Cross Processor Cache Attacks

Gorka Irazoqui
Worcester Polytechnic Institute
girazoqui@wpi.edu

Thomas Eisenbarth
Worcester Polytechnic Institute
teisenbarth@wpi.edu

Berk Sunar
1Worcester Polytechnic Institute
sunar@wpi.edu

ABSTRACT

Multi-processor systems are becoming the de-facto standard across different computing domains, ranging from high-end multi-tenant cloud servers to low-power mobile platforms. The denser integration of CPUs creates an opportunity for great economic savings achieved by packing processes of multiple tenants or by bundling all kinds of tasks at various privilege levels to share the same platform. This level of sharing carries with it a serious risk of leaking sensitive information through the shared microarchitectural components. Microarchitectural attacks initially only exploited core-private resources, but were quickly generalized to resources shared within the CPU.

We present the first fine grain side channel attack that works across processors. The attack does not require CPU co-location of the attacker and the victim. The novelty of the proposed work is that, for the first time the directory protocol of high efficiency CPU interconnects is targeted. The directory protocol is common to all modern multi-CPU systems. Examples include AMD's *HyperTransport*, Intel's *Quickpath*, and ARM's *AMBA Coherent Interconnect*. The proposed attack does not rely on any specific characteristic of the cache hierarchy, e.g. inclusiveness. Note that inclusiveness was assumed in all earlier works. Furthermore, the viability of the proposed covert channel is demonstrated with two new attacks: by recovering a full AES key in OpenSSL, and a full ElGamal key in libgcrypt within the range of seconds on a shared AMD Opteron server.

Keywords

Invalidate+Transfer, Cross-CPU attack, HyperTransport, cache attacks

1. MOTIVATION

Remote servers and cloud computing servers are now more popular than ever due to scalability and low costs. High end users now prefer to remotely access a Virtual Machine or a server that they share with other users rather than buying and maintaining their private hardware. Security plays a crucial role in this scenario since users do not want anyone interfering with their applications. Modern Operating Systems (OSs) now implement permissions and even more advanced sandboxing techniques such as running tasks in virtual machines that ensure isolation and avoid userspace interference.

While sandboxing has been proven effective at the software level, information dependent on a potential victim's activity can still be leaked at the lower layers of the implementation stack, i.e. via shared hardware resources. If the leakage is strong enough, an attacker observing the leakage might be able to steal sensitive fine grain information such as cryptographic keys. To elaborate, *microarchitectural side channel attacks* take advantage of the existing hardware leakage in modern hardware architectures. By following a fundamental computer architecture design and optimization principle, *"make the common case fast"*, computer architects have created machines where access and execution times vary depending on the processed data. Microarchitectural attacks exploit this data dependent behavior as a covert channel from which they infer sensitive information. One of the most popular covert channels exploited in modern processors is the cache, due to its granularity and the lack of any access restrictions[1]. Although their applicability was questioned for a long time, they found an ideal scenario with modern cloud computing and remote server technologies. In fact, these services are designed to host more than one user concurrently in the same server.

Microarchitectural attacks are impervious to access boundaries established at the software level. First starting in the native execution case (a spy process running alongside the victim in the same user space), researchers have shown the effectiveness of such attacks under gradually more restrictive execution environments. For example, the attacks were carried out first inside a single VM and later across VMs [39] with tighter constrains on the attackers privileges. Not in vain, researches have shown to even recover cryptographic keys across VMs, demonstrating the big threat that they can imply.

The first practical implementations of microarchitectural covert channels were studied in 2005, taking advantage of L1 cache

[1]Eliminating caches slows down modern processes by up to 75 times! [17]

leakages [13, 29]. Later, more covert channels like the Branch Prediction Unit (BPU) proved to be as dangerous as caches [11]. These earlier works focused on the native (non-virtualized) setting. Such attacks were largely dismissed as being unrealistic, especially by the industry. As the logic goes, if an attacker was able to smuggle a spy process into the victim's execution space, he had already gained access to significant resources so there was no point in further carrying out a more complicated low-level microarchitectural attack. Therefore, the spy process was not considered to carry any significant meaning in the real world. With the emergence of the compute cloud where unknown and independent parties—the victim and the attacker—run alongside in VMs on the same physical machine, concerns were renewed. However, citing the co-location problem again such concerns were dismissed by cloud service providers. It was not until 2009 when Ristenpart et al. [31] showed the possibility of co-locating two VMs and further extracting keystrokes across VM boundaries in the Amazon EC2 commercial cloud. This study not only demonstrated the grave risks posed by microarchitectural side channel attacks on user's privacy but also reignited research in this direction.

Recent studies uncovered a variety of stronger covert channels along with attacks exploiting them. The most popular leakage is caused by the Last Level Cache (LLC), a resource that is shared across cores and thus works across different cores on the same processor. Attacks exploiting the LLC have proven to recover various types of sensitive information [36, 25, 38] ranging from cryptographic keys to the number of items in a victim's shopping cart. Industry reacted by disabling features that were enabled prior to the discovery of the LLC cache covert channel [7] and by hardening cryptographic libraries [2].

More recently, researchers have also shown that there exists hardware leakage in the memory bus channel [34]. Although this leakage was used to achieve targeted co-location in commercial clouds [33, 35], its ability to recover fine grain information is still an open question. However, the memory bus channel is unique in that it works across processors: it can be used to detect co-location even on multi-CPU systems. This is very relevant, as modern computer systems not only have an increasing number of cores per processor, but also come with an increasing number of processors per system. This trend is not restricted to servers, in fact even mobile platforms now frequently come with at least two separate processors [16].

In this work we present the first cache based cross-processor attack by introducing a new microarchitectural covert channel.

Our Contribution

We present the exploitation of a new covert channel based on the cache coherency protocols implemented in modern processors and multiprocessor systems. In order to retrieve fine grain information from this covert channel we introduce the `Invalidate and Transfer` technique, which relies on shared data between different cores in a system, irrespective of their placement within or across processors. While previously presented cache attacks relied on the inclusiveness of the LLC, our new spy process does not require any special characteristic of the cache hierarchy. Thus it succeeds in those processors where prior cache attacks have not been shown to work, e.g., AMD servers with exclusive caches. Furthermore, we present the first attack that is able to retrieve fine grain information across CPUs in multiple socket servers, thereby we do not rely on CPU co-location to execute the attack. We present the viability of the attack by attacking a software `OpenSSL` implementation of the AES symmetric cipher and a square and multiply `libgcrypt` implementation of the ElGamal public key scheme.

In summary, this work

- introduces the first cross-CPU fine grain side channel attack, i.e., we do not need CPU co-location between attacker and victim to obtain the leakage information
- shows for the first time a directory protocol based attack that does not rely in any specific characteristic of the cache hierarchy. Therefore our attack applies on servers that have not been shown to be vulnerable to microarchitectural side channel attacks such as AMDs Opteron series processors or ARM processors.
- the attack exploits data dependent timing variations in AMD's *HyperTransport*, Intel's *Quickpath* and could be applied to ARM's *AMBA Coherent Interconnect* as well.
- demonstrates the power of the new side channel by recovering a full AES key and a full ElGamal key within a few of seconds.

The rest of the study is divided as follows. We first review the related work in Section 2, we discuss the background knowledge Section 3 and Section 4. The new attack is presented in Section 5 and in 6. The Results are presented in Section 7, before discussing the viability in other scenarios 8. We conclude in Section 9.

