,\tau}$ on states. Informally, two states s and s' are equivalent if they have the same program counter,

and their bank registers and memory mappings coincide on their low part. Given a typing derivation for p with witness (X_h, T), equivalence can be extended to traces[5] of p as follows:

$$\theta = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \dots$$
$$\theta' = s'_0 \xrightarrow{a'_0} s'_1 \xrightarrow{a'_1} s'_2 \xrightarrow{a'_2} s'_3 \dots$$

are equivalent, written $\theta \sim_{X_h,T} \theta'$, iff $i = 0 \dots\ a_i = a'_i$ and $s_i \sim_{X_h,T(pc_i)} s'_i$, where pc_i denotes the program counters of s_i and s'_i (which in particular must coincide).

We say that a program p verifies *LL non-interference* w.r.t. X_h^0, written $\mathsf{LLNI}_{X_h^0}(p)$, iff every two traces θ and θ' obtained by executing p from two initial states s and s' verify:

$$s \sim_{X_h,T(pc_0)} s' \quad \Longrightarrow \quad \theta \sim_{X_h,T} \theta'$$

Note that the definition is implicitly parametrized by (X_h, T).

LL non-interference accurately captures the intended goal of constant-time: indeed, it ensures that programs have the same control flow and perform the same sequence of memory accesses for every pair of executions starting from equivalent initial states.

Proposition 1 (Language-level security for constant-time).
If $X_h^0 \vdash p$ *then* $\mathsf{LLNI}_{X_h^0}(p)$.

This first result proves security against a weak, passive attacker, which can observe the sequence of memory accesses and program counters during program execution, but cannot observe the program memory, or interleave the program's execution with execution of code of its choice. Although we do not establish a connection formally, this model is closely related to a system-level attacker model, called the *non-concurrent attacker model*. In this model, the attacker is a malicious operating system o_a that co-resides with the operating system o_v on which the victim program executes. The attacker initially performs some computations, for instance to set the cache in a state of his choice. Then, the hypervisor performs a context switch and makes the victim operating system active, so that the victim program executes uninterruptedly. Upon termination of the victim program execution, the hypervisor performs a context switch; the attacker becomes active again, and tries to guess from its accumulated observations the secret material, e.g. the secret cryptographic keys, manipulated by the victim program.

4.2 System-level security

Our second soundness theorem establishes a non-interference property for a much stronger model, called the *concurrent attacker model*. The setting of this attacker model is similar to the non-concurrent attacker model, and considers a virtualization platform with a malicious operating system o_a and the victim operating system o_v on which a victim program p executes. However, this model assumes that the attacker is both active and adapative. More explicitly, o_a and o_v execute concurrently under a scheduler controlled by o_a, which decides at each step to execute a sequence of steps of its choice, to force resolution of a pending hypercall, or to switch context in order to give control to the victim o_v. Furthermore, the attacker o_a can observe finely the structure of the cache during execution, but cannot read into the memory of o_v, or read in the cache the values of entries belonging to o_v. At each step, the attacker o_a can use its previous observations to decide how to proceed. This model significantly generalizes the non-concurrent attacker model captured by language-level security and in particular captures the class of access-driven attacks, in which the attacker makes fine-grained observations about the sequence of cache hits and misses.

Formally, we model the attacker model on top of an operational semantics of the virtualization plaform. The semantics is built on top of a rich memory model that accounts for virtual, physical, and machine

[5] We allow infinite traces. Later, we introduce partial traces, which are necessarily finite. Moreover, we assume that s_0 and s'_0 are initial states, i.e. their program counter is set to a distinguished entry point pc_0.

addresses, memory mappings, page tables, TLBs (translation lookaside buffers), and VIPT (virtually indexed physically tagged) cache. Formally, the semantics is modelled as a labelled transition system:

$$t \xhookrightarrow{b} t'$$

where t, t' range over states and b is an action. Informally, a labelled transition as above indicates that the execution of the action b by o in an initial state t leads to a new state t'. Figure 7 provides a representative set of actions considered, including reads and writes, extending or restricting memory mappings, (un)registering memory pages, context and mode switching, and hypercalls. Each action b has an effect $\mathsf{eff}(b)$; see Figure 7 for examples of effects. As in the language-level setting, the visible effects of reads and writes record the addresses that are read and written, but not their value.

Then, we model the attacker as a function $\mathfrak{A}$ that takes as input a partial trace and returns either a tag v if the attacker lets the victim operating system perform the next step of execution, or an action of its choice that it will execute in the next step. Since the choice of the attacker can only depend on its view of the system, we define an equivalence relation $\sim$ on partial traces, and require that $\mathfrak{A}$ is compatible with $\sim$, i.e. $\mathfrak{A}(\theta) = \mathfrak{A}(\theta')$ for every partial traces θ and θ' such that $\theta \sim \theta'$. Equivalence between partial traces is defined from equivalence $\sim$ on states (itself defined formally in Section 6):

$$\theta = t_0 \xhookrightarrow{b_0} t_1 \xhookrightarrow{b_1} t_2 \xhookrightarrow{b_2} \dots \xhookrightarrow{b_{n-1}} t_n$$
$$\theta' = t'_0 \xhookrightarrow{b'_0} t'_1 \xhookrightarrow{b'_1} t'_2 \xhookrightarrow{b'_2} \dots \xhookrightarrow{b'_{n'-1}} t'_{n'}$$

are equivalent, written $\theta \sim \theta'$, iff $n = n'$, and for $i = 0 \dots n-1$, $t_i \sim t'_i$, and if the active OS of t_i is o_v then $\mathsf{eff}(b_i) = \mathsf{eff}(b'_i)$ else if the active OS of t_i is o_a then $b_i = b'_i$.

Given an attacker $\mathfrak{A}$ and a victim program p, one can define the concurrent execution $(\mathfrak{A} \parallel p)[t]$ of $\mathfrak{A}$ and p with initial state t; informally, $(\mathfrak{A} \parallel p)[t]$ is the system trace that interleaves execution of p by o_v and adversarially-chosen code by o_a according to the adversarially-chosen scheduling policy—both captured in the definition of $\mathfrak{A}$. Formally, $(\mathfrak{A} \parallel p)[t]$ is defined recursively: given a partial trace θ for the concurrent execution, one computes $\mathfrak{A}(\theta)$ to determine whether the next action to be executed is the attacker action $\mathfrak{A}(\theta)$, in case $\mathfrak{A}(\theta) \neq \mathsf{v}$, or the next step in the execution of p, in case $\mathfrak{A}(\theta) = \mathsf{v}$.

Given a program p and a set of initial secrets X_h^0, we define an equivalence relation $\sim_{X_h^0}$ on system states; the relation is implicitly parameterized by a mapping of MachIR (or equivalently x86) states to platform states. We say that a program p verifies *SL non-interference* w.r.t. an initial set of high variables X_h^0, written $\mathsf{SLNI}_{X_h}(p)$, iff for every attacker $\mathfrak{A}$ and initial states t and t':

$$[t \sim_{X_h^0} t' \wedge t \sim t'] \implies (\mathfrak{A} \parallel p)[t] \sim (\mathfrak{A} \parallel p)[t']$$

Proposition 2 (System-level security for constant-time).
If $X_h^0 \vdash p$ *then* $\mathsf{SLNI}_{X_h^0}(p)$.

5 Extensions to S-constant-time programs

We now outline an extension of the results of the previous section that accounts for stealth memory. Informally stealth memory provides a distinguished set of stealth addresses such that reading or writing from these addresses has no visible effect. We reflect this property of stealth memory by relaxing the type system to allow secret-dependent memory accesses on stealth addresses. The modified typing rules now involve a set X_s of addresses that must be mapped to stealth memory. The main typing rules are now given in Figure 4. Note that there is no requirement that stealth addresses are high; in practice, stealth addresses often store public tables.

