Last-Level Cache Side-Channel Attacks are Practical

***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 hypervisor or on sharing memory between attacker and victim.**

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 [Ristenpart et al., 2009; Wu et al., 2012; Xu et al., 2011] and even conduct side-channel attacks on cryptographic implementations [Yarom and Falkner, 2014; Zhang et al., 2012].

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.

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 [Ristenpart et al., 2009; Wu et al., 2012; Xu et al., 2011]. An exception is the FLUSH+RELOAD attack [Irazoqui et al., 2014; Yarom and Falkner, 2014], which relies on memory sharing to achieve high resolution. Virtualization vendors explicitly advise against sharing memory between VMs [VMware Inc., 2014], and no IaaS provider is known to ignore this advice [Varadarajan et al., 2014], so this attack also fails in practice.

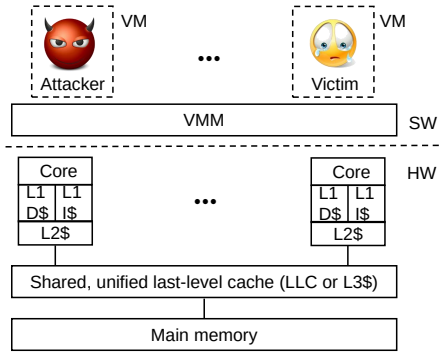


Fig. 1: System model for a multi-core processor

We show that our adaptation of the PRIME+PROBE technique Osvik et al. [2005] 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 the same processor package.

Specifically, we make the following contributions:

- We demonstrate a 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: a novel algorithm for the attacker to probe exactly one cache set without knowing the complicated address virtualization, and using temporal access patterns instead of conventional spatial

access patterns to identify victim's security-critical accesses;

- 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

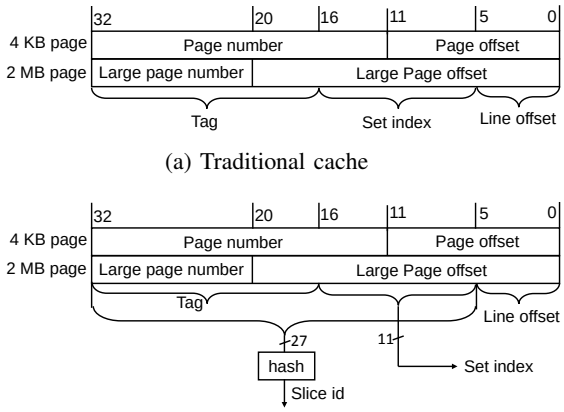
A processes executes in its private virtual address space, composed of *pages*, each representing a contiguous range of addresses. The typical page size is 4 KiB, although processors also support *larger pages*, 2 MiB and 4 GiB on the ubiquitous 64-bit x86 ("x86_64") processor architecture. Each page 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 [VMw, 2008].

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.



(a) Traditional cache

(b) Sliced cache

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

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 a 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 caches, i.e., they are 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

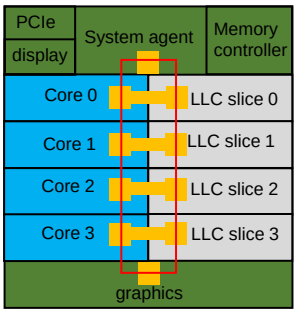


Fig. 3: Ring bus architecture and sliced LLC

bits are 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 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 make place 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. Caches on Intel processors have the *inclusiveness* property: The L_{i+1} cache holds a strict superset of the contents of the L_i [Int, 2012].

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.

Hund et al. [2013] 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 multicore 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 [Osvik et al., 2005], 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 he primes. If V has accessed some 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 attacks on LLC is much harder to achieve than on the L1 caches. We identify the following challenges:

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.

We discuss these challenges in the following subsections, where the meaning and justification of requirements for a successful attack will also become clearer.

