Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses

Jakub Szefer[1]

Yale University, New Haven, CT, USA
jakub.szefer@yale.edu

Abstract. Over last two decades, side and covert channel research has shown variety of ways of exfiltrating information for a computer system. Processor microarchitectural side and covert channel attacks have emerged as some of the most clever attacks, and ones which are difficult to deal with, without impacting system performance. Unlike electromagnetic or power-based channels, microarchitectural side and covert channel do not require physical proximity to the target device. Instead, only malicious or cooperating spy applications need to be co-located on the same machine as the victim. And in some attacks even co-location is not needed, only timing of the execution of the victim as measured by a remote attacker over the network can form a side channel for information leaks. This survey extracts the key features of the processor's microarchitectural functional units which make the channels possible, presents an analysis and categorization of the variety of microarchitectural side and covert channels others have presented in literature, and surveys existing defense proposals. With advent of cloud computing and ability to launch microarchitectural side and covert channels even across virtual machines, understanding of these channels is critical.

1 Introduction

One of the first mentions of what we now call side- or covert-channel attacks was brought up by Lampson in 1973 [36], in his note on the confinement problem of programs. Since then many research papers have explored side and covert channels. From processor architecture perspective, there is an intrinsic connection between the side and covert channels and the characteristics of the underlying hardware. First, these channels exist because of spatial and temporal sharing of processor units among different programs as they execute on the processor. Second, many decades of processor architecture research have resulted in processor optimizations which create fast and slow execution paths, e.g. a simple addition takes much less time to execute than a multiplication operation. Sharing of functional units, and the fast and slow paths, are both characteristics that have allowed the explosion in computational power of modern processors.

Meanwhile, more and more researchers are exploiting the functional unit sharing or the fast and slow paths to present ever more clever side- and covert-channel attacks. Thus on one hand processor architects are adding new features to improve performance, and on another security researchers are exploiting these improvements to show how information can leak, e.g. [7, 31, 63, 50, 61]. Of course, with growing interest in side- and cover-channel attacks, hardware and software defenses have been put forth, e.g. [46, 64, 35, 38]. This survey aims to survey both sides of this arms race and makes number of contributions:

1. elicits key features of the microarchitectural functional units which make the channels possible,
2. presents an analysis of existing microarchitectural side and covert channels, and
3. surveys existing defense proposals.

Analysis of the variety of the side and covert channels reveals that in the presence of sharing of hardware and fast and slow paths, it is the pattern of usage and sharing of these functional units that determines the channel capacity – the information leak rate will vary from potentially close to theoretical maximum to almost zero, depending on how the computer system is used. Thus this work surveys and condenses the key characteristics of hardware's role in the side and covet channels so researchers and developers can better know how their software will behave and how it may be susceptible to attacks.

The different attacks presented in literature thus far vary wildly in their reported information leakage capacity. In idealized experiments, microarchitectural side or covert channels can reach capacities of hundreds kilobits per second, e.g. [31]. In the attacks, however, there are often certain assumptions made about the attacker and victim that help the attacker, for example that the attacker and victim are co-located on same processor core (e.g. in case of cache-based attacks or attacks that leverage the branch predictor). If the attacker is not able to create a situation where they are co-located with a victim for a certain amount of time, the channel capacity can drop significantly. Realistic attacks are in range from fraction of a bit per second, e.g. [49], to few bits per second, e.g. [73].

To remedy the attacks, researchers have shown many defenses. Nevertheless, almost all remain academic proposals. In particular, the designs which focus on eliminating the side and covert channels and their associated attacks often do so at the cost of performance, which is at odds with the desire to improve efficiently of modern processors that are used anywhere from smartphones, cloud computing data centers to high-performance supercomputers used for scientific research. This interplay between microarchitectural side and covert channels on one side and desire to further improve processor performance through microarchitectural enhancements has contributed to lack of practical counter measures in production hardware.

1.1 Scope of the Survey

This survey focuses on side and covert channels which may exist inside a modern processor. This includes processor cores and any functional units inside a multi-core multi-threaded processor such as caches, memory controller, or interconnection network. This work does no look at other components of a computer, e.g. hard drives and associated timing covert channels due hard drive disk head movement [23]. Also, focus in on software attacks on hardware where an attacker process can learn some information about victim process, or cooperating attacker processes can send information between each other. Hardware attacks, such as power analysis side channels [33] or electromagnetic side channels [21] are not in the scope.

1.2 Survey Organization

The survey is organized as follows. Section 2 presets processor microarchitectural features and how they enable information leaks via the side and covert channels. Section 3 describes side and covert channel classification, in relation to the processor features. Section 4 discusses the existing side and covert attacks. Section 5 summarizes various proposals for analysis, detection and defense from side and covert channels. Discussion is presented in Section 6 and we conclude in Section 7.

2 Functional Unit Features Leading to Side and Covert Channels

To understand the problem of microarchitectural side and covert channel attacks, it is first needed to understand the variety of processor microarchitectural features and how they enable information leaks. Figure 1 shows a two-thread SMT processor with 5-stage pipelines, along with the memory subsystem components. The figure shows a single-core, dual threaded (SMT) pipelines with key components found in most processors.

The typical processor pipeline is broken into number of stages: instruction fetch, instruction decode, execute, memory and writeback. At each stage, pipeline is able to perform different operations, and results of these operations are stored in the pipeline buffers (shown as gray rectangles in the figure). At each clock cycle, the pipeline stages take input from the previous stage and proceed to perform its operation. Thus the instructions and data proceed in the pipeline until the results are computed and stored back in register file or written to memory.

Each solid white rectangle in the figure represents a functional unit – a hardware unit responsible for a specific operation or task. Each functional unit has some inputs, can maintain its state, and generates output. The inputs, state, and outputs are affected by the programs and system software running on the system. Each program or system software is composed of code and stream of instructions which are one-by-one executed by the processor[1]. Instruction streams

[1] Modern processor support out-of-order (OoO) execution, which allows instructions to be re-ordered for better performance, but OoO preserves program semantics and instructions are always retired in program order so that OoO execution is transparent to programmers.

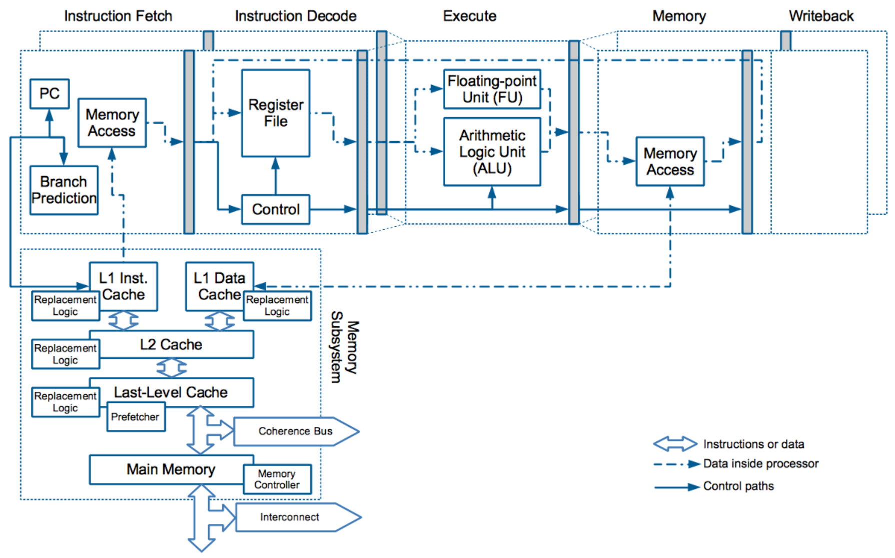

Fig. 1: The prototypical two-thread SMT processor with 5-stage processor pipelines shown with shared execution stage and key components of the memory subsystem.

from different programs and system software alternate according to system software scheduler and hardware policies. Typically, there is strong ring-based protection system which prevents different applications from reading each other's memory or explicitly accessing resources assigned to other programs directly.

