System-level Non-interference for Constant-time Cryptography
Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Luna, David Pichardie

To cite this version:
Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Luna, David Pichardie. System-level Non-interference for Constant-time Cryptography. ACM SIGSAC Conference on Computer and Communications Security, CCS’14, Nov 2014, Scottsdale, United States. pp.1267 - 1279, 10.1145/2660267.2660283. hal-01101950

HAL Id: hal-01101950
https://inria.hal.science/hal-01101950
Submitted on 10 Jan 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
System-level non-interference for constant-time cryptography

Gilles Barthe
IMDEA Software Institute, Spain

Gustavo Betarte
Universidad de la República, Uruguay

Carlos Luna
Universidad de la República, Uruguay

Juan Diego Campo
Universidad de la República, Uruguay

David Pichardie
ENS Rennes/IRISA/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.

Simple approaches for protecting oneself against cache-based attacks are flushing the cache on every context switch and disabling the cache mechanism for critical computations. The first one was formally analyzed in [11]. In addition, both approaches suffer from severe performance penalties [17, 48].

Another approach is to build implementations that do not leak information through the cache. One common strategy is to make implementations constant-time, i.e. do not branch on secrets and do not perform memory accesses that which depend on secrets. There exist constant-time implementations of many cryptographic algorithms, including AES, DES, ECDSA and RC4.

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 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.
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 against 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 different 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 of these mechanisms are transparent to applications, but sacrifice performance: instances include flushing the cache at each context switch [48] or randomizing its layout [50]. 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, 34]; 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 the first 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 [38]. 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 are 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.

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 1 provides a selection of results.

Full version. Additional details are available in the full version of the paper [12].

<table>
<thead>
<tr>
<th>EXAMPLE</th>
<th>LoC</th>
<th>CT</th>
<th>SCT</th>
<th>Stealth cache (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Salsa20</td>
<td>1077</td>
<td>✓</td>
<td>✓</td>
<td>4</td>
</tr>
<tr>
<td>SHA256</td>
<td>419</td>
<td>✓</td>
<td>✓</td>
<td>4</td>
</tr>
<tr>
<td>TEA</td>
<td>70</td>
<td>✓</td>
<td>✓</td>
<td>2</td>
</tr>
<tr>
<td>AES</td>
<td>744</td>
<td>✓</td>
<td>✓</td>
<td>0.25</td>
</tr>
<tr>
<td>Blowfish</td>
<td>279</td>
<td>✓</td>
<td>✓</td>
<td>6</td>
</tr>
<tr>
<td>DES</td>
<td>836</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>RC4</td>
<td>164</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Snow</td>
<td>757</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

A check 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.

Figure 1: Selected experimental 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 [38], 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 latest 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 idiosyncratic 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 \( \mathbb{N} \rightarrow \mathbb{I} \). Each instruction carries its successor(s) node(s) explicitly. The most basic instructions manipulate registers and perform conditional and unconditional jumps: \( \text{op}(\text{op}, r, r, n) \) (register \( r \) is assigned the result of the operation \( \text{op} \) on arguments \( r \); next node is \( n \)), \( \text{goto}(n) \) (unconditional jump to node \( n \)) and \( \text{cond}(c, r, r, n_{\text{then}}, n_{\text{else}}) \) (conditional jump; next node is \( n_{\text{then}} \) or \( n_{\text{else}} \) depending on the boolean value that is obtained by evaluating condition \( c \) on arguments \( r \)). Memory is manipulated through two operations: \( \text{load}(addr, r, r, n) \) (register \( r \) receives the content of the memory at an address that is computed with addressing mode \( addr \) and arguments \( r \); next node is \( n \)) and \( \text{store}(addr, r, r, n) \) (the content of the register \( r \) is stored in memory at an address that is computed with addressing mode \( addr \) and arguments \( r \); next node is \( n \)). \( \zeta \) describes the type of memory chunk that is accessed (of size 1, 2 or 4 bytes). Addressing \( \text{based}(N) \) (resp. \( \text{stack}(\delta) \)) directly denotes the address of a global symbol (resp. of the stack memory block). Pointer arithmetic is performed through addressing mode \( \text{indexd} \). Additional instructions are used to access the activation record of a procedure call, and to perform the call. Figure 2 gives an excerpt of the language instruction set.