2. RELATED WORK

Microarchitectural attacks have been studied for more than 20 years Originally, covert channels like the cache were studied theoretically, as in [23, 23, 30]. Tsunoo et al. [32] were the first ones to practically obtain leakage from a cache side channel attack against DES in 2003. In 2004, with the popularity of the AES cipher, two new cache-based attacks were presented. The first one was implemented by Bernstein [13] by exploiting microarchitectural timing differences observed for different look up table positions. At the same time, Osvik et al. [29] proposed two new spy processes named `Evict + Time` and `Prime and Probe`. While the first one modifies the state of the cache between identical encryptions, the latter one fills the entire cache with attackers data before the encryption and checks which parts of the cache have been used after the encryption. All the attacks, with different number of encryptions required, achieved the full recovery of the AES encryption key.

The proposed attacks motivated the community to analyze the potential threat of microarchitectural side channel attacks. For instance, Bonneau and Mironov further exploited the cache collision attacks and implement a last round side channel attack on AES [14]. Shortly later Acıiçmez et al. exploited similar collision attacks in the first and second round of the AES cipher [9]. Again Acıiçmez et al. implemented the first attack against RSA, by monitoring instruction cache accesses instead of data cache accesses [10].

Although the cache became the main targeted microarchitectural side channel studied by researchers, Acıiçmez et al. [11] also considered the Branch Prediction Unit (BPU) as an exploitable source of leakage for non-constant execution flow software. In particular, they recovered a full RSA key by analyzing the outcome of vulnerable internal branches.

However, due to the raising popularity of multi-core processors and cloud computing systems, microarchitectural side channel attacks were dismissed for a long time due to their limited applicability. Indeed most of the previously proposed attacks targeted core private resources within the same Operating System.

It was in 2009 when Ristenpart et al. [31] proposed mechanisms to achieve co-location in the Amazon EC2 cloud, bringing a whole new scenario where microarchitectural attacks could realistically be applied. In the following years, researchers started exploiting the scenario opened by [31]. Zhang et al. [37] proposed in 2011 a mechanism to detect whether a user is co-residing with any potential attacker in the same core, while in 2012 again Zhang et al. [39] proposed the first successful fine grain side channel attack in the cloud by recovering an ElGamal encryption key. At the same time, Gullasch et al. [21] proposed a new attack on AES that would later acquire the name of `Flush and Reload`.

However, it was in 2013 when the first cross-core side channel attack was studied. Utilizing the same technique as in [21] Yarom et al. [36] studied how the `Flush and Reload` attack applied in the LLC can recover a full RSA key even with VMs that are not co-located in the same core. Shortly later Irazoqui et al. [26] presented a new attack on the AES cipher across VMs, again using the `Flush and Reload` spy process.

The `Flush and Reload` attack was later expanded by a wide range of attacks [12, 27, 38, 20, 19, 22], going from PaaS cloud attacks to cache template attacks. However, this attack is only applicable in the cloud if deduplication is enabled, restricting thereby the applicability of it. In order to overcome this issue, Liu et al. and Irazoqui et al. [18, 25] proposed a new attack in the LLC based on the `Prime and Probe` attack that did not require deduplication, recovering RSA and AES keys respectively. Recently, this attack has been expanded by Inci et al. [24] by showing its applicability in the public Amazon EC2 cloud and by Oren et al. [28] by implementing it in javascript and showing its applicability in web browser scenarios.

3. BACKGROUND

In this section we discuss the strengths and weaknesses of current microarchitectural attacks and discuss why the most powerful one, based on LLC leakage, has not yet been exploited on AMD CPUs.

3.1 Microarchitectural Covert Channels

In the last 10 years many studies have identified and exploited different microarchitectural attacks under very different scenarios. Since the chronological order of these studies has already been discussed in section 2, this section aims at describing the different covert channels already exploited and their applicability.

- **L1 Cache:** The L1 cache was one of the first microarchitectural covert channels that was exploited. It is usually divided into a separate data cache and an instruction cache, each usually several kB (often 32 or 64kB) in size. One of the advantages of the L1 cache is that the attacker can isolate his data-related attacks from the instructions and vice versa. Furthermore, an attacker can monitor the entire L1 cache with a reasonable timing resolution due to its small size. However, distinguishing accesses from the L1 and L2 became a difficult task in modern processors, since they only differ in a few cycles. Furthermore, the L1 (and usually L2) caches are a core-private resource, and therefore are only exploitable when victim and attacker are colocated in the same core. As modern processors incorporate more and more cores in their systems, the applicability of L1 cache attacks reduces drastically.

- **BPU:** The BPU is another microarchitectural component that has been proved to leak information. In order to gather this information, the attacker needs to know whether the executed branch has been mispredicted or not. Thus, having knowledge about how the BPU predicts the branches and about the size of the BTB is crucial to run this kind of attacks. Unfortunately, this information is not released anymore in modern processors. Furthermore, the time difference between a well predicted and a mispredicted branch is not bigger than a few cycles in modern processors. Moreover, the BPU is a core-private resource like the L1 cache, and therefore can only be exploitable in the case of core co-residency.

- **LLC:** The LLC is a recently discovered covert channel that provides many advantages over the previous ones. First, it is a shared resource between cores, and therefore core co-residency is not needed anymore. Second, LLC side channel attacks distinguish between accesses from the LLC and accesses from the memory. In contrast to the previous side channel attacks, distinguishing LLC from memory accesses can be done with a low error rate, since usually they differ in a few tens of cycles. However, these attacks have thus far only been applied to processors where the LLC is inclusive, i.e., for caches where data in the L1 cache is also present in the LLC.

- **Memory Bus:** The memory bus is a covert channel that was discovered in [34] and was later exploited by Varadarajan et al. [33] and Zhang et al. [35] to establish a covert channel in commercial clouds. The method exploits the atomic instruction handling of the CPU and locks the memory bus. Using this lock to send and receive signals, it is possible to send messages covertly, breaking the sandboxing techniques in commercial clouds even across CPUs. Although the covert channel is strong enough to detect co-location, it does not give fine grain information as the previous channels described.

3.2 Why Nobody Attacks AMD Processors

Over the last few years, many cross-core side channel attacks have been introduced to target Intel processors. But none have considered attacking other kinds of servers. Indeed, the utilized covert channels make use of specific characteristics that Intel processors feature. For example, the proposed LLC attacks take advantage of the inclusive cache design in Intel processors. Furthermore, they also rely on the fact that the LLC is shared across cores. Therefore these attacks succeed only when the victim and the attacker are co-located on the same CPU.

These characteristics are not observed in other CPUs, e.g. AMD or ARM. This work focuses on AMD, but the same technique should also succeed in ARM processors, as discussed in Section 8.2. In this sense, AMD servers present two main complications that prevents application of existing side channel attacks:

- AMD tends to have more cores per CPU in high end servers compared to Intel. Indeed, high end AMD servers commonly incorporate 48-cores. The large number of cores reduces the chance of being co-located, i.e. sharing the core with a potential victim. This fact reduces the applicability of core-private covert channels such as L1-Cache and BPU based attacks.
- LLCs in AMD are usually *exclusive* or *non-inclusive*. The former does not allocate a memory block in different level caches at the same time. That is, data is present in only one level of the cache hierarchy. Non-inclusive caches show neither inclusive or exclusive behavior. This means that any memory access will fetch the memory block to the upper level caches first. However, the data can be evicted in the outer or inner caches independently. Hence, accesses to L1 cache cannot be detected by monitoring the LLC, as it is possible on Intel machines.

