Rigorous Analysis of Software Countermeasures against Cache Attacks

Goran Doychev Boris Köpf

IMDEA Software Institute, Spain

{goran.doychev,boris.koepf}@imdea.org

Abstract

CPU caches introduce variations into the execution time of programs that can be exploited by adversaries to recover private information about users or cryptographic keys.

Establishing the security of countermeasures against this threat often requires intricate reasoning about the interactions of program code, memory layout, and hardware architecture and has so far only been done for restricted cases.

In this paper we devise novel techniques that provide support for bit-level and arithmetic reasoning about memory accesses in the presence of dynamic memory allocation. These techniques enable us to perform the first rigorous analysis of widely deployed software countermeasures against cache attacks on modular exponentiation, based on executable code.

CCS Concepts • **Security and privacy** → **Software security engineering**

Keywords Side channel attacks, Countermeasures, Caches

1. Introduction

CPU caches reduce the latency of memory accesses on average, but not in the worst case. Thus, they introduce variations into the execution time of programs, which can be exploited by adversaries to recover secrets, such as private information about users or cryptographic keys [1, 8, 23, 39, 41, 47].

A large number of techniques have been proposed to counter this threat. Some proposals work at the level of the operating system [19, 27, 50], others at the level of the hardware architecture [22, 44, 45] or the cryptographic protocol [17]. In practice, however, software countermeasures are often the preferred choice because they can be easily deployed.

PLDI'17, June 18–23, 2017, Barcelona, Spain
ACM. 978-1-4503-4988-8/17/06...$15.00
http://dx.doi.org/10.1145/3062341.3062388

A common software countermeasure is to ensure that control flow, memory accesses, and execution time of individual instructions do not depend on secret data [9, 31]. While such code prevents leaks through instruction and data caches, hiding all dependencies can come with performance penalties [12].

More permissive countermeasures are to ensure that both branches of each conditional fit into a single line of the instruction cache, to preload lookup tables, or to permit secret-dependent memory access patterns as long as they are secret-*in*dependent at the granularity of cache lines or sets. Such permissive code can be faster and is widely deployed in crypto-libraries such as OpenSSL. However, analyzing its security requires intricate reasoning about the interactions of the program and the hardware platform and has so far only been done for restricted cases [16].

A major hurdle for reasoning about these interactions are the requirements put on tracking memory addresses: On the one hand, static analysis of code with dynamic memory allocation requires memory addresses to be dealt with symbolically. On the other hand, analysis of cache-aligned memory layout requires support for accurately tracking the effect of bit-level and arithmetic operations. While there are solutions that address each of these requirements in isolation, supporting them together is challenging, because the demand for symbolic treatment conflicts with the demand for bit-level precision.

In this paper, we propose novel techniques that meet both requirements and thus enable the automated security analysis of permissive software countermeasures against microarchitectural side-channel attacks in executable code.

Abstract Domains Specifically, we introduce *masked symbols*, which are expressions that represent unknown addresses, together with information about some of their bits. Masked symbols encompass unknown addresses as well as known constants; more importantly, they also support intermediate cases, such as addresses that are unknown except for their least significant bits, which are zeroed out to align with cache line boundaries. We cast arithmetic and bit-level operations on masked symbols in terms of a simple set-based abstract domain, which is a data structure that supports approximating the semantics of programs [14]. We moreover introduce a DAG-based abstract domain to represent sets of traces of masked symbols.

Adversary Models Our novel abstract domains enable us to reason about the security of programs against a wide range of microarchitectural side channel adversaries, most of which are out of the scope of existing approaches. The key observation is that the capability of these adversaries to observe a victim's execution can be captured in terms of projections to some of the bits of the addresses accessed by the victim. This modeling encompasses adversaries that can see the full trace of accesses to the instruction cache (commonly known as the *program counter security model* [36]), but also weaker ones that can see only the trace of memory pages, blocks, or cache banks, with respect to data, instruction, or shared caches.

Bounding Leaks We use our abstract domains for deriving upper bounds on the amount of information that a program leaks. We achieve this by counting the number of observations each of these adversaries can make during program execution, as in [30, 33, 38]. In this paper we perform the counting by applying an adversary-specific projection to the set of masked symbols corresponding to each memory access. We highlight two features of this approach:

- The projection may collapse a multi-element set to a singleton set, for example, in the case of different addresses mapping to the same memory block. This is the key for establishing that some memory accesses do not leak information to some observers, even if they depend on secret data.
- As the projection operates on individual bits, we can compute the adversary's observations on addresses that contain both known and unknown bits. In this way, our counting effectively captures the leak of the program, rather than the uncertainty about the address of the dynamically allocated memory.

Implementation and Evaluation We implement our novel techniques on top of the CacheAudit static binary analyzer [16], and we evaluate their effectiveness in a case study where we perform the first formal analysis of commonly used software countermeasures for protecting modular exponentiation algorithms. The paper contains a detailed description of our case study; here we highlight the following results:

- We analyze the security of the scatter/gather countermeasure used in OpenSSL 1.0.2f for protecting window-based modular exponentiation. Scatter/gather ensures that the pattern of data cache accesses is secret-independent at the level of granularity of cache lines and, indeed, our analysis of the binary executable reports security against adversaries that can monitor only cache line accesses.
- Our analysis of the scatter/gather countermeasure reports a leak with respect to adversaries that can monitor memory accesses at a more fine-grained resolution. This leak has been exploited in the CacheBleed attack [48], where the adversary observes accesses to the individual banks within a cache line. We analyze the variant of scatter/gather published in OpenSSL 1.0.2g as a response to the attack and prove its security with respect to powerful adversaries that can monitor the full address trace.
- Our analysis detects the side channel in the square-and-multiply algorithm in libgcrypt 1.5.2 that has been exploited in [32, 47], but can prove the absence of an instruction cache leak in the square-and-*always*-multiply algorithm used in libgcrypt 1.5.3, for some compiler optimization levels.

Overall, our results illustrate (once again) the dependency of software countermeasures against cache attacks on brittle details of the compilation and the hardware architecture, and they demonstrate (for the first time) how automated program analysis can effectively support the rigorous analysis of permissive software countermeasures.

In summary, our contributions are to devise novel techniques that enable cache-aware reasoning about dynamically allocated memory, and to put these techniques to work in the first rigorous analysis of widely deployed permissive countermeasures against cache side channel attacks.

2. Illustrative Example

We illustrate the scope of the techniques developed in this paper using a problem that arises in implementations of windowed modular exponentiation. There, powers of the base are pre-computed and stored in a table for future lookup. Figure 1 shows an example memory layout of two such pre-computed values p_2 and p_3, each of 3072 bits, which are stored in heap memory. An adversary that observes accesses to the six memory blocks starting at `80eb140` knows that p_2 was requested, which can give rise to effective key-recovery attacks [32].

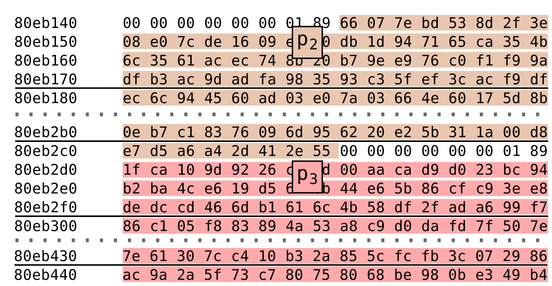


Figure 1: Layout of pre-computed values in main memory, for the windowed modular exponentiation implementation from libgcrypt 1.6.1. Black lines denote the memory block boundaries, for an architecture with blocks of 64 bytes.

Defensive approaches for table lookup, as implemented in NaCl or libgcrypt 1.6.3, avoid such vulnerabilities by accessing *all* table entries in a constant order. OpenSSL 1.0.2f instead uses a more permissive approach that accesses only *one* table entry, however it uses a smart layout of the tables to ensure that the memory blocks are loaded into the cache in a constant order. An example layout for storing 8 pre-computed values is shown in Figure 2. The code that manages such tables consists of three functions, which are given in Figure 3.

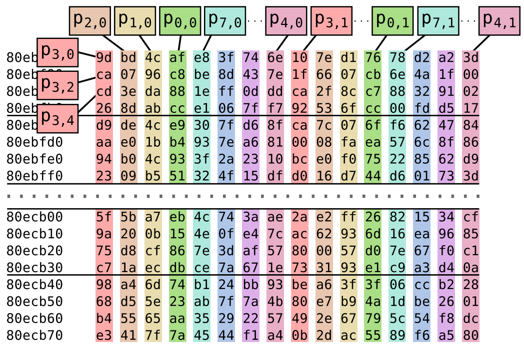


Figure 2: Layout of pre-computed values in main memory, achieved with the scatter/gather countermeasure. Data highlighted in different colors correspond to pre-computed values $p_0, \ldots, p_7$, respectively. Black lines denote the memory block boundaries, for an architecture with blocks of 64 bytes.

- To create the layout, the function `align` aligns a buffer with the memory block boundary by ensuring the least-significant bits of the buffer address are zeroed.
- To write a value into the array, the function `scatter` ensures that the bytes of the pre-computed values are stored `spacing` bytes apart.
- Finally, to retrieve a pre-computed value from the buffer, the function `gather` assembles the value by accessing its bytes in the same order they were stored.

```
align ( buf ):
    return buf − ( buf & ( block_size − 1 ) ) + block_size

scatter ( buf, p, k ):
    for i := 0 to N − 1 do
        buf [k + i ∗ spacing] := p [k][i]

gather ( r, buf, k ):
    for i := 0 to N − 1 do
        r [i] := buf [k + i ∗ spacing]
```

Figure 3: Scatter/gather method from OpenSSL 1.0.2f for aligning, storing and retrieving pre-computed values.

