2015 IEEE Symposium on Security and Privacy

Last-Level Cache Side-Channel Attacks are Practical

Fangfei Liu*†, Yuval Yarom*‡§, Qian Ge§¶, Gernot Heiser§¶, Ruby B. Lee†
* Equal contribution joint first authors.
† Department of Electrical Engineering, Princeton University
Email: {fangfeil,rblee}@princeton.edu
‡ School of Computer Science, The University of Adelaide
Email: yval@cs.adelaide.edu.au
§ NICTA
Email: {qian.ge,gernot}@nicta.com.au
¶ UNSW Australia

***Abstract*—We present an effective implementation of the PRIME+PROBE side-channel attack against the last-level cache. We measure the capacity of the covert channel the attack creates and demonstrate a cross-core, cross-VM attack on multiple versions of GnuPG. Our technique achieves a high attack resolution without relying on weaknesses in the OS or virtual machine monitor or on sharing memory between attacker and victim.**

***Keywords*—Side-channel attack; cross-VM side channel; covert channel; last-level cache; ElGamal;**

I. INTRODUCTION

Infrastructure-as-a-service (IaaS) cloud-computing services provide virtualized system resources to end users, supporting each tenant in a separate virtual machine (VM). Fundamental to the economy of clouds is high resource utilization achieved by sharing: providers co-host multiple VMs on a single hardware platform, relying on the underlying virtual-machine monitor (VMM) to isolate VMs and schedule system resources.

While virtualization creates the illusion of strict isolation and exclusive resource access, in reality the virtual resources map to shared physical resources, creating the potential of interference between co-hosted VMs. A malicious VM may learn information on data processed by a victim VM [32, 42, 43] and even conduct side-channel attacks on cryptographic implementations [45, 47].

Previously demonstrated side channels with a resolution sufficient for cryptanalysis attacked the L1 cache. However, as Figure 1 shows, the L1 Data and Instruction caches (denoted L1 D$ and L1 I$) are private to each processor core. This limits the practicability of such attacks, as VMMs are not very likely to co-locate multiple owners' VMs on the same core. In contrast, the last-level cache (LLC) is typically shared between all cores of a package, and thus constitutes a much more realistic attack vector.

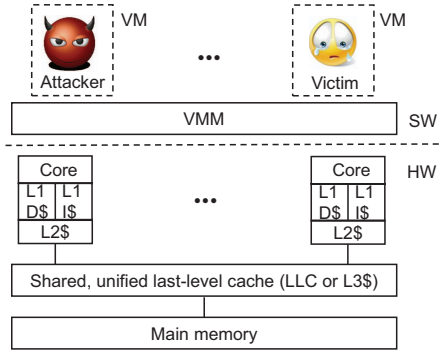


Fig. 1: System model for a multi-core processor

However, the LLC is orders of magnitude larger and much slower to access than the L1 caches, which drastically reduces the temporal resolution of observable events and thus channel bandwidth, making most published LLC attacks unsuitable for cryptanalysis [32, 42, 43]. An exception is the FLUSH+RELOAD attack [22, 45], which relies on memory sharing to achieve high resolution. Virtualization vendors explicitly advise against sharing memory between VMs [39], and no IaaS provider is known to ignore this advice [36], so this attack also fails in practice.

We show that an adaptation of the PRIME+PROBE technique [28] can be used for practical LLC attacks. We exploit hardware features that are outside the control of the cloud provider (inclusive caches) or are controllable but generally enabled in the VMM for performance reasons (large page mappings). Beyond that, we make no assumptions on the hosting environment, other than that the attacker and victim will be co-hosted on

DOI 10.1109/SP.2015.43

the same processor package.

Specifically, we make the following contributions:

- We demonstrate an asynchronous PRIME+PROBE attack on the LLC that does not require sharing cores or memory between attacker and victim, does not exploit VMM weaknesses and works on typical server platforms, even with the unknown LLC hashing schemes in recent Intel processors (Section IV);
- We develop two key techniques to enable efficient LLC based PRIME+PROBE attacks: an algorithm for the attacker to probe exactly one cache set without knowing the virtual-address mapping, and using temporal access patterns instead of conventional spatial access patterns to identify the victim's security-critical accesses (Section IV);
- We measure the achievable bandwidth of the cross-VM covert timing channel to be as high as 1.2 Mb/s (Section V);
- We show a cross-VM side-channel attack that extracts a key from secret-dependent execution paths, and demonstrate it on Square-and-Multiply modular exponentiation in an ElGamal decryption implementation (Section VI);
- We furthermore show that the attack can also be used on secret-dependent data access patterns, and demonstrate it on the sliding-window modular exponentiation implementation of ElGamal in the latest GnuPG version (Section VII).

II. BACKGROUND

A. Virtual address space and large pages

A process executes in its private virtual address space, composed of *pages*, each representing a contiguous range of addresses. The typical page size is 4 KiB, although processors also support *larger pages*, 2 MiB and 1 GiB on the ubiquitous 64-bit x86 ("x64") processor architecture. Each page is mapped to an arbitrary *frame* in physical memory.

In virtualized environments there are two levels of address-space virtualization. The first maps the virtual addresses of a process to a guest's notion of physical addresses, i.e., the VM's *emulated* physical memory. The second maps guest physical addresses to physical addresses of the processor. For our purposes, the guest physical addresses are irrelevant, and we use *virtual address* for the (guest virtual) addresses used by application processes, and *physical address* to refer to actual (host) physical addresses.

Translations from virtual pages to physical frames are stored in *page tables*. Processors cache recently used page table entries in the *translation look-aside buffer* (TLB). The TLB is a scarce processor resource with a small number of entries. Large pages use the TLB more efficiently, since fewer entries are needed to map a particular region of memory. As a result, the performance of applications with large memory footprints, such as Oracle databases or high-performance computing applications, can benefit from using large pages. For the same reason, VMMs, such as VMware ESX and Xen HVM, also use large pages for mapping guest physical memory [38].

B. System model and cache architecture

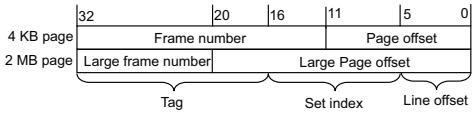
Cloud servers typically have multi-core processors, i.e., multiple processor cores on a chip sharing a last-level cache (LLC) and memory, as indicated in Figure 1.

1) Cache hierarchy: Because of the long access time of main memory compared to fast processors, smaller but faster memories, called caches, are used to reduce the effective memory access time as seen by a processor. Modern processors feature a hierarchy of caches. "Higher-level" caches, which are closer to the processor core are smaller but faster than lower-level caches, which are closer to main memory. Each core typically has two private top-level caches, one each for data and instructions, called level-1 (L1) caches. A typical L1 cache size is 32 KiB with a 4-cycle access time, as in Intel Core and Xeon families.

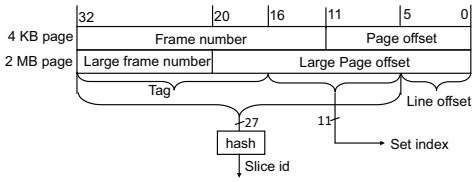
The LLC is shared among all the cores of a multi-core chip and is a *unified* cache, i.e., it holds both data and instructions. LLC sizes measure in megabytes, and access latencies are of the order of 40 cycles. Modern x86 processors typically also support core-private, unified L2 caches of intermediate size and latency. Any memory access first accesses the L1 cache, and on a miss, the request is sent down the hierarchy until it hits in a cache or accesses main memory. The L1 is typically indexed by virtual address, while all other caches are indexed by physical address.

2) Cache access: To exploit spatial locality, caches are organized in fixed-size *lines*, which are the units of allocation and transfer down the cache hierarchy. A typical line size B is 64 bytes. The $log_2 B$ lowest-order bits of the address, called the *line offset*, are used to locate a datum in the cache line.

Caches today are usually set-associative, i.e., organized as S sets of W lines each, called a *W-way set-associative* cache. As shown in Figure 2a, when the cache is accessed, the *set index* field of the address, $log_2 S$ consecutive bits starting from bit $log_2 B$, is used to locate a *cache set*. The remaining high-order bits are



(a) Traditional cache



(b) Sliced cache

Fig. 2: Cache indexing for an 8 GiB address space. A cache (or slice) contains 2,048 sets with 64 B lines.

used as a *tag* for each cache line. After locating the cache set, the tag field of the address is matched against the tag of the W lines in the set to identify if one of the cache lines is a cache hit.

As memory is much larger than the cache, more than W memory lines may map to the same cache set, potentially resulting in cache contention. If an access misses in the cache and all lines of the matching set are in use, one cache line must be evicted to free a cache slot for the new cache line being fetched from the next level of cache or from main memory for the LLC. The cache's *replacement policy* determines the line to evict. Typical replacement policies are approximations to least-recently-used (LRU).

There is often a well-defined relationship between different levels of cache. The *inclusiveness* property of a cache states that the L_{i+1} cache holds a strict superset of the contents of the L_i. The LLC in modern Intel processors is inclusive [20].

Modern Intel processors, starting with the Sandy Bridge microarchitecture, use a more complex architecture for the LLC, to improve its performance. The LLC is divided into per-core *slices*, which are connected by a ring bus (see Figure 3). Slices can be accessed concurrently and are effectively separate caches, although the bus ensures that each core can access the full LLC (with higher latency for remote slices).