Hence, to perform a side channel attack on AMD processors, both of these challenges need to be overcome. Here we present a covert channel that is immune to both complications. The proposed attack is the first side channel attack that works across CPUs that feature non-inclusive or exclusive caches.

4. CACHE COHERENCE PROTOCOLS

In order to ensure coherence between different copies of the same data, systems implement cache coherence protocols. In the multiprocessor setting, the coherency between shared blocks that are cached in different processors (and therefore in different caches) also needs to be maintained. The system has to ensure that each processor accesses the most recent value of a shared block, regardless of where that memory block is cached. The two main categories of cache coherence protocols are *snooping based protocols* and *directory based protocols*. While snooping based protocols follow a decentralized approach, they usually require a centralized data bus that connects all caches. This results in excessive bandwidth need for systems with an increasing number of cores. Directory-based protocols, however, enable point-to-point connections between cores and directories, hence follow an approach that scales much better with an increasing number of cores in the system. We put our focus in the latter one, since it is the prevailing choice in current multiprocessor systems. The directory keeps track of the state of each of the cached memory blocks. Thus, upon a memory block access request, the directory will decide the state that the memory block has to be turned into, both in the requesting node and the *sharing* nodes that have a cached copy of the requested memory block. We analyze the simplest cache coherence protocol, with only 3 states, since the attack that is implemented in this study relies on read-only data. Thus, the additional states applied in more complicated coherency protocols do not affect the flow of our attack.

We introduce the terms `home node` for the node where the memory block resides, `local node` for the node requesting access to the memory block, and `owner node` referring a node that has a valid copy of the memory block cached. This leads to various communication messages that are summarized as follows:

- The memory block cached in one or more nodes can be in either `uncached` state, `exclusive/modified` or `shared`.
- Upon a read hit, the `local node's` cache services the data. In this case, the memory block maintains its state.
- Upon a read miss, the `local node` contacts the home node to retrieve the memory block. The directory knows the state of the memory block in other nodes, so its state will be changed accordingly. If the block is in `exclusive` state, it goes to `shared`. If the block is in `shared` state, it maintains it. In both cases the `local node` then becomes an owner and holds a copy of the shared memory block.
- Upon a write hit, the `local node` sets the memory block to exclusive. The `local node` communicates the nodes that have a cached copy of the memory block to invalidate or to update it.
- Upon a write miss, again the the `home node` will service the memory block. The directory knows the nodes that have a cached copy of the memory block, and therefore sends them either an update or an invalidate message. The `local node` then becomes an owner of the `exclusive` memory block.

In practice, most cache coherency protocols have additional states that the memory block can acquire. The most studied one is the MESI protocol, where the `exclusive` state is divided into the `exclusive` and `modified` states. Indeed, a memory block is `exclusive` when a single node has a clean state of the memory block cached. However, when a cached memory block is modified, it acquires the `modified` state since it is not consistent with the value stored in the memory. A write back operation would set the memory block back to the `exclusive` state.

The protocols implemented in modern processors are variants of the MESI protocol, mainly adding additional states. For instance, the Intel i7 processor uses a MESIF protocol, which adds the additional `forward` state. This state will designate the sharing processor that should reply to a request of a shared memory block, without involving a memory access operation. The AMD Opteron utilizes the MOESI protocol with the additional `owned` state. This state indicates that the memory block is owned by the corresponding cache and is out-of-date with the memory value. However, contrary to the MESI protocol where a transition from `modified` to `shared` involves a write back operation, the node holding the `owned` state memory block can service it to the sharing nodes without writing it back to memory. Note that both the MESIF and MOESI protocol involve a cache memory block forwarding operation. Both the `owned` and the `forward` state suggest that a cache rather than a DRAM will satisfy the reading request. If the access time from cache differs from regular DRAM access times, this behavior becomes an exploitable covert channel.

4.1 AMD HyperTransport Technology

Cache coherency plays a key role in multi-core servers where a memory block might reside in many core-private caches in the same state or in a modified state. In multiple socket servers, this coherency does not only have to be maintained within a processor, but also across CPUs. Thus, complex technologies are implemented to ensure the coherency in the system. These technologies center around the cache directory protocols explained in section 3. The *HyperTransport* technology implemented by AMD processors serves as a good example. To save space here we only focus on the features relevant to the new proposed covert channel. A detailed explanation can be found in [15, 3].

The *HyperTransport* technology reserves a portion of the LLC to act as directory cache in the directory based protocol. This directory cache keeps track of the cached memory blocks present in the system. Once the directory is full, one of the previous entries will be replaced to make room for a new cached memory block. The directory *always* knows the state of any cached memory block, i.e., if a cache line exists in any of the caches, it must also have an entry in the directory. Any memory request will go first through the home node's directory. The directory knows the processors that have the requested memory block cached, if any. The home node initiates in parallel both a DRAM access and a *probe filter*. The probe filter is the action of checking in the directory which processor has a copy of the requested memory block. If any node holds a cached copy of the memory block, a directed probe against it is initiated, i.e., the memory block will directly be fast forwarded from the cached data to the requesting processor. A `directed probe` message does not trigger a DRAM access. Instead, communications between nodes are facilitated via HyperTransport links, which can run as fast as 3 GHz. Figure 1 shows a diagram of how the HyperTransport links directly connect the different CPUs to each other avoiding memory node accesses. Although many execution patterns can arise from this protocol, we will only explain those relevant to the attack, i.e. events triggered over over read-only blocks which we will elaborate on later. We assume that we have processors A and B, refereed to as P_a and P_b, that share a memory block:

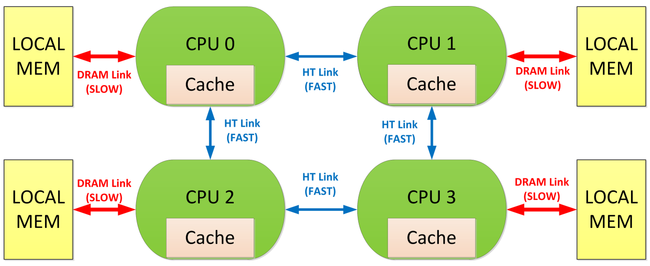


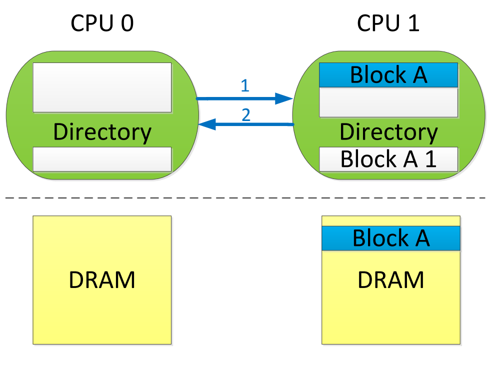
Figure 1: DRAM accesses vs Directed probes thanks to the HyperTransport Links

- If P_a and P_b have the same memory block cached, upon a modification made by P_a, *HyperTransport* will notify P_b that P_a has the latest version of the memory block. Thus, P_a will have to update its version of the block to convert the `shared` block into a `owned` block. Upon a new request made by P_b, *HyperTransport* will transfer the updated memory block cached in P_a.