Definition 2 (S-constant-time). *A program p is S-constant-time with respect to a set of variables X_h^0 and a set of stealth addresses X_s, written $X_s, X_h^0 \vdash p$, if there exists (X_h, T) such that for every $\mathcal{S} \in X_h^0$, $X_h(\mathcal{S}) = \mathsf{High}$ and for all nodes n and all its successors n', there exists τ such that*

$$X_s, X_h \vdash n : T(n) \Rightarrow \tau \quad \wedge \quad \tau \sqsubseteq T(n')$$

$$\frac{p(n) = \mathsf{load}_\varsigma(addr, \boldsymbol{r}, r, n') \qquad PointsTo(n, addr, \boldsymbol{r}) = \mathsf{Symb}(\mathcal{S}) \qquad \tau(\boldsymbol{r}) = \mathsf{High} \implies \mathcal{S} \in X_s}{X_s, X_h \vdash n : \tau \Rightarrow \tau[r \mapsto \tau(\boldsymbol{r}) \sqcup X_h(\mathcal{S})]}$$

$$\frac{p(n) = \mathsf{store}_\varsigma(addr, \boldsymbol{r}, r, n') \quad PointsTo(n, addr, \boldsymbol{r}) = \mathsf{Symb}(\mathcal{S}) \qquad \tau(\boldsymbol{r}) = \mathsf{High} \implies \mathcal{S} \in X_s \qquad \tau(\boldsymbol{r}) \sqcup \tau(r) \sqsubseteq X_h(\mathcal{S})}{X_s, X_h \vdash n : \tau \Rightarrow \tau}$$

Fig. 4. Information flow type system for S-constant-time

$$\frac{p[n] = \mathsf{load}_\varsigma(addr, \boldsymbol{r}, r, n') \qquad [\![addr]\!](\rho, \boldsymbol{r}) = v_{\text{addr}} \qquad v_{\text{addr}} \notin X_s \qquad \mu[v_{\text{addr}}]_\varsigma = v}{(n, \rho, \mu) \xrightarrow{\texttt{read } v_{\text{addr}}} (n', \rho[r \mapsto v], \mu)}$$

$$\frac{p[n] = \mathsf{store}_\varsigma(addr, \boldsymbol{r}, r, n') \qquad [\![addr]\!](\rho, \boldsymbol{r}) = v_{\text{addr}} \qquad v_{\text{addr}} \notin X_s \qquad store(\mu, \varsigma, v_{\text{addr}}, \rho(r)) = \mu'}{(n, \rho, \mu) \xrightarrow{\texttt{write } v_{\text{addr}}} (n', \rho, \mu')}$$

$$\frac{p[n] = \mathsf{load}_\varsigma(addr, \boldsymbol{r}, r, n') \qquad [\![addr]\!](\rho, \boldsymbol{r}) = v_{\text{addr}} \qquad v_{\text{addr}} \in X_s \qquad \mu[v_{\text{addr}}]_\varsigma = v}{(n, \rho, \mu) \xhookrightarrow{\emptyset} (n', \rho[r \mapsto v], \mu)}$$

$$\frac{p[n] = \mathsf{store}_\varsigma(addr, \boldsymbol{r}, r, n') \qquad [\![addr]\!](\rho, \boldsymbol{r}) = v_{\text{addr}} \qquad v_{\text{addr}} \in X_s \qquad store(\mu, \varsigma, v_{\text{addr}}, \rho(r)) = \mu'}{(n, \rho, \mu) \xhookrightarrow{\emptyset} (n', \rho, \mu')}$$

Fig. 5. Modified IR semantics (excerpts)

where $\sqsubseteq$ is the natural lifting of $\sqsubseteq$ from $\mathbb{L}$ to to types.

We automatically infer X_s, X_h and T using Kildall's algorithm.

Memory-trace non-interference is extended to the setting of stealth memory simply by considering a modified labelled operational semantics (see Figure 5) where accessing variables in X_s has no visible effect; the notion of state equivalence remains unmodified. Below we let $\mathsf{LLNI}_{X_s, X_h^0}$ denote the resulting policy.

Proposition 3 (Language-level security for S-constant-time).
If $X_s, X_h^0 \vdash p$ then $\mathsf{LLNI}_{X_s, X_h^0}(p)$.

Given a program p, a set of initial secrets X_h^0, and a set of stealth addresses X_s, we define an equivalence relation $\sim_{X_h^0}$ on system states; the relation is implicitly parameterized by a mapping of MachIR (or equivalently x86) states to platform states that map elements of X_s to stealth addresses. We say that a program p verifies *SL non-interference* w.r.t. an initial set of high variables X_h^0 and a set of stealth addresses X_s, written $\mathsf{SLNI}_{X_s, X_h^0}(p)$, iff for every attacker $\mathfrak{A}$ and initial states t and t':

$$[t \sim_{X_h^0} t' \wedge t \sim t'] \quad \implies \quad (\mathfrak{A} \parallel p)[t] \sim (\mathfrak{A} \parallel p)[t']$$

Proposition 4. *[System-level security for S-constant-time]*
If $X_s, X_h^0 \vdash p$ then $\mathsf{SLNI}_{X_s, X_h^0}(p)$.

6 Formalization of Proposition 4

In this section, we outline the formalization of the proof of system-level security for S-constant-time. We first describe our model of virtualization; then we state an isolation theorem; finally, we sketch how SL non-interference follows.

Simplifications We make several simplifications. The most relevant ones are listed next: i. we take an abstract view of page tables as mappings; ii. we abstract away implementation details such as encoding and size of values, and assume given an abstract type Value of values with a distinguished element $\perp$ to denote undefined values; iii. we consider a single stealth address; iv. we do not model registers. These simplifications do not have any impact on the security analysis.

Policies Our model and henceforth results are parameterized by a write policy and a replacement policy for the cache. They can be instantiated respectively to write back and write through, and to all typical replacement policies, such as LRU, pseudo-LRU or LFU.

Memory model States (SLST) are modelled as 6-tuples that respectively store data about operating systems and about the active operating system, the memory, the hypervisor mapping, the cache and the TLB (translation lookaside buffer); the formal definition appears in Figure 6.

There are three levels of address spaces: *virtual addresses*, which are handled by guest operating systems (OSs) and processes, *physical addresses*, a software abstraction used to provide the illusion of hardware memory to each guest OS and *machine addresses*, which refer to actual hardware memory. Some virtual and machine addresses are marked as *stealth*.