To uniformly distribute memory traffic to the slices, Intel uses a carefully-designed but undocumented hash function (see Figure 2b). It maps the address (excluding the line offset bits) into the *slice id*. Within a slice, the set is accessed as in a traditional cache, so a cache set in the LLC is uniquely identified by the slice id and set index.

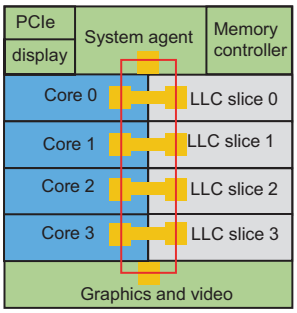


Fig. 3: Ring bus architecture and sliced LLC

Hund et al. [19] found that on Sandy Bridge, only the tag field is used to compute the hash, but we find that this is only true if the number of cores is a power of two. For other core counts, the full address (minus line offset) is used.

III. Challenges in attacking the LLC

A. Attack model

We target information leakage in virtualized environments, such as IaaS clouds. We assume that the attacker controls a VM that is co-resident with the victim VM on the same multi-core processor, as shown in Figure 1. The victim VM computes on some confidential data, such as cryptographic keys. We assume that the attacker knows the crypto software that the victim is running.

We do not assume any vulnerability in the VMM, or even a specific VMM platform. Nor do we assume that attacker and victim share a core, that they share memory, or that the attacker synchronizes its execution with the victim.

B. Prime+Probe

Our LLC-based cross-core, cross-VM attack is based on Prime+Probe [28], which is a general technique for an attacker to learn which cache set is accessed by the victim VM. The attacker, A, runs a spy process which monitors cache usage of the victim, V, as follows:
Prime: A fills one or more cache sets with its own code or data.
Idle: A waits for a pre-configured time interval while V executes and utilizes the cache.
Probe: A continues execution and measures the time to load each set of his data or code that he primed. If V has accessed some cache sets, it will have evicted some of A's lines, which A observes as increased memory access latency for those lines.

As the Probe phase accesses the cache, it doubles as a prime phase for subsequent observations.

C. Overview of challenges for efficient PRIME+PROBE attacks on the LLC

Constructing an efficient PRIME+PROBE attack on the LLC is much harder than on the L1 caches. We identify the following challenges, which we explain in the following subsections:

1) Visibility into one core's memory accesses from another core via LLC;

2) Significantly longer time to probe the large LLC;

3) Identifying cache sets corresponding to security-critical accesses by the victim without probing the whole LLC;

4) Constructing an eviction set that can occupy exactly one cache set in the LLC, without knowing the address mappings;

5) Probing resolution.

D. Visibility of processor-memory activity at the LLC

By design, the higher-level caches, L1 and L2, will satisfy most of the processor's memory accesses, which means that the LLC has less visibility into the victim's memory activity than the L1 caches. Since the attacker only shares the LLC with the victim, if its manipulation of the LLC state does not impact the state of the higher-level caches used by the victim VM, the victim's accesses to its interesting code or data will never reach the LLC and will be hidden to the attacker.

We leverage cache inclusiveness, which lets us replace victim data from the complete cache hierarchy, without access to any of the victim's local caches.

E. Infeasibility of priming and probing the whole LLC

Conventionally, for the L1 caches, the PRIME+PROBE technique primes and probes the entire L1 cache, and uses machine-learning techniques to analyze the cache footprint in order to identify *spatial patterns* associated with the victim's memory activity [4, 8, 31, 47]. This is infeasible to achieve with fine resolution on the LLC, since it is orders of magnitude larger than L1 caches (several MiB versus several KiB). We overcome this challenge by first pinpointing very few cache sets corresponding to relevant security-critical accesses made by the victim, and then we only monitor those cache sets during a prime or probe step, instead of monitoring the whole LLC.

F. Identifying cache sets relevant to security-critical victim code and data

How to identify cache sets relevant to a victim's security-critical accesses, however, is still challenging. This is because the attacker does not know the virtual addresses of those security-critical accesses in the victim's address space, and has no control on how these virtual addresses are mapped to the physical addresses. Our solution to this challenge is to scan the whole LLC by monitoring one cache set at a time, looking for *temporal access patterns* to this cache set that are consistent with the victim performing security-critical accesses. The specific *temporal access patterns* depend on the algorithms used. We delay the detailed discussion of this to Section VI and Section VII, which use a simple square-and-multiply exponentiation and the latest sliding-window exponentiation in GnuPG as case studies to show how algorithm-specific security-critical lines can be identified.

G. Eviction set to occupy exactly one cache set

In order to monitor the victim's accesses to one specific cache set, thus pinpointing whether that cache set is accessed by the victim, the attacker needs to be able to occupy that specific cache set. To achieve this, the attacker can construct an *eviction set* containing a collection of memory lines in its own address space that all map to a specific cache set. Since a cache set contains W cache lines, the eviction set must contain W memory lines in order to evict one complete set. As long as the cache replacement policy replaces older lines before recently loaded ones (e.g., with LRU replacement policy used on Intel processors, or FIFO replacement), touching each line in the eviction set once guarantees that all prior data in the set has been evicted.

Constructing an eviction set for the virtually-indexed L1 cache is trivial: the attacker has full control of the virtual address space, and can arbitrarily choose virtual addresses with the same set index bits. In contrast, the LLC is physically indexed. In order to target a specific cache set, the attacker must at least partially recover the address mapping, which is a challenge, as the VMM's address-space mapping is not accessible to the attacker.

The sliced cache of modern Intel processors (Section II-B2) further complicates the attack: even knowing the physical address of memory lines may not be sufficient for creating the eviction sets, since the *slice id* is unknown to the attacker.

In Section IV, we discuss how we construct the eviction set using large pages and without reverse-engineering the hash function.

H. Probing resolution

Extracting fine-grained information, such as cryptographic keys, requires a fine probing resolution. Since the spy process can run asynchronously, without the need to preempt the victim, the probing resolution

of the LLC is not tied to victim preemption, but is fundamentally limited only by the speed at which the attacker can perform the probe. This is much slower than for an L1 cache, for two reasons.

First, the LLC typically has higher associativity than the L1 cache (e.g., 12 to 24-way versus 4 to 8-way), hence more memory accesses are required to completely prime or probe a cache set.

Second, the probe time increases due to the longer access latency of the LLC (about a factor of 10 for recent Intel processors). Even with all lines resident in the LLC, the attacker, when performing a probe of one LLC set, will still experience misses in the L1 and L2 caches, due to their lower associativity. Furthermore, a miss in the LLC will cause more than 150 cycles latency while a miss in the L1 or L2 cache has a latency of less than 40 cycles.

As a consequence, probing an LLC set is about one order of magnitude slower than probing an L1 cache. In Section V, we characterize the probing resolution of the LLC by measuring the channel capacity of an LLC based covert channel.

IV. Constructing the Eviction Set

A. Methodology

We solve the problem of hidden mappings by utilizing large pages. As discussed in Section II, performance-tuned applications, OSes and VMMs use large pages to reduce TLB contention. Large-page support of the VMM allows a large page in the guest physical memory to be backed up by a large frame in the actual physical memory. For our purpose, large pages eliminate the need to learn the virtual-address mapping used by the OS and VMM: a 2 MiB page is large enough so that all the index bits are within the page offset, thus the cache index bits are invariant under address translation—the LLC is effectively virtually indexed. A side effect of large pages is a reduction of TLB misses and thus interference. But note that large-page mappings are only required for the attacker, we make no assumption on how the victim's address space is mapped.

In recent Intel CPUs, large pages are not sufficient to locate an LLC slice, as memory lines with the same set index bits may be located in different LLC slices.

Instead of following Hund et al. [19] in attempting to reverse-engineer the (likely processor-specific) hash function, we construct *eviction sets* by searching for conflicting memory addresses. Specifically, we allocate a buffer (backed up by large pages) of at least twice the size of the LLC. From this buffer we first select a set of potentially conflicting memory lines, i.e., lines whose addresses have the same set index bits (e.g., address bits 6–16, see Figure 2b).

Algorithm 1: Creating the eviction sets

```
input : a set of potentially conflicting memory lines lines
output: a set of eviction sets for lines, one eviction set for
        each slice

Function probe(set, candidate) begin
    read candidate;
    foreach l in set do
        read l;
    end
    measure time to read candidate;
    return time > threshold;
end

randomize lines;
conflict_set ← {};
foreach candidate ∈ lines do
    if not probe(conflict_set, candidate) then
        insert candidate into conflict_set;
end

foreach candidate in lines − conflict_set do
    if probe(conflict_set, candidate) then
        eviction_set ← {};
        foreach l in conflict_set do
            if not probe(conflict_set − {l}, candidate) then
                insert l into eviction_set;
        end
        output eviction_set;
        conflict_set ← conflict_set − eviction_set;
    end
end
```

We then use Algorithm 1 to create eviction sets for a given set index for each slice. This first creates a *conflict set* that contains a subset of the potentially conflicting memory lines such that for each slice, exactly W memory lines in the conflict set map to the same cache set. The conflict set is, effectively, a union of eviction sets for all the slices (each eviction set contains exactly W lines that map to the cache set in a slice.). The algorithm, then, partitions the conflict set into individual eviction sets, one for each slice.