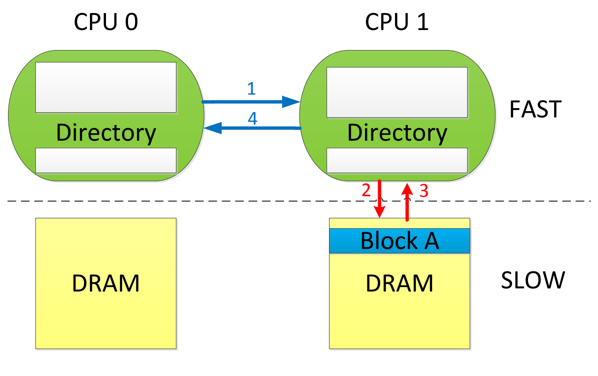
- Similarly, upon a cache miss in P_a, the home node will send a probe message to the processors that have a copy of the same shared memory block, if any. If, for instance, P_b has it, a `directed probe` message is initiated so that the node can service the cached data through the hypertransport links. Therefore, *HyperTransport* reduces the latency of retrieving a memory block from the DRAM by also checking whether someone else maintains a cached copy of the same memory block. Note that this process does not involve a writeback operation.

- When a new entry has to be placed in the directory of P_a, and the directory is full, one of the previously allocated entries has to be evicted to make room for the new entry. This is referred as a `downgrade probe`. In this case, if the cache line is dirty a writeback is forced, and an invalidate message is sent to all the processors (P_b) that maintain a cached copy of the same memory block.

In short, *HyperTransport* reduces latencies that were observed in previously implemented cache coherency protocols by issuing `directed probes` to the nodes that have a copy of the requested memory block cached. The HyperTransport links ensure a fast transfer to the requesting node. In fact, the introduction of HyperTransport links greatly improved the performance and thus viability of multi-CPU systems. Earlier multi-CPU systems relied on broadcast or directory protocols, where a request of a exclusive cached memory block in an adjacent processor would imply a writeback operation to retrieve the up-to-date memory block from the DRAM.



(a) HTLink: fast access



(b) DRAM: slow access

Figure 2: Comparison of a directed probe access across processors: probe satisfied from CPU 1's cache directly via HTLink (a) vs. probe satisfied by CPU 1 via a slow DRAM access (b).

4.2 Intel QuickPath Interconnect Technology

In order to maintain cache coherency across multiple CPUs Intel implements a similar technique to AMD's *HyperTransport* called *Intel QuickPath Interconnect* (QPI) [1, 5]. Indeed, the later one was designed five years latter than the first one to compete with the existing technology in AMD processors. Similar to *HyperTransport*, QPI connects one or more processors through high speed point-to-point links as fast as 3.2GHz. Each processor has a memory controller on the same die to make to improve the performance. As we have already seen with AMD, among other advantages, this interface efficiently manages the cache coherence in the system in multiple processor servers by transferring shared memory blocks through the QPI high speed links. In consequence, the proposed mechanisms that we later explain in this paper are also applicable in servers featuring multi-CPU Intel processors.

5. A NEW CROSS-CPU COVERT CHANNEL

In this section we present a new covert channel based on cache coherency technologies implemented in modern processors. In particular, we focus on AMD processors, which have exclusive caches that in principle are invulnerable to cache side channel attacks although the results can be readily applied to multi-CPU Intel processors as well. In summary,

- We present the first cross-CPU side channel attack, i.e., we show that core co-location is not needed in multi-CPU servers to obtain fine grain information.
- We present a new covert channel that utilizes directory based cache coherency protocols to extract sensitive information.
- We show that the new covert channel succeeds in those processors where cache attacks have not been shown to be possible before, e.g. AMDs exclusive caches.
- We demonstrate the feasibility of our new side channel technique by mounting an attack on a T-table based AES and a square and multiply implementation of El-Gamal schemes.

5.1 Invalidate + Transfer attack

We propose a new spy process that takes advantage of leakages observed in the cache coherency protocol with memory blocks shared between many processors/cores. The spy process does not rely on specific characteristics of the cache hierarchy, like inclusiveness. In fact, the spy process works even across co-resident CPUs that do not share the same cache hierarchy. From now on, we assume that the victim and attacker share the same memory block and that they are located in different CPUs or in different cache hierarchies in the same server.

The spy process is executed in three main steps, which are:

- **Invalidate step:** In this step, the attacker invalidates a memory block that is in his own cache hierarchy. If the invalidation is performed in a shared memory block cached in another cache processors cache hierarchy, the *HyperTransport* will send an invalidate message to them. Therefore, after the invalidation step, the memory block will be invalidated in all the caches that have the same memory block, and this will be uncached from them. This invalidation can be achieved by specialized instructions like `clflush` if they are supported by the targeted processors, or by priming the set where the memory block resides in the cache directory.
- **Wait step:** In this step, the attacker waits for a certain period of time to let the victim do some computation. The victim might or might not use the invalidated memory block in this step.
- **Transfer step:** In the last step, the attacker requests access to the shared memory block that was invalidated. If any processor in the system has cached this memory block, the entry in the directory would have been updated and therefore a `direct probe` request will be sent to the processor. If the memory block was not been used, the home directory will request a DRAM access to the memory block.

The system experiences a lower latency when a `direct probe`

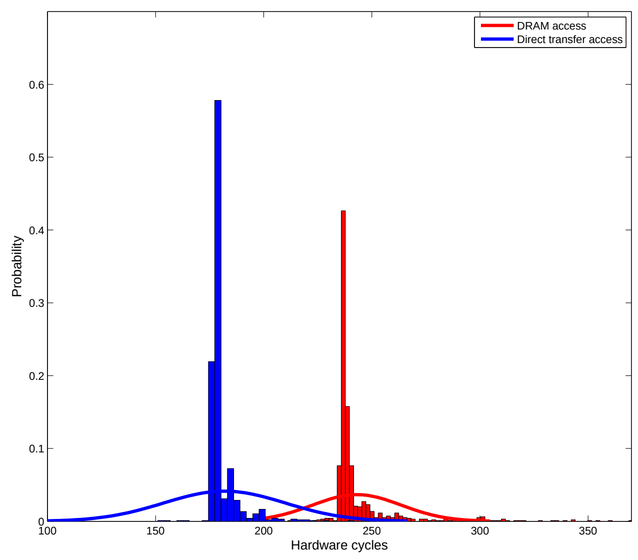


Figure 3: Timing distribution of a memory block request to the DRAM (red) vs a block request to a co-resident processor(blue) in a AMD opteron 6168. The measurements are taken from different CPUs. Outliers above 400 cycles have been removed

is issued, mainly because the memory block is issued from another processors cache hierarchy. This is graphically observed in Figure 2. Figure 2(a) shows a request serviced by the *HyperTransport* link from a CPU that has the same memory block cached. In contrast, Figure 2(b) represents a request serviced by a DRAM access. This introduces a new leakage if the attacker is able to measure and distinguish the time that both actions take. This is the covert channel that will be exploited in this work. We use the `RDTSC` function which accesses the time stamp counter to measure the request time. In case the `RDTSC` function is not available from user mode, one can also create a parallel thread incrementing a shared variable that acts as a counter. We also utilize the `mfence` instruction to ensure that all memory load/store operations have finished before reading the time stamp counter.