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 [Acıiçmez et al., 2010; Brumley and Hakala, 2009; Percival, 2005; Zhang et al., 2012]. This is infeasible to achieve with fine resolution since the LLC is several orders of magnitude larger than L1 caches (several MiB versus several KiB). The solution to this challenge is that we first pinpoint 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 is consistent with the victim performing security-critical accesses. The specific *temporal access patterns* depend on the encryption algorithms used. We delay the detailed discussion on 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 the 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 (subsubsection 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 encryption 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.

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

Secondly, the probe time is slowed by the longer access latency of the LLC (about a factor of 10 for recent Intel processors). Even when all his cache lines remain in a set in the LLC, the attacker will still experience misses in the L1 and L2 caches, when performing a probe of one LLC set, 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 only leads to a latency 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 page 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 much larger than a LLC slice and 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. [2013] 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).

We then use Algorithm 1 to create eviction sets for all the cache sets that share the same set index. This first creates a *conflict set* that contains a maximal subset of the potentially conflicting lines that does not thrash the LLC: exactly W memory lines in the conflicting set map to the same cache set. Note that the conflict set is, effectively, a union of eviction sets for the potentially conflicting lines. The algorithm, then, partitions the conflict set into the individual eviction sets.

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.

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
```

The algorithm creates the conflict set iteratively, adding lines to the conflict set as long as the lines do not conflict with it. Intel's hash function is designed to distribute the selected potentially conflicting lines evenly across all the LLC slices [Int, 2012]. Hence, the set is very likely to contain enough entries to create a maximal conflict set.

To partition the conflict set, the algorithm picks a candidate 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 as the candidate. By iterating over all the members of the conflict set we can find the eviction set for the cahce set of the candidate.

This procedure is repeated for each set index. However, when the number of cores in the processor is

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
```

a power of two, the set index bits are not used for determining the LLC slice, which means determining the slices for a single set index provides enough information for generating the complete complement of eviction sets.

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. [2010]: 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. The 12 `mov` instructions (lines 4 to 15) read the memory lines in the eviction set without loop overheads. Since each `mov` instruction is data-dependent on the previous one, access to the memory lines is fully serialized [Osvik et al., 2005]. 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 [Paoloni, 2010]. However, as noted by Yarom and Falkner [2014], 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 the variations of the probe time.

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 minimise self-evictions, as the oldest line is accessed last [Tromer et al., 2010].

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 timing 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. [2012] 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 [Glover and Grant, 2010]. The sender and the receiver each allocate a buffer of at least the size of the LLC, mapped with 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.

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
VMM	Xen 4.4 (HVM)	VMware ESXi 5.1
Guest OS	Ubuntu 14.04.1 LTS	CentOS 6.5

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 details. 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 pages 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.

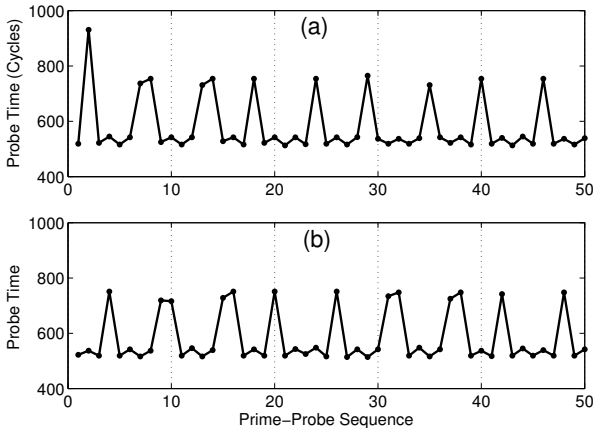


Fig. 4: Sample sequence of receiver's access time on server platform, (a) *set 1*, (b) *set 0*. The sender transmits the sequence "101010...". $T_{mark} = 100$ and $T_{pause} = 3000$ cycles.

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 [Paar and Pelzl, 2010]. 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.

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* [Levenshtein, 1966] 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}.