However, the sole act of executing an instruction and affecting one or more functional units' state can lead to side or covert channel. This is because there is an intrinsic relationship between processor microarchitectural features which allow today's processors to efficiently run various programs, but at the same time optimizations which lead to the side and covert channels.

Microarchitectural side and covert channels are typically timing-based channels. Because of the sharing of functional units among different programs, programs can in general observe timing of the operation of the functional unit and directly or indirectly its output. Knowing the design of the different functional units, timing in turns reveals whether the fast or slow execution path was taken. Finally, knowing one's own operations, or victim's operations, and the timing, a side or covert channel for information leakage can be established.

There are five characteristics of modern processors and their design which lead to microarchitectural-based channels:

1. execution of different instructions takes different amount of time,
2. shared hardware leads to contention,
3. program's behavior (executed instructions) affect state of various functional units,
4. results or timing of instructions is related to state of the functional units, and

5. based on history of executed instructions, entropy in the operation of the functional units changes.

2.1 Instruction Execution Timing

The job of the processor hardware is to perform different computations. Some computations are fundamentally simpler than others. Many logical operations (e.g. AND, OR, NAND, etc.) can be performed in a single processor cycle. Arithmetic operations such as addition also can be done quickly with use of parallel prefix adders or similar hardware circuits. Some operations, however, such as multiplication do not have as efficient hardware implementations. Thus processor designers have in past designed single- and multi-cycle instructions. As the names imply, single-cycle instruction takes one processor cycle. A multi-cycle instruction takes many cycles. The fundamental complexity of certain operations means that the program timing will depend on the instructions in that program. Thus these fast and slow path leads to information leaks when computation is performed, as *execution of different instructions takes different amount of time.*

Eliminating the fast and slow paths would mean making all instructions take as long as the slowest instruction. However, performance implications are tremendous. Different between logical operation and floating-point is on order of 10x cycles[2]. Meanwhile, a memory operation (discussed in detail later) can take over 100s cycles if data has to be fetched from main memory. Consequently there is inverse relationship between the entropy among execution unit timing and the performance. Better performance implies higher entropy and more information leaks.

2.2 Functional Unit Contention

Processors are constrained in area and power budgets. This has led processor designers to opt to re-use and share certain processor units when having separate ones may not on average beneficial. One example is hyper-threading, or simultaneous multithreading (SMT), where there are usually two pipelines per core, as shown in Figure 1. However, the two pipelines share the execution stage and the units therein. Motivation is that, on average, there is a mix of instructions and it is unlikely that two programs executing in parallel, one on each pipeline, will need exactly same functional units. Program A may do addition, while program B is doing memory access, in which case each executes almost as if they had all resources to themselves. A complication comes in when two of the programs from each pipeline attempt to perform same operation. If two programs try, for example, to perform floating point operations, one will be stalled until floating point unit is available. The contention can be reflected in the timing. If a program performs certain operation and it takes longer at some time, then this implies that some other program is also using that same hardware functional unit, leaking information about what another program is doing. Thus information leaks during computation when *shared hardware leads to contention.*

[2] The functional units themselves can be pipelined to lower the overheads.

Reductions in the contention can be addressed by duplicating the hardware. Today, there are many multi-core processors without SMT, where each processor core has all resources to itself. However, equally many processors employ SMT and research results show that SMT yields large performance gains with small area overhead. E.g. SMT chip with two pipelines and shared execution stages is about 6% larger than a singe thread processor [13]. A two thread SMT is likely to remain in production for many years because of the evident benefits. Better performance/area ratios as explicit design goals imply at least some functional unit sharing will exist and in turn contention that leads to information leaks. The contention also becomes quite important in the memory subsystem, as discussed later.

2.3 State-dependent Output of Functional Units

Many functional units inside the processor keep some history of past execution and use the information for prediction purposes. Instructions that are executed form the inputs to the functional units. The state is some function of the current and past inputs. And the output depends on the history. Thus, output of a state-full functional unit depends on past inputs. Observing the current output leaks information about the past computations in which that unit was involved. We give a specific example based on the branch predictor.

Branch Predictor is responsible for predicting which instructions should be executed next when a branch (e.g. `if ... then ... else ...` statement) is encountered. Since the processor pipeline is broken down into stages, the branch instruction is not evaluated unit the second stage. Thus, the hardware needs to guess which instruction to fetch while the branch is being evaluated, should it execute instructions from the `then` path or the `else` path? Only later the hardware goes back and potentially nullifies fetched instructions if it was found that there was as a mis-prediction and instructions from the wrong path were started to execute.

To obtain good performance, branch predictor attempts to learn the branching behavior of programs. Its internal state is built using observation of past branches. Based on the addresses of the branch instructions it builds local and global histories of past branches. When a branch instruction is encountered, branch predictor is looked up based on the address of the branch instruction. If it was seen in the past, there is a taken or not taken prediction. Modern branch predictors can reach below 2 miss-predictions per 1000 instructions [14]. Because of global history component of the branch predictors, different program affect the branch predictor. A pathological program can "train" the branch predictor to mis-predict certain branches. Then, when another program executes, it may experience many mis-predictions leading to longer execution time and thus information leaks about which branches were executed.

Eliminating the branch predictor would deal a hit to the performance and it is unlikely to be removed from modern processors. This is one example of how information leaks will exist as *program's behavior (executed instructions) affect state of various functional units, that later affects others programs' timing.*

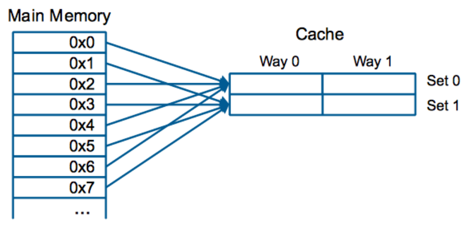

Fig. 2: Example of a two-way set-associative cache. Data from each memory location is assigned to a specific set, based on the address. Multiple ways allow storing multiple pieces of data in a same set. The LRU policy is used within each set.

2.4 Memory Subsystem Functional Units

We dedicate a separate section to the memory subsystem as it has some of the biggest impacts on the programs performance and information leaks. Not coincidentally, vast number of research papers has focused on side and covert channels due to some functional unit in the memory subsystem, e.g. [57] [10] [28] [71] [48] [12] [37] [72] [55] [24] [41] [40] [11] [3] [6] [47] [59] [58].

Caches Recall from Figure 1 that the memory subsystem is composed of different layers of caches. Each L1 cache is located closest to the processor, it is smallest in size, but accessing data in the L1 cache takes about 1 to 2 processor cycles. There are separate L1 caches for instructions and data. L2 cache is a larger cache, but at the cost of taking about 10 processor cycles to access data in the cache. L2 cache can be per processor core or shared between multiple processor cores. L3 cache, also called Last Level Cache (LLC) is the biggest cache in size up to few MB, but accessing data in L3 takes 10s of cycles. Finally, there is the main memory, sized in GBs, but requiring 100 cycles or more to access data.

Processor designers use the cache hierarchy to bring most recently and most often used data into the cache closest to the processor so that when there is memory access or instruction fetch, it can be gotten from one of the caches, rather than requiring going all the way to memory. Unfortunately, the fastest caches closest to the processor are also smallest, so there needs to be some policy of which data to keep in the cache. Often, the policy is some variant of least recently used policy (LRU) that kicks out least recently used data or instructions and keeps most recently used ones. As programs execute on the processor and perform memory accesses, they cause processors to bring into the caches new data, and kick out lest recently used data back to lower cache or eventually to the main memory.

Keeping track of least recently used data in the whole cache is not practical, thus caches are broken down into sets, where each memory location can only be mapped into a specific sets, as shown in Figure 2. Multiple memory addresses are mapped to a set. A cache typically has two or more ways, e.g. in a two-way set associative cache, there are two locations that data from specific set can be placed into. The LRU policy is kept for each set. For example, if memory addresses 0x0, 0x2 and 0x4 are accessed in that order, then when 0x4 is accessed, it will kick out 0x0 from the two-way set associative cache shown in Figure 2 as 0x0 was least recently used from set 0.