The techniques developed in this paper enable automatic reasoning about the effectiveness of such countermeasures, for a variety of adversaries. Our analysis handles the dynamically-allocated address in `buf` from Figure 3 symbolically, but is still able to establish the effect of `align` by considering bit-level semantics of arithmetic operators on symbols: First, the analysis establishes that `buf & (block_size - 1)` clears the most-significant bits of the symbol s; second, when subtracting this value from `buf`, the analysis determines that the result is s, with the least-significant bits cleared; third, the analysis determines that adding `block_size` leaves the least-significant bits unchanged, but affects the unknown bits, resulting in a new symbolic address $s' \neq s$ whose least significant bits are cleared.

Using this information, in `gather`, our analysis establishes that, independently from the value of k, at each iteration of the loop, the most-significant bits of the accessed location are the same. Combining this information with knowledge about the architecture such as the block size, the analysis establishes that the sequence of accessed memory blocks is the same, thus the countermeasure ensures security of scatter/gather with respect to adversary who makes observations at memory-block granularity.

3. Memory Trace Adversaries

In this section we formally define a family of side channel adversaries that exploit features of the microarchitecture, in particular: caches. The difference between these adversaries is the granularity at which they can observe the trace of programs' accesses to main memory. We start by introducing an abstract notion of programs and traces.

3.1 Programs and Traces

We introduce an abstract notion of programs as the transformations of the main memory and CPU register contents (which we collectively call the *machine state*), caused by the execution of the program's instructions. Formally, a program $P = (\Sigma, I, \mathcal{A}, \mathcal{R})$ consists of the following components:

- Σ - a set of *states*
- $I \subseteq \Sigma$ - a set of possible *initial* states
- $\mathcal{A}$ - a set of *addresses*
- $\mathcal{R} \subseteq \Sigma \times \mathcal{A}^* \times \Sigma$ - a *transition relation*

A transition $(\sigma_i, a, \sigma_j) \in \mathcal{R}$ captures two aspects of a computation step: first, it describes how the instruction set semantics operates on data stored in the machine state, namely by updating σ_i to σ_j; second, it describes the sequence of memory accesses $a \in \mathcal{A}^*$ issued during this update, which includes the addresses accessed when fetching instructions from the code segment, as well as the addresses containing accessed data. To capture the effect of one computation step in presence of uncertain inputs, we define the $next$ operator:

$$next(S) = \{t.\sigma_k a_k \sigma_{k+1} \mid t.\sigma_k \in S \wedge (\sigma_k, a_k, \sigma_{k+1}) \in \mathcal{R}\} .$$

A *trace* of P is an alternating sequence of states and addresses $\sigma_0 a_0 \sigma_1 a_1 \ldots \sigma_k$ such that $\sigma_0 \in I$, and that for all $i \in \{0, \ldots, k-1\}$, $(\sigma_i, a_i, \sigma_{i+1}) \in \mathcal{R}$. The set of all traces of P is its *collecting semantics* $Col \subseteq Traces$. In this paper, we only consider terminating programs, and define their collecting semantics as the least fixpoint of the $next$ operator containing I: $Col = I \cup next(I) \cup next^2(I) \cup \ldots$.

3.2 A Hierarchy of Memory Trace Observers

Today's CPUs commonly partition the memory space into units of different sizes, corresponding to virtual memory pages, cache lines, or cache banks. The delays introduced by page faults, cache misses, or bank conflicts enable real-world adversaries to effectively identify the units involved in another program's memory accesses. We explicitly model this capability by defining adversaries that can observe memory accesses at the granularity of each unit, but that cannot distinguish between accesses to positions within the same unit.

Observing a Memory Access On a common n-bit architecture, the most significant $n - b$ bits of each address serve as an identifier for the unit containing the addressed data, and the least significant b bits serve as the offset of the data within that unit, where 2^b is the byte-size of the respective unit.

We formally capture the capability to observe units of size 2^b by projecting addresses to their $n - b$ most significant bits, effectively making the b least significant bits invisible to the adversary. That is, when accessing the n-bit address $a = (x_{n-1}, x_{n-2}, \ldots, x_0)$, the adversary observes

$$\pi_{n:b}(a) := (x_{n-1}, x_{n-2}, \ldots, x_b) .$$

Example 1. *A 32-bit architecture with 4KB pages, 64B cache lines, and 4B cache banks will use bits 0 to 11 for offsets within a page, 0 to 5 for offsets within a cache line, and 0 to 1 for offsets within a cache bank. That is, the corresponding adversaries observe bits 12 to 31, 6 to 31, and 2 to 31, respectively, of each memory access.*

Observing Program Executions We now lift the capability to observe individual memory accesses to full program executions. This lifting is formalized in terms of *views*, which are functions that map traces in Col to sequences of projections of memory accesses to observable units. Formally, the view of an adversary on a trace of the program is defined by

$$view\colon \sigma_0 a_0 \sigma_1 a_1 \ldots \sigma_k \mapsto \pi_{n:b}(a_0)\pi_{n:b}(a_1)\ldots\pi_{n:b}(a_{k-1}) .$$

By considering $\pi_{n:b}$ for different values of b, we obtain a hierarchy of memory trace observers:

- The *address-trace observer* corresponds to $b = 0$; it can observe the full sequence $a_0 a_1 \ldots a_{k-1}$ of memory locations that are accessed. Security against this adversary implies resilience to many kinds of microarchitectural side channels, through cache, TLB, DRAM, and branch prediction buffer.[1] An address-trace observer restricted to instruction-addresses is equivalent to the program counter security model [36].
- The *block-trace observer* can observe the sequence of memory blocks loaded into cache lines. Blocks are commonly of size 32, 64, or 128 bytes, i.e. $b = 5$, 6, or 7. Security against this adversary implies resilience against adversaries that can monitor memory accesses at the level of granularity of cache lines. Most known cache-based attacks exploit observations at the granularity of cache lines, e.g. [32, 40, 49].
- The *bank-trace observer* can observe a sequence of accessed cache banks, a technology used in some CPUs for hyperthreading. An example of an attack at the granularity of cache banks is CacheBleed [48] against the scatter/-gather implementation from OpenSSL 1.0.2f. The platform targeted in this attack has 16 banks of size 4 bytes, i.e. $b = 2$.
- The *page-trace observer* can observe memory accesses at the granularity of accessed memory pages, which are commonly of size 4096 bytes, i.e. $b = 12$. Examples of such attacks appear in [46] and [42].

We denote the views of these observers by $view^{\mathit{address}}$, $view^{\mathit{block}}$, $view^{\mathit{bank}}$, and $view^{\mathit{page}}$, respectively.

Observations Modulo Stuttering For each of the observers defined above we also consider a variant that cannot distinguish between repeated accesses to the same unit (which we call *stuttering*). This is motivated by the fact that the latency of cache misses dwarfs that of cache hits and is hence easier to observe.

For the observer $view^{\mathit{block}}$, we formalize this variant in terms of a function $view^{\mathit{b\text{-}block}}$ taking as input a block-sequence w and replacing the maximal subsequences $B \cdots B$ of each block B in w by the single block B. E.g., $view^{\mathit{b\text{-}block}}$ maps both $AABCDDC$ and $ABBBCCDDCC$ to the sequence $ABCDC$, making them indistinguishable to the adversary. This captures an adversary that cannot count the number of memory accesses, as long as they are guaranteed to be cache hits[2].

4. Static Quantification of Leaks

In this section, we characterize the amount of information leaked by a program, and we show how this amount can be over-approximated by static analysis. While the basic idea is standard (we rely on upper bounds on the cardinality of an adversary's view), our presentation exhibits a new path for performing such an analysis in the presence of low inputs. In this section we outline the basic idea, which we instantiate in Sections 5 and 6 for the observers defined in Section 3.

Quantifying Leaks As is common in information-flow analysis, we quantify the amount of information leaked by a program about its secret (or *high*) inputs in terms of the maximum number of observations an adversary can make, for any valuation of the public (or *low*) input [29, 33, 43].

To reflect the distinction between high and low inputs in the semantics, we split the initial state into a low part I_{lo} and a high part I_{hi}, i.e., $I = I_{lo} \times I_{hi}$. We split the

[1] We do not model, or make assertions about, the influence of advanced features such as out-of-order-execution.

[2] Here we rely on the (weak) assumption that the second B in any access sequence $\cdots BB \cdots$ is guaranteed to hit the cache.

collecting semantics into a family of collecting semantics Col_λ with $I = \{\lambda\} \times I_{hi}$, one for each $\lambda \in I_{lo}$, such that $Col = \bigcup\limits_{\lambda} Col_\lambda$.

Formally, we define *leakage* as the maximum cardinality of the adversary's view w.r.t. all possible low inputs:

$$\mathcal{L} := \max_{\lambda \in I_{lo}}(|view(Col_\lambda)|)\ . \tag{1}$$

The logarithm of this number corresponds to the number of leaked *bits*, and it comes with different interpretations in terms of security. For example, it can be related to a lower bound on the expected number of guesses an adversary has to make for successfully recovering the secret input [34], or to an upper bound on the probability of successfully guessing the secret input in one shot [43]. Note that a leakage $\mathcal{L}$ of 1 (i.e. 0 bits) corresponds to non-interference.