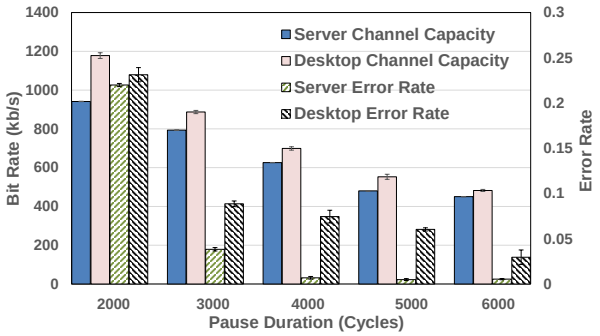


Fig. 5: Channel capacity and error rate of the covert channel.

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 associativities 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 [Wu et al., 2012] for an LLC-based covert channel.

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 [Rivest et al., 1978] and ElGamal [ElGamal, 1985] decryptions, leaking the exponent e may lead to the recovery of the private key.

The square-and-multiply algorithm [Gordon, 1998] 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 $r_i = b^{\lfloor e/2^i \rfloor} \bmod m$ 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: $b^e \bmod m$

$r \leftarrow 1$
for *i from* $n-1$ *downto* 0 **do**
  $r \leftarrow r^2 \bmod m$
  if $e_i = 1$ **then** $r \leftarrow r \cdot b \bmod m$;
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 [Acıiçmez and Schindler, 2008; Yarom and Falkner, 2014; Zhang et al., 2012]. 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

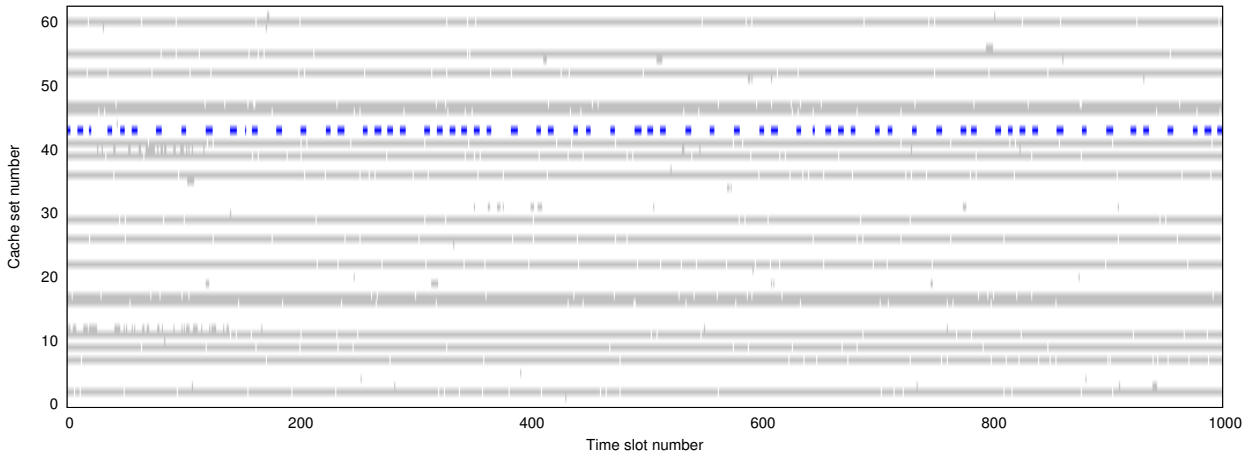


Fig. 6: Traces of cache sets activity. The highlighted trace is for the code of the squaring operation.

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 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.