Such design of the caches lends itself easily to contention and interference, which in turn leads to information leakage. The typical information leakage is due to timing that reveals whether some data is in the cache or not. Accessing data in L1 takes 1 to 2 cycles, while data in memory can take up to 100 cycles. Eliminating such leakages is difficult. Ideally, the cache replacement logic could search whole cache for least recently used data, rather than just within a set. The Z-cache is one step towards that, however, its complexity has prevent it from being implemented [51]. Proposals for randomized caches (discussed in more detail later) have also been put forward [64]. However, currently if caches are removed, each memory access could take 100 or more cycles, which is not realistic to eliminate the cache hierarchy. Again, *execution of different (memory) instructions takes different amount of time* leading to potential for side or covert channels.

Prefetcher Another component of the memory subsystem is the prefetcher which is used in microprocessors to speed up the execution of a program by speculatively brining in data or instructions into the caches. The goal of a processor cache prefetcher is to predict which memory locations will be accessed in near future and prefetch these locations into the cache. By predicting memory access patterns of applications the prefetcher brings in the needed data into the cache, so that when the application accesses the memory, it will already be in the cache or a stream buffer[3], avoiding much slower access to the main memory.

Hardware prefetchers attempts to automatically calculate what data and when to prefetch. The prefetchers usually work in chunks of size of the last level cache (LLC) blocks. Sequential prefetchers prefetch block x+1 when block x is accessed. An improvement, which is most often used in today's processors, is the stride prefetcher which attempts to recognize sequential array accessed, e.g. block x, block x+20, block x+40, etc. [52].

Because hardware stride prefetcher fetches multiple blocks ahead, it will sometimes bring in data that the application is not going to use. However, depending on the physical memory allocation, that prefethed data may actually be used by another application. When the application accesses memory and measures timing, the blocks which were prefetched based on pattern detected for the other application will be accessible more quickly. In addition, if the exact details of the prefetcher algorithm are known, it is possible to trace back which addresses and how many addresses were accessed by the other application. Prefetcher re-

[3] Some prefetchers place prefetched data in a dedicated stream buffer to limit cache pollution, stream buffer nevertheless is like a cache and accessing data in stream buffer is much faster than going to main memory.

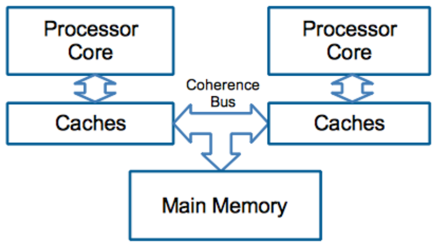

Fig. 3: Example of a dual-core platform where two processing cores share one memory controller and the main memory (DRAM).

moval would have largest penalty for regular applications that have many array accesses.

Memory Controller The memory controller and the DRAM controller in the main memory are responsible for managing data going to and from the processor and the main memory. The memory controller contains queues for request from the processor (reads and writes, usually coming form the last level cache), it has to schedule these request and arbiter between different caches making request and deal with the DRAM device resource contention.

The memory control which is a share resource, becomes a point of contention, as shown in Figure 3. In this example, two processor cores are each connected to the same memory controller and memory chip. Requests from each processor core need to be ordered and queued up by processing by the memory. Dynamically changing memory demand from one processor core will affect memory performance of the other core. While the memory controller attempts to achieve fairness, it is not always possible to balance out memory traffic from different cores. In particular, today's DRAM is typically divided up into pages and data from within DRAM is first brought into a row buffer before being actually processed (reads sent data back to the requesting processor from the buffer, or writes update it with incoming data). Many of today's chips use open-page policy that gives some preference to reads or writes to currently opened page (i.e. on in the row buffer). Memory accesses with lots of spatial locality may get preference as they hit in the open page – giving overall better performance as opening / closing new pages is expensive in terms of energy and time. Because of such optimization, again *shared hardware leads to contention* which in turn can be basis for side or covert channels.

Interconnect Modern processors have replaced a typical bus that connected multiple processors and memory with more advance interconnects, such as Intel's Quick Path Interconnect (QPI). The interconnect is used by to send data between processors and also for memory accesses in non-uniform memory architectures (NUMA) where main memory is divided up and separate DRAM chips

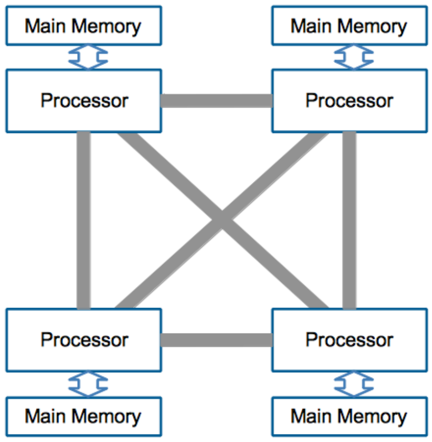

Fig. 4: Example of 4-processor setup with point-to-point (QPI-like) interconnect between them. Each processor has its own associated memory, but all processors can access the other memories via the interconnect, a typical NUMA configuration for systems with a single global address space.

and memory controllers are located near each processor, as shown in Figure 4. Such arrangement gives each processor fast access to local memory, yet still large total system memory. However, timing of memory accesses can reveal information, such as accessing data in the DRAM chip close to the processor is faster than accessing remotely located DRAM at another core. In addition, locking and atomic operations can lock down the interconnect making memory accesses stall. Thus memory access timing can reveal state of the interconnect and leak information about what other processes are doing.

3 Side and Covert Channel Classification

With the understanding of the processor functional units and their operation, we proceed to explore the different channels and actual attacks and defenses. First, we begin by classifying different types of attacks.

A covert communication channel is a communication channel that was not intended or designed to transfer information between a sender and a receiver. A side channel is similar to a covert channel, but the sender does not intend to communicate information to the receiver, rather sending (i.e. leaking) of information is a side effect of the implementation and the way the computer hardware is used.

Covert channels are important when considering intentional information exfiltration where one program manipulates state of the system according to some protocol and another observers the changes to read the "messages" that are sent to it. Covert channels are a concern because even when there is explicit isolation, e.g. each program runs in its own address space and can't directly read and write another program's memory, the covert channel through processor hardware features allows the isolation mechanisms to be bypassed.

Side channels are important when considering unintentional information leaks. When considering side channels, there is usually the "victim" process that uses a computer system and the way the system is used can be observed by an "attacker" process.

Side and covert channels can be generally categorized as timing-based, access-based, or trace-based channels. Timing-based channels rely on timing of various operations to leak information, e.g. [6, 34, 9]. For example, one process performs many memory accesses so that memory accesses of another process are slowed down. Access-based channels rely on accessing some information directly, e.g. [47, 44, 27, 73, 49]. For example, one process probes the cache state by timing its own memory accesses. Trace-based channels rely on measuring exact execution of a program, e.g. [3]. For example, attacker obtains sequence of memory accesses and whether they are cache hits or misses based on the power measurements.

Trace-based channels usually require some physical proximity, for example, to obtain the power traces. In this work, we focus on microarchitectural channels and attacks that can be done remotely, thus the focus is narrowed down to the timing-based and access-based channels.

In context of the processor, both timing-based and access-based channels have a timing component that is observable by a potential attacker. Especially, while access-based attacks are built on operations that access certain resource, such accesses perturb timing of another process' operations. In particular, we differentiate these as internal timing and external timing attacks, shown in Figure 5. In internal timing, the attacker measures its own execution time. Based on knowledge of what it (the attacker) is doing, e.g. which cache lines it accessed, and the timing of its own operations, the attacker can deduce information, e.g. which cache lines were being accessed by other applications on that processor. In external timing, the attacker measures execution time of the victim, e.g. how long it takes to encrypt a piece of data; knowing the timing and what the victim is doing the attacker can deduce some information, e.g. was there addition or

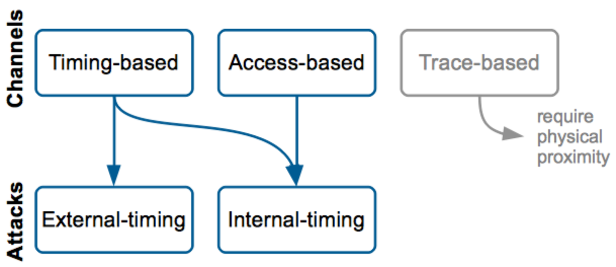

Fig. 5: Relation between channel types and attack types.

multiplication done during the encryption, potentially leaking bits of information about the encryption key[4].