$\mathsf{Va}, \mathsf{Pa}, \mathsf{Ma}$		virtual, physical and machine address
OSId		OS identifier
HC	$::= \mathit{new} \mid \mathit{del} \mid \mathit{lswitch} \mid \mathit{pin} \mid \mathit{unpin} \mid \mathit{none}$	hyper calls
OSData	$::= \mathsf{Pa} \times \mathsf{HC}$	OS data
$\mathsf{GuestOSs}$	$::= \mathsf{OSId} \to \mathsf{OSData}$	guest OSs
$\mathsf{OSActivity}$	$:= \mathit{running} \mid \mathit{waiting}$	exec modes
$\mathsf{ActiveOS}$	$::= \mathsf{OSId} \times \mathsf{OSActivity}$	active OS
$\mathsf{PageContent}$	$:= RW(\mathsf{Value}) \mid PT(\mathsf{Va} \to \mathsf{Ma}) \mid \mathit{none}$	page content
$\mathsf{PageOwner}$	$:= \mathit{Hyp} \mid OS(\mathsf{OSId}) \mid \mathit{none}$	page owner
Page	$:= \mathsf{PageContent} \times \mathsf{PageOwner} \times \mathsf{Bool}$	memory page
Memory	$::= \mathsf{Ma} \to \mathsf{Page}$	memory map
$\mathsf{HyperMap}$	$::= \mathsf{OSId} \to \mathsf{Pa} \to \mathsf{Ma}$	hypervisor map
$\mathsf{CacheData}$	$:= \mathsf{Va} \times \mathsf{Ma} \mapsto \mathsf{Page}$	cache data
$\mathsf{CacheIndex}$	$:= \mathsf{Va} \to \mathsf{Index}$	cache index
$\mathsf{CacheHistory}$	$:= \mathsf{Index} \to \mathsf{Hist}$	cache history
Cache	$::= \mathsf{CacheData} \times \mathsf{CacheIndex} \times \mathsf{CacheHistory}$	VIPT cache
TLB	$::= \mathsf{Va} \mapsto \mathsf{Ma}$	TLB
SLST	$::= \mathsf{GuestOSs} \times \mathsf{ActiveOS} \times \mathsf{HyperMap} \times \mathsf{Memory} \times \mathsf{Cache} \times \mathsf{TLB}$	System level state

Fig. 6. System level state

The first component of a state records for each OS, drawn from a set OSId of OS identifiers: i. a physical address pointing to its current page table; ii. its pending hypercall. Hypercalls are priviledged functionalities exported by the hypervisor to the guest OSs; there is at most one pending hypercall per OS.

The second component of a state stores the current *active operating system* (ActiveOS) together with its activity mode. The active OS is either *running* or *waiting* for a hypercall to be resolved.

The third component of the state stores the *platform memory* (Memory). The memory is modelled as a function from machine addresses to *memory pages*; contrary to separation kernels, pages are allocated on demand. Each page contains: .i an *owner* (PageOwner); .ii a flag indicating whether the page can be cached or not[6]; .iii a *content* (PageContent). A page owner is either the hypervisor or a guest OS; pages may not have owners. The page content is either a *readable/writable value* or an OS *page table*. Page tables are used by guest OSs for mapping the virtual addresses used by running applications to machine addresses. Neither applications nor guest OSs have permission to read or write page tables; these actions can only be performed by the hypervisor.

[6] To properly deal with the problems posed by aliasing in VIPT caches, pages mapped by two different virtual addresses are flagged as non-cacheable.

The fourth component of the state stores the *hypervisor mapping* (HyperMap). This mapping is used to translate physical page addresses to machine page addresses and is under control of the hypervisor, which can allocate and deallocate machine memory.

The fifth component of the state stores a Virtually Indexed Physically Tagged (VIPT) *data cache* (Cache). The cache is used to speed up data fetch and store, and consists of a collection of data blocks or *cache lines* that are accessed by cache indices. The cache consists of: i. a bounded map[7] from pairs of virtual and machine addresses to memory pages, ii. a history (used by the replacement policy) and, iii. a static mapping from virtual addresses to cache indices. Each entry is tagged with a machine address. This avoids the need of flushing the cache on every context switch. Since caches are usually set associative, there are many virtual addresses that map to the same index. All data that is accessed using the same index is called a *cache line set*. We select one cache index and one particular virtual address (*stealth_va*) in its cache line set for stealth use. All other virtual addresses in that cache line set are reserved and cannot be used either by the guest operating systems or the hypervisor. It is relatively straightforward to extend the definitions to a set of stealth addresses.

The final component of the state stores the *Translation Lookaside Buffer* (TLB), which is used to improve virtual address translation speed. The TLB is modelled as a bounded map from virtual to machine addresses. It is used in conjunction with the current page table of the active OS to speed up translation of virtual to machine addresses. The TLB is flushed on context switch and updates are done simultaneously in the page table, so its management is simpler than the cache (we do not need to record the TLB access history, as it is not necessary to write back evicted TLB entries).

State invariants The model formalizes a notion of valid state that captures several well-formedness conditions, and an *exclusion* property, which is crucial for proving isolation, and ensures that stealth and non-stealth addresses cannot be mapped to the same cache line set. Both properties are preserved by execution; for exclusion, this is achieved by a careful treatment of allocation in the operational semantics.

Platform semantics Figure 7 lists some representative actions and their effects. Figures 8 and 9 present, respectively, the semantics of two important actions: `write` and `new_sm`. The complete semantics of the virtualization platform is presented in [12].

We use some helper functions to manipulate the components of the state. These functions are explained in the description of the actions semantics. There is, for example, a function *cache_add* that is used to add entries in the cache. It returns the new cache and an optional entry selected for replacement. The function *cache_add* is parameterized by an abstract replacement policy that determines which elements are evicted from a full cache, and guarantees that the *inertia* property, as defined in [32], holds for the cache: when adding an entry to the cache in a virtual address *va*, if an eviction occurs, the evicted address is in the same cache line set as *va*.

Attacker model and state equivalence We let the attacker observe: i. its current page table; ii. its pending hypercalls; iii. the identity of the active operating system; iv. its activity when active; v. its own readable/writable memory pages; vi. the values of its own cache entries; vii. the memory layout of the victim, as defined by the page metadata (owner and cacheable status) of the victim memory pages; viii. the layout of the non-stealth part of the cache; ix. the cache history. The attacker cannot, however, directly read, write, or observe page table or the hypervisor mappings (either its own or the victim). This is because these mappings are maintained by the hypervisor, and guest OSs have no access to them. Moreover, the attacker cannot observe the values held in the memory or cache entries of the victim. This very strong adversary model captures the kind of attacks we are interested in: if two states differ in one of these observable components, the execution of an action might replace an attacker entry in the cache, potentially leading to a cache-based attack. On the other hand, we prove that if an action is executed in two states that are equivalent from the attacker's view, the attacker cache entries are equal in the resulting states.

[7] A bounded map is a finite map whose domain must have size less than some fixed positive constant.

Action	Informal description	Effect
`read` va	Guest OS reads virtual address va	$\emptyset$ if va is Stealth `read` va otherwise
`write` va val	Guest OS writes value val in va	$\emptyset$ if va is Stealth `write` va otherwise
`new` va pa	Hypervisor extends the non stealth memory of the active OS with $va \mapsto ma$	`new` va pa
`del` va	Hypervisor deletes mapping for va from current memory mapping of the active OS	`del` va
`new_sm` $stealth\_va$ pa	Hypervisor extends the stealth memory of the active OS with $stealth\_va \mapsto ma$	$\emptyset$
`switch` o	Hypervisor sets o to be the active OS	`switch` o
`lswitch` pa	Hypervisor changes the current memory mapping of the active OS to be pa	`lswitch` pa
`hcall` c	An OS requires privileged service c to be executed by the hypervisor	`hcall` c
`chmod`	The hypervisor gives the execution control to the active OS	`chmod`
`page_pin` pa t	The memory page that corresponds to pa is registered and classified with type t	`page_pin` pa t
`page_unpin` pa	The memory page of the active OS that corresponds to pa is un-registered	`page_unpin` pa

Fig. 7. Selected actions and their effects

Action `write` va val

Guest OS writes value val in va

Rule

$$\frac{\begin{array}{c} aos\_act = (aos, running) \qquad get\_page\_mem(t, va) = (ma, pg) \qquad pg = (RW\ \_, OS\ aos, b) \\ cache\_add(cache, va, ma, (RW\ val, OS\ aos, b)) = (cache', (ma', pg')) \\ mem[ma' := pg'][ma := (RW\ val, OS\ aos, b)]_{pol} = mem' \qquad tlb[va := ma] = tlb' \end{array}}{t = (oss, aos\_act, hyp, mem, cache, tlb) \xrightarrow{\texttt{write}\ va\ val} (oss, aos\_act, hyp, mem', cache', tlb')}$$

$$\frac{\begin{array}{c} aos\_act = (aos, running) \qquad get\_page\_mem(t, va) = (ma, pg) \qquad pg = (RW\ \_, OS\ aos, b) \\ cache\_add(cache, va, ma, (RW\ val, OS\ aos, b)) = (cache', \bot) \\ mem[ma := (RW\ val, OS\ aos, b)]]_{pol} = mem' \qquad tlb[va := ma] = tlb' \end{array}}{t = (oss, aos\_act, hyp, mem, cache, tlb) \xrightarrow{\texttt{write}\ va\ val} (oss, aos\_act, hyp, mem', cache', tlb')}$$

Precondition

The action `write` va val requires that the active OS aos is running. Furthermore, the virtual address va is mapped to a machine address ma and a readable/writable page pg in the current page table of the active OS (get_page_mem).