The timing distributions of both the DRAM access and the directed transfer access are shown in Figure 3, where 10,000 points of each distribution were taken in a 48-core 4 CPU AMD Opteron 6168. The x-axis represents the hardware cycles, while the y-axis represents the density function. The measurements are taken across processors. The blue distribution represents a `directed probe` access, i.e., a co-resident CPU has the memory block cached, whereas the red distribution represents a DRAM access, i.e., the memory block is not cached anywhere. It can be observed that the distributions differ in about 50 cycles, fine grain enough to be able to distinguish them. However, the variance in both distributions is very similar, in contrast to LLC covert channels. Nevertheless, we obtain a covert channel that works across CPUs and that does not rely on the inclusiveness property of the cache.

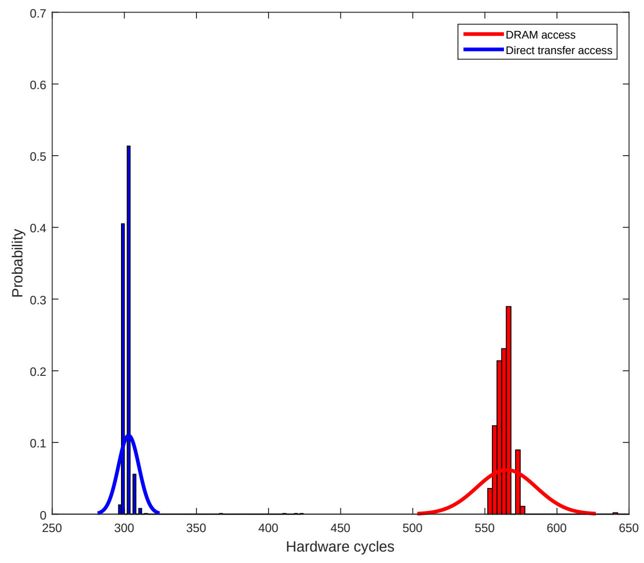


Figure 4: Timing distribution of a memory block request to the DRAM (red) vs a block request to a co-resident core(blue) in a dual core Intel E5-2609. The measurements are taken from the same CPU. Outliers above 700 cycles have been removed

We also tested the viability of the covert channel in a dual socket Intel Xeon E5-2609. Intel utilizes a similar technique to the *HyperTransport* technology called `Intel Quick Path Interconnect`. The results for the Intel processor are shown in Figure 4, again with processes running in different CPUs. It can be observed that the distributions are even more distinguishable in this case.

6. EXPLOITING THE NEW COVERT CHANNEL

In the previous section, we presented the viability of the covert channel. Here we demonstrate how one might exploit the covert channel to extract fine grain information. More concretely, we present two attacks:

- a symmetric cryptography algorithm, i.e. table based `OpenSSL` implementation of AES, and
- a public key algorithm, i.e. a square-and-multiply based `libgcrypt` implementation of the ElGamal scheme.

6.1 Attacking Table based AES

We test the granularity of the new covert channel by mounting an attack in a software implementation of AES, as in [26]. We use the C OpenSSL reference implementation, which uses 4 different T-tables along 10 rounds for AES-128. The first 9 rounds is composed of 4 main operations: `AddRoundKey`,`Subbbytes`,`ShiftRows`,`Mixcolumns`. The last round executes the same operations except the `Mixcolumns` operation. Thus, as in [26], we mount a last round attack, i.e., we assume that the ciphertext is known to the attacker.

In the attack, we monitor a memory block belonging to each one of the T-tables. Each memory block contains 16 T-Table positions and it has a certain probability, 8% in our particular case, of not being used in any of the 10 rounds of an encryption. Thus, applying our `Invalidate + Transfer` attack and recording the ciphertext output, we can know when the monitored memory block *has not* been used. For this purpose we `invalidate` the memory block before the encryption and try to probe it after the encryption. In a noise free scenario, the monitored memory block will not be used for 240 ciphertext outputs with 8% probability, and it will not be used for the remaining 16 ciphertext with 0% probability (because they directly map through the key to the monitored T-table memory block). Although microarchitectural attacks suffer from different microarchitectural sources of noise, we expect that the `Invalidate + Transfer` can still distinguish both distributions.

Once we know the ciphertext values belonging to both distributions, we can apply the equation:

$$K_i = T[S_j] \oplus C_i$$

to recover the key. Since the last round of AES involves only a Table look up and a XOR operation, knowing the ciphertext and the T-table block position used is enough to obtain the key byte candidate that was used during the last AES round. Since a cache line holds 16 T-table values, we XOR each of the obtained ciphertext values with all the 16 possible T-table values that they could map to. Clearly, the key candidate will be a common factor in the computations with the exception of the observed noise which is eliminated via averaging. As the AES key schedule is revertible, knowing one of the round keys is equivalent to knowing the full encryption key.

6.2 Attacking Square and Multiply ElGamal Decryption

We test the viability of the new side channel technique with an attack on a square and multiply `libgcrypt` implementation of the public key ElGamal algorithm, as in [39]. An ElGamal encryption involves a cyclic group of order p and a generator g of that cyclic group. Then Alice chooses a number $a \in \mathbb{Z}_p^*$ and computes her public key as the 3-tuple (p, g, g^a) and keeps a as her secret key.

To encrypt a message m, Bob first chooses a number $b \in \mathbb{Z}_p^*$ and calculates $y_1 = g^b$ and $y_2 = ((g^a)^b) * m$ and sends both to Alice. In order to decrypt the message, Alice utilizes her secret key a to compute $((y_1)^{-a}) * y_2$. Note that, if a malicious user recovers the secret key a he can decrypt any message sent to Alice.

Our target will be the y_1^{-a} that uses the square and multiply technique as the modular exponentiation method. It bases its procedure in two operations: a square operation followed by a modulo reduction and a multiplication operation followed by a modulo reduction. The algorithm starts with the intermediate state $S = b$ being b the base that is going to be powered, and then examines the secret exponent a from the most significant to the least significant bit. If the bit is a 0, the intermediate state is squared and reduced with the modulus. If in the contrary the exponent bit is a 1, the intermediate state is first squared, then it is multiplied with the base b and then reduced with the modulus. Algorithm 1 shows the entire procedure.

Algorithm 1 Square and Multiply modular exponentiation

```
Input: Ciphertext $c \in \mathbb{Z}_N$, Exponent $a$
Output: $c^d \mod N$
$a_b = bitwise(a)$                ▷ Convert exponent $a$ to bit string
$S = c j = len(a)$                ▷ Exponentiation Step
while $j > 0$ do
    $S = S^2 \mod N$
    if $e_j == 1$ then
        $S = S * c \mod N$
    end if
    $j = j - 1$
end while
return $S$
```

As it can be observed the algorithm does not implement a constant execution flow, i.e., the functions that will be used directly depend on the bit exponent. If the square and multiply pattern is known, the complete key can be easily computed by converting them into ones and zeros. Indeed, our `Invalidate + Transfer` spy process can recover this information, since functions are stored as shared memory blocks in cryptographic libraries. Thus, we mount an attack with the `Invalidate + Transfer` to monitor when the square and multiplication functions are utilized.

