Journal of Hardware and Systems Security
https://doi.org/10.1007/s41635-018-0046-1

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

Jakub Szefer[1]

Received: 13 February 2018 / Accepted: 27 August 2018

Abstract

Over the last two decades, side and covert channel research has shown a variety of ways of exfiltrating information for a computer system. Processor microarchitectural timing-based 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 timing-based 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 application, as measured by a remote attacker, can lead to 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 for cybersecurity.

Keywords Side channels · Covert channels · Microarchitecture · Survey

1 Introduction

One of the first mentions of what we now call side or covert channel attacks was brought up by Lampson in 1973 [42], in his note on the confinement problem of programs. Since then, many research papers have explored timing-based side and covert channels. From a processor architecture perspective, there is an intrinsic connection between the timing-based 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 for 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 the one hand, processor architects are adding new features to improve performance, and on the other, security researchers are exploiting these improvements to show how information can leak (e.g., [7, 36, 57, 69, 71]). Of course, with growing interest in side and cover channel attacks, hardware and software defenses have been put forth (e.g., [41, 45, 53, 72]). This survey aims to show both sides of this arms race and makes a number of contributions:

1. Elicits key features of the microarchitectural functional units which make the channels possible.
2. Analyzes existing microarchitectural side and covert channels.
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

✉ Jakub Szefer
jakub.szefer@yale.edu

[1] Department of Electrical Engineering, Yale University, 10 Hillhouse Ave., New Haven, CT 06510, USA

Published online: 13 September 2018

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. This work surveys and condenses the key characteristics of hardware's role in the side and covert 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 greatly in their reported information leakage capacity. In idealized experiments, microarchitectural side or covert channels can reach capacities of hundreds of kilobits or even megabits per second (e.g., [36]). 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 the 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.

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 efficiency of modern processors that are used anywhere from smartphones and cloud computing data centers to high-performance supercomputers used for scientific research. This interplay between microarchitectural side and covert channels on the one side and the desire to further improve processor performance through microarchitectural enhancements on the other side has likely contributed to the 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 not look at other components of a computer, e.g., hard drives and associated timing covert channels due to hard drive disk head movement [28]. Also, focus is 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 [39] or electromagnetic side channels [26], are not in the scope.

1.2 Survey Organization

The survey is organized as follows. Section 2 describes side and covert channel classification. Section 3 presents processor microarchitectural features and how they enable information leaks via the side and covert channels. 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 conclusion is in Section 7.

2 Side and Covert Channel Classification

This section explores the different types of side and covert channels, actual attacks, and defenses. First, we begin by classifying different types of attacks.

According to [25], a covert 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 the state of a 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 cannot 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, 11, 40]). 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., [32, 51, 54, 56, 83]). For example, one process probes the cache state by observing latency to determine if data was hit or miss in the cache. 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 survey, we focus on microarchitectural channels and attacks that can be done remotely; thus, the focus is narrowed down to 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 Fig. 1. 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 multiplication done during the encryption, potentially leaking bits of information about the encryption key. Note, external timing channels often require many iterations to correlate timing information; however, basic principle is the same that an attacker observes a victim's timing and the victim's timing depends on the operations it performs.

3 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

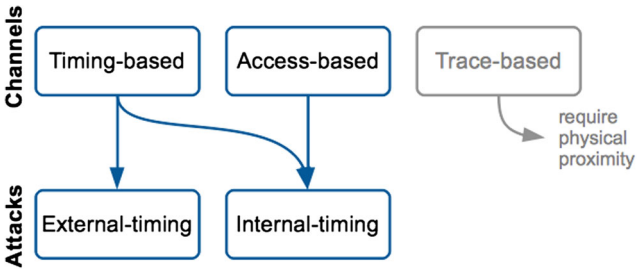

Fig. 1 Relation between channel types and attack types. The channel types are timing-based or access-based; however, in both cases, the attacker measures timing to make their observations. The external timing attacks involve attacker measuring the victim's execution time, while internal timing attacks involve attacker measuring their own timing to deduce how state of the system changed due to victim's actions

the variety of processor microarchitectural features and how they enable information leaks. Figure 2 shows a two-thread simultaneous multithreading (SMT) processor with five-stage pipelines, along with the memory subsystem components. The figure shows a single-core, dual-threaded pipeline 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 executed one-by-one by the processor. 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. Instruction streams from different programs and system software alternate according to system software scheduler and hardware policies. Typically, there is a 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 turn 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.