\[ \begin{array}{c|c|c}
\mathbb{N} & n & \text{CFG nodes} \\
\mathbb{R} & r & \text{register names} \\
\mathbb{S} & S & \text{global variable names} \\
\& \lor \text{addr} & \text{based}(S) & \text{based addressing} \\
& \text{stack}(\delta) & \text{stack position} \\
& \text{indexed} & \text{indexed addressing} \\
\zeta \lor \text{op} & \text{addrof}(addr) & \text{symbol address} \\
& \text{move} & \text{register move} \\
& \text{arith}(a) & \text{arithmetical operation} \\
I \lor \text{instr} & \text{op}(\text{op}, \overrightarrow{r}, r, n) & \text{register operation} \\
& \text{load}(\text{addr}, \overrightarrow{r}, r, n) & \text{memory load} \\
& \text{store}(\text{addr}, \overrightarrow{r}, r, n) & \text{memory store} \\
& \text{goto}(n) & \text{static jump} \\
& \text{cond}(c, \overrightarrow{r}, n_{\text{then}}, n_{\text{else}}) & \text{conditional static jump} \\
\end{array} \]

Figure 2: Instruction set

Semantics. Values are either numeric values \( \text{Vnum}(r) \) or pointer values \( \text{Vptr}(b, \delta) \) with \( b \) a memory block name and \( \delta \) a block offset. We let \( \& \text{SP} \) denote the memory block that stores the stack. A state \( (n, p, \mu) \) is composed of the current CFG node \( n \), the register bank \( p \in \mathbb{R} \rightarrow \text{Val} \) and a CompCert memory \( \mu \in \text{Mem} \).

The operational semantics is modelled with judgments:

\[ s \xrightarrow{s'} s' \]

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 \( e \), where \( e \) is either a read effect \( \text{read} x \) (with \( x \) an address), or a write effect \( \text{write} x \), or the null effect \( \text{null} \). Note that effects model the addresses that are read and written, but not their values. Figure 3 presents selected rules of the semantics. Note that an instruction like \( \text{store}(\text{stack}(\delta), [], r, n) \) will assign the four stack positions \( \delta, \delta + 1, \delta + 2 \) 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
points-to type system. As usual, we consider a lattice of security levels \( S \) such that \( S \) contains all 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 [43], 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 [12]. 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 \( \text{PointsTo}(n, addr, \bar{r}) \) at every node \( n \) for a memory access with an addressing mode \( addr \) and arguments \( \bar{r} \). Hence, if node \( n \) contains an instruction \( \text{load}(addr, \bar{r}, r, n') \) or an instruction \( \text{store}(addr, \bar{r}, r, n') \), 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: i. \( \text{Symb}(S) \), which represents pointer values \( \text{Vptr}(b, \delta) \) such that \( b \) is equal to the memory address \( k \in S \) of the global variable \( S \); ii. \( \text{Stack}(\delta) \), which represents the pointer value \( \text{Vptr}(k, \delta, S) \).

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

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

Programs are assigned types \( (X_h, T) \), where \( X_h \in S \rightarrow L \) is a global type, and \( T \in \mathbb{N} \rightarrow (\mathbb{N} + \mathbb{R}) \rightarrow L \) is a mapping from program nodes to local types. \( X_h \) is a flow-insensitive global type which assigns a security level \( X_h[S] \) for every global variable \( S \in S \). \( T \) 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 \( \text{Vptr}(k, S, \delta) \) 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 \( \tau_1 \) and \( \tau_2 \) are local types, i.e. \( \tau_1, \tau_2 \in (\mathbb{N} + \mathbb{R}) \rightarrow L \). The type system enforces a set of constraints on \( X_h[n], T[n] \) and \( T[r] \). Typing rules are given in Figure 4; we note \( \tau(n) \) for \( \bigcup_{T[n] = \tau} \). The rule for \( \text{load}(addr, \bar{r}, r, n') \) simply updates the security level of \( r \) with the supremum of the security levels of \( \bar{r} \).