Note that 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 recent Intel processors, the main requirements being inclusive caches and the availability of large-page mappings. It will also work on other implementations of the square-and-multiply algorithm, and in fact 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 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. 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 consistently, 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 clear bits, or 16–17 time slots for set 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 this line: 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. I.e. the pattern has 6 slots of no activity, followed by five with activity and another six 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 then 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

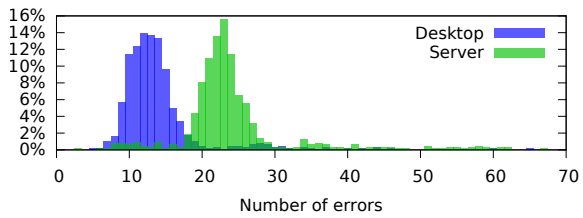


Fig. 7: Distribution of capture errors

With this optimisation, we require in 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 and compare to the ground truth.

Figure 7 shows the distribution of the number of capture errors on the desktop and the server platforms. Out of 817 captured 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. We note that the both the number of failed captures and the number of errors is significantly larger in the server platform. This contrasts with the lower error rate observed for the covert channel on the server platform.

VII. Attacking Sliding-Window Exponentiation

In the previous section we showed how to recover secret-dependent execution paths. We now show that the approach can also be used to observe secret-dependent data access patterns. As an example we use an implementation of the sliding-window exponentiation algorithm [Bos and Coster, 1989].