Postcondition

There are two rules for the `write` action, one in which an entry is evicted from the cache when the written page is added, and the other in which no entry is evicted. In both cases the resulting state differs in the value val of the page associated to the pair (va, ma) in the cache $cache$, and in the TLB tlb. If $cache\_add$ returns an entry (ma', pg') that was evicted from the cache, the memory in ma' is updated with pg'. The final value in memory of the page in ma is dependent on the write policy in use ($mem[ma := page]_{pol}$ updates the page in ma with $page$ in write-through policies, and it leaves it unchanged in write-back ones).

Fig. 8. Semantics of action `write`

Action `new_sm` *stealth_va pa*

Hypervisor extends the stealth memory of the active OS with $stealth\_va \mapsto ma$

Rule

$$\frac{\begin{array}{c} aos\_act = (aos, waiting) \qquad oss[aos] = (pa', New\ stealth\_va\ pa) \\ get\_page\_hyp(t, aos, pa) = (ma, pg) \qquad pg = (RW\ \_, OS\ aos, true) \\ \neg memory\_alias(mem, stealth\_va, ma) \qquad get\_page\_hyp(t, aos, pa') = (ma', cpt) \qquad cpt[stealth\_va] = \emptyset \\ oss[aos := (pa', None)] = oss' \qquad cpt[stealth\_va := ma] = cpt' \qquad mem[ma' := cpt'] = mem' \\ cache\_add(cache, stealth\_va, ma, pg) = (cache', \_) \qquad tlb[stealth\_va := ma] = tlb' \end{array}}{t = (oss, aos\_act, hyp, mem, cache, tlb) \xrightarrow{\texttt{new\_sm}\ stealth\_va\ \ pa} (oss', aos\_act, hyp, mem', cache', tlb')}$$

Precondition

The action `new_sm` *stealth_va pa* requires that the active OS *aos* is waiting for the hypervisor to extend its current page table *cpt* with *stealth_va*. The physical address *pa* maps to the machine address *ma* and page *pg* in the hypervisor mapping of *aos* (*get_page_hyp*). This page *pg* must be readable/writable and cacheable. Also, no page table can map a virtual address to *ma* (no *memory_alias*), and *stealth_va* is not mapped in *cpt*. This is needed in order to guarantee that the stealth page *pg* in *ma* is always cached and that no aliased pages are cached.

Postcondition

textbfPostcondition In the resulting state, the pending hypercall of *aos* is removed. The current page table *cpt* and *tlb* are updated with the mapping of *stealth_va* to *ma*. Furthermore, the new stealth page is immediately stored in *cache*.

Fig. 9. Semantics of action `new_sm`

Dynamic allocation is a known difficulty when reasoning about state equivalence; in our setting, the difficulty manifests itself in the definition of equivalence for memory and hypervisor mappings. In an object-oriented setting, this difficulty is normally solved using partial bijections [9]. However, we model both memory allocation and deallocation via the `pin` and `unpin` actions; unfortunately, the partial bijection approach breaks in this setting[8] and we do not know any formal proof of soundness of an information flow type syetm for a language with allocation and deallocation. Fortunately, we can define state equivalence without using partial bijections; instead, we rely on the hypervisor mapping physical addresses, which are the same in both executions.

Formally, state equivalence $\sim$ is defined as the conjunction of four equivalence relations for OS information, cache history, hypervisor mapping, and memory mapping. The first two relations are straightforward. We define equivalence of hypervisor mappings below; equivalence of memory is defined similarly.

Definition 3 (Equivalence of hypervisor mappings). *Two states t and t' have equivalent hypervisor mappings for the attacker ($t \sim^{hyp} t'$) if for every physical address pa, readable/writable page pg and machine address ma:*

- *if $get\_page\_hyp(t, o_a, pa) = (ma, pg)$, there exists ma' such that $get\_page\_hyp(t', o_a, pa) = (ma', pg)$;*
- *if $get\_page\_hyp(t, o_v, pa) = (ma, pg)$, and no page table maps stealth_va to ma, then there exists ma' such that $get\_page\_hyp(t', o_v, pa) = (ma', pg')$, where pg and pg' are equal except in their contents;*

and reciprocally for t'.

Figure 10 provides a pictural representation of the equivalence: we require that the attacker readable/writable pages are the same for *hyp* and *hyp'*. Furthermore, the layout of the non-stealth memory pages of the victim must be the same (non-stealth pages should have the same owner, and same cacheable flag, but arbitrary value).

[8] The approach requires that the partial bijection grows during execution. With deallocation, one would require that the final partial bijection is a superset of a subset of the original one, which is vacuous.

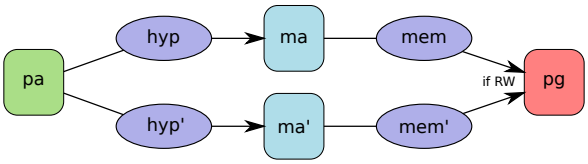

Fig. 10. Equivalence of hypervisor mappings

Unwinding lemmas The equivalence relation $\sim$ is kept invariant by the execution of a victim stealth action. Furthermore, if the same attacker action or two victim actions with the same effect are executed in two equivalent states, the resulting states are also equivalent. These results are variations of standard unwinding lemmas [42]. In the sequel, we write t^{o_v} and t^{o_a} respectively to denote states where o_v and o_a are the active operating system.

Lemma 1 (o_a step-consistent unwinding). *Assume $s_1^{o_a} \xhookrightarrow{a} s_1'$, and $s_2^{o_a} \xhookrightarrow{a} s_2'$. If $s_1 \sim s_2$ then $s_1' \sim s_2'$.*

Lemma 2 (o_v step-consistent unwinding). *Assume $s_1^{o_v} \xhookrightarrow{a} s_1'$, and $s_2^{o_v} \xhookrightarrow{a'} s_2'$. If $\mathsf{eff}(a) = \mathsf{eff}(a')$ and $s_1 \sim s_2$, then $s_1' \sim s_2'$.*

The proofs of these lemmas critically rely on the *inertia* property of cache [32]: upon adding a virtual address to the cache, the evicted virtual address, if any, is in the same cache line set as the added one; and on the *exclusion* property: the hypervisor ensures that guest operating systems can only allocate virtual addresses that are not in the same cache line set as the stealth virtual addresses.