There are two rules for \( \text{store}(addr, \bar{r}, r, n') \). The first one considers the case where the value is loaded from a global variable \( S \). In this case, the typing rule requires that all registers are low, i.e. \( \tau(r) = \text{Low} \), 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 \( X_h(S) \) of the variable. The second rule considers the case where the value is loaded from a stack position at offset \( \delta \). 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 \( \delta \); 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 \( \text{store} \) are similar to the rules for \( \text{load} \). If the \( \text{store} \) is performed on a global variable, we again require \( \tau(\bar{r}) = \text{Low} \) to make sure the dereferenced pointer does not leak secrets. The constraint \( \tau(r) \subseteq X_h(S) \) propagates the security level of the stored value. For a \( \text{store} \) 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 \( X_h[p] \), written \( X_h[p] \vdash p \), if there exists \((X_h, T)\) such that for every \( S \in X_h[p] \), \( X_h(S) = \text{High} \) and for all nodes \( n \) and all its successors \( n' \), there exists \( \tau \) such that:

\[
X_h \vdash n : T(n) = \tau \quad \land \quad \tau \subseteq T(n')
\]
where \( \subseteq \) is the natural lifting of \( \subseteq \) from \( \mathbb{L} \) to types.

We automatically infer \( X_b \) and \( T \) using Kildall’s algorithm [33].

4. SOUNENESS OF TYPE SYSTEM

We capture the soundness of the static analyses with respect to two distinct non-interference properties. The first property is expressed 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, relates them informally to existing threat models, and provides 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 \((X_b, T)\) consisting of a global type and a local type an equivalence relation \(\sim_{X_b,T}\) 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 \((X_b, T)\), equivalence can be extended to traces\(^2\) of \(p\) as follows:

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

are equivalent, written \(\sim_{X_b,T} \theta \equiv \theta'\), if \(i = 0 \ldots a_i = a'_i\) and \(s_i \sim_{X_b,T(pci)} s'_i\), where \(pci\) 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. \(X_b\), written \(\text{LLNI}_{X_b}(p)\), iff for every two traces \(\theta\) and \(\theta'\) obtained by executing \(p\) from initial states \(s\) and \(s'\): \(s \sim_{X_b,T(pci)} s' \implies \theta \sim_{X_b,T} \theta'\)

Note that the definition is parameterized by \((X_b, T)\).

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 2 (LANGUAGE-LEVEL SECURITY).** If a program \(p\) is typable, i.e. \(X_b \vdash p\), then \(p\) is LL non-interfering, i.e. \(\text{LLNI}_{X_b}(p)\).

Proposition 2 states that constant-time programs verify LL non-interference with respect to the set of secrets \(X_b\). It 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 its 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 adaptive. 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 platform. The semantics is built on top of a rich memory model that accounts for virtual, physical, and machine 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:

\[
t \xrightarrow{t, \ell} t'
\]

where \(t, \ell\) range over states and \(b\) is an action. Informally, a labelled transition as above indicates that the execution of the action \(b\) by \(o_v\) in an initial state \(t\) leads to a new state \(t'\). Figure 9 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 \(\text{eff}(b)\); see Figure 9 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 \( \theta \) and \( \theta' \) such that \( \theta \sim \theta' \). Equivalence between partial traces is defined from equivalence \( \sim \) on states (itself defined formally in Section 6):

\[
\theta = t_0 \triangleright_\mathfrak{A} t_1 \triangleright_\mathfrak{A} t_2 \triangleright_\mathfrak{A} \ldots \triangleright_\mathfrak{A} t_n
\]