The algorithm uses the function *probe*, which checks whether a candidate memory line conflicts with a set of lines. That is, whether accessing the set of lines evicts the candidate from the LLC. The function first reads data from the candidate line, ensuring that the line is cached. It then accesses each of the memory lines in the set. If, after accessing the set, reading the candidate takes a short time, we know that it is still cached and accessing the lines in the set does not evict it. If, on the other hand, the read is slow, we can conclude that the set contains at least W lines from the same cache set as the candidate, and accessing these forces the eviction of the candidate.

The algorithm creates the conflict set iteratively, adding lines to the conflict set as long as the lines do

Listing 1: Code for probing one 12-way cache set

```
1   lfence
2   rdtsc
3   mov %eax, %edi
4   mov (%r8), %r8
5   mov (%r8), %r8
6   mov (%r8), %r8
7   mov (%r8), %r8
8   mov (%r8), %r8
9   mov (%r8), %r8
10  mov (%r8), %r8
11  mov (%r8), %r8
12  mov (%r8), %r8
13  mov (%r8), %r8
14  mov (%r8), %r8
15  mov (%r8), %r8
16  lfence
17  rdtsc
18  sub %edi, %eax
```

not conflict with it. Intel's hash function is designed to distribute the selected potentially conflicting lines evenly across all the LLC slices [20]. Hence, a buffer of twice the size of the LLC is large enough to construct the desired conflict set.

To partition the conflict set, the algorithm picks a candidate from memory lines that did not make it into the conflict set. The algorithm iterates over the members of the conflict set, checking whether after removing the member, the candidate still conflicts with the conflict set. If removing the member removes the conflict, we know that the member is in the same cache set of the same LLC slice as the candidate. By iterating over all the members of the conflict set we can find the eviction set for the cache set of the candidate.

It takes about 0.2 seconds for determining the slices of a single set index. When the number of cores in the processor is a power of two, the set index bits are not used for determining the LLC slice.Therefore, given the eviction sets for one set index, it is straightforward to construct eviction sets for other set index, without the need to repeat the above procedure for each set index. Otherwise, Algorithm 1 has to be repeated for every set index.

B. *Implementing the* PRIME+PROBE *attack*

Once eviction sets are created, we can implement the PRIME+PROBE attack. The implementation follows the pointer-chasing technique of Tromer et al. [34]: We organize all the memory lines in each eviction set as a linked list in a random order. The random permutation prevents the hardware from prefetching memory lines in the eviction set.

Listing 1 shows the assembly code we use to probe one set of the cache. The input in register `%r8` is the head pointer of the linked list, and the `rdtsc` instruction (lines 2 and 17) is used to measure the time to traverse the list. Each of the 12 `mov` instructions (lines 4 to 15) reads one memory line in the eviction set, which stores the pointer to the next memory line. Since each `mov` instruction is data-dependent on the previous one, access to the memory lines is fully serialized [28]. Upon completion, register `%eax` contains the measured time.

The `lfence` instructions (lines 1 and 16) protect against instruction re-ordering and out-of-order completion. It ensures that all preceding load instructions complete before progressing, and that no following loads can begin execution before the `lfence`. Intel recommends using the `cpuid` instruction for full serialization of the instruction stream [30]. However, as noted by Yarom and Falkner [45], because the `cpuid` instruction is emulated by the VMM, it is less suitable for the purpose of measuring the timing in our attack.

C. *Optimizations*

Several optimizations on the scheme above are possible, to minimize the probe time as well as its variations.

Thrashing: As mentioned in Section II, probing the cache implicitly primes it for the subsequent observation. However, due to the cache's (approximate) LRU replacement policy, using the same traversal order in the prime and probe stages may cause thrashing, i.e., self-eviction by the attacker's own data: If the victim evicts a line, it will be the attacker's oldest. On probing that evicted line, it will evict the second-oldest, leading to a miss on every probe. By using a doubly-linked list to reverse the traversal order during the probe stage, we minimize self-evictions, as the oldest line is accessed last [34].

Interaction with higher-level caches: The attacker data is also partially cached in higher-level caches. Retrieving data from the higher cache levels is faster than reading it from the LLC, hence variations in the L1 and L2 contents affect the cache probe time and introduce noise to its measurements. For example, with an 8-way L1 cache and a timing difference of about 30 cycles between L1 access and LLC access, the total variation can reach 240 cycles—much larger than the difference between LLC and memory access. The interaction of higher-level caches tends to have less effect when the associativity of the LLC is much higher than that of the L1 and L2 caches, since the L1 and L2 caches can only hold a small portion of the eviction set for the LLC. An optimization is that instead of measuring the total probe time, one can measure the time of every load from the eviction set. This approach reduces the noise from the multiple levels of caching, but at the cost of an increased probe time.

V. Probing resolution via channel capacity measurements

Next, we study the probing resolution and the effectiveness of our proposed technique using a willing transmitter, i.e., by constructing a covert channel and characterizing the channel capacity. The covert channel protocol is shown in Algorithm 2. It is similar to the timing-based cache-channel protocol of Wu et al. [42] but more efficient, because we use the technique from Section IV to create an exact eviction set.

Since the sender and the receiver execute concurrently without synchronization, we use a return-to-zero (RZ) self-clocking encoding scheme [14]. The sender and the receiver each allocate a buffer of at least the size of the LLC, backed up by large pages. They agree on two arbitrarily chosen cache-set indices (preferably without interferences from non-participating memory accesses). The sender picks from its allocated buffer two memory lines, *line 0* and *line 1*, that map to the two agreed cache set indices. To send a "1", the sender continuously accesses *line 1* for an amount of time, T_{mark}. Similarly, the sender continuously accesses *line 0* for a time equal to T_{mark} to send a "0". The RZ encoding scheme makes sure there is enough time between two consecutive bits by busy waiting for an amount of time, T_{pause}.

Before monitoring the sender, the receiver first uses Algorithm 1 to create eviction sets *set 1* and *set 0* for *line 1* and *line 0*. It then uses PRIME+PROBE to continuously monitor the two sets. To maximize channel capacity, we set the idle interval between successive probes to zero.

We construct the covert channel on a Dell server platform with the Xen VMM, and a HP desktop platform with VMware ESXi, Table I shows their configurations. We use the default configurations for the VMMs, which have large page support to transparently back up the large pages in the guest physical memory with large frames in the host physical memory. The only special configuration is that for Xen, we need to use native mode for the "rdtsc" instruction to avoid being emulated by the VMM.

Figure 4 shows a sample sequence of the receiver's measurements when interleaved bits of ones and zeros are transmitted. The figure clearly shows that the peaks of *set 0* occur during the troughs of *set 1* and vice versa, corresponding to the "101010..." sequence. The receiver can get more than one sample for each mark, and the pause duration is long enough to avoid overlapping between "1" and "0". The experiment indicates that the threshold value should be 700 cycles. Note that the mark duration seen by the receiver is much longer than T_{mark} of the sender (100 cycles), since the receiver's probe takes much longer time than 100 cycles.

Algorithm 2: Covert channel protocol

line 1: cache line accessed by the sender to send "1".
line 0: cache line accessed by the sender to send "0".
set 1: eviction set conflicting with *line 1*.
set 0: eviction set conflicting with *line 0*.
$D_{send}[N]$: N bits data to transmit by the sender.

Sender Operations:

```
for i ← 0 to N − 1 do
    if D_send[i] = 1 then
        for an amount of time T_mark do
            access line 1;
        end
        busy loop for an amount of time T_pause;
    else
        for an amount of time T_mark do
            access line 0;
        end
        busy loop for an amount of time T_pause;
    end
end
```

Receiver Operations:

```
for an amount of time T_monitor do
    probe set 1 in forward direction;
    probe set 0 in forward direction;
    probe set 1 in backward direction;
    probe set 0 in backward direction;
end
```

TABLE I: Experimental platform specifications.

	Dell R720 (server)	HP Elite 8300 (desktop)
Processor Model	Intel Xeon E5 2690	Intel Core i5-3470
Microarchitecture	Sandy Bridge	Sandy Bridge
Clock Frequency	2.9 GHz	3.2 GHz
# of Cores (slices)	8	4
LLC	20-way 20 MiB	12-way 6 MiB
Cache line size	64 B	64 B
VMM	Xen 4.4 (HVM)	VMware ESXi 5.1
Guest OS	Ubuntu 14.04.1 LTS	CentOS 6.5

The covert channel suffers from various transmission errors, including bit loss, insertion of extra bits or bit flips. We conduct an experiment to measure the effect of the pause duration on channel capacity and error rate. The sender generates a long pseudo-random bit sequence (PRBS) with a period of $2^{15}-1$ using a linear feedback shift register (LFSR) with a width of 15 [29]. The LFSR can exhaust all the 2^{15} states except the all-zero state in one period, therefore the maximum number of consecutive ones and consecutive zeros is 15 and 14, respectively. To decode the signal, the receiver probes both *set 0* and *set 1*. It produces a "1" for each sequence of consecutive probes in *set 1* that take longer than a threshold value (e.g., 700 cycles for the server and 400 cycles for the desktop), and a "0" for each sequence of consecutive probes in *set 0* that are above the threshold.