Static Bounds on Leaks For quantifying leakage based on Equation 1, one needs to determine the size of the range of $view$ applied to the fixpoint Col_λ of the $next$ operator, for all $\lambda \in I_{lo}$ – which is infeasible for most programs.

For *fixed* values $\lambda \in I_{lo}$, however, the fixpoint computation can be tractably approximated by abstract interpretation [14]. The result is a so-called abstract fixpoint $Col^\sharp$ that represents a superset of Col_λ, based on which one can over-approximate the range of $view$ [30]. One possibility to obtain bounds for the leakage that hold for *all* low values is to compute one fixpoint w.r.t. all possible I_{lo} rather than one for each single $\lambda \in I_{lo}$ [16]. The problem with this approach is that possible variation in low inputs is reflected in the leakage, which can lead to imprecision.

Secret vs Public, Known vs Unknown Inputs The key to our treatment of low inputs is that we classify variables along two dimensions.

- The first dimension is whether variables carry secret (or *high*) or public (or *low*) data. High variables are represented in terms of the set of all possible values the variables can take, where larger sets represent more uncertainty about the values of the variable. Low data is represented in terms of a singleton set.
- The second dimension is whether variables represent values that are known at analysis time or not. Known values are represented by constants whereas unknown values are represented as symbols.

Example 2. *The set* $\{1, 2\}$ *represents a high variable that carries one of two known values. The set* $\{s\}$ *represents a low variable that carries a value* s *that is not known at analysis time. The set* $\{1\}$ *represents a low variable with known value* 1*. Combinations such as* $\{1, s\}$ *are possible; this example represents a high variable, one of its possible values is unknown at analysis time.*

While existing quantitative information-flow analyses that consider low inputs rely on explicit tracking of path relations [7, 13], our modeling allows us to identify – and factor out – variations in observable outputs due to low inputs even in simple, set-based abstractions. This enables us to compute fixpoints $Col^\sharp(s)$ containing symbols, based on techniques that are known to work on intricate low-level code, such as cryptographic libraries [16]. The following example illustrates this advantage.

Example 3. *Consider the following program, where variable* x *is initially assigned a pointer to a dynamically allocated memory region. We assume that the pointer is low but unknown, which we model by* $x = \{s\}$*, for some symbol* s*. Depending on a secret bit* $h \in \{0, 1\}$ *this pointer is increased by 64 or not.*

```
x:= malloc(1000);
if h then
    x := x+64
```

For an observer who can see the value of x *after termination, our analysis will determine that leakage is bounded by* $\mathcal{L} \leq |\{s, s + 64\}| = 2$*. This bound holds for* any *value that* s *may take in the initial state* λ*, effectively separating uncertainty about low inputs from uncertainty about high inputs.*

In this paper we use low input to model dynamically allocated memory locations, as in Example 3. That is, we rely on the assumption that locations chosen by the allocator do not depend on secret data. More precisely, we assume that the initial state contains a pool of low but unknown heap locations that can be dynamically requested by the program.

5. Masked Symbol Abstract Domain

Cache-aware code often uses Boolean and arithmetic operations on pointers in order to achieve favorable memory alignment. In this section we devise the *masked symbol domain*, which is an abstract domain that enables the static analysis of such code in the presence of dynamically allocated memory, i.e., when the base pointer is unknown.

5.1 Representation

The masked symbol domain is based on finite sets of what we call *masked symbols*, which are pairs (s, m) consisting of the following components:

1. a *symbol* $s \in Sym$, uniquely identifying an unknown value, such as a base address;
2. a *mask* $m \in \{0, 1, \top\}^n$, representing a pattern of known and unknown bits. We abbreviate the mask $(\top, \ldots, \top)$ by $\top$.

The i-th bit of a masked symbol (s, m) is called *masked* if $m_i \in \{0, 1\}$, and *symbolic* if $m_i = \top$. Masked bits are known at analysis time, whereas symbolic bits are not. Two special cases are worth pointing out: The masked symbol $(s, \top)$, with $\top$ as shorthand for $(\top, \ldots, \top)$, represents a vector of unknown bits, and (s, m) with $m \in \{0, 1\}^n$ represents the bit-vector m. In that way, masked symbols generalize both unknown values and bitvectors.

We use finite sets of masked symbols to represent the elements of the masked symbol domain, that is, $\mathcal{M}^\sharp = \mathcal{P}(Sym \times \{0,1,\top\}^n)$.

5.2 Concretization

We now give a semantics to elements of the masked symbol domain. This semantics is parametrized w.r.t. valuations of the symbols. For the case where masked symbols represent partially known heap addresses, a valuation corresponds to one specific layout of the heap.

Technically, we define the concretization of elements $x^\sharp \in \mathcal{M}^\sharp$ w.r.t. a function $\lambda\colon Sym \to \{0,1\}^n$ that maps symbols to bit-vectors:

$$\gamma_\lambda^{\mathcal{M}^\sharp}(x^\sharp) = \{\lambda(s) \odot m \mid (s,m) \in x^\sharp\}$$

Here $\odot$ is defined bitwise by $(\lambda(s) \odot m)_i = m_i$ if $m_i \in \{0,1\}$, and $\lambda(s)_i$ if $m_i = \top$.

The function λ takes the role of the low initial state, for which we did not assume any specific structure in Section 4. Modeling λ as a mapping from symbols to bitvectors is a natural refinement to an initial state consisting of multiple components that are represented by different symbols.

5.3 Counting

We now show that the precise valuation of the symbols can be ignored for deriving upper bounds on number of observations that an adversary can make about a set of masked symbols. For this we conveniently interpret a symbol s as a vector of identical symbols $(s,\ldots,s)$, one per bit.[3] This allows us to apply the adversary's $view$ (see Section 3) on masked symbols as the respective projection π to a subset of observable masked bits.

Given a set of masked symbols, we count the observations with respect to the adversary by applying π on the set's elements and taking the cardinality of the resulting set.

Example 4. *The projection of the set*

$$x^\sharp = \{(s,(0,0,1)),(t,(\top,\top,1)),(u,(1,1,1))\}$$

of (three bit) masked symbols to their two most significant bits yields the set $\{(0,0),(t,t),(1,1)\}$*, i.e., we count three observations. However, the projection to their least significant bit yields only the singleton set* $\{1\}$*, i.e., the observation is determined by the masks alone.*

The next proposition shows that counting the symbolic observations after projecting, as in Example 4, yields an upper bound for the range of the adversary's view, for *any* valuation of the symbols. We use this effect for static reasoning about cache-aware memory alignment.

[3] We use this interpretation to track the provenance of bits in projections; it does *not* imply that the bits of $\lambda(s)$ are identical.

Proposition 1. *For every $x^\sharp \in \mathcal{M}^\sharp$, every valuation $\lambda\colon Sym \to \{0,1\}^n$, and every projection π mapping vectors to a subset of their components, we have $|\pi(\gamma_\lambda^{\mathcal{M}^\sharp}(x^\sharp))| \leq |\pi(x^\sharp)|$.*

Proof. This follows from the fact that equality of π on (s,m) and (s',m') implies equality of π on $\gamma_\lambda^{\mathcal{M}^\sharp}(s,m)$ and $\gamma_\lambda^{\mathcal{M}^\sharp}(s',m')$, for all λ. To see this, assume there is a symbolic bit in $\pi(s,m)$. Then we have $s = s'$, and hence $\lambda(s) = \lambda(s')$. If there is no symbolic bit, the assertion follows immediately. $\square$

5.4 Update

We now describe the effect of Boolean and arithmetic operations on masked symbols. We focus on operations between pairs of masked symbols; the lifting of those operations to sets is obtained by performing the operations on all pairs of elements in their product. The update supports tracking information about known bits (which are encoded in the mask) and about the arithmetic relationship between symbols. We explain both cases below.

5.4.1 Tracking Masks

Cache-aware code often aligns data to the memory blocks of the underlying hardware.

Example 5. *The following code snippet is produced when compiling the* `align` *function from Figure 3 with* `gcc -O2`*. The register* `EAX` *contains a pointer to a dynamically allocated heap memory location.*

```
AND 0xFFFFFFC0, EAX
ADD 0x40, EAX
```

The first line ensures that the 6 least significant bits of that pointer are set to 0*, thereby aligning it with cache lines of* 64 *bytes. The second line ensures that the resulting pointer points into the allocated region while keeping the alignment.*

We support different Boolean operations and addition on masked symbols that enable us to analyze such code. The operations have the form $(s'',m'') = \mathtt{OP}^\sharp(s,m),(s',m')$, where the right-hand side denotes the inputs and the left-hand side the output of the operation. The operations are defined bit-wise as follows:

$\mathtt{OP} = \mathtt{AND}$ or $\mathtt{OP} = \mathtt{OR}$: $m''_i = \mathtt{OP}\, m_i, m'_i$, for all i such that $m_i, m'_i \in \{0,1\}$, i.e., the abstract $\mathtt{OP}^\sharp$ extends the concrete `OP`. Whenever m_i or m'_i is absorbing (i.e., 1 for `OR` and 0 for `AND`), we set m''_i to that value. In all other cases, we set $m''_i = \top$.
The default is to introduce a fresh symbol for s'', unless the logical operation acts neutral on all symbolic bits, in which case we can set $s'' = s$. This happens in two cases: first, if the operands' symbols coincide, i.e. $s = s'$; second, if one operand is constant, i.e. $m' \in \{0,1\}^n$, and $m_i = \top$ implies that m'_i is neutral (i.e., 0 for `OR` and 1 for `AND`).