7. EXPERIMENT SETUP AND RESULTS

In this section we present the test setup in which we implemented and executed the `Invalidate+Transfer` spy process together with the results obtained for the AES and ElGamal attacks.

7.1 Experiment Setup

In order to prove the viability of our attack, we performed our experiments on a 48-core machine featuring four 12-core AMD Opteron 6168 CPUs. This is an university server which has not been isolated for our experiments, i.e., other users are utilizing it at the same time. Thus, the environment is a realistic scenario in which non desired applications are running concurrently with our attack.

The machine runs at 1.9GHz, featuring 3.2GHz HyperTransport links. The server has 4 AMD Opteron 6168 CPUs, with 12 cores each. Each core features a private 64KB 2-way L1 data cache, a private 64KB L1 instruction cache and a 16-way 512KB L2 cache. Two 6MB 96-way associative L3 caches—each one shared across 6 cores—complete the cache hierarchy. The L1 and L2 caches are core-private resources, whereas the L3 cache is shared between 6 cores. Both the L2 and L3 caches are exclusive, i.e., data can be allocated in *exactly one* cache level at a time. This is different to the inclusive LLC where most of the cache spy processes in literature have been executed.

The attacks were implemented in a RedHat enterprise server running the linux 2.6.23 kernel. The attacks do not require root access to succeed, in fact, we did not have sudo rights on this server. Since ASLR was enabled, the targeted functions addresses were retrieved by calculating the offset with respect to the starting point of the library. All the experiments were performed across CPUs, i.e., attacker and victim

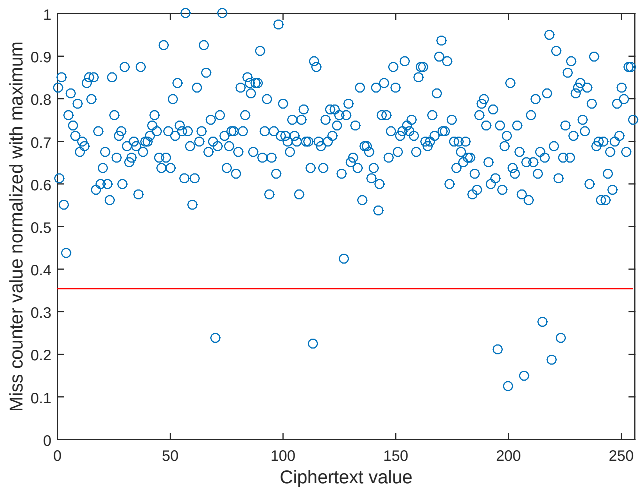


Figure 5: Miss counter values for each ciphertext value, normalized to the average

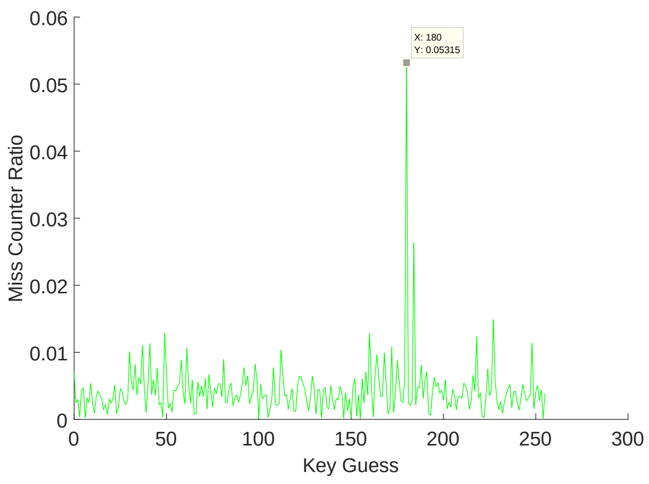


Figure 6: Correct key byte finding step, iterating over all possible keys. The maximum distance is observed for the correct key

do not reside in the same CPU and do not share any LLC. To ensure this, we utilized the `taskset` command to assign the CPU affinity to our processes.

Our targets were the AES C reference implementation of OpenSSL and the ElGamal square and multiply implementation of libgcrypt 1.5.2. The libraries are compiled as shared, i.e., all users in the OS will use the same shared symbols. In the case of AES we assume we are synchronized with the AES server, i.e., the attacker sends plaintext and receives the corresponding ciphertexts. As for the ElGamal case, we assume we are not synchronized with the server. Instead, the attacker process simply monitors the function until valid patterns are observed, which are then used for key extraction.

7.2 AES Results

As explained in Section 6, in order to recover the full key we need to target a single memory block from the four T-tables. However, in the case that a T-table memory block starts in the middle of a cache line, monitoring only 2 memory blocks is enough to recover the full key. In fact, there exists a memory block that contains both the last 8 values of T0 and the first 8 values of T1. Similarly there exists a memory block that contains the last 8 values of T2 and the first 8 values of T3. Since this is the case for our target library, we only monitor those two memory blocks to recover the entire AES key.

We store both the transfer timing and the ciphertext obtained by our encryption server. In order to analyze the results, we implement a miss counter approach: we count the number of times that each ciphertext value sees a miss, i.e. that the monitored cache line was not loaded for that ciphertext value. An example of one of the runs for ciphertext number 0 is shown in Figure 5. The 8 ciphertext values that obtain the lowest scores are the ones are the ones corresponding to the cache line, thereby revealing the key value.

In order to obtain the key, we iterate over all possible key

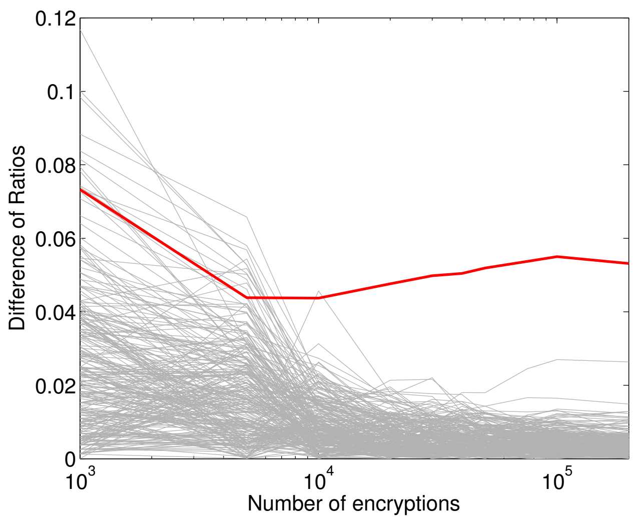


Figure 7: Difference of Ratios over the number of encryptions needed to recover the full AES key. The correct key (bold red line) is clearly distinguishable from 20,000 encryptions.

byte values and compute the last round of AES only for the monitored T-table values, and then group the miss counter values of the resulting ciphertexts in one set. We group in another set the miss counter of the remaining 248 ciphertext values. Clearly, for the correct key, the distance between the two sets will be maximum. An example of the output of this step is shown in Figure 6, where the y-axis represents the miss counter ratio (i.e., ratio of the miss counter value in both sets) and the x-axis represents the key byte guess value. It can be observed that the ratio of the correct key byte (180) is much higher than the ratio of the other guesses.