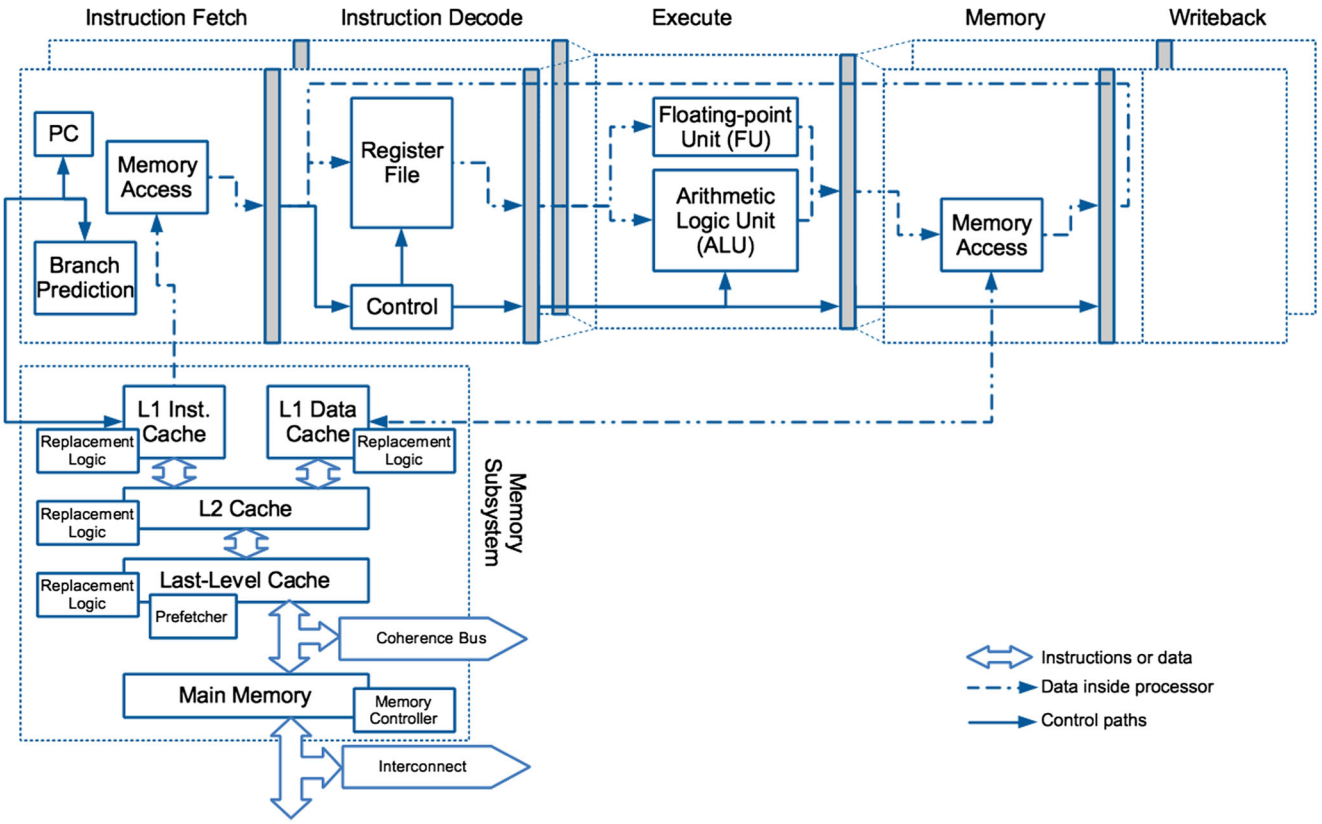

Fig. 2 The prototypical two-thread SMT processor with 5-stage processor pipelines shown with shared execution stage and key components of the memory subsystem

There are six 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 affects state of the functional units.
   (a) Results or timing of instructions is related to state of the functional units.
   (b) Based on history of executed instructions, entropy in the operation of the functional units changes.
4. Memory subsystem with its cache hierarchy, prefetchers, and other functional units contributes to timing channels.