$\mathtt{OP} = \mathtt{XOR}$: $m''_i = \mathtt{XOR}\, m_i, m'_i$, for all i such that $m_i, m'_i \in \{0,1\}$, i.e., the abstract $\mathtt{XOR}^\sharp$ extends the concrete XOR. Whenever the symbols coincide, i.e. $s = s'$, we further set $m''_i = 0$, for all i with $m_i = m'_i = \top$. In all other cases, we set $m''_i = \top$.
The default is to introduce a fresh symbol for s''. We can avoid introducing a fresh symbol and set $s'' = s$ in case one operand is a constant that acts neutral on each symbolic bit of the other, i.e., if $m' \in \{0,1\}^n$ and if $m_i = \top$ implies $m'_i = 0$.

$\mathtt{OP} = \mathtt{ADD}$: Starting from $i = 0, 1, \ldots$, and as long as $m_i, m'_i \in \{0,1\}$, we compute m''_i according to the standard definition of ADD[4]. As soon as we reach i with $m_i = \top$ or $m'_i = \top$, we set $m''_j = \top$ for all $j \geq i$.
The default is to use a fresh symbol s'', unless one operand is a constant that acts neutral on the symbolic most-significant bits of the other, i.e., if $m' \in \{0,1\}^n$ and for all $j \geq i$, $m_j = \top$ implies $m'_j = 0$ and $c_j = 0$; then we keep the symbol, i.e., $s'' = s$.

$\mathtt{OP} = \mathtt{SUB}$: We compute SUB similarly to ADD, where the borrow bit takes the role of the carry bit. Here, we use the additional rule that whenever the symbols coincide, i.e. $s = s'$, we further set $m''_i = 0$, for all i with $m_i = m'_i = \top$.

Example 6. *Consider again Example 5 and assume that* EAX *initially has the symbolic value* $(s, \top)$*. Executing Line 1 yields the masked symbol*

$$(s, (\top \cdots \top 000000)) \,, \tag{2}$$

Executing Line 2, we obtain $(s', (\top \cdots \top 000000))$*, for a fresh* s'*. This masked symbol points to the beginning of some (unknown) cache line. In contrast, addition of* 0x3F *to* (2) *would yield* $(s, (\top \cdots \top 111111))$*, for which we can statically determine containment in the same cache line as* (2).

5.4.2 Tracking Offsets

Control flow decisions in low-level code often rely on comparisons between pointers. For analyzing such code with sufficient precision, we need to keep track of their relative distance.

Example 7. *Consider the function* gather *from Figure 3. When compiled with* gcc -O2*, the loop guard is translated into a comparison of pointers. The corresponding pseudocode looks as follows:*

```
y := r + N
for x := r; x ≠ y; x++ do
    *x = buf [k + i * spacing]
```

Here, r *points to a dynamic memory location. The loop terminates whenever pointer* x *reaches pointer* y.

[4] ADD between two bit-vectors x and y determines the i-th bit of the result r as $r_i = x_i \oplus y_i \oplus c_i$, where c_i is the carry bit. The carry bit is defined by $c_i = (x_{i-1} \wedge y_{i-1}) \vee (c_{i-1} \wedge x_{i-1}) \vee (c_{i-1} \wedge y_{i-1})$, with $c_0 = 0$.

In this section we describe a mechanism that tracks the distance between masked symbols, and enables the analysis of comparisons, such as the one in Example 7.

Origins and Offsets Our mechanism is based on assigning to each masked symbol an *origin* and an *offset* from that origin. The origin of a symbol is the masked symbol from which it was derived by a sequence addition operations, and the offset tracks the cumulated effect of these operations.

$$\mathit{orig}\colon \mathcal{M} \to \mathcal{M} \qquad \mathit{off}\colon \mathcal{M} \to \mathbb{N}$$

Initially, $\mathit{orig}(x) = x$ and $\mathit{off}(x) = 0$, for all $x \in \mathcal{M}$.

For convenience, we also define a partial inverse of *orig* and *off* describing the *successor* of an origin at a specific offset. We formalize this as a function $\mathit{succ}\colon \mathcal{M} \times \mathbb{N} \to \mathcal{M} \cup \{\bot\}$ such that $\mathit{succ}(\mathit{orig}(x), \mathit{off}(x)) = x$.

Addition of Offsets When performing an addition of a constant to a masked symbol, the mechanism first checks if there is already a masked symbol with the required offset. If such a symbol exists, it is reused. If not, the addition is carried out and memorized.

More precisely, the result of performing the addition $y = \mathtt{ADD}^\sharp\, x, c$ of a masked symbol $x \in \mathcal{M}$ with a constant $c \in \{0,1\}^n$ is computed as follows:

1. If $\mathit{succ}(\mathit{orig}(x), \mathit{off}(x) + c) = x'$ for some masked symbol x', then we set $y = x'$.
2. If $\mathit{succ}(\mathit{orig}(x), \mathit{off}(x) + c) = \bot$, then we compute $y = \mathtt{ADD}^\sharp\, x, c$, as described in Section 5.4.1, and update $\mathit{orig}(y) = \mathit{orig}(x)$, $\mathit{off}(y) = \mathit{off}(x) + c$, and $\mathit{succ}(\mathit{orig}(y), \mathit{off}(y)) = y$.

Note that we restrict to the case in which one operand is a constant. In case *both* operands contain symbolic bits, for the result y (obtained according to Section 5.4.1) we set $\mathit{orig}(y) = y$ and $\mathit{off}(y) = 0$.

5.4.3 Tracking Flag Values

Our analysis is performed at the level of disassembled x86 binary code, where inferring the status of CPU flags is crucial for determining the program's control flow. We support limited derivation of flag values; in particular, we determine the values of the zero (ZF) and carry flags (CF) as follows.

For the Boolean and addition operations described in Section 5.4.1, we determine the value of flag bits as follows:

- If at least one masked bit of the result is non-zero, then ZF = 0.
- If the operation does not affect the (possibly symbolic) most-significant bits of the operands, then CF = 0.

For comparison and subtraction operations, we rely on offsets for tracking their effect on flags. Specifically, for $\mathtt{CMP}^\sharp\, x, y$ and $\mathtt{SUB}^\sharp\, x, y$, with source x and target y, we determine the value of flags as follows:

- If $x = y$, then ZF = 1;
- If $\mathit{orig}(x) = \mathit{orig}(y)$ and $\mathit{off}(x) \neq \mathit{off}(y)$ then ZF = 0;

In any other case, we assume that all combinations of flag values are possible.

Example 8. *Consider again the code in Example 7. Termination of the loop is established by an instruction* `CMP` x, y, *followed by a conditional jump in case the zero flag is not set. In our analysis, we infer the value of the zero flag by comparing the offsets of* x *and* y *from their common origin* r.

6. Memory Trace Abstract Domain

In this section, we present the *memory trace domain*, which is a data structure for representing the set of traces of possible memory accesses a program can make, and for computing the number of observations that the observers defined in Section 3.2 can make.

6.1 Representation

We use a directed acyclic graph (DAG) to compactly represent sets of traces of memory accesses. This generalizes a data structure that has been previously used for representing sets of traces of cache hits and misses [16].

A DAG $t^\sharp$ from the memory trace domain $\mathcal{T}^\sharp$ is a tuple (V, E, r, L, R). The DAG has a set of vertices V representing program points, with a root $r \in V$ and a set of edges $E \subseteq V \times V$ representing transitions. We equip the DAG with a vertex labeling $L\colon V \to \mathcal{M}^\sharp$ that attaches to each vertex a set of masked symbols representing the memory locations that may have been accessed at this point, together with a repetition count $R\colon V \to \mathcal{P}(\mathbb{N})$ that tracks the number of times each address has been accessed. During program analysis, we maintain and manipulate a single DAG, which is why we usually keep $t^\sharp$ implicit.

6.2 Concretization

Each vertex v in $t^\sharp$ corresponds to the set of traces of memory accesses performed by the program from the root up to this point of the analysis. This correspondence is formally given by a concretization function $\gamma_\lambda^{\mathcal{T}^\sharp}$ that is parameterized by an instantiation $\lambda\colon Sym \to \{0,1\}^n$ of the masked symbols occurring in the labels (see Section 5), and is defined by:

$$\gamma_\lambda^{\mathcal{T}^\sharp}(v) = \bigcup_{v_0 \cdots v_k} \{a_0^{r_0} \cdots a_k^{r_k} \mid a_i \in \gamma_\lambda^{\mathcal{M}^\sharp}(L(v_i)), r_i \in R(v_i)\},$$

where $v_0 \cdots v_k$, with $v_0 = r$ and $v_k = v$, ranges over all paths from r to v in $t^\sharp$. That is, for each such path, the concretization contains the adversary's observations (given by the concretizations of the labels of its vertices) and their number (given by the repetition count).

6.3 Counting

We devise a counting procedure that over-approximates the number of observations different adversaries can make. The key feature of the procedure is that the bounds it delivers are *independent* of the instantiation of the symbols.

$$cnt^\pi(v) = |R(v)| \cdot |\pi(L(v))| \cdot \sum_{(u,v) \in E} cnt^\pi(u) \,, \qquad (3)$$

with $cnt^\pi(r) = 1$. For the stuttering observers, we replace the factor $|R(v)|$ from the expression in (3) by 1, which captures that those observers cannot distinguish between repetitions of accesses to the same unit.