4 Existing Side- and Covert-Channel Attacks

Variety of clever internal and external timing attacks have been demonstrated in literature in past years. We present grouping of the attacks based on whether they target computer system without or with virtualization. Naturally, more attacks have been presented for non-virtualized systems as these have been around for longer[5].

4.1 Attacks in Non-Virtualized Settings

Caches Traditional side [34] and covert [67] channels based on timing have been explored since 1990s. One of first theoretical uses of processor cache memory as a side-channel was shown in 2002 [45]. Timing attacks based on shared data caches have been widely studied since in literature, focusing on attacks on cryptographic protocols and algorithms, and leveraging processor data caches [57] [10] [28] [71] [48] [12] [37] [72] [55] [24] [41] [40] [11] [3] [6] [47] [59] [58]. These side and covert channels leverage the underlying properties of the caches where accessing specific memory location is guaranteed to place data in a specific cache set. When one program's (attacker's) data gets kicked out of the cache, which can be measured using timing, that means that another program (victim) has accessed data that maps to the same cache set. The hardware properties, combined with frequent AES or other cryptographic implementations that use table lookups during operation, allow attackers to make educated guesses about which table locations were accessed by the victim, and thus guess bits of cryptographic keys.

In addition to data caches (D-cache), channels through instruction caches (I-cache) have been demonstrated [1] [2]. I-cache channels rely on the fact that instruction cache misses increase the execution time of the applications. As example, researchers have mounted these attacks against OpenSSL RSA implementation and took advantage of the fact that OpenSSL uses different functions to compute modular multiplications and square operations when doing RSA computations. The different functions leave different footprints in the I-cache. Attacker is able to fill I-cache with its own instructions, when the victim (OpenSSL RSA) runs it kicks out different instruction form I-cache and attacker now knows if multiplication or square function was executed in the victim.

Most of the attacks focus on first level caches, but recent work [70] utilized shared last level cache (LLC) attack to extract cryptographic keys. The L1 caches

[4] External timing channels often require many iterations to correlate timing information, however, basic principle is the same that attacker observes a victim's timing and the victim's timing depends on the operations it performs.

[5] IBM mainframes and similar computer systems have had virtualization-like features long before today's commodity PCs; however very few attacks, if any, have been explored in that setting.

are smallest, so it is easiest to manipulate their contents by accessing data or instructions. Higher level caches (L2, LLC) are larger, can be shared among different processor cores, and store both instructions and data. This contributes to more noise in cache based attacks, leading to lesser bandwidth for side or covert channels through these caches. Nevertheless, attacks get better every year.

Data Path Units Beyond caches, integer and floating-point units have been found to be sources of information leaks [7]. These are classical examples of contention that causes different timing and can leak information. When two or more programs access a share resource that is busy, one of them is stalled. Stalls mean longer execution time, and allow the program to conclude that another program is using the unit. More recently, similar idea has been applied to the new AES instructions in Intel's processors [61].

In addition, covert channels using exceptions on speculative load instructions and shared functional units in simultaneously multithreading (SMT) processors have been shown [63]. SMT processors allow multiple processor pipelines share some hardware, on assumption that on average not all programs will do exactly same functional units so fewer units are implemented. However, when the average case does not hold, contention arises causing timing channels.

Control Path Units Side-channel attacks through branch prediction units [5] [4] [31] have been shown as well. Similarly to caches where hits (data is found in the cache) or misses (data is not in the cache) cause different timing, branch prediction hits and misses give different timing. Attackers can "train" a branch predictor by forcefully executing branches that tend to hit (or miss) at certain address and then when another process runs its branches at the trained addresses will be predicted to hit (or miss). The hit or miss timing can be measured to form a covert channel.

System Bus and Interconnect Channels have also been presented based on the Quick Path Interconnect (QPI) lock mechanism in Intel's processors [61]. Some channels become rediscovered over time, e.g. with first processor bus contention channel presented in 1995 [26] and recently applied to the modern memory bus [50].

The interconnect and memory bus are a shared resource, and these channels are further example of how sharing of a resource leads to contention, which affects timing and forms a channel. These attacks rely on the fact that the interconnect can be locked for use by a specific program when it is doing a type of atomic operation, or bus is not available when another "memory hog" program has many memory requests.

4.2 Attacks in Virtualized Settings

With advent of cloud computing, researchers have moved their attention to side and covert channels in virtualized environments. The attacks in non-virtualized settings can be extended to virtualized settings. The attacks tend to be less effective as (in addition to the operating system) there is the hypervisor and many other virtual machines. This creates a noisy environment. However, researches have been able to overcome this. It can be expected that variety of the channels from non-virtualize environments will become viable in virtualized environment, even if there are no such current attacks listed below.

Caches One of the first channels was a cross-VM covert channel that exploited the shared cache [49]. Such cross-VM side channels have been used to extract private keys from VMs [73]. Researchers have moreover leveraged L2 caches to create covert channels in virtualized environments [69]. Interesting resource-freeing attacks improve an attacker's VM's performance by forcing interference with a competing VM [60]. Leveraging some hypervisor software features in combination with hardware memory, researchers have utilized sharing of redundant pages in memory deduplication [39] [53] as channels as well.

Like their non-virtualized counterparts, these channels rely on the fact that programs can fill the caches with data by making memory accesses, and later read back data and time the reads to see which memory was kicked out by another program, allowing them to make educated guess about which data or instructions were executed.

System Bus and Interconnect Researchers have exploited the memory bus as a high-bandwidth covert channel medium. Specifically, memory controllers have been shown to be sources of potential channels and leaks [68]. Again, like their non-virtualized counterparts, bus contention can open up timing channels.

4.3 Attack Analysis

The attacks that researchers have presented, and keep presenting, have a direct correlation between number of attacks and their bandwidth vs. how much performance improvement the associated functional unit offers. Caches stand out as the major source of attacks, and they are also a major performance improving feature. Caches were first ones explores for attacks in non-virtualize environments, and also first ones in virtualize environments. Memory controller and related functional units also are a source of attacks, but to a lesser degree. Finally there are units such as branch predictor which give very small bandwidth channels. It's difficult to quantify the contribution of the different units to the performance of the processor as they it is heavily dependent on the software that is running on the computer. Nevertheless, intuitively, there are many more memory operations than branch instructions, so the branch predictor has smaller effect on the performance – and in turn has smaller bandwidth as source of information leaks.

Timing channels arising due to contention are also quite frequent, and are more often exploited for covert channels. Re-use of functional units and their sharing leads to contention as processors are designed for the average case where not all software needs all functional units at the same time, meanwhile attacks create pathological code that forcefully tries to use same functional units as other code at the same time so as to bring about the contention.

Researchers and software writers should focus on analyzing their applications to understand what operations and functional units they use, to determine how different side and covert channels may affect them. The more the software uses functional units that have most impact on performance, the more it is susceptible to attacks.

4.4 Estimates of Existing Attack Bandwidths

Large number of research papers do not clearly state specific bit per second rates, but rather show that they were able to recover a number of secret key bit or bits of sensitive information for their target application. Nevertheless, some of the bandwidths can be well estimated by looking at the researchers' experiments and their experimental setup. Most of the rates are in ranges of kilobits per second (Kbps) in optimal or idealized setting, and reduce to bits per second (bps) or less in realistic settings.