3.1 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) 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 the past, designed single- and multi-cycle instructions. As the names imply, single-cycle instruction takes one processor cycle to execute. A multi-cycle instruction takes many cycles to execute. The fundamental complexity of certain operations means that the program timing will depend on the instructions in that program. By observing timing, attackers can potentially learn if the victim is executing fast instructions or the slow instructions, e.g., in number of non-constant time cryptographic software implementations, different instructions are used depending on the secret key bits. The fast and slow paths lead to information leaks, 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. Difference between logical operation and floating-point is on order of 10× cycles. However, the functional units themselves can be pipelined to lower the overheads. Meanwhile, a memory operation (discussed in detail later) can take over 100 s of cycles if data has to be fetched from the main

memory. Consequently, there is direct relationship between the entropy among execution units timing and the performance. Better performance implies higher entropy and more information leaks.

3.2 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 be beneficial. One example is hyper-threading, or simultaneous multithreading (SMT), where there are usually two or more pipelines per core, e.g., as shown in Fig. 2. 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 programs executing in parallel, one on each pipeline, will need exactly same functional units. Program A may do addition, while program B does memory access, in which case, each executes almost as if they had all the resources to themselves. A complication comes in when two of the programs from each pipeline attempt to perform the same operation. If two programs try, for example, to perform floating point operations, one will be stalled until the floating point unit is available. The contention is reflected in the timing. If a program performs a certain operation and it takes longer, 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.

Reductions in the contention can be addressed by duplicating the hardware. Today, there are multi-core processors without SMT, where each processor core has all resources to itself. However, likely equally, many processors employ SMT and research results show that it gives large performance gains with small overhead in area. E.g., SMT chip with two pipelines and shared execution stages is about 6% larger than a singe-thread processor [15]. A two or more 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.

3.3 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 of the executed instructions. Thus, output of a stateful functional unit depends on past inputs. Observing the current output leaks information about the past computations in which that unit was involved. A specific example based on the branch predictor is given below.

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 until 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 misprediction and that wrong instructions were fetched.

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 [16]. To achieve such good prediction rate, branch predictors collect global histories (based on execution of all programs) and local histories (based on specific memory addresses). Because of global history component of the branch predictors, different programs affect the branch predictor and thus each other. A pathological program can "train" the branch predictor to mispredict certain branches. Then, when another program executes, it may experience many mispredictions 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) affects state of various functional units that later affects others programs' timing.

3.4 Memory Subsystem Timing Channels

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 have focused on side and covert channels due the memory subsystem (e.g., [3, 6, 12–14, 29, 33, 43, 47, 48, 54, 55, 62, 64, 65, 81, 82]).

3.4.1 Caches

Recall from Fig. 2 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 megabytes, but accessing data in L3 takes 10 s of cycles. Finally, there is the main memory, sized in gigabytes, but requiring 100 s of cycles to access.

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 the memory. Unfortunately, the fastest caches, closest to the processor, are also smallest, so there needs to be some policy on which data to keep in the cache. Often, the policy is some variant of the least recently used (LRU) policy 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 the processor to bring into the caches new data, and kick out least recently used data back to lower-level caches 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 set, as shown in Fig. 3. 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.

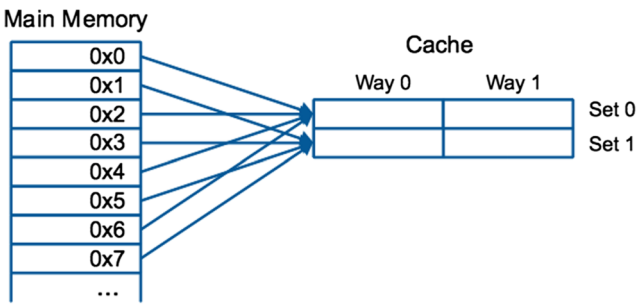

Fig. 3 Example of a two-way set-associative cache. Data from each memory location is assigned to a specific set, based on the address. Multiple ways allow storing multiple pieces of data in the same set. The LRU policy is used within each set

Such design of the caches lends itself easily to contention and interference, which in turn leads to information leakage that is typically due to timing. The leakage can reveal whether some data is in the cache or not. Accessing data in L1 takes 1 to 2 cycles, while data in memory can take 100 cycles or more. Eliminating such leakages is difficult. Ideally, any data could be stored anywhere in the cache and the cache replacement logic could search the whole cache for least recently used data, rather than just within a set. To this end, a fully associative cache is not as susceptible to timing attack as all data is mapped to one set and a miss does not carry as much information to the attacker. A fully associative cache is, however, expensive in terms of power and performance. Some proposals to break the relationship between the data accessed and the cache set include randomized caches (discussed in more detail later) [72]. However, again, if execution of different (memory) instructions takes different amount of time, this can lead to potential side or covert channels.