\[
\theta' = t'_0 \triangleright_\mathfrak{A} t'_1 \triangleright_\mathfrak{A} t'_2 \triangleright_\mathfrak{A} \ldots \triangleright_\mathfrak{A} t'_n
\]

are equivalent, written \( \theta \sim \theta' \), if \( n = n' \), and for \( i = 0 \ldots n-1 \), \( t_i \sim t'_i \), and if the active OS of \( t_i \) is \( o_a \), then \( \text{eff}(b_i) = \text{eff}(b'_i) \) 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 \( \langle \mathfrak{A} \parallel p \rangle[t] \) of \( \mathfrak{A} \) and \( p \) with initial state \( t \); informally, \( \langle \mathfrak{A} \parallel p \rangle[t] \) is the system trace that interleaves execution of \( p \) by \( o_a \) and adversarially-chosen code by \( o_b \), according to the adversarially-chosen scheduling policy—both captured in the definition of \( \mathfrak{A} \). Formally, \( \langle \mathfrak{A} \parallel p \rangle[t] \) is defined recursively: given a partial trace \( \theta \) 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 \( \mathfrak{A}(\theta) \neq v \), or the next step in the execution of \( p \), in case \( \mathfrak{A}(\theta) = v \).

Given a program \( p \) and a set of initial secrets \( X_0^p \), we define an equivalence relation \( \sim_{X_0^p} \) 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 \( X_0^p \), written \( \text{SLNI}_{X_0^p}(p) \), if for every attacker \( \mathfrak{A} \) and initial states \( t \) and \( t' \):

\[
[t \sim_{X_0^p} t' \land t \sim t'] \implies \langle \mathfrak{A} \parallel p \rangle[t] \sim \langle \mathfrak{A} \parallel p \rangle[t']
\]

**Proposition 3 (System-level security).**

If \( p \) is typable, i.e. \( X_0^p : p \), then \( p \) is SL non-interfering, i.e. \( \text{SLNI}_{X_0^p}(p) \).

Proposition 3 states that constant-time programs verify SL non-interference with respect to the set of secrets \( X_0^p \). It proves security against a strong, active attacker, which can interleave the program’s execution with execution of code of its choice.

## 5. EXTENSIONS TO S-CONSTANT-TIME

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 5. Note that there is no requirement that stealth addresses are high; in practice, stealth addresses often store public tables.

**Definition 4 (S-constant-time).** A program \( p \) is S-constant-time with respect to a set of variables \( X_0^p \) and a set of stealth addresses \( X_s \), written \( X_0^p, X_s : p \), if there exists \( X_s, T \) such that for every \( S \in X_0^p \), \( X_s, T(S) \) is High and for all states \( n \) and all its successors \( n' \), there exists \( \tau \) such that:

\[
X_s, X_n : n : T(n) \Rightarrow \tau \land \tau \subseteq T(n')
\]

where \( \subseteq \) is the natural lifting of \( \subseteq \) from \( L \) to types.

**Figure 5: Information flow rules for S-constant-time**

\[
\frac{p[n] = \text{load} \left( addr, r, r', n' \right)}{\text{PointsTo}(n, addr, r) = \text{Symb}(S)}
\]

\[
\tau(r) = \text{High} \implies S \in X_s
\]

\[
X_s, X_n : n : T(r) \Rightarrow \tau \land \tau \subseteq X_h(S)
\]

\[
\tau(r) = \text{High} \implies S \in X_s
\]

**Figure 6: Modified IR semantics (excerpts)**

We automatically infer \( X_s, X_h \) and \( T \) using Kildall’s algorithm.

LL non-interference is extended to the setting of stealth memory simply by considering a modified labelled operational semantics (see Figure 6) where accessing variables in \( X_s \) has no visible effect; the notion of state equivalence remains unmodified. Below we let \( \text{LLNI}_{X_s, X_h^p}(p) \) denote the resulting policy.

**Proposition 5 (Language-level security).**

If \( X_0^p, X_s^p : p \) then \( \text{LLNI}_{X_s, X_h^p}(p) \).