Proposition 2. *For all* $\lambda\colon Sym \to \{0,1\}^n$ *we have*

$$|view(\gamma_\lambda^{\mathcal{T}^\sharp}(v))| \leq cnt^\pi(v)$$

Proof. $cnt^\pi(v)$ recursively sums over all paths from r to v and weights each vertex with the size of π applied to its label. From Proposition 1 it follows that this size is larger than the the number of concrete observations, regardless of how the symbols are instantiated. This yields the assertion. □

6.4 Update and Join

The memory trace domain is equipped with functions for update and join, which govern how sets of traces are extended and merged, respectively.

The *update* of an abstract element $t^\sharp$ receives a vertex v representing a set of traces of memory accesses, and it extends v by a new access to a potentially unknown address, represented by a set of masked symbols $x^\sharp \in \mathcal{M}^\sharp$. Technically:

1. If the set of masked symbols is not a repetition (i.e. if $L(v) \neq x^\sharp$), the update function appends a new vertex v' to v (adding (v, v') to E), and it sets $L(v') = x^\sharp$ and $R(v') = \{1\}$.

2. Otherwise (i.e. if $L(v) = x^\sharp$), it increments the possible repetitions in $R(v)$ by one.

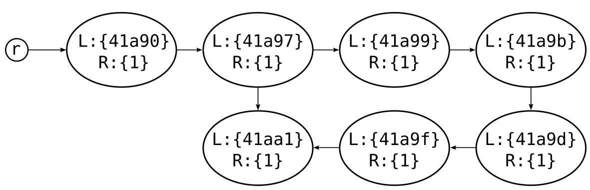
The *join* for two vertices v_1, v_2 first checks whether those vertices have the same parents and the same label, in which case v_1 is returned, and their repetitions are joined. Otherwise, a new vertex v' with $L(v') = \{\epsilon\}$ is generated and edges (v_1, v') and (v_2, v') are added to E.

Implementation Issues To increase precision in counting and compactness of the representation, we apply the projection π when applying the update function, and we delay joins until the next update is performed. In this way we only maintain the information that is relevant for the final counting step and can encode accesses to the same block as stuttering. This is relevant, for example, when an if-statement fits into a single memory block.

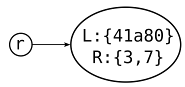
Example 9. *Consider the following snippet of x86 assembly, corresponding to a conditional branch in libgcrypt 1.5.3:*

```
41a90: mov 0x80(%esp),%eax
41a97: test %eax,%eax
41a99: jne 41aa1
41a9b: mov %ebp,%eax
41a9d: mov %edi,%ebp
41a9f: mov %eax,%edi
41aa1: sub $0x1,%edx
```

Figure 4 shows the corresponding DAGs for an address-trace observer (Figure 4a) and for a block-trace observer (Figure 4b) of the instruction cache. For both observers, the counting procedure reports two traces, i.e., a leak of 1 *bit. For the stuttering block-trace observer, however, the counting procedure determines that there is only one possible observation.*



(a) Address-trace observer.



(b) Block-trace observer.

Figure 4: DAGs that represent the traces corresponding to the assembly code in Example 9, for an architecture with cache lines of 64 bytes.

7. Soundness

In this section we establish the correctness of our approach. We split the argument in two parts. The first part explains how we establish leakage bounds w.r.t. *all* valuations of low variables. The second part explains the correctness of the abstract domains introduced in Section 5 and Section 6.

7.1 Global Soundness and Leakage Bounds

We formalize the correctness of our approach based on established notions of local and global soundness [14], which we slightly extend to cater for the introduction of fresh symbols during analysis.

For this, we distinguish between the symbols in $Sym_{lo} \subseteq Sym$ that represent the low initial state and those in $Sym \setminus Sym_{lo}$ that are introduced during the analysis. A low initial state in I_{lo} is given in terms of a valuation of low symbols $\lambda \colon Sym_{lo} \to \{0,1\}^n$. When introducing a fresh symbol s (see Section 5), we need to extend the domain of λ to include s. We denote by $Ext(\lambda)$ the set of all functions $\bar{\lambda}$ with $\bar{\lambda}\restriction_{dom(\lambda)} = \lambda$, $dom(\bar{\lambda}) \subseteq Sym$, and $ran(\bar{\lambda}) = \{0,1\}^n$.

With this, we formally define the *global soundness* of the fixpoint $Col^\sharp$ of the abstract transition function $next^\sharp$ as follows:

$$\forall \lambda \in I_{lo}\, \exists \bar{\lambda} \in Ext(\lambda) : Col_\lambda \subseteq \gamma_{\bar{\lambda}}\left(Col^\sharp\right) . \qquad (4)$$

Equation (4) ensures that for all low initial states λ, every trace of the program is included in a concretization of the symbolic fixpoint, for *some* valuation $\bar{\lambda}$ of the symbols that have been introduced during analysis.

The *existence* of $\bar{\lambda}$ is sufficient to prove our central result, which is a bound for the leakage w.r.t. all low initial states.

Theorem 1. *Let $t^\sharp \in \mathcal{T}^\sharp$ be the component in $Col^\sharp$ representing memory traces, and $v \in t^\sharp$ correspond to the final state. Then*

$$\mathcal{L} = \max_{\lambda \in I_{lo}} |view(Col_\lambda)| \leq cnt^\pi(v)$$

The statement follows because set inclusion of the fixpoints implies set inclusion of the projection to memory traces: $view(Col_\lambda) \subseteq view(\gamma_{\bar{\lambda}}(Col^\sharp))$. The memory trace of the abstract fixpoint is given by $view(\gamma_{\bar{\lambda}}^{\mathcal{T}^\sharp}(v))$, and Proposition 2 shows that $cnt^\pi(v)$ delivers an upper bound, for any $\bar{\lambda}$.

7.2 Local Soundness

We say that an abstract domain is *locally sound* if the abstract $next^\sharp$ operator over-approximates the effect of the concrete $next$ operator (here: in terms of set inclusion). Formally we require that, for all abstract elements $a^\sharp$,

$$\forall \lambda, \exists \bar{\lambda} \in Ext(\lambda) : next\left(\gamma_\lambda(a^\sharp)\right) \subseteq \gamma_{\bar{\lambda}}(next^\sharp(a^\sharp)) . \quad (5)$$

From Local to Global Soundness It is a fundamental result from abstract interpretation [14] that local soundness (5) implies global soundness (4). When the fixpoint is reached in a finite number of steps, this result immediately follows for our modified notions of soundness, by induction on the number of steps. This is sufficient for the development in our paper; we leave an investigation of the general case to future work

Local Soundness in CacheAudit It remains to show the local soundness of abstract transfer function $next^\sharp$. In our case, this function is inherited from the CacheAudit static analyzer [16], and it is composed of several independent and locally sound abstract domains. For details on how these domains are wired together, refer to [16].

Here, we focus on proving local soundness conditions for the two components we developed in this paper, namely the masked symbol and the memory trace domains.

Masked Symbol Domain The following lemma states the local soundness of the Masked Symbol Domain, following (5):

Lemma 1 (Local Soundness of Masked Symbol Domain)**.** *For all operands $\mathit{OP} \in \{\mathit{AND}, \mathit{OR}, \mathit{XOR}, \mathit{ADD}, \mathit{SUB}\}$, we have*

$$\forall x_1^\sharp, x_2^\sharp, \lambda, \exists \bar{\lambda} \in Ext(\lambda) :$$
$$\mathit{OP}(\gamma_\lambda^{\mathcal{M}^\sharp}(x_1^\sharp), \gamma_\lambda^{\mathcal{M}^\sharp}(x_2^\sharp)) \subseteq \gamma_{\bar{\lambda}}^{\mathcal{M}^\sharp}(\mathit{OP}^\sharp(x_1^\sharp, x_2^\sharp))$$

Proof. For the proof, we consider two cases:

- the operation preserves the symbol. Then the abstract update coincides with the concrete update, with $\bar{\lambda} = \lambda$ and $next\left(\gamma_{\lambda}^{\mathcal{M}^\sharp}(a^\sharp)\right) = \gamma_{\lambda}^{\mathcal{M}^\sharp}(next^\sharp(a^\sharp))$. This is because the abstract operations are defined such that the symbol is only preserved when we can guarantee that the operation acts neutral on all symbolic bits.
- the operation introduces a fresh symbol s''. Then we simply define $\bar{\lambda}(s'')$ such that $\bar{\lambda}(s'') \odot m'' = \mathtt{OP}^\sharp(\lambda(s) \odot m, \bar{\lambda}(s') \odot m')$. This is possible because the concrete bits in m'' coincide with the operation, and the symbolic bits can be set as required by $\bar{\lambda}$.