Isolation We first define a relation to capture that two traces perform the same sequence of actions from the attacker's view:

$$\frac{\mathsf{eff}(b_1) = \mathsf{eff}(b_2) \quad \Theta_1 \approx \Theta_2}{t_1^{o_v} \xhookrightarrow{b_1} \Theta_1 \approx t_2^{o_v} \xhookrightarrow{b_2} \Theta_2} \qquad \frac{\Theta_1 \approx \Theta_2}{t_1^{o_a} \xhookrightarrow{b} \Theta_1 \approx t_2^{o_a} \xhookrightarrow{b} \Theta_2}$$

We then define equivalence of traces:

$$\frac{t_1 \sim t_2 \quad \Theta_1 \sim \Theta_2}{t_1^{o_v} \xhookrightarrow{b_1} \Theta_1 \sim t_2^{o_v} \xhookrightarrow{b_2} \Theta_2} \qquad \frac{t_1 \sim t_2 \quad \Theta_1 \sim \Theta_2}{t_1^{o_a} \xhookrightarrow{b} \Theta_1 \sim t_2^{o_a} \xhookrightarrow{b} \Theta_2}$$

Theorem 1 (OS isolation). *Let Θ and Θ' be execution traces such that $\Theta \approx \Theta'$. If $t_1 \sim t_1'$, with t_1 and t_1' the first states of traces Θ and Θ' respectively, then $\Theta \sim \Theta'$, i.e. Θ and Θ' are indistinguishable traces for the attacker system o_a.*

The proof of the theorem follows from the unwinding lemmas by co-induction on the execution traces.

System-level security for S-constant-time We define a relation between MachIR instructions and system-level actions, such that an instruction is related to an action if they have the same effect. In order to do this we use a mapping from language variables to virtual addresses that guarantees that program variables marked as stealth by the type system are mapped to stealth addresses in the platform. The relation between instructions and actions is naturally extended to programs and traces. With this extended relation, we define the concurrent execution of an attacker and a victim program ($(\mathfrak{A} \parallel p)[t]$), and state Proposition 4. The proof of this proposition is a direct consequence of Theorem 1, and shows that S-constant-time programs are protected to cache-based attacks in virtualization platforms.

7 Evaluation

We have successfully applied our approach to a representative set of cryptographic implementations, including some that are vulnerable to cache-based attacks on common platforms, and constant-time algorithms that

were specifically designed to avoid such attacks. In all cases, we picked standard and publicly available implementations of the constructions, and after very minor modifications of the code[9], compiled them using CompCert, and run our certified stealth verifier on the MachIR (or equivalently x86) programs output by the compiler. Figure 11 summarizes the list of examples analyzed, and provides in each case the number of variables marked as stealth, and the amount of stealth memory that is required to execute the program securely. The remaining of this section provides a brief explanation of each example considered.

Example	LoC	CT	SCT	Stealth cache (KB)
Salsa20	1077	✓		
SHA256	419	✓		
TEA	70	✓		
AES	744		✓	4
Blowfish	279	×	✓	4
DES	836	×	✓	2
RC4	164	×	✓	0.25
Snow	757	×	✓	6

A tick in the CT or SCT column respectively indicates whether programs are constant-time or S-constant-time. For the latter, the last column gives the amount of stealth cache required to run the application. All constant-time applications are also S-constant-time with 0KB stealth cache.

Fig. 11. Experimental results

AES Advanced Encryption Standard (AES) is a symmetric encryption algorithm that was selected by NIST in 2001 to replace DES. AES is now used very widely and is anticipated to remain the prevailing blockcipher for the next 20 years. Although NIST claimed the selected algorithm resilient against side-channels, AES is a prominent example of an algorithm in which the sequence of memory accesses depend on the cryptographic key.

Most applications of AES require that encryption and decryption be very efficient; therefore, the AES specification advises using S-boxes and other lookup tables to bypass expensive operations, such as arithmetic in the field GF(2^8). As a result of using S-boxes, most AES implementations are vulnerable to cache-based attacks, and fail to comply with even the weakest security guarantees. In 2005, Bernstein [17] reports on a simple timing attack which allows to recover AES keys by exploiting the correlation between execution time and cache behavior during computation. Shortly afterwards, Tromer, Osvik, and Shamir [46] report on several cache-based attacks against AES, including an effective attack that does not require knowledge of the plaintexts or the ciphertexts. Further improvements are reported by Bonneau and Mironov [19], Aciiçmez, Schindler and Koç [2], and Canteaut, Lauradoux and Seznec [21]. More recently, Bangerter, Gullasch and Krenn [30] report on a new cache-based attack in which key recovery is performed in almost real-time, and Ristenpart *et al* [40] show that cache-based attacks are not confined to closed systems, and can be realized in cloud architectures based on virtualization.

As a testcase for our approach, we have applied stealth verification to the PolarSSL implementation of AES. Stealth verification is able to prove that 4kB of stealth memory is sufficient to execute AES securely.

DES and BlowFish Data Encryption Standard (DES) and BlowFish are symmetric encryption algorithms that were widely used until the advent of AES. They are designed under the same principles as AES, and their implementation also relies on S-boxes. Cache-based attacks against DES and BlowFish are reported by Tsunoo *et al* [47] and Kelsey *et al* [31] respectively. We have applied stealth verification to PolarSSL implementations of both algorithms; again, our tool proves that only a small amount of stealth memory (resp. 2kB and 4kB) is required for the programs to execute securely.

[9] We have modified some examples to declare some arrays as global. This is a consequence of the relative coarseness of the alias analysis, and could be solved by formalizing a more precise value analysis.

SNOW Snow is a stream cipher used in standards such as the 3GPP encryption algorithms. Its implementation relies on table lookups for clocking its linear feedback shift register (LFSR). Cache-based attacks against SNOW—and similar LFSR-based ciphers—are reported by Leander, Zenner, and Hawkes [35]. We have applied stealth verification on a ECRYPT implementation of SNOW; our tool proves that SNOW can be executed securely with 6kB of stealth memory.

RC4 RC4 is a stream cipher introduced by Rivest in 1987 and used in cryptographic standards such as SSL and WPA. It is based on a pseudo-random generator that performs table lookups. Chardin, Fouque and Leresteux [22] present a cache-based attack against RC4. Analyzing the PolarSSL implementation of RC4 with stealth verification proves that the program can execute securely with only 0.25kB of stealth memory.

TEA, Salsa20, SHA256 We have applied stealth verification to some cryptographic algorithms that carefully avoid performing table lookups with indices dependent on secrets: Tiny Encryption Algorithm, a block cipher designed by Needham and Wheeler; Salsa20, a stream cipher designed by Bernstein, and SHA256. For the latter, we consider the input to be secret, with the intention to demonstrate that SHA256 is suitable to be used in password hashing. In all cases, stealth verification establishes that the programs are secure without using stealth memory.

8 Related work

Side-channel attacks in cryptography Kocher [34] presents a pratical timing attack on RSA and suggests that many vectors, including the cache, can be exploited to launch side-channel attacks. Aciiçmez and Schindler [1] demonstrate that not only data cache, but also instruction cache attacks are also effective. Over the last decade, researchers have developed abstract models of cryptography that capture side-channels, and developed constructions that are secure in these models, see e.g. [28] for a survey.

Analysis tools for cache-based attacks To our best knowledge, the first automated checker for constant-time is due to Adam Langley[10]. However, there is no formal publication attached to the implementation, no proof of soundness and no extensive evaluation.

CacheAudit [26] is an abstract-interpretation based framework for estimating the amount of leakage through the cache in straightline x86 executables. CacheAudit has been used to show that several applications do not leak information through the cache and to compute an upper bound for the information leaked through the cache by AES. These guarantees hold for a single run of the program, i.e. in the non-concurrent attacker model. A follow-up [14] provides an upper bound for the leakage of AES in the concurrent attacker model; the result is stated in an abstract setting, and under some restrictions. The results of [14] cannot be used to assert the security of constant-time programs against concurrent cache attacks.