One of the first side-channel attacks was the 0.001 bps Bernstein's AES cache attack using L1 cache collisions [9]. Bonneau improved the attack to about 1 bps [11]. Around same time, Percival reported attacks with about 3200 Kbps using L1 cache-based covert channel, 800 Kbps using L2 cache-based covert channel, which reduce to few Kbps when they were done in a realistic setting [47]. Recently about 0.5 bps channels due to branch predictor [20] were presented.

Other set of researchers have focused on virtualization and virtual machines. Ristenpart, et al., showed 0.006 bps memory bus contention channel across VMs [49], and also 0.2 bps cross-VM L2 access-driven attack [49]. Xu, et al., presented 262 bps L2 cache-based covert channel in a virtualized environment [69]. Zhang, et al., show 0.02 bps L1 cache-based covert channel across VMs, using IPIs to force attacker VM to interrupt victim VM [73]. Wu, et al., show 100 bps channel on Amazon's EC2 due to shared main-memory interface in symmetric multi-processors [68]. Hunger, et al., show up to 624 Kbps channel when sender and receiver can have a well optimized and have aligned clock signals [31]. These are summarized in Table 1.

4.5 Attack Bandwidth Analysis

The Trusted Computer System Evaluation Criteria (TCSEC), or more commonly *The Orange Book*, sets the basic requirement for trusted computer systems [43]. The Orange Book specifies that a channel bandwidth exceeding a rate of 100 bps is a high bandwidth channel. It can be seen from Table 1 that many idealized attacks are well above that rate, but there is also a quite large variance with the reported or estimated bandwidths for actual attacks. The cache based attacks are highest in bandwidth as potential attackers are able to affect specific cache sets by executing memory accesses to particular addresses that map to the desired set. Other functional units let potential attackers affect the units state

Attack	**Functional Unit**	**Channel Bandwidth**
Non-virtualized Environments		
[9]	L1 cache	0.001 bps
[20]	Branch predictor	0.5 bps
[11]	L1 cache	1 bps
[47]	L2 cache	800 Kbps (idealized)
[47]	L1 cache	3200 Kbps (idealized)
Virtualized Environments		
[49]	Memory bus	0.006 bps
[73]	L1 cache	0.02 bps
[49]	L2 cache	0.2 bps
[68]	Memory controller	100 bps
[69]	L2 cache	262 bps
[31]	Various	624 Kbps (idealized)

Table 1: Bandwidth estimate for various side and covert channel attacks reported in literature

less directly. For example, many branch instructions are needed to re-train the branch predictor, which leads to lesser bandwidth channel.

Figure 6 shows the selected attacks and their bandwidths as a function of year in which they were presented. Bandwidth in bps on y-axis is plotted on log-scale for easier reading. It may seem unusual that as years progress, new published attacks do not necessarily have bandwidth better than prior attacks (due to expectation that new research should beat prior work). It should be noted however, that some of the newer attacks are based on functional units that contribute less to the performance, so the bandwidth is less. The contributions of these attacks are clever ways of, for example, leveraging branch predictor. To help see overall trend, a trend line was added to the figure to show the overall trend of the attacks and their bandwidths. Clearly, bandwidths are getting higher and have reached the bounds set by TCSEC for “high bandwidth channels.”

When considering idealized attacks (shown as triangles in the figure) which are on the order of 100s Kbps, the ”high bandwidth” boundary has long been passed in their case. These attacks, however, are usually specific to a single program, which often is the AES encryption algorithm. The attacks ten to be also synchronous, where the attacker and victim are executing synchronized (e.g. attacker and victim alternate execution on same processors). The synchronous attacks tend to have better bandwidth. For example, the L1 cache-based covert channel across VMs used IPIs to exactly force execution of the attacker to interleave with the victim. Dedicated attacker can thus come up with clever ways of improving bandwidth by having more synchronous attacks, approaching the idealized attack scenarios.

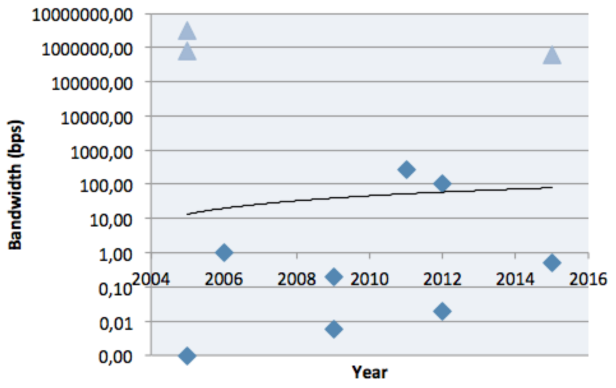

Fig. 6: Scatter plot of attacks from Table 1 along with a trend line showing roughly 10x improvement in attack bandwidth over last 10 years for the non-idealized attacks (squares). Idealized attacks (triangles) show much greater bandwidth, but these are for specific cases, such as attacks on particular table lookup based AES software implementation.

5 Analysis and Defense of Processor-based Side and Covert Channels

Microarchitectural side and covert channel research is not all on attacks, with much effort put into detection and defense against such attacks. Detection of, and defending against, side and covert channels is a difficult task. We first look at detection-related work, and then explore potential defense, many of which can be deploy today, but at cost of performance.

5.1 Analyzing Side and Covert Channels Susceptibility

Microarchitectural side and covert channels arise because of the functional unit sharing and the fast and slow execution paths. Whenever there is contention or re-use of the units, attackers can leverage that for timing channels. However, the side and covert channels are not only due to hardware, but also due to how the software uses the hardware. A simple example is that if there is only one program using floating point unit, there will not be contention and an information leak. Only when another program that uses that unit (and uses it at the same time) will floating point unit leak information. Thus, detection has focused on design-time approaches that attempt to understand how much information could leak and run-time approaches that try to detect unit sharing or the fast and slow execution paths. The two are compared in Figure 7.

Design-Time Approaches One of the first works looked at shared resource matrix methodology to identify storage and timing channels [32]. Researchers have tried to define formal basis for hardware information flow security by providing a method to separate timing flows from other flows of information [42]. Also, a side-channel vulnerability factor has been presented as a metric for measuring information leakage in processors [17] [16].

Such approaches are designed to be deployed at design time, but thus far it is not clear if any processor manufactures use them. The intuition is that design-time only approaches are fundamentally needed, but the side and covert channels depend both on the hardware and how it is used. Thus run-time approaches complement them. Nevertheless, it would be desired that processors come with some metrics of side and covert channel susceptibility, but today that is not available.

Run-Time Approaches A number of run-time approaches have been proposed. Detection based on entropy-based approach [22] or dynamically tracking conflict patterns over the use of shared processor hardware have been shown [15]. Attempts to detect malware through analyzing existing performance counters have been proposed [18], or by using other hardware supported lower-level processor features [54].

One new innovative run-time approach that stands out uses groups of VMs and L2 cache contention to detect attackers VMs [72]. The key idea in their proposal is to invert the usual application of side channels, and use the timing to observe if other expected VMs are executing or if there is an unexpected VM accessing memory. According to authors, by analyzing cache usage through memory timings, "friendly" VMs coordinate to avoid portions of the cache and can detect the activity of a co-resident "foe" VM.

These approaches attempt to measure and understand how the software is using the hardware. By obtaining insights into the running software, it is possible to detect if there may be contention between different programs leading to side or covert channel, or if there is malware that has unusual sequences of instructions being executed signaling that it may be a piece of malware leveraging such channels.

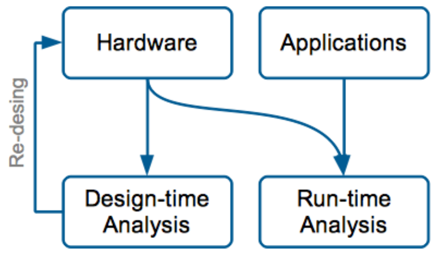

Fig. 7: Design-time analysis takes into account the hardware properties. Run-time approaches can combine both hardware and software properties, but are done post-design and are not able to influence the re-design of hardware, which could minimize the side and covert channel capacities.