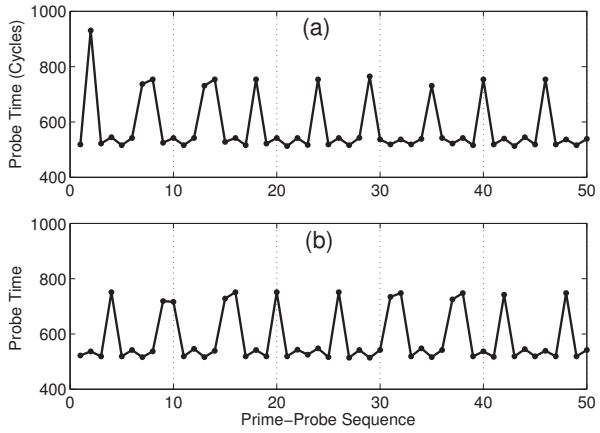


Fig. 4: Sample sequence of receiver's access time on server platform, (a) *set 1*, (b) *set 0*. The sender transmits the sequence "101010...". $T_{mark} = 100$ and $T_{pause} = 3000$ cycles.

To estimate the error rate, we first identify a complete period of the received PRBS to synchronize the received signals with the sent PRBS, and then calculate the *edit distance* [25] of one complete period of the sent PRBS and the received data. The edit distance calculates the number of insertion, deletion, or substitution operations required to transform a string to another string. In the measurement, we fix T_{mark} as 100 cycles and vary T_{pause}.

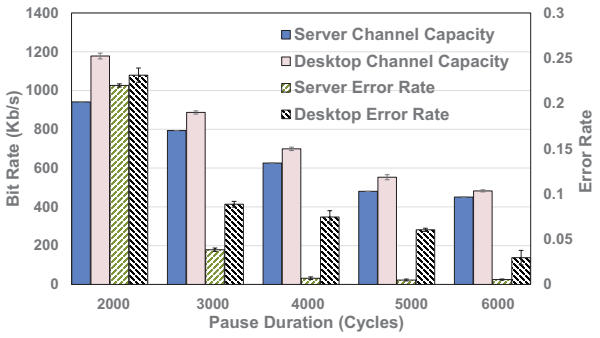


Fig. 5: Channel capacity and error rate of the covert channel.

Figure 5 shows the resulting channel capacity and error rate. As can be expected, increasing the pause duration reduces the error rate, but also the channel capacity. With a small pause time, synchronization may be lost during long sequences of the same bit value. Consequently, it is hard to determine the number of consecutive bits of the same value, resulting in many bit-loss errors. As T_{pause} increases, the error rate drops for both the server and the desktop, until leveling out at about 0.5% and 3%, respectively, where insertion and flip errors dominate the bit-loss errors.

The desktop shows a higher error rate than the server, and larger variance. This is a result of the desktop's lower LLC associativity, as the interaction with higher-level caches tends to be stronger when the associativity of the caches is similar. Perhaps counter-intuitively, the figure also shows that the channel capacity on the generally more powerful server is about 20% less than that of the desktop, also a result of the server's higher LLC associativity (and thus higher probe time) as well as a slightly lower clock rate.

The key takeaway from Figure 5 is the high overall bandwidth, on the desktop 1.2 Mb/s with an error rate of 22%. Although the probe time for the LLC is much longer than that of the L1 cache, this is balanced by the efficiency gain from the concurrent execution of sender and receiver. Our observed bandwidth of 1.2 Mb/s is about 6 times that of the highest previously reported channel capacity [42] for an LLC-based covert channel.

For an apples-to-apples comparison of channel bandwidth, we need to know the error rate in the experiments of Wu et al. [42]. Unfortunately, this error rate is not mentioned, and the test conditions are not compatible with ours. Nevertheless, note that if we reduce the channel bandwidth to 600 Kb/s the error rate on the server platform drops to below 1%. Thus, the data transfer rate of our method is three times faster than the raw transfer rate reported in [42].

We have also experimented with longer mark durations, where the receiver can collect more than one sample per mark. This can further reduce the error rate, but the effects are not as pronounced as for changing the pause duration, so we do not pursue this further.

VI. Attacking the Square-and-Multiply Exponentiation Algorithm

We now show how our approach can be used to leak a secret by recovering a secret-dependent execution path. We use as a case study the square-and-multiply implementation of modular exponentiation.

A. Square-and-multiply exponentiation

Modular exponentiation is the operation of raising a number b to the power e modulo m. In both RSA [33] and ElGamal [11] decryptions, leaking the exponent e may lead to the recovery of the private key.

The square-and-multiply algorithm [15] computes $r = b^e \bmod m$ by scanning the bits of the binary representation of the exponent e. Given a binary representation of e as $(e_{n-1} \cdots e_0)_2$, square-and-multiply calculates a sequence of intermediate values $r_{n-1}, \ldots, r_0$ such that

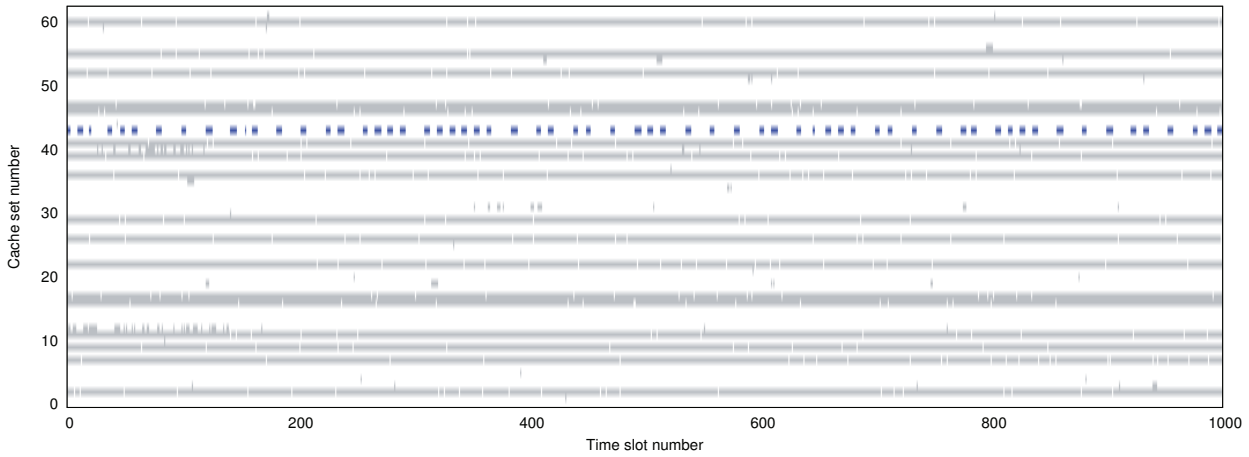


Fig. 6: Traces of activity in cache sets. The highlighted trace at cache set 43 is for the code of the squaring operation.

$r_i = b^{\lfloor e/2^i \rfloor} \bmod m$ using the formula $r_{i-1} = r_i^2 b^{e_{i-1}}$. Algorithm 3 shows a pseudo-code implementation of square-and-multiply.

Algorithm 3: Square-and-Multiply exponentiation

input : base b, modulo m, exponent $e = (e_{n-1} \cdots e_0)_2$
output: $b^e \bmod m$

```
r ← 1
for i from n − 1 downto 0 do
    r ← r^2 mod m
    if e_i = 1 then
        r ← r · b mod m
    end
end
return r
```

The multiplications and modulo reductions directly correspond to the bits of the exponent: each occurrence of square-reduce-multiply-reduce corresponds to a one bit, while each occurrence of square-reduce not followed by a multiply corresponds to a zero bit. Consequently, an attacker process that can trace the execution of the square-and-multiply exponentiation algorithm can recover the exponent [2, 45, 47]. We now show how we can attack this algorithm using the technique developed in Section IV. The main challenge is finding the cache sets that hold the relevant victim code.

By their nature, side-channel attacks are very specific to the details of what is being attacked. Here we develop an attack against the implementation of square-and-multiply found in GnuPG version 1.4.13. For the victim we use the default build, which compiles the code with a high level of optimization (`-O2`), but leaves the debugging information in the binary. The debugging information is not loaded during run time, and does not affect the performance of the optimized code. The victim repeatedly executes the GnuPG binary to decrypt a short file encrypted with a 3,072 bit ElGamal public key. GnuPG uses Wiener's table to determine the key length to use. For a 3,072 bit ElGamal, Wiener's table returns a value of 269. GnuPG adds a 50% safety margin, resulting in a key length of 403 bits.

The technique we use is fairly independent of the specifics of the hardware platform. We apply it to our two experimental platforms of Table I, but it will work on all processors that have inclusive caches and large-page mappings. It will also work on other implementations of the square-and-multiply algorithm, and in fact on any algorithm whose execution path depends on secret information.

B. Implementing the attack

The core idea of the attack is to monitor the use of the squaring operation. While processing a "1" bit, the squaring is followed by a modulo reduction, which is followed by a multiply and another reduction. In contrast, for a "0" bit, after the squaring there is only one reduction, which will then be followed by the squaring for the next bit. Hence, by observing the time between subsequent squarings, we can recover the exponent.

We trace cache-set activity looking for this access pattern. To trace a cache set, we divide time into fixed slots of 5,000 cycles each, which is short enough to get multiple probes within each squaring operation. Within each time slot, we prime the cache set, wait to the end of the time slot and then probe the cache set.

Figure 6 shows the traces of several cache sets. Each line shows a trace of a single cache set over 1,000 time slots. Shaded areas indicate time slots in which activity was detected in the traced cache set.