A. Sliding-window exponentiation

Given an exponent e, the sliding-window representation with a window size S of the exponent is a sequence of windows w_i, each of length $L(w_i)$ bits, where w_i is either 0 or an odd number between 1 and $2^S - 1$ and $1 \leq L(w_i) \leq S$. 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, 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
```

For each odd window value, v, the algorithm precomputes the multiplier b^v, storing it in $g[(v-1)/2]$. It then scans the exponent from the most to the least significant bit, executing a square for each bit and multiplying by the pre-computed window value whenever reaching the least significant bit of a non-zero window. In order to mitigate against recovery of the square-and-multiply sequence with the FLUSH+RELOAD attack [Yarom and Falkner, 2014], GnuPG uses the multiply function to calculate squares.Thus, the GnuPG implementation of the sliding-window algorithm performs a sequence of multiplication operations.

The sequence of squares and multiplies executed by the sliding-window algorithm leaks information about the exponent. Each multiply operation in the exponentiation phase of the algorithm indicates the position of a non-zero window. Knowing the positions of non-zero windows discloses some information on exponent bits: Because the windows are odd, the exponent bits at the positions where multiplications take place are "1".

A second leak is the pattern of access to pre-computed multipliers. By extracting the sequence of square operations and the pre-computed values used in the multiply operations, an attacker can completely recover the exponent [Percival, 2005]. Our attack targets this leak.

B. Overview of the attack

The attack recovers the sequence of multiplications executed, identifying those that implement squaring and

the window value used for the other multiplications. As mentioned, this information is sufficient for recovering the exponent. 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.

For each pre-computed multiplier, $g[i]$, we define a multiplier *use vector* V_i that identifies the multiplication operations in which $g[i]$ is one of the operands. Our attack strategy is to use the PRIME+PROBE technique from Section IV to capture activity in a cache set of each of the pre-computed multipliers $g[i]$ and use this activity data to recover the use vector V_i.

While in theory the strategy is straightforward, the attack has to overcome the following challenges.

1) Unknown multiplier locations: Each decryption executes in a separate process. The implementation uses heap memory (i.e. `malloc()`) for the array, g, of multipliers. While the code of each of these processes tends to use the same physical addresses (and thus cache sets), as a result the victim OS's use of the page cache, the heap locations differ between executions. Hence, multiplier location information obtained during one exponentiation cannot be used for locating the multiplier in subsequent exponentiations.

We overcome this by observing multiple exponentiations to determine activity in each cache set of the LLC. Each observation has a non-zero probability of being a cache set used by a multiplier during the observed exponentiation. Hence, by observing enough exponentiations, our attack can expect to observe the use of all of the pre-computed multipliers. Section VII-G analyzes the number of exponentiations that the attacker needs to observe.

2) Elusive cache use patterns: It is not enough to observe a cache set used by a multiplier. For the observation to be useful, the attacker must know that the observed cache set is used for a multiplier *and* must know which multiplier it observes. The only information the attacker can use is the pattern of access to the cache set. However, each multiplier is only used sporadically during an exponentiation.

To identify the multiplier cache set, we combine several strategies. While we do not know the exact usage pattern, we do have some statistical information on the usage of the pre-computed multipliers (Section VII-D). We use this information to discard observations that do not observe multiplier sets and to identify the specific multiplier when we know that we observe a multiplier.

To amplify the signal and distinguish multiplier cache sets from random noise, we set the number of observed exponentiations such that we are likely to observe multiple uses of each pre-computed multiplier. Uses of the same multiplier will show the same access pattern, by finding similar patterns we can identify the multipliers (Section VII-E).

3) Asynchronous operation: The attack is asynchronous: The attacker does not control the victim and does not know what the victim does and when. The attacker, therefore, needs to build a model of the victim's activity based on the victim's effect on the shared cache. In particular, the attacker does not know when the victim computes the exponentiation, and within each exponentiation, the timing of each multiplication operation. Both are required for the attacker to analyze the activity in the observed cache set.

We find the timing by probing a cache set used for the victim's multiplication code in parallel with any other observation. Identifying the cache set of the multiplication operation is similar to finding the code of the squaring operation in Section VI. We discuss the use of the multiplication cache set in Section VII-C

4) Noise: Other programs and system activity generate noise in the form of activity in observed cache sets. By correlating the data collected in multiple observations, we are able to remove most of the capture noise and recover the use vectors V_i (Section VII-F).

Putting it all together, our attack follows this outline:

1) Find a cache set of the multiplication code.