5.2 Defending Side and Covert Channels

The analysis of the hardware and software has lead researchers to propose a variety of defenses. Since the channels depend both on the hardware, and the software that is running on that hardware, the defenses have focused on hardware approaches and software approaches.

Hardware Approaches To mitigate side and covert channels, hardware architectural changes have been proposed including partitioning or time-multiplexing caches [46] [64], which have since been improved [35]. Cache design which dynamically reserves cache lines for active threads and prevents other co-executing threads from evicting reserved lines have also been shown [19]. Such approaches essentially reservers a subset of the cache for the protected program. Other programs are not able to interfere with these reserved cache blocks. This prevents internal-timing, but external-timing attacks are still possible since measuring the protected program's timing from outside still can reveal some patterns about memory it is accessing. In addition, other applications cannot use the reserved cache blocks, effectively cutting down on cache size and the performance benefits it brings.

One of the best proposals focus on new type of randomized caches [65]. Today's commodity hardware has caches where the mapping between memory and cache sets is fixed and same for all applications. Recall in Figure 2 memory address 0x0, 0x2, 0x4, etc. all mapped to set 0, regardless of which application is running. Randomized caches in effect change this mapping for each application, e.g. application A has 0x0 mapped to set 1 and application B has 0x0 mapped to set 0. This thwarts internal-timing, but external-timing may still be an issue. While not designed with security as a goal, the Z-cache [51] may have some of the similar properties where it searchers among many cache sets to find least recently used block for replacement. Thus, effective set size is larger, reducing applications' contention for same set.

Most recently, work has turned to on-chip networks and ensuring non-interference [66] or providing timing channel protection [62]. In [66] authors re-design the interconnect to allow precise scheduling of the packets that flow across the interconnect. Data from each processor are grouped together and carried together in "waves" while strictly non-interfering with other data transfers. In [62] observe, as we do, that shared resources imply that applications affect each other's timing through interference and contention. The defense proposal is again to partition the network temporally and limit how much each processor can use the network so as to limit the interference.

Hardware-supported mechanism have also been added for enforcing strong non-interference [56]. The authors propose "execution leases" which allow to temporally partition the processor's resources and lease them to an application so that others cannot interfere with the usage of these resources. This temporal partitioning again focuses on un-doing the original design where resources are shared at very fine granularity, and instead making it coarser, leaving to small

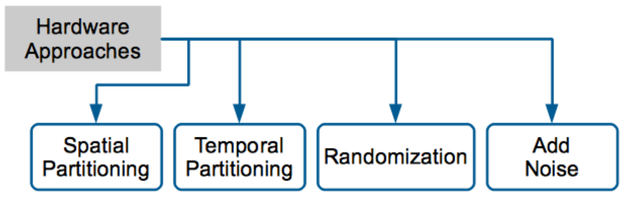

Fig. 8: Main hardware approaches to microarchitectural side and covert channel defense.

potential leaks. The tradeoff is the performance impact of locking parts of processor for exclusive use of an application. The longer application can keep the resources, the better leak protection, but also more negative impaction performance of other applications.

Processor architects have also proposed the introduction of random noise to hardware counters [38]. The key to any timing attacks is to be able to obtain a timing reference, either within the application or somewhere from outside. In [38] authors' approach is to limit the granularity and precision of timekeeping and performance counters mechanisms. By introducing more uncertainty in the timing, potential attackers are limited in their ability to get a good point of reference. Nevertheless, many applications are networked and can use external sources of timing. Both inside and outside the computer system needs to be considered even when focusing on microarchitectural channels.

Software Approaches Researchers have suggested clearing out leftover state in caches through frequent cache flushes [44]. This is clearly performance degrading technique, nevertheless it is able to prevent side and covert channels, as all application data and code is flushed from memory on context switch and when application runs again it observes the long timing of main memory accesses. If the scheduling periods are long, application is able to fill up the cache and benefit from it. However, when scheduling periods are short, essentially the application will have to get all data from memory as the caches are flushed constantly. External timing attacks are prevented, and internal-timing can also be thwarted if the scheduling periods are short.

Outside of processor caches, to deal with the branch predictor based channels, clearing branch predictor on a context switch has been suggested [5] as well. Again, periodical clearing of the predictor state makes the current predictions not depend on past inputs seen by the predictor, thus reducing information leak. However, such a defense is also a performance hit as branch predictors rely on learning the branching history of the running programs to give good branch predictor hit rate.

Addition of noise has also been suggested. For example, [29], explores reducing channels by introducing fuzzy time. In it, a collection of techniques is introduced that reduce the bandwidths of covert timing channels by making all clocks available to a process noisy; this includes system time stamp counters and

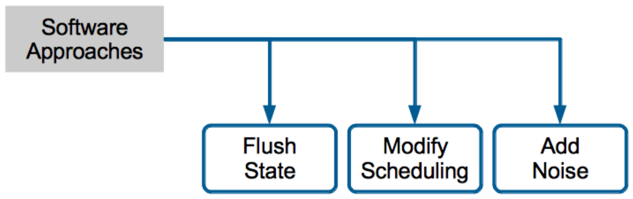

Fig. 9: Main software approaches to microarchitectural side and covert channel defense.

inputs from disk drives or network cards. In [25], authors introduce similar fuzzy time noise to defeat bus contention channel.

Other works focus on making the time and all events deterministic, such as in deterministic OSes designs [8]. Rather than randomize the timing and add noise, all events are delivered at deterministic instants. Such approaches are not easily applied to caches or hardware features, but do help in virtualized environments where the hypervisor can control precisely delivery of packets or other timing information.

An example of pro-active attempt is the lattice scheduler which is a process scheduler that schedules applications using access class attributes to minimize potential contention channels [30]. In general applications can be scheduled such that the contention is minimized. While existing schedulers do not do this, taking hardware into consideration a scheduler can run different processes on different cores, or time multiplex them such as to limit the contention. Of course, this assumes availability of many processes to be run and fairness of today's OS or hypervisor schedulers may be violated.

6 Discussion

Based on the analysis, we make a number of observations and recommendations for researchers and software writers:

1. *Sharing of functional units by different programs and fast and slow execution paths lead to side and covert channels that attackers can exploit.*

It has been shown by the authors of the surveyed works that a wide variety of functional units can be sources of internal and external timing channels that attackers can leverage to leak information. Ever more clever attacks emerge each year, showing that processor caches were the early targets of analysis and attacks and now other functional units in the processor are getting exploited. As long as software uses instructions which timing could be affected by contention in the functional units, there will be vulnerabilities. Likewise, timing differences between execution of different operations lead to vulnerabilities as well.

2. *Covert and side channel capacities continue to increase, with real and idealized attacks reaching beyond 100 bps.*

Given current hardware implementation, software running on commodity systems should assume that existing side and covert channels have passed the lower bounds set by TCSEC for "high bandwidth channels." This is true for virtualized environments. Especially, the strong isolation mechanisms in today's hypervisors are not able to prevent variety of side and covert channels. In addition, virtualization may make things worst, as users are outsourcing their computations to the cloud where they are co-located with unknown other users.

3. *New functional units cannot be assumed free from side or covert channel vulnerabilities.*

Section 2 listed variety of functional units and how performance optimizations embedded in these units are unlikely to be removed, thus leaving theses units as potential sources of side and covert channels. It would be hoped that even if old functional units cannot be changed, new ones could be better designed. Meanwhile, as discussed in the section on attacks, even the newest units such as the dedicated AES hardware can be basis for contention and lead to timing-based channels. For example, switching from software based AES implementations, to avoid cache channels, to hardware AES instructions does not fully solve information leaks. Thus when re-coding software to avoid one type of side or covert channel vulnerability, care must be taken to understand what new channels may be opened up.