Given a program \( p \), a set of initial secrets \( X_0^p \), and a set of stealth addresses \( X_s \), we define an equivalence relation \( \sim_{X_0^p} \) 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 \( X_0^p \) and a set of stealth addresses \( X_s \), written \( \text{SLNI}_{X_s, X_h^p}(p) \), if for every attacker \( \mathfrak{A} \) and initial states \( t \) and \( t' \):

\[
[t \sim_{X_0^p} t' \land t \sim t'] \implies \langle \mathfrak{A} \parallel p \rangle[t] \sim \langle \mathfrak{A} \parallel p \rangle[t']
\]

**Proposition 6 (System-level security).**

If \( X_0^p, X_s^p : p \) then \( \text{SLNI}_{X_s, X_h^p}(p) \).

## 6. FORMALIZATION

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 ⊥ to denote undefined values; iii. we consider a single stealth address; iv. we do not model registers. These simplifications do not impact 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 7.

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.

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 hypervisor mapping, Hypercalls are privileged functionalities exported by the hypervisor to the guest OSs; there is at most one pending hypervisor 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 hypervisor 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\(^3\); 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.

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\(^4\) 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.** Our formalization considers a representative set of actions to read and write from memories, manage memory mappings, request and perform privileged actions, and switch between operating systems and the hypervisor; see Figure 9. Figure 10 presents the semantics of two important actions: write (write value in virtual address) and new_sm (extends the stealth memory of the active OS with a new mapping).

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 [34], 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.

\(^3\)To properly deal with the problems posed by aliasing in VIPT caches, pages mapped by two different virtual addresses are flagged as non-cacheable.

\(^4\)A bounded map is a finite map whose domain must have size less than some fixed positive constant.
va, pa, ma

OSId := new | del | lswitch | pin | unpin | none
HC := Pa × HC
OSData := OSId → OSData
GuestOSs := OSId → OSData
OSActivity := running | waiting
ActiveOS := OSId × OSActivity

PageContent := RW (Value) | PT(Va → Ma) | none
PageOwner := hyp | OS(OSId) | none
Page := PageContent × PageOwner × Bool

Memory := Ma → Page
HyperMap := OSId → Pa → Ma
CacheData := Va × Ma → Page
CacheIndex := Va → Index

CacheHistory := Index → Hist
Cache := CacheData × CachIndex × CacheHistory

TLB := Va → Ma

SLST := GuestOSs × ActiveOS × HyperMap × Memory × Cache × TLB

System level state

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.

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 setting5 and we do not know any formal proof of soundness of an information flow type system 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.

5The 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.

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 7. (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 \( \text{get\_page\_hyp}(t, o, pa) = (ma, pg) \), there exists \( ma' \) such that \( \text{get\_page\_hyp}(t', o, pa) = (ma', pg) \);
- if \( \text{get\_page\_hyp}(t, o, pa) = (ma, pg) \), and no page table maps stealth\_va to \( ma \), then there exists \( ma' \) such that \( \text{get\_page\_hyp}(t', o, pa) = (ma', pg') \), where \( pg \) and \( pg' \) are equal except in their contents;

and reciprocally for \( t' \).

Figure 8 provides a pictural representation of the equivalence: we require that the attacker readable/writable pages are the same for \( hyp \) and \( hyp' \). Furthermore, if the same attacker action or two victim actions with the same effect are executed in two equivalent states,

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

\[
\text{eff}(b_1) = \text{eff}(b_2) \quad \Theta_1 \approx \Theta_2 \\
\text{if } t_1^o \cup \Delta_1 \Theta_1 \approx \text{eff}(a') \quad \text{and } s_1 \sim s_2, \quad \text{then } s_1^o \sim s_2^o.
\]

We then define equivalence of traces:

\[
\begin{align*}
& t_1 \sim t_2 \quad \Theta_1 \sim \Theta_2 \\
& t_1^o \cup \Delta_1 \Theta_1 \sim t_2^o \cup \Delta_2 \Theta_2 \\
& t_1^o \cup \Delta_1 \Theta_1 \sim t_2^o \cup \Delta_2 \Theta_2
\end{align*}
\]

**Theorem 10 (OS isolation).** Let \( \Theta \) and \( \Theta' \) 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 \( \Theta \) and \( \Theta' \) respectively, then \( \Theta \sim \Theta' \), i.e. \( \Theta \) and \( \Theta' \) are indistinguishable traces for the attacker system \( \alpha_o \).