2) Scan each set in the LLC. Use PRIME+PROBE to collect a trace of activity in both the scanned set and the multiplication cache set.
3) Process each trace to identify the timing of exponentiations and the individual multiplications in each exponentiation. For each exponentiation produce an *activity vector* which identifies the scanned set's activity during each multiplication. Except for noise, we expect the activity vector of a multiplier's cache line to be identical to its use vector.
4) Cluster the activity vectors to find groups of similar vectors.
5) Analyze the clusters to recover the use vectors V_i, and caclulate the exponent.

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.

C. Creating the multiplication activity vectors

As discussed above, the attack collectes parallel traces of two cache sets, the multiplication and the scanned set. We now discuss the process of converting such traces into activity vectors. Recall that an activity vector represents the activity in the scanned cache set

during a single exponentiation, identifying the multiplication operations in the exponentiation during which the scanned cache set shows activity.

We use the information in the trace of the multiplication set to split the trace into exponentiations, and to identify the multiplication operations within each trace. We identify the exponentiations by finding sequences of high activity in the multiplication cache set.

Identifying the multiplication operations in the trace seems straightforward: each consequtive sequence of time slots in which activity is detected in the multiplication cache set corresponds to a single multiplication operation. 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 causedby 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 the multiplication operations. With the multiplication operation identified, we can generate the multiplication activity vector by checking for activity in the scanned cache set during the time slots that correspond to each multiplication operation.

D. Multiplier access patterns

To find the access pattern of each multiplier, we need to look at Algorithm 4. The sequence of multiplications has two phases: first the multipliers are pre-computed, then the exponent is calculated.

The precomputing phase consists of 2^{S-1} multiplications. The first of these multiplications squares b ($g[0]$) to calculate s. The i^{th} multiplication accesses $g[i-1]$ to calculate $g[i]$. Thus, $g[0]$ is accessed in the first two multiplications and the $g[i]$ is accessed in the $(i+2)^{\text{th}}$ multiplication.

Because we do not know the value of the exponent, we have less information about access to the multipliers during the exponentiation phase. We can, however, calculate some statistical data on expected use. GnuPG uses algorithm 14.85 of Menezes et al. [1997] for calculating the windows w_i. For a window size of S, the expected distance between non-zero windows is S. 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 pre-computed multiplier to be used $2^{1-S} \cdot n/(S+1)$ times during the exponentiation.

When choosing the private key, GnuPG uses the so-called Wiener's table to determine the key length to use. For a 3,072-bit ElGamal key, Wiener's table returns a value of 269. GnuPG adds a 50% safety margin, resulting in a key length of 403 bits. For an exponent of 403 bits, GnuPG uses window-size $S = 4$. Hence, we expect each pre-computed multiplier to be accessed $2^{1-4} \cdot 403/(4+1) \approx 10$ times during the exponentiation.

The expected number of multiplications in an exponentiation is 492: 8 for the pre-computation, 403 for the squaring in the exponentiation and 81 for the non-zero windows. Hence, the use of each pre-computed multiplier is sparse, and because the positions in which the multipliers are used are random, it is hard to distinguish the use of the multipliers from sporadic random noise.

E. Clustering the multiplication activity vectors

In the absence of noise, the activity vectors of cache sets used for a multiplier are identical to the use vector of that multiplier, so activity vectors of all cache sets of a 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 identify activity vectors of multiplier activity.

Our attack collects enough activity vectors to have a sufficiently high probability that we captured multiple vectors for each multiplier. To group similar vectors, we use a hierarchical clustering algorithm [Hastie et al., 2009] with the edit distance [Levenshtein, 1966] between the vectors as the measure of similarity.

The clustering algorithm is quadratic in the number of vectors. To reduce the processing time, we only cluster vectors that have an activity level similar to the expected. That is, we remove vectors that show too little or too much activity (less than 5 or over 20 active multiplications). We also remove vectors that do not show activity within the first few multiplications.

F. Recovering the exponent

The final step is recovering the use vectors of the exponentiations. We need to identify the clusters that correspond to multiplier activity and process them to correct capture errors. Unlike previous steps, which are automated, this step requires manual processing.

We find that identifying the clusters of the components is fairly straightforward. When a group of similar vectors is observed, we can easily detect that the activity matches expectations, as discussed in Section VII-D.

To explain how we correct capture errors, we look at a sample cluster in Figure 8, showing a cluster with 16 activity vectors. Each horizontal line represents a vector.

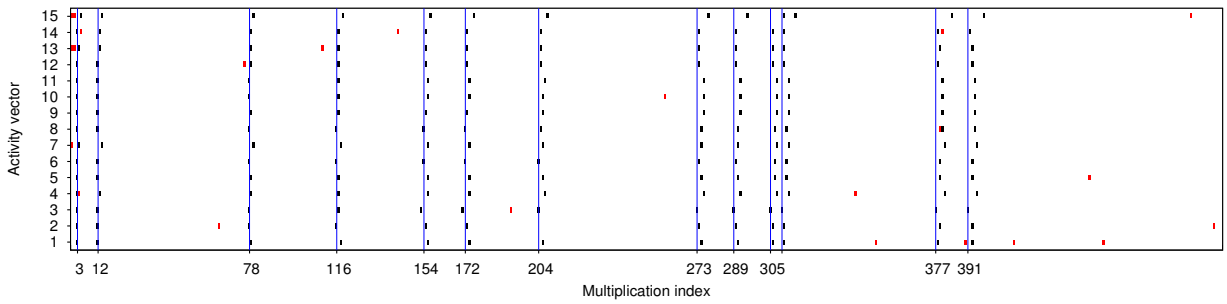


Fig. 8: A cluster of activity vectors

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 use vector 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 vectors in the cluster.