Finally we calculate the number of encryptions needed to

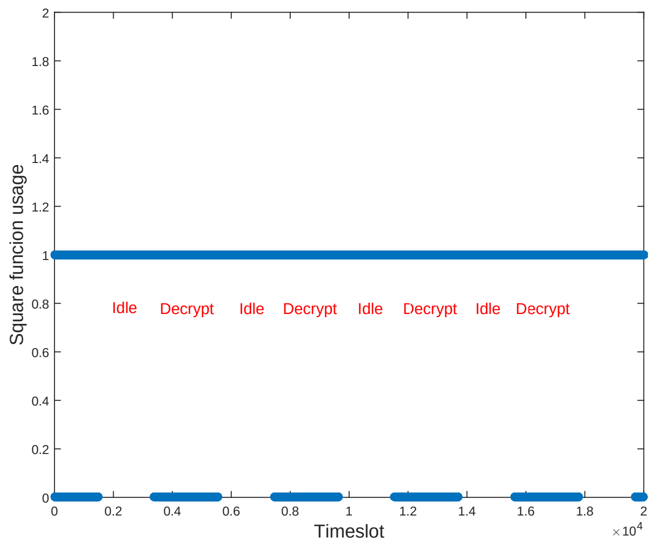


Figure 8: Trace observed by the `Invalidate+Transfer`, where 3 4 decryption operations are cached. The decryption stages are clearly visible when the square function usage gets the 0 value

recover the full AES key. This is shown in Figure 7, where the y-axis again represents the ratios and the x-axis represents the number of encryptions. As it can be observed, the correct key is not distinguishable before 10,000 traces, but from 20,000 observations, the correct key is clearly distinguishable from the rest. We conclude that the new method succeeds in recovering the correct key from 20,000 encryptions.

7.3 ElGamal Results

Next we present the results obtained when the attack aims at recovering an ElGamal decryption key. We target a 2048 bit ElGamal key. Remember that, unlike in the case of AES, this attack does not need synchronization with the server, i.e., the server runs continuous decryptions while the attacker continuously monitors the vulnerable function. Since the modular exponentiation creates a very specific pattern with respect to both the square and multiply functions, we can easily know when the exponentiation occurred in the time. We only monitor a single function, i.e., the square function. In order to avoid speculative execution, we do not monitor the main function address but the following one. This is sufficient to correctly recover a very high percentage of the ElGamal decryption key bits. For our experiments, we take the time that the invalidate operation takes into account, and a minimum waiting period of 500 cycles between the `invalidate` and the `transfer` operation is sufficient to recover the key patterns. Figure 8 presents a trace where 4 different decryptions are caught. A 0 in the y-axis means that the square function is being utilized, while a 1 the square function is not utilized, while the x-axis represents the time slot number. The decryption stages are clearly observable when the square function gets a 0 value.

Please recall that the execution flow caused by a 0 bit in the exponent is `square+reduction`, while the pattern caused

Table 1: Summary of error results in the RSA key recovery attack.

Traces analysed	20
Maximum error observed	3.47%
Minimum error observed	1.9%
Average error	2.58%
Traces needed to recover full key	5

by a 1 bit in the exponent is `square+reduction+multiply+reduction`. Since we only monitor the square operation, we reconstruct the patterns by checking the distance between two square operations. Clearly, the distance between the two square operations in a 00 trace will be smaller than the distance between the two square operations in a 10 trace, since the latter one takes an additional multiplication function. With our waiting period threshold, we observe that the distance between two square operations without the multiplication function varies from 2 to 4 `Invalidate+Transfer` steps, while the distance between two square operations varies from 6 to 8 `Invalidate+Transfer` steps. If the distance between two square operations is lower than 2, we consider it part of the same square operation. An example of such a trace is shown in Figure 9. In the figure, S refers to a square operation, R refers to a modulo reduction operation and M refers to a multiply operation. The x-axis represents the time slot, while the y axis represents whether the square function was utilized. The 0 value means that the square function was utilized, whereas the 1 value means that the square function was not utilized. The pattern obtained is $SRMRSRSRMRSRSRMRSRSRMRSRMRSRSRMRS$ $RMRSRSRMRSRS$, which can be translated into the key bit string 101010110101010.

However, due to microarchitectural sources of noise (context switches, interrupts, etc) the recovered key has still some errors. In order to evaluate the error percentage obtained, we compare the obtained bit string with the real key. Any insertion, removal or wrong guessed bit is considered a single error. Table 1 summarizes the results. We evaluate 20 different key traces obtained with the `Invalidate+Transfer` spy process. On average, they key patterns have an error percentage of 2.58%. The minimum observed error percentage was 1.9% and the maximum was 3.47%. Thus, since the errors are very likely to occur at different points in order to decide the correct pattern we analyse more than one trace. On average, 5 traces are needed to recover the key correctly.

8. VIABILITY OF THE COVERT CHANNEL IN OTHER SCENARIOS

We demonstrated the covert channel in a shared server setting. However, there are other scenarios where the proposed covert channel exists and can be exploited as shown earlier.

8.1 Cloud Computing Scenario

The Platform as a Service cloud computing model is becoming increasingly popular with well known examples like Apprenda, VMware/EMC co-owned Pivotal, Red Hat Openshift, Salesforce Heroku, and AWS Elastic Beanstalk. The PaaS provider delivers software and hardware tools for use in application development. The applications that belong

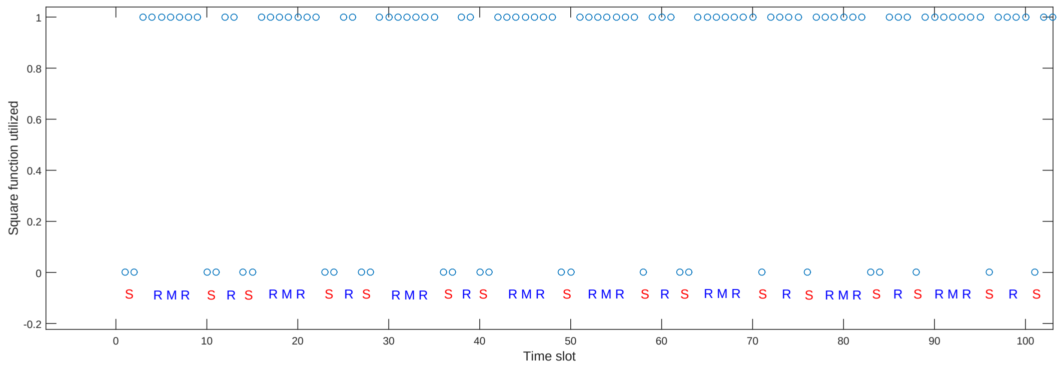


Figure 9: Trace observed by the `Invalidate+Transfer`, converted into square and multiply functions. The y-axis shows a 0 when the square function is used and a 1 when the square function is not used

to different users, are executed within the same OS. Thus, both attacker and victim share parts of the memory space as demonstrated in [38]. This scenario is essentially the same scenario used in the attack proposed in this paper. Therefore, the proposed `Invalidate + Transer` should also be applicable in PaaS clouds. IaaS clouds provide each user with a dedicated OS where applications can be executed. In this particular case, the `Invalidate + Transer` would work when the hypervisor implements deduplication techniques, i.e., if memory pages are shared across VMs.