Language-based protection mechanisms There has been significant work to develop language-based protection methods against side-channel attacks. Agat [3] defines an information flow type system that only accepts statements branching on secrets if the branches have the same pattern of memory accesses, and a type-directed transformation to make programs typable. Molnar et al [38] define the program counter model, which is equivalent to path non-interference, and give a program transformation for making programs secure in this model. Coppens et al [24] use selective if-conversion to remove high branches in programs. Zhang et al [49] develop a contract-based approach to mitigate side-channels. Enforcement of contracts on programs is performed using a type system, whereas informal analyses are used to ensure that the hardware comply with the contracts. They prove soundness of their approach. However, they do not consider the concurrent attacker model and they do not provide an equivalent of system-level non-interference. Stefan *et al* [45] also show how to eliminate cache-based timing attacks, but their adversary model is different.

More recently, Liu et al [37] define a type system that an information flow policy called memory-trace non-interference in the setting of oblivious RAM. Their type system has similar motivations has ours, but operates on source code and deals with a different attacker model.

[10] See `https://github.com/agl/ctgrind/`.

OS verification OS verification is an active field of research, with an increasing emphasis of machine-checked proofs [44]. One recent breakthrough is the machine-checked refinement proof of an implementation of the seL4 microkernel [33]. Subsequent machine-checked developments prove that seL4 enforces integrity, authority confinement [43] and intransitive non-interference [39]. The formalization does not model cache nor side-channel attacks.

Dam et al [25] formally verify information flow security for a simple separation kernel for ARMv7. The verification is based on an extant model of ARM in HOL, and relates an ideal model in which the security requirements hold by construction with a real model that faithfully respects the system behavior. Extending the approach to handle the cache is left for further work.

Our model of virtualization is inspired from recent work [11] which proves isolation in an idealized model of virtualization with a shared cache. However their model is based on a virtually indexed virtually tagged (VIVT) cache and assumes that the cache implements a write through policy, and is flushed upon context switch; thanks to these assumptions, the cache is always consistent with the memory of the current operating system. This coincidence allows lifting without much difficulty the isolation result of earlier work [10], which does not consider the cache. In particular, the unwinding lemmas of [10] can be used *mutatis mutandis*, without the need to be reproved in this extended setting. In comparison, our notion of state equivalence is significantly more involved, and as a result the proof of isolation is far more complex.

Stealth memory Stealth memory is introduced in [29] as a flexible system-level mechanism to protect against cache-based attacks. This flexibility of stealth memory is confirmed by a recent implementation and practical evaluation [32]. The implementation, called StealthMem, is based on Microsoft Hyper-V hypervisor, and is reasonably efficient (around 5% overhead for the SPEC 2006 benchmarks and less than 5% for cryptographic algorithms). Both [29, 32] lack a rigorous security analysis and language-based support for applications.

Verified cryptographic implementations There is a wide range of methods to verify cryptographic implementations: type-checking, see e.g. [18], deductive verification, see e.g. [27], code generation, see e.g. [20] and model extraction, see e.g. [4]. However, these works do not consider side-channels. Recently, Almeida et al [5] extend the EasyCrypt framework [13] to reason about the security of C-like implementations in idealized models of leakage, such as the Program Counter Model, and leverage CompCert to carry security guarantees to executable code; moreover they, instrument CompCert with a simple check on assembly programs to ensure that a source C program that is secure in the program counter model is compiled into an x86 program that is also secure in this model.

Verified compilation and analyses CompCert [36] is a flagship verified compiler that has been used and extended in many ways; except for [5], these works are not concerned with security. Type-preserving and verifying compilation are alternatives that have been considered for security purposes; e.g. Chen et al [23] and Barthe et al [16] develop type-preserving compilers for information flow.

Formal verification of information flow analyses is an active area of research; e.g. Barthe et al [15] and Amtoft et al [6] formally verify type-based and logic-based methods for enforcing information flow policies in programs. More recently, Azevedo et al [8] formally verify a clean-slate design that enforces information flow.

9 Final remarks

Constant-time cryptography is an oft advocated solution against cache-based attacks. In this work, we have developed an automated analyzer for constant-time cryptography, and given the first formal proof that constant-time programs are indeed protected against concurrent cache-based attacks. Moreover, we have extended our analysis to the setting of stealth memory; to this end, we have developed the first formal security analysis of stealth memory. Our results have been formalized in the Coq proof assistant, and our analyses have been validated experimentally on a representative set of algorithms.

Bibliography

[1] O. Aciiçmez and W. Schindler. A vulnerability in rsa implementations due to instruction cache analysis and its demonstration on openssl. In *CT-RSA'08*, volume 4964 of *LNCS*, pages 256–273. Springer, 2008.

[2] O. Aciiçmez, W. Schindler, and Çetin Kaya Koç. Cache based remote timing attack on the AES. In *CT-RSA 2007*, volume 4377 of *LNCS*, pages 271–286. Springer, 2007.

[3] J. Agat. Transforming out Timing Leaks. In *Proceedings POPL'00*, pages 40–53. ACM, 2000.

[4] M. Aizatulin, A. D. Gordon, and J. Jürjens. Computational verification of c protocol implementations by symbolic execution. In *CCS 2012*, pages 712–723. ACM, 2012.

[5] J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir. Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations. In *CCS 2013*, 2013.

[6] T. Amtoft, J. Dodds, Z. Zhang, A. W. Appel, L. Beringer, J. Hatcliff, X. Ou, and A. Cousino. A certificate infrastructure for machine-checked proofs of conditional information flow. In *POST 2012*, volume 7215 of *LNCS*, pages 369–389. Springer, 2012.

[7] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, 1994.

[8] A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach. A verified information-flow architecture. In *POPL 2014*. ACM, 2014.

[9] A. Banerjee and D. Naumann. Stack-based access control for secure information flow. *Journal of Functional Programming*, 15:131–177, Mar. 2005. Special Issue on Language-Based Security.

[10] G. Barthe, G. Betarte, J. Campo, and C. Luna. Formally verifying isolation and availability in an idealized model of virtualization. In *FM 2011*, pages 231–245. Springer-Verlag, 2011.

[11] G. Barthe, G. Betarte, J. Campo, and C. Luna. Cache-Leakage Resilient OS Isolation in an Idealized Model of Virtualization. In *CSF 2012*, pages 186–197, 2012.

[12] G. Barthe, G. Betarte, J. Campo, C. Luna, and D. Pichardie. An idealized model of virtualization with stealth memory: complete action semantics and non-interference. Technical report, June 2014. Available from `http://www.fing.edu.uy/inco/grupos/gsi/proyectos/VirtualCert/Doc/Publicaciones/ReportesTecnicos/model_description.pdf`.

[13] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella-Béguelin. Computer-aided security proofs for the working cryptographer. In *CRYPTO 2011*, volume 6841 of *LNCS*, Heidelberg, 2011.

[14] G. Barthe, B. Köpf, L. Mauborgne, and M. Ochoa. Leakage resilience against concurrent cache attacks. In *POST*, 2014.

[15] G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference java bytecode verifier. In *ESOP 2007*, pages 125–140, 2007.

[16] G. Barthe, T. Rezk, and D. A. Naumann. Deriving an information flow checker and certifying compiler for java. In *S&P 2006*, pages 230–242. IEEE Computer Society, 2006.

[17] D. J. Bernstein. Cache-timing attacks on AES, 2005. Available from author's webpage.

[18] K. Bhargavan, C. Fournet, and A. D. Gordon. Modular verification of security protocol code by typing. In *POPL 2010*. ACM, 2010.