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 \( (\mathcal{G} \parallel p)[t] \), and state Proposition 5. The proof of this proposition is a direct consequence of Theorem 10, and shows that S-constant-time programs are protected to cache-based attacks in virtualization platforms.

**7. EVALUATION**

We have tested the effectiveness of our type systems on two sets of examples. The first set of examples consists of small programs that violate the constraints of constant-time algorithms, e.g. branch on secret values. The second set of examples consists of 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 performing very minor modifications of the code\(^6\), compiled them using CompCert, and run our certified type system on the MachIR (or equivalently x86) programs output by the compiler. Figure 1 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.

**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 \( \text{GF}(2^8) \).

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 [48] 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], Acı́mez, Schindler and Ko¸c [2], and Cantaut, 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 [42] show that cache-based attacks are not confined to closed systems, and can be realized in cloud architectures based on virtualization. In a different line of work, Kasper and Schwabe [31] report on a constant-time implementation of AES.

As a test case for our approach, we have applied our S-constant-time type system to the PolarSSL implementation of AES. Our type system 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 [49] and Kelsey et al [32] respectively. We have applied our S-constant-time type system 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.

---

\(^6\)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, Zemen, and Hawkes [37]. We have applied our S-constant-time type system on an 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 Lerestuex [22] present a cache-based attack against RC4. Analyzing the PolarSSL implementation of RC4 with our S-constant-time type system proves that the program can execute securely with only 0.25kB of stealth memory.

**TEA, Salsa20, SHA256.** We have applied our constant-time type system 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, our type system establishes that the programs are secure without using stealth memory.

**RSA.** RSA is a widely used encryption algorithm. We have applied our constant-time type system to implementations of modular exponentiation. As expected, our type system rejects implementations that branch on secrets and accepts constant-time implementations.

### 8. RELATED WORK

**Side-channel attacks in cryptography.** In [36], Kocher presents a practical timing attack on RSA and suggests that many vectors, including the cache, can be exploited to launch side-channel attacks. Acıcmez 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.** CtGrind\(^7\) is an extension of ValGrind that can be used to check automatically that an implementation is constant-time.

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 an abstract version of the concurrent attacker model; however, the bound is only valid under strong restrictions, e.g. on scheduling. Moreover, the results of [14] cannot be used to assert the security of constant-time programs against concurrent cache attacks.

**Language-based protection mechanisms.** Many authors have developed 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 [40] 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 [51] 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 [47] also show how to eliminate cache-based timing attacks, but their adversary model is different.

More recently, Liu et al [39] 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.

**OS verification.** OS verification is an active field of research [46]. One recent breakthrough is the machine-checked refinement proof of an implementation of the seL4 microkernel [35]. Subsequent machine-checked developments prove that seL4 enforces integrity, authority confinement [45] and intransitive non-interference [41]. 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.

---

\(^7\)It was developed circa 2010 by Adam Langley and is available from https://github.com/agl/ctgrind/.
**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 [34]. 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, 34] 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 [38] 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. One direction for future work is to extend our analysis to constant-time programs which branch on secrets.

**Acknowledgements.** We are grateful to Martín Abadi for suggesting to look at stealth memory. The work of G. Betarte, J. Campo and C. Luna was partially funded by project CSIC/Convocatoria 2012, Proyectos I + D, VirtualCert - Fase II, Uruguay.