As the figure demonstrates, some cache sets are accessed almost continuously, others are almost never accessed, whereas others are accessed sporadically. The highlighted cache set at line 43 is the only one exhibiting the activity pattern we expect for the squaring code, with a typical pulse of activity spanning 4–5 time slots. The pauses between pulses are either around six time slots, for "0" bits, or 16–17 time slots for "1" bits. In addition, there are some variations in the pattern, including pulses of a single time slot and pauses of over 20 time slots.

We can easily read the bit pattern of the exponent from line 43 in Figure 6: Reading from the left, we see two pulses followed by short intervals, indicating two "0" bits. The next pulse is followed by a longer interval, indicating a "1" bit. The resulting bit pattern is 001001111111101...

To identify the cache set we correlate the trace of the cache set with a pattern that contains a single pulse: the pattern has 6 slots of no activity, followed by 5 with activity and another 6 without activity. We count the number of positions in the trace that have a good match with the pattern and mark traces that have a large number of matches as potential candidates for being the squaring cache set. We pass candidates to the user for the decision on whether the trace is, indeed, of the squaring cache set.

C. Optimization

Rather than searching all cache sets, we can leverage some information on the GnuPG binary to reduce the search space. In many installations, the GnuPG binary is part of a standard distribution and is, therefore, not assumed to be secret. An attacker that has access to the binary can analyze it to find the page offset of the squaring code. As there is some overlap between the (4 KiB) page offset and the cache set index (Figure 2), the attacker only needs to search cache sets whose set index matches the page offset of the victim code. This reduces the search space by a factor of 64.

D. Results

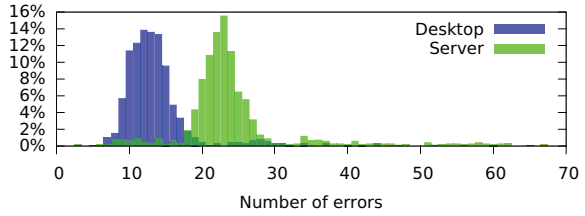


Fig. 7: Distribution of capture errors

With this optimization, we require on average about 120 executions of the victim to locate the cache set of the squaring code on the desktop platform. As the LLC on the server platform is twice as big, we need about 240 executions there. Once found, we collect the access information and use a shell script to parse the results.

Figure 7 shows the distribution of the number of capture errors on the desktop and the server platforms. Out of 817 exponentiations captured on the desktop platform, we drop 46, where the observed exponent is significantly longer or significantly shorter than the expected 403 bits. For the server platform we collect 959 and drop 297. Figure 7 shows the distribution of the number of capture errors over the remaining 771 (desktop) and 662 (server) captured exponentiations.

VII. Attacking the Sliding-Window Exponentiation

In the previous section we showed how to recover secret-dependent execution paths. We now show that our attack can also be used to observe secret-dependent data access patterns. As an example we use an implementation of the sliding-window exponentiation algorithm [7].

A. Sliding-window exponentiation

Given an exponent e, the sliding-window representation of the exponent is a sequence of windows w_i, each of length $L(w_i)$ bits. A window w_i can be either a zero window with a string of "0"s, or a non-zero window that starts with a "1" and ends with a "1". For a sliding window representation with a window size S, the length of the non-zero window satisfies $1 \leq L(w_i) \leq S$, hence the value of a non-zero window is an odd number between 1 and $2^S - 1$. Algorithm 4 computes an exponentiation given the sliding-window representation of the exponent.

Algorithm 4: Sliding-window exponentiation

input : window size S, base b, modulo m,
N-bit exponent e represented as n windows w_i of length $L(w_i)$
output: $b^e \bmod m$

```
//Precomputation
g[0] ← b mod m
s ← MULT(g[0], g[0]) mod m
for j from 1 to 2^(S−1) do
    g[j] ← MULT(g[j − 1], s) mod m
end
//Exponentiation
r ← 1
for i from n downto 1 do
    for j from 1 to L(w_i) do
        r ← MULT(r, r) mod m
    end
    if w_i ≠ 0 then r ← MULT(r, g[(w_i − 1)/2]) mod m;
end
return r
```

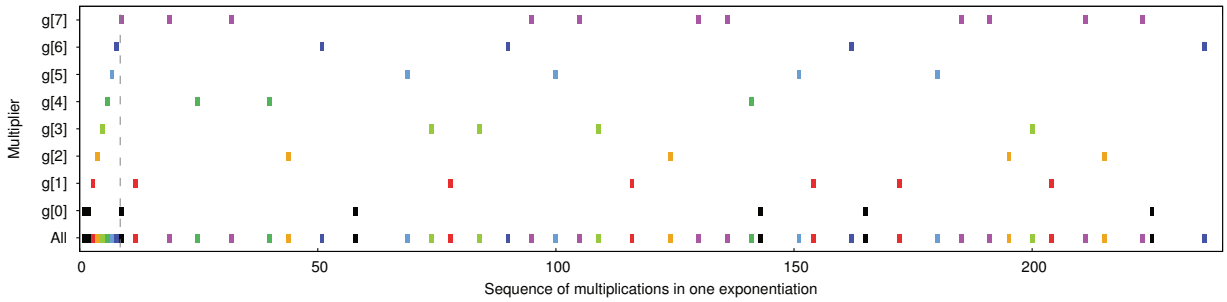


Fig. 8: The usage patterns of the multipliers $g[i] = b^{2i+1}$ in an exponentiation. The vertical line separates the pre-computation and the body of the exponentiation. The x-axis shows the first 240 of 492 multiplications in the exponentiation.

The algorithm scans the exponent from the most significant to the least significant bit, executing a square for each bit. Whenever it reaches the least significant bit of a non-zero window, a multiplication is performed. The multiplier used is $b^v \bmod m$, where v is the value of the non-zero window. For better performance, the algorithm pre-computes multipliers $b^v \bmod m$ for each odd value of v ($1 \leq v \leq 2^S - 1$) and stores them in a table $g[i]$, which can be unambiguously indexed with $i = (v-1)/2$. For example, when $S = 4$, we compute 8 multipliers, $b^1, b^3, b^5, ..., b^{15} \bmod m$, which can be found in $g[0], g[1], g[2], ..., g[7]$, respectively.

To thwart recovery of the square-and-multiply sequence with the FLUSH+RELOAD attack [45], GnuPG uses the multiply routine to calculate squares. Unless explicitly mentioned otherwise, we use multiplication to refer to the execution of the multiply routines, including both the square and the true multiply. Thus, the GnuPG implementation of the sliding-window algorithm performs a sequence of multiplication operations.

However, this still leaks information, since the true multiply (last statement in Algorithm 4) looks up the pre-computed multiplier table using the value of the non-zero window. An attacker can determine the position and value (and hence the length) of the non-zero window by knowing when and which pre-computed multiplier is used, which is sufficient to completely recover the exponent [31].

As in Section VI, we tailor the attack to a specific implementation of the algorithm. In this case, we use the latest version (1.4.18) of GnuPG. The victim repeatedly decrypts a short file encrypted with a 3,072 bit ElGamal public key.

B. Multiplier usage pattern

The core idea of our attack is to *monitor the use of each pre-computed multiplier in the multiplication operations*. We define *multiplier usage pattern* U_i for a multiplier $g[i]$ as a bit vector indicating whether $g[i]$ is used as an operand in each multiplication operation of a single exponentiation.

As a concrete example, Figure 8 displays the multiplier usage patterns (each horizontal line represents a usage pattern for one multiplier) for the first 240 multiplications out of the 492 multiplications executed during a single decryption with a 3,072 bit ElGamal key. This has a 403-bit exponent, for which GnuPG uses window size $S = 4$, resulting in $2^{S-1} = 8$ multipliers. The exponentiation requires 8 multiplications in the pre-computation phase, 403 for squaring (one for each bit), as well as 81 for the true multiply operation, for a total of 492. For a 3,072-bit ElGamal key, the size of the multiplier is also 3,072 bits, which may occupy 6–7 cache lines.

The usage pattern of the multipliers in the 8 multiplications of the pre-computation phase is very regular: The first squares b ($g[0]$) to calculate s, while the next 7 multiply $g[i-2]$ with s to calculate the next odd multiplier $g[i-1]$. Thus $g[0]$ is used as an operand in the first two multiplications, and $g[i]$ ($1 \leq i \leq 6$) is used in the $(i+2)^{\text{nd}}$ multiplication. $g[7]$ is special, since it is not used as an operand in the pre-computation phase, but there is a write access to $g[7]$ at the end of the pre-computation phase, which will be captured in the first time slot of the first multiplication in the exponentiation body. We also include this write access to $g[7]$ in the usage pattern since it is useful to identify $g[7]$.

The multiplier usage patterns in the exponentiation phase are irregular. However, we can calculate some statistical data on the expected use. For a window size of S, the expected distance between non-zero windows is $S+1$ [27]. Thus, on average, we expect $N/(S+1)$ non-zero windows in the exponent. With 2^{S-1} different values of non-zero windows, we expect each multiplier to be used $2^{1-S} \cdot N/(S+1)$ times during the exponentiation. Hence, for window size $S = 4$, we expect each multiplier to be accessed $2^{1-4} \cdot 403/(4+1) \approx 10$ times

during the exponentiation.

By combining the usage patterns for all the multipliers, recovering the key is straightforward: the multiplications that do not use any of the multipliers are identified as square operations, the remaining multiplications are true multiply operations and the multipliers can also be identified.