In the figure we can see another noise effect: the further we advance in the exponentiation, the more the activity vectors 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 (Section VII-C), our fix is, evidently, not perfect.

To correct for these errors, we process the vectors from left to right. We decide on the position of each active multiplication, and re-align all the vectors to have the activity in that position. This removes the spurious multiplications as we progress through the vectors.

We note that in most activity vectors, the third one is the first to show cache activity (Figure 8. We can, therefore conclude that this cluster traces activity in $g[1]$. By processing the clusters of all of the pre-computed multipliers we recover the use vectors, which we use to calculate the exponent.

G. Calculating the number of required observations

We have seen that the attack requires multiple activity vectors of cache sets of each pre-computed multiplier. The question that remains is how many exponentiations we need to observe in order to collect enough activity vectors for the attack.

Assuming we observe a fixed number of exponentiations in each cache set, the probability that a randomly chosen observation "hits" a cache set used for a specific multiplier is given by the ratio between the number of cache lines the multiplier occupies and the total number of cache lines. 3,072 bit multipliers occupy 6–7 cache lines. Consequently, a scan that observes one exponentiation in each cache set is expected to yield 6–7 activity vectors for each multiplier.

To be able to use an activity vector of a cache set, the noise level during the observation should not be too high. As shown in Figure 6, some cache sets are constantly active, on our desktop platform, this is about a third of all sets. Thus, we expect to find four or five useable activity vectors for each multiplier in a scan that observes one exponentiation in each cache set.

For effectively identifying multipliers, including redundancy for errors correction, we need 10–15 activity vectors for each multiplier. By collecting observations of four exponentiations in each cache set, the expected number of useable activity vectors in each multiplier is between 16 and 20, and we are very likely to get more than 10 vectors in each.

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 attack 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 the cache set. Due to the larger cache size, we collect more observations on the server platform. We filter over 97% of the observations because they do not match the expected activity of a

multiplier, leaving 700–1000 interesting activity vectors, which we pass to the clustering algorithm. This offline attack takes less than a minute, leaving us with a list of clusters. The average cluster size is around 20 vectors, with a minimum of 5. In all test cases, we require about 10 minutes of manual processing of the clusters to completely recover the use vectors and break the key.

VIII. Related work

A. Prime+Probe

This technique has been used for attacks against several processor caches, including the L1 data cache [Osvik et al., 2005; Percival, 2005; Tromer et al., 2010; Zhang et al., 2012], L1 instruction cache [Acıiçmez, 2007; Acıiçmez and Schindler, 2008; Acıiçmez et al., 2010] and the branch prediction cache [Acıiçmez et al., 2007]. All these caches are core-private, and the attacks exploit either hyper-threading or time multiplexing of the core.

Zhang et al. [2012] 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, and have a much faster attack.

B. LLC based covert channel

Percival [2005] 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. [2009] experiment with L2 covert channels in a cloud environment, achieving a bandwidth of about 0.2 b/s. Xu et al. [2011] 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. [2012] achieve a transfer rate of over 190 kb/s.

Due to the low channel capacity, an LLC based covert channel typically only leaks course-grain information. For example, the attacks of Ristenpart et al. [2009] leak information about co-residency, traffic rates and keystroke timing. Zhang et al. [2011] use an L2 side channel to detect non-cooperating co-resident VMs.

We show a much higher bandwidth LLC based covert channel, and can leak fine-grained information like secret key bits.

C. LLC based side channel attacks

Yarom and Falkner [2014] show that when attacker and victim share memory, e.g. in the form of shared libraries, the technique of Gullasch et al. [2011] can achieve an efficient cross-VM, cross-core, LLC attack. The attack recovers the private key from the square-and-multiply exponentiation of GnuPG 1.4.13 by observing as little as one or two RSA decryptions. The same technique has been used in other scenarios [Benger et al., 2014; Irazoqui et al., 2014; van de Pol et al., 2014; Yarom and Benger, 2014; Zhang et al., 2014].

Our attack does not require shared 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.

IX. Mitigation

A. Fixing GnuPG

One countermeasure is fixing GnuPG to prevent information leaks. One approach is exponent blinding, which splits the exponent into two parts [Clavier and Joye, 2001]. Modular exponentiation is performed on each part which are then combined to obtain the result.

An alternative is a constant-time implementation, that does not contain any conditional statements or memory references that depend on secret data. Techniques for constant-time implementations have been explored, for example, in Bernstein et al. [2012]. 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 [Cock et al., 2014].

However, 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.

B. Avoiding resource contention