4. *Many functional units exist which do not have shown attacks, but which do contribute to to the fast and slow execution paths can could become future side and covert channels.*

Analysis of the processor hardware reveals units such as the prefetcher that keep internal state base on past inputs and their output is dependent on these inputs, potentially leaking information. Recall, hardware prefetchers attempt to automatically calculate what data and when to prefetch into the cache in anticipation that an application will use them. Because hardware stride prefetcher fetches multiple blocks ahead, it will sometimes bring in data that the (victim) application is not going to use. However, depending on the physical memory allocation, that prefethed data may actually be used by another (attacker) application. When the attacker application accesses memory and measures timing, the blocks which were prefetched based on pattern detected for the victim application will be accessible more quickly by the attacker. Such is a simple theoretical example of a prefetcher attack.

5. *Flushing state of different functional units, modified scheduling of applications to avoid contention, resource partitioning and adding noise are software defenses available today.*

Despite the above dangers, much research has been put into detection and prevention of side channels. Fuzzy timing, clearing state of functional units, spatial and temporal partitioning and randomization are all techniques available to today's software. They should be leveraged when writing software and system software. With move to cloud computing, OS should assume it may be running inside a virtual machine and should employ these mechanisms. Likewise, hypervisors have no way to verify intent of the guest VMs and can leverage the mechanisms to protect VMs.

Clearly, side and covert channels in today's processors are a source of potential danger. Ongoing work is begin done by architecture and hardware communities to bring about hardware free of information leaks. Until the hardware becomes available, researchers and software writers should be mindful of what operations their applications perform and how they could be affected by the side and covert channels due to sharing of functional units or the fast and slow execution paths inside the processor.

7 Conclusion

Over last two decades, side and covert channel research has shown variety of, often very clever, ways of exfiltrating information for a computer system. Processor microarchitectural side and covert channel attacks have emerged as some of the most clever attacks, and ones which are difficult to deal with, without impacting system performance. This survey extracted the key features of the processor's microarchitectural functional units which make the channels possible, presented an analysis and categorization of the variety of microarchitectural side and covert channels others have presented in literature, and surveyed existing defense proposals.

Processor architects continue to come up with new processor optimizations which create a fast and slow execution paths or re-use and sharing of functional units for better energy efficiently, power or area. Meanwhile, more and more researchers are exploiting the functional unit sharing or the fast and slow paths to present ever more clever side and covert channel attacks. This work surveyed both sides of this arms race, which continues today. Especially, with advent of cloud computing and ability to co-locate VMs with other VMs on a cloud computing data center servers, understanding of these channels is critical as users have less and less control over environment where the software runs.

References

1. Aciiçmez, O.: Yet another microarchitectural attack:: exploiting i-cache. In: Proceedings of the 2007 ACM workshop on Computer security architecture. pp. 11–18. ACM (2007)
2. Acıiçmez, O., Brumley, B.B., Grabher, P.: New results on instruction cache attacks. In: Cryptographic Hardware and Embedded Systems, CHES 2010, pp. 110–124. Springer (2010)
3. Acıiçmez, O., Koç, Ç.K.: Trace-driven cache attacks on aes (short paper). In: Information and Communications Security, pp. 112–121. Springer (2006)
4. Acıiçmez, O., Koç, Ç.K., Seifert, J.P.: Predicting secret keys via branch prediction. In: Topics in Cryptology–CT-RSA 2007, pp. 225–242. Springer (2006)

5. Aciiçmez, O., Koç, Ç.K., Seifert, J.P.: On the power of simple branch prediction analysis. In: Proceedings of the 2nd ACM symposium on Information, computer and communications security. pp. 312–320. ACM (2007)
6. Aciiçmez, O., Schindler, W., Koç, Ç.K.: Cache based remote timing attack on the aes. In: Topics in Cryptology–CT-RSA 2007, pp. 271–286. Springer (2006)
7. Aciicmez, O., Seifert, J.P.: Cheap hardware parallelism implies cheap security. In: Fault Diagnosis and Tolerance in Cryptography, 2007. FDTC 2007. Workshop on. pp. 80–91. IEEE (2007)
8. Aviram, A., Hu, S., Ford, B., Gummadi, R.: Determinating timing channels in compute clouds. In: Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop. pp. 103–108. CCSW '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1866835.1866854
9. Bernstein, D.J.: Cache-timing attacks on aes (2005)
10. Bogdanov, A., Eisenbarth, T., Paar, C., Wienecke, M.: Differential cache-collision timing attacks on aes with applications to embedded cpus. In: CT-RSA. vol. 10, pp. 235–251. Springer (2010)
11. Bonneau, J., Mironov, I.: Cache-collision timing attacks against aes. In: Cryptographic Hardware and Embedded Systems-CHES 2006, pp. 201–215. Springer (2006)
12. Brumley, B.B., Hakala, R.M.: Cache-timing template attacks. In: Advances in Cryptology–ASIACRYPT 2009, pp. 667–684. Springer (2009)
13. Burns, J., Gaudiot, J.L.: Smt layout overhead and scalability. Parallel and Distributed Systems, IEEE Transactions on 13(2), 142–155 (Feb 2002)
14. Championship Branch Prediction (2014), http://www.jilp.org/cbp2014/, accessed August 2015
15. Chen, J., Venkataramani, G.: Cc-hunter: Uncovering covert timing channels on shared processor hardware. In: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture. pp. 216–228. IEEE Computer Society (2014)
16. Demme, J., Martin, R., Waksman, A., Sethumadhavan, S.: A quantitative, experimental approach to measuring processor side-channel security. Micro, IEEE 33(3), 68–77 (2013)
17. Demme, J., Martin, R., Waksman, A., Sethumadhavan, S.: Side-channel vulnerability factor: a metric for measuring information leakage. In: ACM SIGARCH Computer Architecture News. vol. 40, pp. 106–117. IEEE Computer Society (2012)
18. Demme, J., Maycock, M., Schmitz, J., Tang, A., Waksman, A., Sethumadhavan, S., Stolfo, S.: On the feasibility of online malware detection with performance counters. ACM SIGARCH Computer Architecture News 41(3), 559–570 (2013)
19. Domnitser, L., Jaleel, A., Loew, J., Abu-Ghazaleh, N., Ponomarev, D.: Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. ACM Transactions on Architecture and Code Optimization (TACO) 8(4), 35 (2012)
20. Evtyushkin, D., Ponomarev, D., Abu-Ghazaleh, N.: Covert channels through branch predictors: a feasibility study. In: Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy. p. 5. ACM (2015)
21. Gandolfi, K., Mourtel, C., Olivier, F.: Electromagnetic analysis: Concrete results. In: Cryptographic Hardware and Embedded SystemsCHES 2001. pp. 251–261. Springer (2001)
22. Gianvecchio, S., Wang, H.: Detecting covert timing channels: an entropy-based approach. In: Proceedings of the 14th ACM conference on Computer and communications security. pp. 307–316. ACM (2007)