C. Identify multiplier cache set

A key step of our attack is identifying which cache set corresponds to which multiplier. Similar to the attack in Section VI, we need to scan every cache set for a long sequence of time slots, e.g. spanning several executions of ElGamal decryption. However, there are three significant differences compared with the attack in Section VI:

1) Since we want to get the use of a multiplier during the multiplication operations, a naive trace of cache-set activity will not tell when a multiplication operation starts and ends. Therefore, our measurements of the cache-set activities must be aligned with the sequence of multiplication operations.

2) The locations of the multipliers are not only unknown, but also differ for every execution of the exponentiation. This is because the multipliers are stored in heap memory that is dynamically allocated for every execution of the exponentiation. Hence, multiplier location information obtained during one exponentiation cannot be used for locating the multiplier in subsequent exponentiations. Furthermore, for every execution of the exponentiation, a monitored cache set may contain a multiplier, or may not contain any multipliers; the probability depends on the size of a multiplier and the total number of cache sets. This is in contrast with the attack in Section VI where the location of the cache set containing the squaring code is fixed for different executions of the exponentiation.

3) The access to the multiplier is sparse and irregular, as can be seen in Figure 8. In Section VI, we showed that the access to those cache sets containing the squaring code shows regular temporal patterns that can be easily recognized. However, since we do not know when, during an exponentiation, each multiplier is used, we have very little information on the temporal access pattern of a multiplier.

In a nutshell, our strategy is as follows:

1) To trace the cache-set activities in the multiplication operations, we simultaneously monitor two cache sets: one is the scanned cache set that may potentially contain a multiplier, and the other is the cache set containing the multiplication code. In this way, the sequence of multiplications serves as the "clock" for all the measurements.

2) Although we do not know the expected temporal access pattern for a multiplier, we do know that cache sets corresponding to the same multiplier must show similar cache-set activities even in the presence of noise, since they must all follow the usage pattern of that multiplier. Therefore, we can cluster the scanned cache sets into groups, each group showing similar cache-set activities during the multiplication operations.

3) Since the multiplier usage pattern at the precomputation phase is regular and distinct, we can leverage this knowledge to identify which group corresponds to which multiplier. The statistical information on the use of the multiplier during the exponentiation phase can also be leveraged to clean out some noise for the clustering.

Putting it all together, our attack follows this outline:

1) Find the cache sets that contains the multiplication code.
2) Collect cache-set activity traces for all the cache sets (Section VII-D).
3) Filter out traces that do not match the expected statistical properties of multiplier access (Section VII-E).
4) Cluster the traces to amplify the signal (Section VII-F).
5) Analyze the clusters to recover the multiplier usage patterns and calculate the exponent (Section VII-G).

With the exception of the last step, the attack is automated. In Section VII-H we present the results of running the attack on the experimental platforms.

D. Collect cache-set activity trace patterns

The purpose of this step is to create *trace patterns* for the cache-set activities during the victim multiplication operations. A trace pattern is a bit vector of length m, the number of multiplications in an exponentiation. Each bit describes whether a cache set is accessed or not in the multiplication operation. In the absence of noise, if a cache set is used for a multiplier, we expect the trace pattern of the cache set to match the usage pattern of that multiplier. However, noise is present, both due to system activity and due to capture errors. Nevertheless, we still expect the trace pattern of the cache set to be similar to the usage pattern of the multiplier.

To collect trace patterns, we first reuse the method of Section VI-B to identify a cache set used for the multiplication code. Next, we scan cache-set activities for every cache set, each scan lasts for 35,000 time slots (of 5,000 cycles each). For each scan, we simultaneously monitor the scanned cache set and the cache set containing the multiplication code. Within each time slot, we prime both cache sets, and at the end of the

time slot we probe for activity in both sets. We use the information obtained from the cache set containing the multiplication code to identify when the victim performs the exponentiation and when the multiplication operations start and end. We then check for activity in the scanned cache set during these multiplications to construct the trace patterns. We leave the detailed description on how we handle noise in the cache set containing multiplication code to Appendix A.

Note that because a decryption takes about 7,000 time slots, each scan of 35,000 time slots contains 4–5 complete exponentiations. Therefore, we can get 4–5 samples of trace patterns in each scan. We refer readers to Appendix B for a discussion of the number of trace patterns we need to collect.

E. Filter out unexpected trace patterns

Since the size of the multipliers is much smaller than the size of the LLC, most of the trace patterns created during the scan do not correspond to cache sets used for multipliers. To reduce the amount of data we process in the next phase, we leverage the statistical knowledge of the multiplier usage patterns to filter out trace patterns that are unlikely to be a multiplier.

As discussed in Section VII-B, during the first few multiplication operations, $g[0]$ is accessed twice and other multipliers are accessed once in sequence. A multiplier is expected to be used 10 times during the subsequent exponentiation phase. Therefore, we can discard trace patterns that show too little or too much activity (< 5 or > 20 multiplications). Furthermore, we remove trace patterns that do not show activity within the first few multiplications.

F. Clustering trace patterns

In the absence of noise, the trace patterns for cache sets used for the same multiplier are identical to the multiplier usage pattern, so usage patterns of all cache sets containing the same multiplier would be identical. With moderate noise we cannot expect them to be identical, but they should at least be similar. We rely on this similarity to group trace patterns for a multiplier together, thereby converging towards the actual usage of that multiplier in multiplication operations. Because our attack collects enough trace patterns, we are highly likely to capture multiple trace patterns for each multiplier.

To group similar trace patterns, we use a hierarchical clustering algorithm [17] with the edit distance [25] between the trace patterns as the measure of similarity. The clustering algorithm is quadratic in the number of trace patterns. Figure 9 demonstrates a cluster of trace patterns, representing the usage pattern of a multiplier.

G. Identify corresponding multipliers for clusters

The final step is recovering usage patterns of all the pre-computed multipliers, which directly exposes the exponent, as we discussed in Section VII-B. We find that identifying the clusters representing usage of multipliers is fairly straightforward. According to the statistical information on multiplier usage (Section VII-B), we can easily locate those clusters that match expectations. Therefore, we focus on error correction and on mapping clusters to corresponding multipliers. Unlike previous steps, which are automated, this step requires manual processing.

To explain how we capture errors, we look at a sample cluster in Figure 9, showing a cluster with 15 trace patterns. Each horizontal line represents a trace pattern of the multiplier cache set, indexed by the multiplication sequence. The shaded areas are the multiplication indices in which the scanned cache set shows activity. The solid vertical lines show the ground-truth activity, i.e., the usage pattern for the multiplier, as obtained from the victim's key. Red marks indicate activity detected due to noise in the scanned cache line. Because the noise is independent of the multiplication activity, it can be easily identified by comparing all the trace patterns in the cluster.

In the figure we can see another noise effect: the further we advance in the exponentiation, the more the trace patterns deviate from the ground truth. We believe that this deviation is caused by short pauses in the victim operation which result in our attack interpreting a single multiplication operation as two separate multiplications. While we do try to correct these (Appendix A), our fix is, evidently, not perfect.

For correcting errors, we process the trace patterns from left to right. We re-align all the trace patterns in a cluster based on their common positions that have access to the corresponding multiplier. This removes the spurious accesses as we progress through the trace patterns.

Lastly, we assign trace patterns to pre-computed multipliers, $g[i]$, by comparing the patterns in the first few multiplications with the knowledge of the pre-computation phase in Algorithm 4. For the cluster in Figure 9, the trace patterns indicate that the first multiplication operation of that multiplier occurs at the third index. According to the sequence of multiplications in the pre-computation stage, we conclude that this cluster represents usage patterns of multiplier $g[1]$. By processing all clusters with the same technique, we are able to identify usage patterns of all pre-computed multipliers.

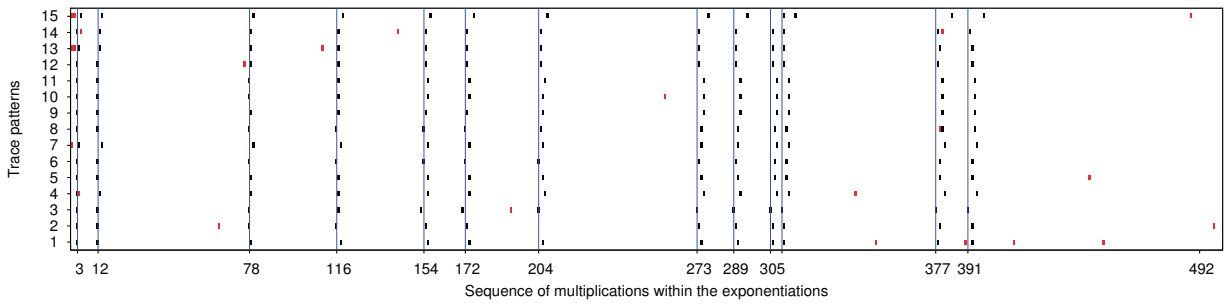


Fig. 9: A cluster of traces

H. Results

We test the attack five times on each platform, each time with a different key. In each attack, the victim runs in one VM, repeatedly executing the GnuPG program, decrypting a short text file. The attacker runs continuously in another VM, observing activity in the shared LLC. Table II summarizes the results.