Flag values are correctly approximated as all flag-value combinations are considered as possible unless the values can be exactly determined. □

Memory Trace Domain The following lemma states the soundness of the memory trace domain:

Lemma 2 (Local Soundness of Memory Trace Domain)**.**

$$\forall \lambda, \exists \bar{\lambda} \in Ext(\lambda):$$
$$upd\left(\gamma_{\lambda}^{\mathcal{T}^\sharp}(t^\sharp), \gamma_{\bar{\lambda}}^{\mathcal{M}^\sharp}(x^\sharp)\right) \subseteq \gamma_{\bar{\lambda}}^{\mathcal{T}^\sharp}(upd^\sharp(t^\sharp, x^\sharp))$$

Proof. The local soundness of the memory trace domains follows directly because the update does not perform any abstraction with respect to the sets of masked symbols it appends; it just yields a more compact representation in case of repetitions of the same observation. □

8. Case Study

This section presents a case study, which leverages the techniques developed in this paper for the first rigorous analysis of a number of practical countermeasures against cache side channel attacks on modular exponentiation algorithms. The countermeasures are from releases of the cryptographic libraries libgcrypt and OpenSSL from April 2013 to March 2016. We report on results for bits of leakage to the adversary models presented in Section 3.2 (i.e., the logarithm of the maximum number of observations the adversaries can make, see Section 4), due to instruction-cache (I-cache) accesses and data-cache (D-cache) accesses.[5] As the adversary models are ordered according to their observational capabilities, this sheds light into the level of provable security offered by different protections.

8.1 Analysis Tool

We implement the novel abstract domains described in Sections 5 and 6 on top of the CacheAudit open source static analyzer [16]. CacheAudit provides infrastructure for parsing, control-flow reconstruction, and fixed point computation. Our novel domains extend the scope of CacheAudit by providing support for (1) the analysis of dynamically allocated memory, and for (2) adversaries who can make fine-grained observations about memory accesses. The source code is publicly available[6]. For all considered instances, our analysis takes between 0 and 4 seconds on a t1.micro virtual machine instance on Amazon EC2.

8.2 Target Implementations

The target of our experiments are different side-channel countermeasures for modular exponentiation, which we analyse at x86 binary level. Our testbed consists of C-implementations of ElGamal decryption [18] with 3072-bit keys, using 6 different implementations of modular exponentiation, which we compile using `gcc 4.8.4`, on a 32-bit Linux machine.

We use the ElGamal implementation from the libgcrypt 1.6.3 library, in which we replace the source code for modular exponentiation (`mpi-pow.c`) with implementations containing countermeasures from different versions of libgcrypt and OpenSSL. For libgcrypt, we consider versions 1.5.2 and 1.5.3, which implement square-and-multiply modular exponentiation, as well as versions 1.6.1 and 1.6.3, which implement sliding-window modular exponentiation. Versions 1.5.2 and 1.6.1 do not implement a dedicated countermeasure against cache attacks. For OpenSSL, we consider versions 1.0.2f and 1.0.2g, which implement fixed-window modular exponentiation with two different countermeasures against cache attacks. We integrate the countermeasures into the libgcrypt 1.6.3-implementation of modular exponentiation.

The current version of CacheAudit supports only a subset of the x86 instruction set, which we extended with instructions required for this case study. To bound the required extensions, we focus our analysis on the regions of the executables that were targeted by exploits and to which the corresponding countermeasures were applied, rather than the whole executables. As a consequence, the formal statements we derive only hold for those regions. In particular, we do not analyze the code of the libgcrypt's multi-precision integer multiplication and modulo routines, and we specify that the output of the memory allocation functions (e.g. `malloc`) is symbolic (see Section 5).

8.3 Square-and-Multiply Modular Exponentiation

The first target of our analysis is modular exponentiation by square-and-multiply [35]. The algorithm is depicted in Figure 5 and is implemented, e.g., in libgcrypt version 1.5.2. Line 5 of the algorithm contains a conditional branch whose condition depends on a bit of the secret exponent. An attacker who can observe the victim's accesses to instruction or data caches may learn which branch was taken and identify the value of the exponent bit. This weakness has been

[5] We also analyzed the leakage from accesses to shared instruction- and data-caches; for the analyzed instances, the leakage results were consistently the maximum of the I-cache and D-cache leakage results.

[6] `http://software.imdea.org/cacheaudit/memory-trace`

shown to be vulnerable to key-recovery attacks based on prime+probe [32, 49] and flush+reload [47].

In response to these attacks, libgcrypt 1.5.3 implements a countermeasure that makes sure that the squaring operation is always performed, see Figure 6 for the pseudocode. It is noticeable that this implementation still contains a conditional branch that depends on the bits of the exponent in Lines 7–8, namely the copy operation that selects the outcome of both multiplication operations. However, this has been considered a minor problem because the branch is small and is expected to fit into the same cache line as preceding and following code, or to be always loaded in cache due to speculative execution [47]. In the following, we apply the techniques developed in this paper to analyze whether the expectations on memory layout are met.[7]

```
1  r := 1
2  for i := |e| − 1 downto 0 do
3      r := mpi sqr(r)
4      r := mpi mod(r, m)
5      if e_i = 1 then
6          r := mpi_mul(b, r)
7          r := mpi_mod(r, m)
8      return r
```

Figure 5: Square-and-multiply modular exponentiation

```
1  r := 1
2  for i := |e| − 1 downto 0 do
3      r := mpi_sqr(r)
4      r := mpi_mod(r, m)
5      tmp := mpi_mul(b, r)
6      tmp := mpi_mod(tmp , m)
7      if e_i = 1 then
8          r := tmp
9      return r
```

Figure 6: Square-and-always-multiply exponentiation

Observer	*address*	*block*	*b-block*
I-Cache	1 bit	1 bit	1 bit
D-Cache	1 bit	1 bit	1 bit

(a) Square-and-multiply from libgcrypt 1.5.2

Observer	*address*	*block*	*b-block*
I-Cache	1 bit	1 bit	0 bit
D-Cache	0 bit	0 bit	0 bit

(b) Square-and-*always*-multiply from libgcrypt 1.5.3

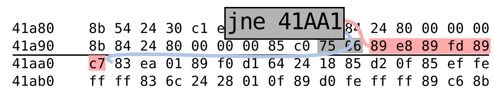
Figure 7: Leakage of modular exponentiation algorithms to observers of instruction and data caches, with cache line size of 64 bytes and compiler optimization level `-O2`.

Results The results of our analysis are given in Figure 7 and Figure 8.

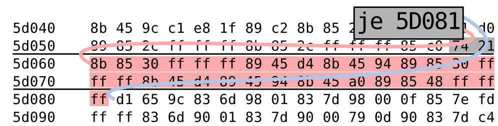
[7] Note that we analyze the branch in Lines 7–8 for one iteration; in the following iteration the adversary may learn the information by analyzing which memory address is accessed in Line 3 and 4.

Observer	*address*	*block*	*b-block*
I-Cache	1 bit	1 bit	1 bit
D-Cache	1 bit	1 bit	1 bit

Figure 8: Leakage of square-and-always-multiply from libgcrypt 1.5.3, with cache line size of 32 bytes and compiler optimization level `-O0`.



(a) Compiled with the default gcc optimization level -O2. Regardless whether the jump is taken or not, first block `41a80` is accessed, followed by block `41aa0`. This results in a 0-bit *b-block*-leak.



(b) Compiled with gcc optimization level -O0. The memory block `5d060` is only accessed when the jump is taken. This results in a 1-bit *b-block*-leak.

Figure 9: Layout of libgcrypt 1.5.3 executables with 32-byte memory blocks (black lines denote block boundaries). The highlighted code corresponds to the conditional branching in lines 7–8 in Figure 6. The red region corresponds to the executed instructions in the if-branch. The blue curve points to the jump target, where the jump is taken if the if-condition does not hold.

- Our analysis identifies a 1-bit data cache leak in square-and-multiply exponentiation (line 2 in Figure 7a), due to memory accesses in the conditional branch in that implementation. Our analysis confirms that this data cache leak is closed by square-and-always-multiply (line 2 in Figure 7b).
- Line 1 of Figures 7a and Figure 7b show that both implementations leak through instruction cache to powerful adversaries who can see each access to the instruction cache. However, for weaker, stuttering block-trace (*b-block*) observers that cannot distinguish between repeated accesses to a block, square-and-always-multiply does *not* leak, confirming the intuition that the conditional copy operation is indeed less problematic than the conditional multiplication.
- The comparison between Figure 7b and Figure 8 demonstrates that the effectiveness of countermeasures can depend on details such as cache line size and compilation strategy. This is illustrated in Figure 9, which shows that more aggressive compilation leads to more compact code that fits into single cache lines. The same effect is observable for data caches, where more aggressive compilation avoids data cache accesses altogether.

```
if e0 == 0 then
    base_u := bp
    base_u_size := bsize
else
    base_u := b_2i3[e0 − 1]
    base_u_size := b_2i3size[e0 − 1]
```

Figure 10: Table lookup from libgcrypt 1.6.1. Variable `e0` represents the window, right-shifted by 1. The lookup returns a pointer to the first limb of the multi-precision integer in `base_u`, and the number of limbs in `base_u_size`. The first branch deals with powers of 1 by returning pointers to the base.

8.4 Windowed Modular Exponentiation

In this section we analyze windowed algorithms for modular exponentiation [35]. These algorithms differ from algorithms based on square-and-multiply in that they process multiple exponent bits in one shot. For this they commonly rely on tables filled with pre-computed powers of the base. For example, for moduli of 3072 bits, libgcrypt 1.6.1 pre-computes 7 multi-precision integers and handles the power 1 in a branch, see Figure 10. Each pre-computed value requires 384 bytes of storage, which amounts to 6 or 7 memory blocks in architectures with cache lines of 64 bytes. Key-dependent accesses to those tables can be exploited for mounting cache side channel attacks [32].