[19] J. Bonneau and I. Mironov. Cache Collision Timing Attacks Against AES. In *CHES '06*, 2006.

[20] D. Cadé and B. Blanchet. From computationally-proved protocol specifications to implementations. In *ARES 2012*, pages 65–74. IEEE Computer Society, 2012.

[21] A. Canteaut, C. Lauradoux, and A. Seznec. Understanding cache attacks. Rapport de recherche RR-5881, INRIA, 2006.

[22] T. Chardin, P.-A. Fouque, and D. Leresteux. Cache timing analysis of RC4. In *ACNS 2011*, volume 6715 of *LNCS*, pages 110–129, 2011.

[23] J. Chen, R. Chugh, and N. Swamy. Type-preserving compilation of end-to-end verification of security enforcement. In *PLDI 2010*, pages 412–423. ACM, 2010.

[24] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *S&P 2009*, pages 45–60, 2009.

[25] M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz. Formal verification of information flow security for a simple ARM-based separation kernel. In *CCS 2013*, pages 223–234, 2013.

[26] G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke. Cacheaudit: A tool for the static analysis of cache side channels. In *Usenix Security 2013*, 2013.

[27] F. Dupressoir, A. D. Gordon, J. Jürjens, and D. A. Naumann. Guiding a General-Purpose C Verifier to Prove Cryptographic Protocols. In *CSF 2011*, pages 3–17. IEEE Computer Society, 2011.

[28] S. Dziembowski and K. Pietrzak. Leakage-resilient cryptography. In *FOCS*, pages 293–302. IEEE Computer Society, 2008.

[29] U. Erlingsson and M. Abadi. Operating system protection against side-channel attacks that exploit memory latency. Technical Report MSR-TR-2007-117, Microsoft Research, 2007.

[30] D. Gullasch, E. Bangerter, and S. Krenn. Cache games - bringing access-based cache attacks on AES to practice. In *S&P 2011*, pages 490–505, 2011.

[31] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Side Channel Cryptanalysis of Product Ciphers. *Journal of Computer Security*, 8(2–3):141–158, 2000.

[32] T. Kim, M. Peinado, and G. Mainar-Ruiz. Stealthmem: system-level protection against cache-based side channel attacks in the cloud. In *USENIX Security 2012*, pages 11–11, Berkeley, CA, USA, 2012. USENIX Association.

[33] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *SOSP 2009*, pages 207–220. ACM, 2009.

[34] P. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.

[35] G. Leander, E. Zenner, and P. Hawkes. Cache Timing Analysis of LFSR-Based Stream Ciphers. In *IMACC 2009*, volume 5921 of *LNCS*, pages 433–445. Springer, 2009.

[36] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL 2006*, pages 42–54. ACM, 2006.

[37] C. Liu, M. Hicks, and E. Shi. Memory trace oblivious program execution. In *CSF 2013*, pages 51–65, 2013.

[38] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *ICISC 2005*, pages 156–168, 2005.

[39] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. G., and G. Klein. sel4: from general purpose to a proof of information flow enforcement. In *S&P 2013*, pages 415–429, 2013.

[40] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud! Exploring information leakage in third-party compute clouds. In *CCS 2009*, pages 199–212. ACM Press, 2009.

[41] V. Robert and X. Leroy. A formally-verified alias analysis. In *CPP*, pages 11–26, 2012.

[42] J. M. Rushby. Noninterference, Transitivity, and Channel-Control Security Policies. Technical Report CSL-92-02, SRI International, 1992.

[43] T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein. seL4 enforces integrity. In *ITP 2011*, Nijmegen, The Netherlands, 2011.

[44] Z. Shao. Certified software. *Commun. ACM*, 53(12):56–66, 2010.

[45] D. Stefan, P. Buiras, E. Z. Yang, A. Levy, D. Terei, A. Russo, and D. Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In J. Crampton, S. Jajodia, and K. Mayes, editors, *ESORICS*, volume 8134 of *Lecture Notes in Computer Science*, pages 718–735. Springer, 2013.

[46] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on AES, and countermeasures. *J. Cryptology*, 23(1):37–71, 2010.

[47] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi. Cryptanalysis of DES implemented on computers with cache. In *CHES 2003*, volume 2779 of *LNCS*, pages 62–76. Springer, 2003.

[48] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *ISCA 2007*, pages 494–505. ACM, 2007.

[49] D. Zhang, A. Askarov, and A. C. Myers. Predictive mitigation of timing channels in interactive systems. In *CCS 2011*, pages 563–574. ACM, 2011.

A Alias type system

We first define the set *alias* of alias information. An element of *alias* is of one of the forms: i. Num, which represents only a numerical value; ii. $\mathsf{Symb}(\mathcal{S})$, which represents either a non-pointer value, or a pointer value that points to the variable $\mathcal{S}$; iii. $\mathsf{Stack}(\delta)$, which represents either a non-pointer value, or $\mathsf{Vptr}(\&\mathsf{SP}, \delta)$. We do not consider dynamic allocation and analyse only one single procedure where all function calls have been inlined. Hence $\mathsf{Symb}(\mathcal{S})$ and $\mathsf{Stack}(\delta)$ are the only symbolic informations we need. Alias informations are partially ordered with $\mathsf{Num} \subseteq \mathsf{Symb}(\mathcal{S})$ and $\mathsf{Num} \subseteq \mathsf{Stack}(\delta)$. This partial order is lifted to alias maps by the standard pointwise lifting.

Programs are assigned types of the form:

$$A \in \mathbb{N} \to (\mathbb{S} + \mathbb{N} + \mathbb{R}) \to \mathit{alias}$$

where for every node n: i. $A[n](S)$ gives the flow-sensitive points-to information of the global variable $\mathcal{S} \in \mathbb{S}$; ii. $A[n](\delta)$ gives the flow-sensitive information of the stack cell at address $\mathsf{Vptr}(\&\mathsf{SP}, \delta)$ ($\delta \in \mathbb{N}$); iii. $A[n](r)$ gives the flow-sensitive information of the register $r \in \mathbb{R}$.

The typing rules are given in Figure 12, and are mostly standard. Each load and store instructions require two cases: one for stack access and one other for global variable access. For a load at node n, in a register r and on a pointer value that points to a global variable $\mathcal{S}$, we transfer the abstract content of $A[n](\mathcal{S})$ to the abstract information $A[n'](r)$ of the successor node n'. The treatment is similar for a load on a pointer value that points to a stack position. For a store at node n, from a register r to a pointer value that points to a stack position $\mathsf{Vptr}(\&\mathsf{SP}, \delta)$, we update the abstract content $A[n'](\delta)$ of the successor node n'. If the store is performed on a global variable, we only perform a weak update: $A[n'](\mathcal{S})$ must contain $A[n](r)$ (the content of the stored value), but also $A[n](\mathcal{S})$ (the previous content), since the $\mathsf{Symb}(\mathcal{S})$ may represent several concrete memory cells. The type system will reject some programs if a pointer value, at a specific program point, may points-to a different symbol, depending on the path that has been followed to reach this point. This restriction could be lifted by considering sets (rather than singletons) of symbolic addresses.

Definition 4 (Alias-well-typed programs). *A program p is alias-well-typed with respect to a alias map A, if $A \vdash n : p[n]$ holds for all nodes n in p, and at the initial node n_0 of the program, $A[n_0]$ contains a correct abstraction of the initial program memory.*

We forge the initial abstraction of the memory by scanning the declaration of the global variables. If a global is initialised with a numeric constant, its initial abstraction is Num. CompCert also allows to initialise a global with the address of an other global variable $\mathcal{S}$. In this case, the initial abstraction is $\mathsf{Symb}(\mathcal{S})$.