TABLE II: Results of attack on sliding window.

	server	desktop
Online attack time	27m	12m
Offline analysis time	60s	30s
Manual processing time	10m	10m
Observed exponentiations	79,900	33,600
Interesting exponentiations	1,035	734
Average cluster size	20.4	17.7
Minimum cluster size	12	5

Most of the attack time is spent on the online attack, collecting observations of cache sets. Due to the larger cache size, we collect more observations on the server platform. We filter out over 97% of the observations because they do not match the expected activity of a multiplier, leaving 700–1000 interesting traces, which we pass to the clustering algorithm. This offline phase takes less than a minute, leaving us with a list of clusters. The average cluster size is around 20 traces, with a minimum of 5. In all test cases, we require about 10 minutes of an expert to do the manual processing of the clusters to completely recover the actual usage of multipliers and recover the key.

VIII. Related work

A. Prime+Probe

This technique has been used for attacks against several processor caches, including the L1 data cache [28, 31, 34, 47], L1 instruction cache [1, 2, 4] and the branch prediction cache [3]. All these caches are core-private, and the attacks exploit either hyper-threading or time multiplexing of the core.

Zhang et al. [47] use Prime+Probe to implement a cross-VM attack on the square-and-multiply implementation of GnuPG version 1.4.13. The attack relies on exploiting a weakness in the Xen scheduler and on having a non-zero probability of the spy and victim time-sharing the same core. The attack requires six hours of constant decryptions for collecting enough data to break the key. In contrast, we use the LLC as an attack vector, which is used by all cores, and do not need to trick the scheduler to share a processor core, resulting in a much faster attack.

B. LLC based covert channel

Percival [31] describes an L2 covert channel with a capacity of 100 KiB/s, but does not explain how the attack recovers the address mapping. Ristenpart et al. [32] experiment with L2 covert channels in a cloud environment, achieving a bandwidth of about 0.2 b/s. Xu et al. [43] extend this attack, reporting an L2-based channel with a capacity of 233 b/s. By focusing on a small group of cache sets, rather than probing the whole cache, Wu et al. [42] achieve a transfer rate of over 190 Kb/s. By accurately mapping the cache sets, our attack achieves a much higher bandwidth (up to 1.2 Mb/s) than prior work.

C. LLC based side channel attacks

Due to the low channel capacity, an LLC-based side channel typically only leaks course-grain information. For example, the attacks of Ristenpart et al. [32] leak information about co-residency, traffic rates and keystroke timing. Zhang et al. [46] use an L2 side channel to detect non-cooperating co-resident VMs. Our attack improves on this work by achieving a high granularity that enables leaking of cryptographic keys.

Yarom and Falkner [45] show that when attacker and victim share memory, e.g. shared libraries, the technique of Gullasch et al. [16] can achieve an efficient cross-VM, cross-core, LLC attack. The same technique has been used in other scenarios [5, 22, 35, 44, 48]. Our attack removes the requirement for sharing memory, and is powerful enough to recover the key from the latest GnuPG crypto software which uses the more advanced

sliding window technique for modular exponentiation, which is impossible using FLUSH+RELOAD attacks.

In concurrent work Irazoqui et al. [23] describes the use of large pages for mounting a synchronous LLC PRIME+PROBE attack against the last round of AES.

IX. MITIGATION

A. Fixing GnuPG

One approach to fixing GnuPG for preventing information leaks is exponent blinding, which splits the exponent into two parts. Modular exponentiation is performed on each part, followed by combining the results [9].

An alternative is a constant-time implementation that does not contain any conditional statements or secret-dependent memory references. Techniques for constant-time implementations have been explored, for example, in Bernstein et al. [6]. These approaches can be tricky to get right, and recent work has demonstrated that a "constant-time" implementation of OpenSSL is still susceptible to timing attacks at least on the ARM architecture [10].

When reporting the vulnerability to the GnuPG team, we also provided a patch, which changes the exponentiation algorithm to a fixed-window exponentiation, and ensures that the access patterns to the multipliers do not depend on exponent bits. This measure falls short of a constant-time implementation because the implementation of the multiplication and modular reduction are not constant time. While it is possible to leak information from these implementations [12, 13], we are not aware of a micro-architectural attack that can exploit this weakness.

B. Avoiding resource contention

While fixing GnuPG is clearly desirable, this does not address the general issue of maintaining isolation and preventing information leaks in a multi-tenant environment. Since the root cause of LLC attacks is resource contention, the most effective countermeasure is to eliminate the resource contention. This can be achieved with different granularity.

1) Avoid co-residency: This is the coarse-grained partitioning of the resource: simply disallowing VMs from different tenants to be hosted on the same processor package, which prevents sharing of the LLC among the attacker and the victim VMs. However, this approach is fundamentally at odds with the core motivation of cloud computing: reducing cost by increasing resource utilization through sharing. Given the steady increase in core counts, the economics will shift further in favor of sharing.

2) Cache partitioning: Cache partitioning is a form of fine-grained resource partitioning. There are several approaches to partition the cache.

One approach is to partition the cache by sets. This can be achieved through page coloring, where frames of different color are guaranteed to map to different cache sets [26]. The VMM can manage the allocation of host-physical memory so that VMs from different tenants are mapped to frames of disjoint colors. Coloring frames complicates the VMM's resource management and leads to memory wastage due to fragmentation. It is also incompatible with the use of large pages, and thus foregoes their performance benefits.

STEALTHMEM [24] proposes a smarter way to utilize the page coloring technology by only reserving very few colors, known as stealth pages, for each physical core to store the security-sensitive code and data. It ensures that the security-sensitive code and data will not have cache conflicts with other code and data. However, this approach does not eliminate the LLC-based covert channel.

The latest Intel processors provide *cache allocation technology* (CAT), which partitions the cache by ways [21]. CAT defines several classes of service (COS), and each COS can be allocated a subset of ways in each cache set. This can be used to partition the LLC between COSes. The architecture presently supports up to four COSes, which may not be sufficient in many cloud environments. Further research is required to study the use of CAT and its effectiveness as a countermeasure to LLC-based attacks.

Fine-grained cache partitioning can also be done dynamically using special load and store instructions that can lock a security-critical cache line into the cache, as in the *partition-locked cache* (PLcache) proposed by Wang and Lee [40]. However, the PLcache design only locks data in the cache, it is not clear how it can be extended to lock instructions.

C. Run-time diversification

Other proposals for secure cache designs randomize the memory-to-cache mapping [40, 41]. This randomizes resource contention, so an attacker cannot extract useful information. These designs have been applied to the L1 data cache without causing performance degradation, but use on the much larger LLC has not been investigated to date.

Fuzzy time approaches disrupt the timing measurement by adding noise or slowing it down, or reduce the accuracy of the clock [18, 37]. The drawback is that it may impact other benign applications that require the access to the high-resolution timers.

X. Conclusions

We have presented a method for implementing an LLC-based PRIME+PROBE attack. We have demonstrated that the LLC presents a high-bandwidth channel, in the worst case exceeding 1 Mb/s. We have then shown that the approach can be used to mount cross-core, cross-VM side channel attacks that leak keys effectively for crypto code with secret-dependent execution paths as well as with secret-dependent data access. We have demonstrated these attacks against implementations of ElGamal decryption in GnuPG. The attack is very effective, taking a few seconds to break the key when used against old versions of GnuPG and between 12 and 27 minutes for the latest version.

Our assumptions are minimal: we rely on cache inclusiveness and utilize large-page mappings in the attacker, and assume that the VMM uses large frames to map guest physical memory. Beyond that we make no assumptions on the environment or the victim, other than it repeatedly decrypts text using the same key. In particular, we do not require memory sharing across VMs, do not exploit VMM weaknesses and show that the same attack works with different VMMs and different hardware platforms.

Given these weak assumptions, we believe that our attack is eminently practical, and as such presents a real threat against keys used by cloud-based services. However, we have not tested the attack in a noisy environment, nor in a real cloud settings. Experimenting in such environments to measure the effects of noise on the attack is left for future work.

Given the dependence on large-page mappings, the easiest countermeasure would be to disable large pages in the VMM, but this will result in a performance penalty for all clients of the cloud provider, whether or not they are potential targets—the provider will most likely not be too keen to use this defense. Also, it might be possible to adapt our attack to work without large pages (although at a reduced efficiency).

While fixing GnuPG would defeat our specific attacks, this will not prevent information leaks from other software. In principle, any frequently-executed secret computation that is not constant-time is vulnerable to the attack. Leveraging hardware support for LLC partitioning might be the most promising defense, but whether those mechanisms work in practice remains to be seen.

Acknowledgments

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program. This work was also supported by DHS/AFRL FA8750-12-2-0295 and NSF CNS-1218817. We thank Carlos Rozas and Frank Mckeen from Intel for their support of this work when Fangfei Liu was an intern at Intel Labs.

Appendix A
Handling noise in the multiplication cache set

The technique of Section VII-D relies on identifying the sequence of multiplication operations from the activity in the multiplication cache set. In the absence of noise, this is a trivial task. However, two types of noise complicate the process: occasional gaps within a single multiplication operation and the merging of multiple multiplication operations to a single sequence of activity. We suspect that the former is caused by short bursts of system activity and that the latter is caused by our probing process occasionally failing to evict all victim lines from the multiplication cache set.

To clean this noise, we remove short gaps of inactivity in the multiplication cache set and break sequences of activity longer than twice the expected length of a multiplication operation. We use the cleaned result to identify multiplication operations. With the multiplication operation identified, we can generate the trace patterns by checking for activity in the other probed cache set during each multiplication operation.