8.2 Mobile Devices

ARM devices implement a technique called *AMBA Cache Coherent Interconnect* that facilitates fast interprocessor connections very similar to the *HyperTransport* protocol exploited in this paper. This technology helps to maintain cache coherency across ARM CPUs using a `snoop filter` protocol supported by a cache directory architecture [8, 4, 6]. Thus, upon a shared memory read miss, the `snoop filter` checks whether the same memory block is cached in an adjacent processor. If successful, a direct cache-to-cache link will be established, thereby eliminating the need for a slow DRAM access.

9. CONCLUSION

We presented a new covert channel exploiting cache coherence protocols which recovers information leakage caused by the data access time difference. The new attack exploits the fact that data cached anywhere in the multiprocessor system has lower access times than memory accesses facilitated by fast interconnects such as AMD *HyperTransport* and Intel *QuickPath*. The attack thus can retrieve fine-grain information even when victim and attacker are located in different processors on the same system. Even further, the new covert channel does not rely on specific properties of cache hierarchies like inclusiveness. This was a common assumption in previous attacks. The coherence protocol ensures that the data is found independently of where in the cache it is stored. Thus, the new attack can be applied in processors where cache attacks have not been demonstrated before such as in AMD processors. We proved the viability of the new attack by recovering a full AES key and a full ElGamal key across co-located AMD CPUs.

Acknowledgements

This work is supported by the National Science Foun- dation, under grant CNS-1318919. We would like to thank Edward Burnham for his support on the server configuration.

10. REFERENCES

[1] An introduction to the QuickPath Interconnect. `http://www.intel.com/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf`.

[2] Fix Flush and Reload in RSA. `https://lists.gnupg.org/pipermail/gnupg-announce/2013q3/000329.html`.

[3] HyperTransport Technology white paper. `http://www.hypertransport.org/docs/wp/ht_system_design.pdf`.

[4] Increasing Performance and Introducing CoreLink CCI-500. `https://community.arm.com/groups/processors/blog/2015/02/03/extended-system-coherency--part-3--corelink-cci-500`.

[5] Intel QuickPath Architecture. `http://www.intel.com/pressroom/archive/reference/whitepaper_QuickPath.pdf`.

[6] Introduction to AMBA 4 ACE. `https://www.arm.com/files/pdf/CacheCoherencyWhitepaper_6June2011.pdf`.

[7] Transparent Page Sharing: new default setting. `http://blogs.vmware.com/security/2014/10`.

[8] Verifying ARM AMBAő 5 CHI Interconnect-Based SoCs Using Next-Generation VIP. `http://www.synopsys.com/Company/Publications/DWTB/Pages/dwtb-verifying-arm-amba5-chi-2014Q4.aspx`.

[9] Aciiçmez, O., Schindler, W., and Çetin K. Koç. Cache Based Remote Timing Attack on the AES. In *Topics in Cryptology CT-RSA 2007, The Cryptographers Track at the RSA Conference 2007*, pp. 271–286.

[10] Aciiçmez, O. Yet Another MicroArchitectural Attack: Exploiting I-Cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*.

[11] Aciiçmez, O., K. Koç, c., and Seifert, J.-P. Predicting secret keys via branch prediction. In *Topics in Cryptology CT-RSA 2007*, vol. 4377. pp. 225–242.

[12] Benger, N., van de Pol, J., Smart, N. P., and Yarom, Y. "ooh aah... just a little bit" : A small amount of side channel can go a long way. In *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings* (2014), pp. 75–92.

[13] Bernstein, D. J. Cache-timing attacks on AES, 2004. URL: `http://cr.yp.to/papers.html#cachetiming`.

[14] Bonneau, J., and Mironov, I. Cache-Collision Timing Attacks against AES. In *CHES 2006*, vol. 4249 of *Springer LNCS*, pp. 201–215.

[15] Conway, P., Kalyanasundharam, N., Donley, G., Lepak, K., and Hughes, B. Cache hierarchy and memory subsystem of the amd opteron processor. *IEEE Micro 30*, 2 (Mar. 2010), 16–29.

[16] Cornero, M., and Anyuru, A. Multiprocessing in mobile platforms: the marketing and the reality. `http://etn.se/images/expert/FD-SOI-eQuad-white-paper.pdf`.

[17] Crane, S., Homescu, A., Brunthaler, S., Larsen, P., and Franz, M. Thwarting cache side-channel attacks through dynamic software diversity. In *Network And Distributed System Security Symposium, NDSS* (2015), vol. 15.

[18] Fangfei Liu and Yuval Yarom and Qian Ge and Gernot Heiser and Ruby B. Lee. Last level cache side channel attacks are practical. In *S&P 2015.*

[19] Gruss, D., Bidner, D., and Mangard, S. Practical Memory Deduplication Attacks in Sandboxed Javascript. In *ESORICS 2015* (2015).

[20] Gruss, D., Spreitzer, R., and Mangard, S. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., Aug. 2015), USENIX Association, pp. 897–912.

[21] Gullasch, D., Bangerter, E., and Krenn, S. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. SP '11, pp. 490–505.

[22] Gülmezoglu, B., Inci, M. S., Apecechea, G. I., Eisenbarth, T., and Sunar, B. A faster and more realistic flush+reload attack on AES. In *Constructive Side-Channel Analysis and Secure Design - 6th International Workshop, COSADE 2015* (2015), pp. 111–126.

[23] Hu, W.-M. Lattice Scheduling and Covert Channels. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy.*

[24] Inci, M. S., Gulmezoglu, B., Irazoqui, G., Eisenbarth, T., and Sunar, B. Seriously, get off my cloud! cross-vm rsa key recovery in a public cloud. Cryptology ePrint Archive, Report 2015/898, 2015. `http://eprint.iacr.org/`.

[25] Irazoqui, G., Eisenbarth, T., and Sunar, B. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing and its Application to AES. In *36th IEEE Symposium on Security and Privacy (S&P 2015).*

[26] Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Wait a Minute! A fast, Cross-VM Attack on AES. In *RAID* (2014), pp. 299–319.

[27] Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. Lucky 13 strikes back. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2015), ASIA CCS '15, ACM, pp. 85–96.

[28] Oren, Y., Kemerlis, V. P., Sethumadhavan, S., and Keromytis, A. D. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 1406–1418.

[29] Osvik, D. A., Shamir, A., and Tromer, E. Cache Attacks and Countermeasures: The Case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06.

[30] Page, D. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel, 2002.

[31] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pp. 199–212.

[32] Tsunoo, Y., Saito, T., Suzaki, T., and Shigeri, M. Cryptanalysis of DES implemented on computers with cache. In *Proc. of CHES 2003, Springer LNCS* (2003), pp. 62–76.

[33] Varadarajan, V., Zhang, Y., Ristenpart, T., and Swift, M. A placement vulnerability study in multi-tenant public clouds. In *24th USENIX Security Symposium (USENIX Security 15)* (2015), pp. 913–928.

[34] Wu, Z., Xu, Z., and Wang, H. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *USENIX Security symposium* (2012), pp. 159–173.

[35] Xu, Z., Wang, H., and Wu, Z. A measurement study on co-residence threat inside the cloud. In *24th USENIX Security Symposium (USENIX Security 15)* (2015).

[36] Yarom, Y., and Falkner, K. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pp. 719–732.

[37] Zhang, Y., Juels, A., Oprea, A., and Reiter, M. K. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy.*

[38] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security.*

[39] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security.*