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 [Liedtke et al., 1997]. 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. Coloring also requires knowledge of the cache indexing function, which, on modern Intel processors, is undocumented, as discussed in subsubsection II-B2.

STEALTHMEM [Kim et al., 2012] 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 a mechanism, called cache QoS enforcement (CQE), which partitions the cache by ways [Int, 2014]. CQE defines several classes of service (COS), and each COS can be allocated a subset of ways in each cache set. With proper resource allocation for the COS, CQE can ensure that each COS gets a disjoint of the cache, achieving resource isolation, effectively giving each COS a private LLC. At present, a maximum of four COSs are supported, which may not be sufficient in many cloud environments. Further research is required to study the use of CQE and its effectiveness as a countermeasure to our attack.

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 the Partitioned Locked cache (PLcache) proposed in [Wang and Lee, 2007]. However, PLcache is proposed for locking data in the cache, it is not clear how it can be extended to lock instructions.

C. Run-time diversification

Secure cache designs that can randomize the memory-to-cache mapping have been proposed [Wang and Lee, 2007, 2008]. Instead of avoiding, these approaches randomize resource contention, so an attacker cannot extract useful information. These designs have been applied to the L1 data cache without causing performance degradation. However, it is not clear how they perform on the much larger LLC.

Fuzzy time approaches disrupt the timing measurement by adding noise or slowing it down, or reduce the accuracy of the clock [Hu, 1991; Vattikonda et al., 2011]. 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 encryption in GnuPG. The attack is very effectictive, 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 pages to map guest physical memory. Beyond that we make no assumptions on the environment or the victim, other that it repeatedly encrypts 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 hypervisors 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.

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 defence. 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. Hardware support for cache partitioning might be the most promising defence, but whether those mechanisms work in practice remains to be seen.

REFERENCES

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

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.

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.

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.

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.

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.

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

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

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

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.

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

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

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

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

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.

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

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

Intel 64 and IA-32 Architectures Optimization Reference Manual, Intel Corporation, Apr 2012.

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

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.

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.

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

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.

A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, C. Press, Ed. CRC Press, 1997.

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.

C. Paar and J. Pelzl, *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer-Verlag New York Inc, 2010.

G. Paoloni, *How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures*, Intel Corporation, Sep 2010.

C. Percival, "Cache missing for fun and profit," http://www.daemonology.net/papers/htt.pdf, 2005.

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.

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.

E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks in AES, and countermeasures," *J. Cryptology*, no. 2, pp. 37–71, Jan 2010.

J. van de Pol, N. P. Smart, and Y. Yarom, "Just a little bit more," Cryptology ePrint Archive, Report 2014/434, Jun 2014.

V. Varadarajan, T. Ristenpart, and M. Swift, "Scheduler-based defenses against cross-VM side-channels," in *USENIX Security*, San Diego, CA, US, Aug 2014.

B. C. Vattikonda, S. Das, and H. Shacham, "Eliminating fine grained timers in Xen," in *CCSW*, Chicago, IL, US, Oct 2011, pp. 41–46.

Large Page Performance, VMware Inc., Palo Alto, CA, US, 2008.

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.

Z. Wang and R. B. Lee, "New Cache Designs for Thwarting Software Cache-based Side Channel At-

tacks," in *ISCA*, San Diego, CA, US, Jun 2007, pp. 494–505.

——, "A Novel Cache Architecture with Enhanced Performance and Security," in *MICRO*, Como, IT, Nov 2008, pp. 83–93.

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.

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.

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/.

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.

Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter, "Homealone: Co-residency detection in the cloud via side-channel analysis," in *IEEE Symp. Security & Privacy*, Berkeley, CA, US, May 2011, pp. 313–328.

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.

——, "Cross-tenant side-channel attacks in PaaS clouds," in *CCS*, 2014.