Figure 10 shows an example of a trace of the two cache sets and the results of identifying the multiplication operations. The first, fourth and fifth multiplications show no indication of noise in the multiplication cache set. In the sixth multiplication we notice that no activity is indicated in the multiplication cache set during the second time slot. Because this gap is short, we include this time slot in the multiplication.

The trace captures activity in the multiplication cache set during time slots 11 to 22. As this period is longer than twice the expected length of a multiplication, we treat it as the second type of noise and split it into two multiplications.

Thus, the trace in Figure 10 spans six multiplications. The other cache set shows activity during the first and sixth of these. That is, the trace pattern from this figure is 100001. We ignore activity captured in the other cache set that in time slots outside the multiplication operations, e.g. in time slot 26.

Appendix B
Calculating the number of required observations

We have seen that the attack collects traces for all the cache sets on LLC during multiple exponentiations. The question that remains is how many exponentiations

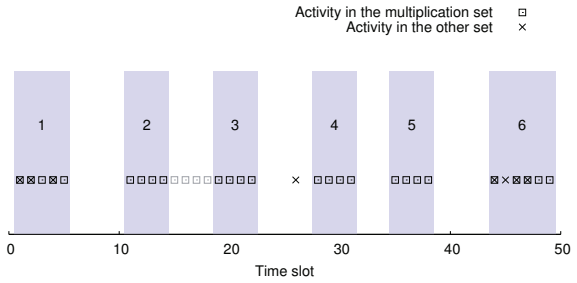


Fig. 10: Traces of the multiplication and another cache sets, indicating time spans of multiplication operations (shaded) and the multiplication index.

we need to observe in order to collect enough traces for the attack.

For effectively identifying multipliers, including redundancy for error correction, we need 10–15 traces for each multiplier. On our experimental platforms, a 3,072 bit multiplier maps to 6–7 cache sets on the LLC. If we collect one trace from each cache set with zero error rate, we expect to obtain processable traces from 6–7 cache sets. However, as shown in Figure 6, some cache sets are constantly active on our desktop platform, demonstrating an error rate for a third of all sets. Hence, we anticipate to obtain four or five usable traces for each multiplier during an exponentiation. By collecting observations for four exponentiations from each cache set, the expected number of reliable traces for each multiplier is between 16 and 20, which satisfies our requirements.

REFERENCES

[1] O. Acıiçmez, "Yet another microarchitectural attack: exploiting I-Cache," in *Comp. Security Arch. WS*, Fairfax, VA, US, Nov 2007, pp. 11–18.

[2] O. Acıiçmez and W. Schindler, "A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL," in *CT-RSA*, San Francisco, CA, US, Apr 2008, pp. 256–273.

[3] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "On the power of simple branch prediction analysis," in *ASIACCS*, Singapore, Mar 2007, pp. 312–320.

[4] O. Acıiçmez, B. B. Brumley, and P. Grabher, "New results on instruction cache attacks," in *CHES*, Santa Barbara, CA, US, Apr 2010, pp. 110–124.

[5] N. Benger, J. van de Pol, N. P. Smart, and Y. Yarom, "'Ooh aah..., just a little bit': A small amount of side channel can go a long way," in *CHES*, Busan, KR, Sep 2014, pp. 75–92.

[6] D. J. Bernstein, T. Lange, and P. Schwabe, "The security impact of a new cryptographic library," in *Conf. Cyptology & Inform. Security Latin America*, Santiago, CL, Oct 2012, pp. 159–176.

[7] J. Bos and M. Coster, "Addition chain heuristics," in *CRYPTO*, Santa Barbara, CA, US, Aug 1989, pp. 400–407.

[8] B. B. Brumley and R. M. Hakala, "Cache-timing template attacks," in *ASIACRYPT*, 2009, pp. 667–684.

[9] C. Clavier and M. Joye, "Universal exponentiation algorithm a first step towards *Provable* SPA-resistance," in *CHES*, Paris, FR, May 2001, pp. 300–308.

[10] D. Cock, Q. Ge, T. Murray, and G. Heiser, "The last mile: An empirical study of some timing channels on seL4," in *CCS*, Scottsdale, AZ, US, Nov 2014, pp. 570–581.

[11] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *Trans. Inform. Theory*, no. 4, pp. 469–472, Jul 1985.

[12] D. Genkin, A. Shamir, and E. Tromer, "RSA key extraction via low-bandwidth acoustic cryptanalysis," in *CRYPTO*, Santa Barbara, CA, US, Aug 2014, pp. 444–461.

[13] D. Genkin, L. Pachmanov, I. Pipman, and E. Tromer, "Stealing keys from PCs by radio: Cheap electromagnetic attacks on windowed exponentiation," Cryptology ePrint Archive, Report 2015/170, Feb 2015, http://eprint.iacr.org/.

[14] I. Glover and P. Grant, *Digital Communications*. Prentice Hall, 2010.

[15] D. M. Gordon, "A survey of fast exponentiation methods," *J. Algorithms*, no. 1, pp. 129–146, Apr 1998.

[16] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games — bringing access-based cache attacks on AES to practice," in *Symp. Security & Privacy*, Oakland, CA, US, may 2011, pp. 490–595.

[17] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference and Prediction*, 2nd ed. New York, NY, US: Springer Science+Business Media, 2009.

[18] W.-M. Hu, "Reducing timing channels with fuzzy time," in *Symp. Security & Privacy*, Oakland, CA, US, May 1991, pp. 8–20.

[19] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space ASLR," in *Symp. Security & Privacy*, San Francisco, CA, US, May 2013, pp. 191–205.

[20] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Intel Corporation, Apr 2012.

[21] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2*, Intel Corporation, Jun 2014.

[22] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! a fast, cross-VM attack on AES," in *RAID*, Gothenburg, SE, Sep 2014, pp. 299–319.

[23] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A shared cache attack that works across cores and defies VM sandboxing—and its application to AES," in *IEEE: Security & Privacy*, San Jose, CA, US, May 2015.

[24] T. Kim, M. Peindo, and G. Mainer-Ruiz, "STEALTHMEM: System-level protection against cache-based side channel attacks in the Cloud," in *USENIX Security*, Bellevue, WA, US, Aug 2012.

[25] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *Soviet Physics Doklady*, p. 707, Feb 1966.

[26] J. Liedtke, N. Islam, and T. Jaeger, "Preventing denial-of-service attacks on a μ-kernel for WebOSes," in *6th HotOS*, Cape Cod, MA, US, May 1997, pp. 73–79.

[27] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, C. Press, Ed. CRC Press, 1997.
[28] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," http://www.cs.tau.ac.il/~tromer/papers/cache.pdf, Nov 2005.
[29] C. Paar and J. Pelzl, *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer-Verlag New York Inc, 2010.
[30] G. Paoloni, *How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures*, Intel Corporation, Sep 2010.
[31] C. Percival, "Cache missing for fun and profit," http://www.daemonology.net/papers/htt.pdf, 2005.
[32] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off my cloud: Exploring information leakage in third-party compute clouds," in *CCS*, Chicago, IL, US, Nov 2009, pp. 199–212.
[33] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *CACM*, no. 2, pp. 120–126, Feb 1978.
[34] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks in AES, and countermeasures," *J. Cryptology*, no. 2, pp. 37–71, Jan 2010.
[35] J. van de Pol, N. P. Smart, and Y. Yarom, "Just a little bit more," in *CT-RSA*, 2015.
[36] V. Varadarajan, T. Ristenpart, and M. Swift, "Scheduler-based defenses against cross-VM side-channels," in *USENIX Security*, San Diego, CA, US, Aug 2014, pp. 687–702.
[37] B. C. Vattikonda, S. Das, and H. Shacham, "Eliminating fine grained timers in Xen," in *CCSW*, Chicago, IL, US, Oct 2011, pp. 41–46.
[38] *Large Page Performance*, VMware Inc., Palo Alto, CA, US, 2008.
[39] VMware Inc., "Security considerations and disallowing inter-virtual machine transparent page sharing," VMware Knowledge Base 2080735 http://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=2080735, Oct 2014.
[40] Z. Wang and R. B. Lee, "New Cache Designs for Thwarting Software Cache-based Side Channel Attacks," in *ISCA*, San Diego, CA, US, Jun 2007, pp. 494–505.
[41] ——, "A Novel Cache Architecture with Enhanced Performance and Security," in *MICRO*, Como, IT, Nov 2008, pp. 83–93.
[42] Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyper-space: High-speed covert channel attacks in the cloud," in *USENIX Security*, Bellevue, WA, US, 2012, pp. 159–173.
[43] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting, "An exploration of L2 cache covert channels in virtualized environments," in *CCSW*, Chicago, IL, US, Oct 2011, pp. 29–40.
[44] Y. Yarom and N. Benger, "Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack," Cryptology ePrint Archive, Report 2014/140, Feb 2014, http://eprint.iacr.org/.
[45] Y. Yarom and K. Falkner, "FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack," in *USENIX Security*, San Diego, CA, US, Aug 2014, pp. 719–732.
[46] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter, "Homealone: Co-residency detection in the cloud via side-channel analysis," in *Symp. Security & Privacy*, Berkeley, CA, US, May 2011, pp. 313–328.
[47] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *CCS*, Raleigh, NC, US, Oct 2012, pp. 305–316.
[48] ——, "Cross-tenant side-channel attacks in PaaS clouds," in *CCS*, Scottsdale, AZ, US, Nov 2014, pp. 990–1003.