We consider three countermeasures, which are commonly deployed to defend against this vulnerability. They have in common that they all *copy* the table entries instead of returning a pointer to the entry.

```
1  // Retrieves r from p[k]
2  secure_retrieve ( r , p , k):
3      for i := 0 to n − 1 do
4          for j := 0 to N − 1 do
5              v := p[i][j]
6              s := (i == k)
7              r[j] := r[j] ^ ((0 − s) & ( r[j] ^ v))
```

Figure 11: A defensive routine for array lookup with a constant sequence of memory accesses, as implemented in libgcrypt 1.6.3.

- The first countermeasure ensures that in the copy process, a constant sequence of memory locations is accessed, see Figure 11 for pseudocode. The expression on line 7 ensures that only the k-th pre-computed value is actually copied to `r`. This countermeasure is implemented, e.g. in NaCl and libgcrypt 1.6.3.
- The second countermeasure stores pre-computed values in such a way that the i-th byte of all pre-computed values resides in the same memory block. This ensures that when the pre-computed values are retrieved, a constant sequence of memory blocks will be accessed. This so-called scatter/gather technique is described in detail in Section 2, with code in Figure 3, and is deployed, e.g. in OpenSSL 1.0.2f.
- The third countermeasure is a variation of scatter/-gather, and ensures that the gather-procedure performs a constant sequence of memory accesses (see Figure 12). This countermeasure was recently introduced in OpenSSL 1.0.2g, as a response to the CacheBleed attack, where the adversary can use *cache-bank conflicts* to make finer-grained observations and recover the pre-computed values despite scatter/-gather. For example, the pre-computed values in Figure 2 will be distributed to different cache banks as shown in Figure 13, and cache-bank adversaries can distinguish between accesses to $p_0, \ldots, p_3$ and $p_4, \ldots, p_7$.

```
defensive_gather( r, buf, k ):
    for i := 0 to N−1 do
        r[i] := 0
        for j:= 0 to spacing − 1 do
            v := buf[j + i∗spacing]
            s := (k == j)
            r[i] := r[i] | (v & (0 − s))
```

Figure 12: A defensive implementation of gather (compare to Figure 3) from OpenSSL 1.0.2g.

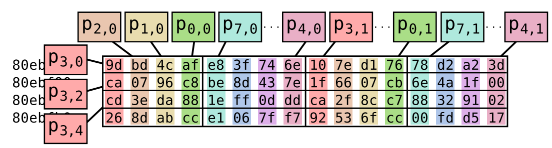


Figure 13: Layout of pre-computed values in cache banks, for a platform with 16 banks of 4 bytes. The shown data fits into one memory block, and the cells of the grid represent the cache banks.

Results Our analysis of the different versions of the table lookup yields the following results[8]:

- Figure 14a shows the results of the analysis of the unprotected table lookup in Figure 10. The leakage of one bit for most adversaries is explained by the fact that they can observe which branch is taken. The layout of the conditional branch is demonstrated in Figure 15a; lowering the optimization level results in a different layout (see Figure 15b), and in this case our analysis shows that the I-Cache *b-block*-leak is eliminated.
- Figure 14a also shows that more powerful adversaries that can see the exact address can learn $\log_2 7 = 2.8$ bits per access. The static analysis is not precise enough to determine that the lookups are correlated, hence it reports that at most 5.6 bits are leaked.

[8] We note that sliding-window exponentiation exhibits further control-flow vulnerabilities, some of which we also analyze. To avoid redundancy with Section 8.3, we focus the presentation of our results on the lookup-table management.

Observer	*address*	*block*	*b-block*
I-Cache	1 bit	1 bit	1 bit
D-Cache	5.6 bit	2.3 bit	2.3 bit

(a) Leakage of secret-dependent table lookup in the modular exponentiation implementation from libgcrypt 1.6.1.

Observer	*address*	*block*	*b-block*
I-Cache	0 bit	0 bit	0 bit
D-Cache	0 bit	0 bit	0 bit

(b) Leakage in the patch from libgcrypt 1.6.3.

Observer	*address*	*block*	*b-block*
I-Cache	0 bit	0 bit	0 bit
D-Cache	1152 bit	0 bit	0 bit

(c) Leakage in the scatter/gather technique, applied to libgcrypt 1.6.1.

Observer	*address*	*block*	*b-block*
I-Cache	0 bit	0 bit	0 bit
D-Cache	0 bit	0 bit	0 bit

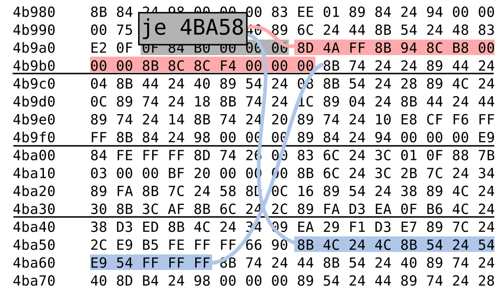
(d) Leakage in the defensive gather technique from OpenSSL 1.0.2g, applied to libgcrypt 1.6.1.

Figure 14: Instruction and data cache leaks of different table lookup implementations. Note that the leakage in Figure 14a accounts for copying a pointer, whereas the leakage in Figure 14b and 14c refers to copying multi-precision integers.

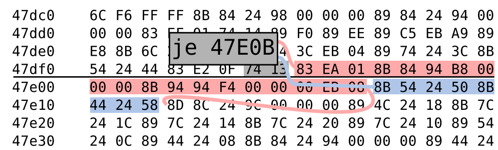
- Figure 14b shows that the defensive copying strategy from libgcrypt 1.6.3 (see Figure 11) eliminates all leakage to the cache.
- Figure 14c shows that the scatter/gather copying-strategy eliminates leakage for any adversary that can observe memory accesses at the granularity of memory blocks, and this constitutes the first proof of security of this countermeasure. For adversaries that can see the full address-trace, our analysis reports a 3-bit leakage for each memory access, which is again accumulated over correlated lookups because of imprecisions in the static analysis.
- Our analysis is able to detect the leak leading to the CacheBleed attack [48] against scatter/gather. The leak is visible when comparing the results of the analysis in Figure 14c with respect to address-trace and block-trace adversaries, however, its severity may be over-estimated due to the powerful address-trace observer. For a more accurate analysis of this threat, we repeat the analysis for the bank-trace D-cache observer. The analysis results in 384-bit leak, which corresponds to one bit leak per memory access, accumulated for each accessed byte due to analysis imprecision (see above). The one-bit leak in the i-th memory access is explained by the ability of this observer to distinguish between the two banks within which the i-th byte of all pre-computed values fall.
- Figure 14d shows that defensive gather from OpenSSL 1.0.2g (see Figure 12) eliminates all leakage to cache.

8.5 Discussion

A number of comments are in order when interpreting the bounds delivered by our analysis.



(a) Compiled with the default gcc optimization level -O2. If the jump is taken, first block `4b980`, followed by block `4ba40`, followed by `4b980` again. If the branch is not taken, only block `4b980` is accessed.



(b) Compiled with gcc optimization level -O1. Regardless whether the jump is taken or not, first block `47dc0` is accessed, followed by block `47e00`.

Figure 15: Layout of executables using libgcrypt 1.6.1. The highlighted code corresponds to a conditional branch (blue: if-branch, red: else-branch). Curves represent jump targets.

Use of Upper Bounds The results we obtain are upper bounds on the leaked information. Results of zero leakage present a proof of the absence of leaks. Positive leakage bounds, however, are not necessarily tight and do not correspond to proofs of the presence of leaks. The reason for this is that the amount of leaked information may be over-estimated due to imprecision of the static analysis, as is the case with the D-Cache leak shown on Figure 14c.

Use of Imperfect Models The guarantees we deliver are only valid to the extent to which the models used accurately capture the aspects of the execution platform relevant to known attacks. A recent empirical study of OS-level side channels on different platforms [11] shows that advanced microarchitectural features may interfere with the cache, which can render countermeasures ineffective.

Alternative Attack Targets In our analysis, we assume that heap addresses returned by malloc are low values. For analyzing scenarios in which the heap addresses themselves are the target of cache attacks (e.g., when the goal is to reduce the entropy of ASLR [25]), heap addresses would need to be modeled as high data.

8.6 Performance of Countermeasures

We conclude the case study by considering the effect of the different countermeasures on the performance of modular exponentiation. For the target implementations (see Section 8.2), we measure performance as the clock count

algorithm	square and multiply		sliding window			
countermeasure (CM)	no CM	always multiply	no CM	scatter/ gather	access all bytes	defensive gather
implementation	libgcrypt 1.5.2	libgcrypt 1.5.3	libgcrypt 1.6.1	openssl 1.0.2f	libgcrypt 1.6.3	openssl 1.0.2g
instructions $\times 10^6$	90.32	120.62	73.99	74.21	74.61	75.29
cycles $\times 10^6$	75.58	100.73	61.58	61.65	62.20	62.28

(a) Different versions of modular exponentiation.

algorithm	sliding window		
countermeasure	scatter/ gather	access all bytes	defensive gather
implementation	openssl 1.0.2f	libgcrypt 1.6.3	openssl 1.0.2g
instructions	2991	8618	13040
cycles	859	3073	5579

(b) Only multi-precision-integer retrieval step.

Figure 16: Performance measurements for libgcrypt 1.6.3 ElGamal decryption, for 3072-bit keys.

(through the `rdtsc` instruction), as well as the number of performed instructions (through the `PAPI` library [37]), for performing exponentiations, for a sample of random bases and exponents. We make 100,000 measurements with an Intel Q9550 CPU.