**References**


<table>
<thead>
<tr>
<th>Action</th>
<th>Informal Description</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>read va</td>
<td>Guest OS reads virtual address va</td>
<td>$\emptyset$ if va is Stealth</td>
</tr>
<tr>
<td>write va val</td>
<td>Guest OS writes value val in va</td>
<td>$\emptyset$ if va is Stealth</td>
</tr>
<tr>
<td>new va pa</td>
<td>Hypervisor extends non-stealth memory of active OS with va $\mapsto$ ma</td>
<td>new va pa</td>
</tr>
<tr>
<td>new_sm stealth va pa</td>
<td>Hypervisor extends stealth memory of active OS with stealth va $\mapsto$ ma</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>switch o</td>
<td>Hypervisor sets o to be active OS</td>
<td>switch o</td>
</tr>
<tr>
<td>lswitch pa</td>
<td>Hypervisor changes the current memory mapping of active OS to be pa</td>
<td>lswitch pa</td>
</tr>
<tr>
<td>hcall c</td>
<td>An OS requires privileged service c to be executed by the hypervisor</td>
<td>hcall c</td>
</tr>
<tr>
<td>chmod pa t</td>
<td>Memory page corresponding to pa is registered and classified with type t</td>
<td>page_pin pa t</td>
</tr>
<tr>
<td>page_unpin pa</td>
<td>Memory page of active OS that corresponds to pa is un-registered</td>
<td>page_unpin pa</td>
</tr>
</tbody>
</table>

**Figure 9: Selected actions and their effects**

**Action write va val**

Guest OS writes value val in va

<table>
<thead>
<tr>
<th>Rule</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$aos_{act} = (aos, running)$</td>
<td>get_page_mem(t, va) $= (ma, pg)$</td>
</tr>
<tr>
<td>$pg = (\text{RW}_O a\text{os}, b)$</td>
<td>$pol = (\text{RW}^t_v a\text{os}, aos, b)$</td>
</tr>
<tr>
<td>$cache_add(cache, va, ma, pol) = (cache', (ma', pg'))$</td>
<td></td>
</tr>
<tr>
<td>$mem[ma'] := pg'[ma := pol]$</td>
<td>$tlb[va := ma] = tlb'$</td>
</tr>
<tr>
<td>$t = (oss, aos, act, hyp, mem, cache, tlb)$</td>
<td>$t' = (oss, aos, act, hyp, mem', cache', tlb')$</td>
</tr>
</tbody>
</table>

**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, 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).

**Action new_sm stealth va pa**

Add stealth va $\mapsto$ ma to stealth memory of active OS

<table>
<thead>
<tr>
<th>Rule</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$aos_{act} = (aos, waiting)$</td>
<td>get_page_hyp(t, aos, pa) $= (ma, pg)$</td>
</tr>
<tr>
<td>$pg = (\text{RW}_O aos, true)$</td>
<td>$\neg memory_alias(mem, stealth_va, ma)$</td>
</tr>
<tr>
<td>$get_page_hyp(t, aos, pa') = (ma', cpt)$</td>
<td>$cpt[stealth_va] = \emptyset$</td>
</tr>
<tr>
<td>$oss[aos := (pa', \text{None})] = oss'$</td>
<td>$cpt[stealth_va := ma'] = cpt'$</td>
</tr>
<tr>
<td>$mem[ma' := cpt'] = mem'$</td>
<td>$cache_add(cache, stealth_va, ma, pg) = (cache', _)$</td>
</tr>
<tr>
<td>$tlb[stealth_va := ma] = tlb'$</td>
<td>$t = (oss', aos, act, hyp, mem, cache, tlb)$</td>
</tr>
<tr>
<td>$t' = (oss', aos, act, hyp, mem', cache', tlb')$</td>
<td>$t new_sm stealth_va, pa, t'$</td>
</tr>
</tbody>
</table>

**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 page stealth va and no aliases pages are cached.

**Postcondition** 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.

**Figure 10: Semantics of write and new_sm actions**