23. Gold, B., Linde, R., Cudney, P.: Kvm/370 in retrospect. In: Security and Privacy, 1984 IEEE Symposium on. pp. 13–13. IEEE (1984)
24. Grabher, P., Großschädl, J., Page, D.: Cryptographic side-channels from low-power cache memory. In: Cryptography and Coding, pp. 170–184. Springer (2007)
25. Gray III, J.W.: On introducing noise into the bus-contention channel. In: Research in Security and Privacy, 1993. Proceedings., 1993 IEEE Computer Society Symposium on. pp. 90–98. IEEE (1993)
26. Gray III, J.W.: Countermeasures and tradeoffs for a class of covert timing channels (1994)
27. Gullasch, D., Bangerter, E., Krenn, S.: Cache games–bringing access-based cache attacks on aes to practice. In: Security and Privacy (SP), 2011 IEEE Symposium on. pp. 490–505. IEEE (2011)
28. Henricksen, M., Yap, W.S., Yian, C.H., Kiyomoto, S., Tanaka, T.: Side-channel analysis of the k2 stream cipher. In: Information Security and Privacy. pp. 53–73. Springer (2010)
29. Hu, W.M.: Reducing timing channels with fuzzy time. In: Research in Security and Privacy, 1991. Proceedings., 1991 IEEE Computer Society Symposium on. pp. 8–20. IEEE (1991)
30. Hu, W.M.: Lattice scheduling and covert channels. In: Research in Security and Privacy, 1992. Proceedings., 1992 IEEE Computer Society Symposium on. pp. 52–61. IEEE (1992)
31. Hunger, C., Kazdagli, M., Rawat, A., Dimakis, A., Vishwanath, S., Tiwari, M.: Understanding contention-based channels and using them for defense. In: High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on. pp. 639–650. IEEE (2015)
32. Kemmerer, R.A.: Shared resource matrix methodology: An approach to identifying storage and timing channels. ACM Transactions on Computer Systems (TOCS) 1(3), 256–277 (1983)
33. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Advances in CryptologyCRYPTO99. pp. 388–397. Springer (1999)
34. Kocher, P.C.: Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In: Advances in CryptologyCRYPTO96. pp. 104–113. Springer (1996)
35. Kong, J., Aciiçmez, O., Seifert, J.P., Zhou, H.: Hardware-software integrated approaches to defend against software cache-based side channel attacks. In: High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on. pp. 393–404. IEEE (2009)
36. Lampson, B.W.: A note on the confinement problem. Communications of the ACM 16(10), 613–615 (1973)
37. Leander, G., Zenner, E., Hawkes, P.: Cache timing analysis of lfsr-based stream ciphers. In: Cryptography and Coding, pp. 433–445. Springer (2009)
38. Martin, R., Demme, J., Sethumadhavan, S.: Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In: ACM SIGARCH Computer Architecture News. vol. 40, pp. 118–129. IEEE Computer Society (2012)
39. Miłós, G., Murray, D.G., Hand, S., Fetterman, M.A.: Satori: Enlightened page sharing. In: Proceedings of the 2009 conference on USENIX Annual technical conference. pp. 1–1 (2009)
40. Neve, M., Seifert, J.P.: Advances on access-driven cache attacks on aes. In: Selected Areas in Cryptography. pp. 147–162. Springer (2007)

41. Neve, M., Seifert, J.P., Wang, Z.: A refined look at bernstein's aes side-channel analysis. In: Proceedings of the 2006 ACM Symposium on Information, computer and communications security. pp. 369–369. ACM (2006)
42. Oberg, J., Meiklejohn, S., Sherwood, T., Kastner, R.: A practical testing framework for isolating hardware timing channels. In: Proceedings of the Conference on Design, Automation and Test in Europe. pp. 1281–1284. EDA Consortium (2013)
43. DoD 5200.28-STD, Department of Defense Trusted Computer System Evaluation Criteria (1983), http://csrc.nist.gov/publications/history/dod85.pdf
44. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: the case of aes. In: Topics in Cryptology–CT-RSA 2006, pp. 1–20. Springer (2006)
45. Page, D.: Theoretical use of cache memory as a cryptanalytic side-channel. IACR Cryptology ePrint Archive 2002, 169 (2002)
46. Page, D.: Partitioned cache architecture as a side-channel defence mechanism. IACR Cryptology ePrint Archive 2005, 280 (2005)
47. Percival, C.: Cache missing for fun and profit (2005)
48. Rebeiro, C., Mukhopadhyay, D., Takahashi, J., Fukunaga, T.: Cache timing attacks on clefia. In: Progress in Cryptology-INDOCRYPT 2009, pp. 104–118. Springer (2009)
49. Ristenpart, T., Tromer, E., Shacham, H., 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. pp. 199–212. ACM (2009)
50. Saltaformaggio, B., Xu, D., Zhang, X.: Busmonitor: A hypervisor-based solution for memory bus covert channels. Proceedings of EuroSec (2013)
51. Sanchez, D., Kozyrakis, C.: The zcache: Decoupling ways and associativity. In: Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on. pp. 187–198. IEEE (2010)
52. Shen, J.P., Lipasti, M.H.: Modern processor design: fundamentals of superscalar processors. Waveland Press (2013)
53. Suzaki, K., Iijima, K., Yagi, T., Artho, C.: Software side channel attack on memory deduplication. SOSP POSTER (2011)
54. Tang, A., Sethumadhavan, S., Stolfo, S.J.: Unsupervised anomaly-based malware detection using hardware features. In: Research in Attacks, Intrusions and Defenses, pp. 109–129. Springer (2014)
55. Tiri, K., Acıiçmez, O., Neve, M., Andersen, F.: An analytical model for time-driven cache attacks. In: Fast Software Encryption. pp. 399–413. Springer (2007)
56. Tiwari, M., Li, X., Wassel, H.M., Chong, F.T., Sherwood, T.: Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture. pp. 493–504. ACM (2009)
57. Tromer, E., Osvik, D.A., Shamir, A.: Efficient cache attacks on aes, and countermeasures. Journal of Cryptology 23(1), 37–71 (2010)
58. Tsunoo, Y.: Cryptanalysis of block ciphers implemented on computers with cache. preproceedings of ISITA 2002 (2002)
59. Tsunoo, Y., Saito, T., Suzaki, T., Shigeri, M., Miyauchi, H.: Cryptanalysis of des implemented on computers with cache. In: Cryptographic Hardware and Embedded Systems-CHES 2003, pp. 62–76. Springer (2003)
60. Varadarajan, V., Kooburat, T., Farley, B., Ristenpart, T., Swift, M.M.: Resource-freeing attacks: improve your cloud performance (at your neighbor's expense). In: Proceedings of the 2012 ACM conference on Computer and communications security. pp. 281–292. ACM (2012)

61. Wang, Y., Ferraiuolo, A., Suh, G.E.: Timing channel protection for a shared memory controller. In: High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on. pp. 225–236. IEEE (2014)
62. Wang, Y., Suh, G.E.: Efficient timing channel protection for on-chip networks. In: Networks on Chip (NoCS), 2012 Sixth IEEE/ACM International Symposium on. pp. 142–151. IEEE (2012)
63. Wang, Z., Lee, R.B.: Covert and side channels due to processor architecture. In: Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual. pp. 473–482. IEEE (2006)
64. Wang, Z., Lee, R.B.: New cache designs for thwarting software cache-based side channel attacks. In: ACM SIGARCH Computer Architecture News. vol. 35, pp. 494–505. ACM (2007)
65. Wang, Z., Lee, R.B.: A novel cache architecture with enhanced performance and security. In: Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on. pp. 83–93. IEEE (2008)
66. Wassel, H.M., Gao, Y., Oberg, J.K., Huffmire, T., Kastner, R., Chong, F.T., Sherwood, T.: Surfnoc: a low latency and provably non-interfering approach to secure networks-on-chip. ACM SIGARCH Computer Architecture News 41(3), 583–594 (2013)
67. Wray, J.C.: An analysis of covert timing channels. In: Research in Security and Privacy, 1991. Proceedings., 1991 IEEE Computer Society Symposium on. pp. 2–7. IEEE (1991)
68. Wu, Z., Xu, Z., Wang, H.: Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In: USENIX Security symposium. pp. 159–173 (2012)
69. Xu, Y., Bailey, M., Jahanian, F., Joshi, K., Hiltunen, M., Schlichting, R.: An exploration of l2 cache covert channels in virtualized environments. In: Proceedings of the 3rd ACM workshop on Cloud computing security workshop. pp. 29–40. ACM (2011)
70. Yarom, Y., Falkner, K.E.: Flush+ reload: a high resolution, low noise, l3 cache side-channel attack. IACR Cryptology ePrint Archive 2013, 448 (2013)
71. Zenner, E.: A cache timing analysis of hc-256. In: Selected Areas in Cryptography. pp. 199–213. Springer (2009)
72. Zhang, Y., Juels, A., Oprea, A., Reiter, M.K.: Homealone: Co-residency detection in the cloud via side-channel analysis. In: Security and Privacy (SP), 2011 IEEE Symposium on. pp. 313–328. IEEE (2011)
73. Zhang, Y., Juels, A., Reiter, M.K., 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. pp. 305–316. ACM (2012)