Figure 16a summarizes our measurements. The results show that the applied countermeasure for square and multiply causes a significant slow-down of the exponentiation. A smaller slow-down is observed with sliding-window countermeasures as well; this slow-down is demonstrated in Figure 16b, which shows the performance of the retrieval of pre-computed values, with different countermeasures applied.

9. Related Work

We begin by discussing approaches that tackle related goals, before we discuss approaches that rely on similar techniques.

Transforming out Timing Leaks Agat proposes a program transformation for removing control-flow timing leaks by equalizing branches of conditionals with secret guards [2], which he complements with an informal discussion of the effect of instruction and data caches in Java byte code [3]. Molnar et al. [36] propose a program transformation that eliminates control-flow timing leaks, together with a static check for the resulting x86 executables. Coppens et al. [12] propose a similar transformation and evaluate its practicality. The definitions in Section 3 encompass the adversary model of [36], but also weaker ones; they could be used as a target for program transformations that allow for limited forms for secret-dependent behavior.

Constant-time Software Constant-time code defeats timing attacks by ensuring that control flow, memory accesses, and execution time of individual instructions do not depend on secret data. Constant-time code is the current gold standard for defensive implementations of cryptographic algorithms [9].

A number of program analyses support verifying constant-time implementations. Almeida et al. [5] develop an approach based on self-composition that checks absence of timing leaks in C-code; Almeida et al. [4] provide a tool chain for verifying constaint-time properties of LLVM IR code. Similar to the dynamic analysis by Langley [31], our approach targets executable code, thereby avoiding potential leaks introduced by the compiler [26]. Moreover, it supports verification of more permissive interactions between software and hardware – at the price of stronger assumptions about the underlying hardware platform.

Quantitative Information Flow Analysis Technically, our work draws on methods from quantitative information-flow analysis (QIF) [10], where the automation by reduction to counting problems appears in [7, 38], and has subsequently been refined in several dimensions [13, 24, 28, 30].

Specifically, our work builds on CacheAudit [16], a tool for the static quantification of cache side channels in x86 executables. The techniques developed in this paper extend CacheAudit with support for precise reasoning about dynamically allocated memory, and a rich set of novel adversary models. Together, this enables the first formal analysis of widely deployed countermeasures, such as scatter/gather.

Abstract Interpretation We rely on basic notions from abstract interpretation [14] for establishing the soundness of our analysis. However, the connections run deeper: For example, the observers we define (including the stuttering variants [21]) can be seen as abstractions in the classic sense, which enables composition of their views in algebraic ways [15]. Abstract interpretation has also been used for analyzing information flow properties [6, 20]. Reuse of the machinery developed in these papers could help streamline our reasoning. We leave a thorough exploration of this connection to future work.

10. Conclusions

In this paper we devise novel techniques that provide support for bit-level and arithmetic reasoning about pointers in the presence of dynamic memory allocation. These techniques enable us to perform the first rigorous analysis of widely deployed software countermeasures against cache side-channel attacks on modular exponentiation, based on executable code.

Acknowledgments

We thank Roberto Giacobazzi, Francesco Logozzo, Laurent Mauborgne, and the anonymous reviewers for their helpful feedback. This work was supported by Ramón y Cajal grant RYC-2014-16766, Spanish projects TIN2012-39391-C04-01 StrongSoft and TIN2015-70713-R DEDETIS, and Madrid regional project S2013/ICE-2731 N-GREENS.

References

[1] O. Aciiçmez, W. Schindler, and Ç. K. Koç. Cache based remote timing attack on the AES. In *CT-RSA*, pages 271–286. Springer, 2007.

[2] J. Agat. Transforming out timing leaks. In *POPL 2000*, pages 40–53. ACM, 2000.

[3] J. Agat. Transforming out timing leaks in practice: An experiment in implementing programming language-based methods for confidentiality, 2000.

[4] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. Verifying constant-time implementations. In *USENIX Security Symposium*. USENIX, 2016.

[5] J. B. Almeida, M. Barbosa, J. S. Pinto, and B. Vieira. Formal verification of side-channel countermeasures using self-composition. *Sci. Comput. Program.*, 78(7):796–812, 2013.

[6] M. Assaf, D. A. Naumann, J. Signoles, E. Totel, and F. Tronel. Hypercollecting semantics and its application to static analysis of information flow. In *POPL*. ACM, 2017.

[7] M. Backes, B. Köpf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *SSP*, pages 141–153. IEEE, 2009.

[8] D. Bernstein. Cache-timing attacks on AES. `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`, 2005.

[9] D. J. Bernstein, T. Lange, and P. Schwabe. The security impact of a new cryptographic library. In *LATINCRYPT*, pages 159–176. Springer, 2012.

[10] D. Clark, S. Hunt, and P. Malacaria. A static analysis for quantifying information flow in a simple imperative language. *JCS*, 15(3):321–371, 2007.

[11] D. Cock, Q. Ge, T. Murray, and G. Heiser. The last mile: An empirical study of timing channels on sel4. In *CCS*. ACM, 2014.

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

[13] P. M. Corina Pasareanu, Sang Phan. Multi-run side-channel analysis using symbolic execution and max-smt. In *CSF*. IEEE, 2016.

[14] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[15] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*. ACM Press, 1979.

[16] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke. Cacheaudit: A tool for the static analysis of cache side channels. *ACM Transactions on Information and System Security*, 18(1):4:1–4:32, June 2015.

[17] S. Dziembowski and K. Pietrzak. Leakage-resilient cryptography. In *FOCS*. IEEE, 2008.

[18] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.

[19] Ú. Erlingsson and M. Abadi. Operating system protection against side-channel attacks that exploit memory latency. Technical report, 2007.

[20] R. Giacobazzi and I. Mastroeni. Abstract Non-Interference: Parameterizing Non-Interference by Abstract Interpretation. pages 186–197. ACM, 2004.

[21] R. Giacobazzi and I. Mastroeni. Timed Abstract Non-Interference. In *Proc. 3rd International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS 2005)*, volume 3829 of *LNCS*, pages 289–303. Springer, 2005.

[22] S. Gueron. Intel Advanced Encryption Standard (AES) Instructions Set. `http://software.intel.com/file/24917`, 2010.

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

[24] J. Heusser and P. Malacaria. Quantifying information leaks in software. In *ACSAC*, pages 261–269. ACM, 2010.

[25] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space aslr. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 191–205. IEEE, 2013.

[26] T. Kaufmann, H. Pelletier, S. Vaudenay, and K. Villegas. When constant-time source yields variable-time binary: Exploiting curve25519-donna built with msvc 2015. In *International Conference on Cryptology and Network Security*, pages 573–582. Springer, 2016.

[27] T. Kim, M. Peinado, and G. Mainar-Ruiz. StealthMem: System-level protection against cache-based side channel attacks in the cloud. In *19th USENIX Security Symposium*. USENIX, 2012.

[28] V. Klebanov. Precise quantitative information flow analysis using symbolic model counting. In F. Martinelli and F. Nielson, editors, *Proceedings, International Workshop on Quantitative Aspects in Security Assurance (QASA)*, 2012.

[29] B. Köpf and D. Basin. An Information-Theoretic Model for Adaptive Side-Channel Attacks. In *CCS*, pages 286–296. ACM, 2007.

[30] B. Köpf and A. Rybalchenko. Approximation and randomization for quantitative information-flow analysis. In *CSF*, pages 3–14. IEEE, 2010.

[31] A. Langley. Checking that functions are constant time with valgrind. https://www.imperialviolet.org/2010/04/01/ctgrind.html, 2010. Accessed: 15 April 2017.

[32] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*, pages 605–622. IEEE Computer Society, 2015.

[33] G. Lowe. Quantifying Information Flow. In *Proc. 15th IEEE Computer Security Foundations Symposium (CSFW 2002)*, pages 18–31. IEEE, 2002.

[34] J. L. Massey. Guessing and Entropy. In *ISIT*, page 204. IEEE, 1994.

[35] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[36] 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*. Springer, 2006.

[37] P. J. Mucci, S. Browne, C. Deane, and G. Ho. Papi: A portable interface to hardware performance counters. In *DoD HPCMP Users Group Conference*, 1999.

[38] J. Newsome, S. McCamant, and D. Song. Measuring channel capacity to distinguish undue influence. In *PLAS*, pages 73–85. ACM, 2009.

[39] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In *CT-RSA*, volume 3860 of *LNCS*, pages 1–20. Springer, 2006.

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

[41] 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*, pages 199–212. ACM, 2009.

[42] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing page faults from telling your secrets. In *ASIACCS*. ACM, 2016.

[43] G. Smith. On the foundations of quantitative information flow. In *FoSSaCS*. Springer, 2009.

[44] M. Tiwari, J. Oberg, X. Li, J. Valamehr, T. E. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. In *ISCA*, pages 189–200. ACM, 2011.

[45] Z. Wang and R. B. Lee. A novel cache architecture with enhanced performance and security. In *MICRO*, 2008.

[46] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *SSP*. IEEE, 2015.

[47] Y. Yarom and K. Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium*, 2014.

[48] Y. Yarom, D. Genkin, and N. Heninger. CacheBleed: A timing attack on OpenSSL constant time RSA. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 346–367. Springer, 2016.

[49] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In *CCS*. ACM, 2012.

[50] Y. Zhang and M. K. Reiter. Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 827–838. ACM, 2013.