We note that the relative simplicity of our analysis is derived from the specific guarantees provided by the CompCert memory model, and more particularly of the strong isolation between global variables. The main complexity of our type system lies in its treatment of stack manipulation, which requires specific care because the CompCert memory model does not enforce any separation guarantee for it. The type system makes sure every access to the stack is performed through constant address in order to track finely this part of the memory.

$$
\begin{array}{rl}
\mathit{alias} ::= & \\
\quad | & \mathsf{Num} \quad \text{numerical value} \\
\quad | & \mathsf{Symb}(\mathcal{S}) \quad \text{points to any cell allocated for symbol } \mathcal{S} \\
\quad | & \mathsf{Stack}(\delta) \quad \text{points to the } \delta^{\text{th}} \text{ stack cell}
\end{array}
$$

$$
\begin{array}{rl}
\mathcal{A}[\![\mathsf{indexed}]\!](a,[r_1;r_2]) & = \mathcal{A}[\![+]\!]([a(r_1);a(r_2)]) \\
\mathcal{A}[\![\mathsf{global}(S)]\!](a,\boldsymbol{r}) & = \mathsf{Symb}(S) \\
\mathcal{A}[\![\mathsf{stack}(\delta)]\!](a,[]) & = \mathsf{Stack}(\delta) \\
 & = \mathsf{Num} \quad \text{otherwise}
\end{array}
$$

$$
\begin{array}{rl}
\mathcal{A}[\![\mathsf{addrof}(\mathit{addr})]\!](a,\boldsymbol{r}) & = \mathcal{A}[\![\mathit{addr}]\!](a,\boldsymbol{r}) \\
\mathcal{A}[\![\mathsf{move}]\!](a,[r]) & = a(r) \\
\mathcal{A}[\![\mathsf{arith}(a)]\!](a,\boldsymbol{r}) & = \mathcal{A}[\![a]\!](a[\boldsymbol{r}])
\end{array}
$$

$$
\frac{A[n][r \mapsto \mathcal{A}[\![op]\!](A[n],\boldsymbol{r})] \subseteq A[n']}{A \vdash n : \mathsf{op}(op,\boldsymbol{r},r,n')}
$$

$$
\frac{\mathcal{A}[\![\mathit{addr}]\!](A[n],\boldsymbol{r}) = \mathsf{Symb}(S) \qquad A[n][r \mapsto A[n](S)] \subseteq A[n']}{A \vdash n : \mathsf{load}_\varsigma(\mathit{addr},\boldsymbol{r},r,n')}
$$

$$
\frac{\mathcal{A}[\![\mathit{addr}]\!](A[n],\boldsymbol{r}) = \mathsf{Stack}(\delta)\} \qquad A[n][r \mapsto A[n](\delta)] \subseteq A[n']}{A \vdash n : \mathsf{load}_\varsigma(\mathit{addr},\boldsymbol{r},r,n')}
$$

$$
\frac{\mathcal{A}[\![\mathit{addr}]\!](A[n],\boldsymbol{r}) = \mathsf{Symb}(S) \qquad A[n][S \mapsto A[n](r) \cup A[n](S)] \subseteq A[n']}{A \vdash n : \mathsf{store}_\varsigma(\mathit{addr},\boldsymbol{r},r,n')}
$$

$$
\frac{\mathcal{A}[\![\mathit{addr}]\!](A[n],\boldsymbol{r}) = \mathsf{Stack}(\delta) \qquad A[n][\delta \mapsto A[n](r)] \subseteq A[n']}{A \vdash n : \mathsf{store}_\varsigma(\mathit{addr},\boldsymbol{r},r,n')}
$$

$$
\frac{A[n] \subseteq A[n']}{A \vdash n : \mathsf{goto}(n')} \qquad \frac{A[n] \subseteq A[n_{then}] \quad A[n] \subseteq A[n_{else}]}{A \vdash n : \mathsf{cond}(c,\boldsymbol{r},n_{then},n_{else})}
$$

Fig. 12. Alias type system

$$\frac{\forall \mathcal{S}, X_h(\mathcal{S}) = \mathsf{Low} \implies \forall \delta, \forall \varsigma, \mu_1[\mathsf{Vptr}(b,i)]_\varsigma = \mu_2[\mathsf{Vptr}(b,i)]_\varsigma}{\mu_1 \sim_{X_h} \mu_2}$$

$$\frac{\forall \delta, \tau(\delta) = \mathsf{Low} \implies \mu_1[\mathsf{Vptr}(\mathsf{SP},\delta)]_1 = \mu_2[\mathsf{Vptr}(\mathsf{SP},\delta)]_1}{\mu_1 \sim_\tau \mu_2}$$

$$\frac{\forall r,\ \tau(r) = \mathsf{Low} \implies \rho_1(r) = \rho_2(r)}{\rho_1 \sim_\tau \rho_2}$$

$$\frac{\rho_1 \sim_\tau \rho_2 \qquad \mu_1 \sim_\tau \mu_2 \qquad \mu_1 \sim_{X_h} \mu_2}{(n, \rho_1, \mu_1) \sim_{X_h,\tau} (n, \rho_2, \mu_2)}$$

Fig. 13. State equivalence relations

B Formalization of Proposition 3

State equivalence is defined according to the rules of Figure 13. Two states $s_1 = (n_1, \rho_1, \mu_1)$ and $s_2(n_2, \rho_2, \mu_2)$ are equivalent with respect to a global type X_h and a local type τ, written $s_1 \sim_{X_h,\tau} s_2$, iff $n_1 = n_2$, and the two memories μ_1 and μ_2 are *global-equivalent* with respect to X_h and *stack-equivalent* with respect to the local type τ, and if the two register banks ρ_1 and ρ_2 are *register-equivalent*. Two memories μ_1 and μ_2 are *global-equivalent* ($\mu_1 \sim_{X_h} \mu_2$) if for every low global variable $\mathcal{S}$ (*i.e.* $X_h(\mathcal{S}) = \mathsf{Low}$), every load (whatever offset and memory chunk) returns the same result for μ_1 and μ_2. They are *stack-equivalent* ($\mu_1 \sim_\tau \mu_2$) if for each low stack offset δ (*i.e.* $\tau(\delta) = \mathsf{Low}$), reading one byte in μ_1 ($\mu_1[\mathsf{Vptr}(\mathsf{SP},\delta)]_1$) at this stack position, gives the same result as reading one byte at the same position in μ_2 ($\mu_2[\mathsf{Vptr}(\mathsf{SP},\delta)]_1$). Two register banks ρ_1 and ρ_2 are *register-equivalent* ($\rho_1 \sim_\tau \rho_2$) if they coincide on every low register r ($\tau(r) = \mathsf{Low}$).

The proof of soundness of the type system relies on key *unwinding lemma* and a *monotonicity lemma* (a.k.a. weakening lemma).

Lemma 3 (Unwinding). *Let s_1 and s_2 be two equivalent states, i.e. $st_1 \sim_{X_h,\tau} st_2$. If the judgment type $X_h \vdash n : \tau \Rightarrow \tau'$ holds at the common node n of s_1 and s_2; if two steps are performed from each state ($s_1 \xrightarrow{a_1} s_1'$ and $s_2 \xrightarrow{a_2} s_2'$) then the resulting states are equivalent ($s_1' \sim_{X_h,\tau'} s_2'$ and the effect a_1 and a_2 are the same.*

Lemma 4 (Monotonicity). *If $st_1 \sim_{X_h,\tau} st_2$ and $\tau \sqsubseteq \tau'$ holds then $st_1 \sim_{X_h,\tau'} st_2$ holds too.*