3.4.2 TLBs

Translation look-aside buffers (TLBs) are small cache-like functional units which are used to store translation between virtual addresses and physical addresses. Just like caches, they are susceptible to side and covert channel attacks as the mapping between the virtual address and the TLB set into which it is mapped is known and is similar to a set-associative cache. Some works have explored TLB-based attacks, including in Intel's SGX architecture [68].

3.4.3 Prefetcher

Another component of the memory subsystem is the prefetcher, which is used in microprocessors to improve the execution speed of a program by speculatively bringing 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, avoiding much slower access to the main memory. 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.

Hardware prefetchers attempt 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, and block x + 40 [59].

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. While currently no prefetcher-based attack has been presented in literature, one can likely be created based on the behavior just described. Disabling a prefetcher could prevent such attack, but has performance penalty.

3.4.4 Speculative Execution

Related to memory and prefetchers is the speculative execution feature supported by modern processors. Processors can speculatively fetch and execute in, e.g., when a branch is not yet know to be correctly predicted, and later cancel the instructions if there was a misprediction.

As has been shown by recent Spectre [38] and Meltdown [44] vulnerabilities, this can lead to security attacks. If speculative execution alters the state of the processor, any side effects have to be undone if the speculation was wrong. If this is not done, private data of a process (that was fetched speculatively) may remain in the processor, allowing another process to potentially access it. Spectre and Meltdown leverage speculative execution to modify the state of the processor cache. Because processor cache state is not properly leaned up in today's processors after mis-speculation is detected, a cache timing attack can be used to learn sensitive information that was use to modify cache state during the speculative execution.

3.4.5 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 from 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 controller which is a share resource becomes a point of contention, as shown in Fig. 4. 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 for handling 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 the 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., one 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 and 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.

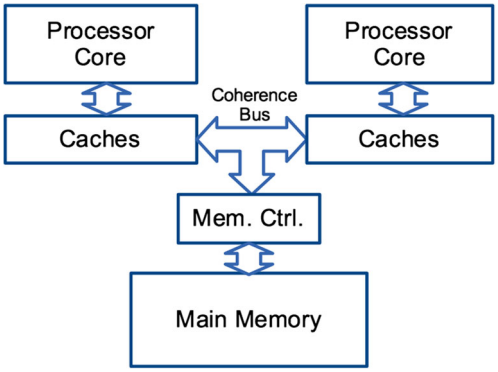

Fig. 4 Example of a dual-core platform with two processing cores sharing one memory controller and DRAM

3.4.6 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) [69]. The interconnect is used to send data between processors and also for memory accesses in non-uniform memory architectures (NUMA) where the main memory is divided up and separate DRAM chips and memory controllers are located near each processor, as shown in Fig. 5. 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.

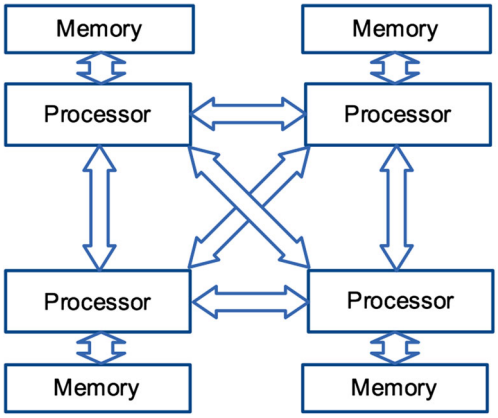

Fig. 5 Example of 4-processor setup with point-to-point (QPI-like) interconnect between them. Each processor has its own associated memory, but all processors can access the other memories via the interconnect, a typical NUMA configuration for systems with a single global address space

Thus, memory access timing can reveal the state of the interconnect and leak information about what other processes are doing.

3.5 Virtualization Features

Modern processor contain virtualization features, such as Intel's VT-x [66]. The main addition to support virtualization in modern processors is addition of the new privilege level, the "ring -1," where the hypervisor system management software, also called virtual machine monitor, runs. Along with the new privilege level, modern processors add hardware support for nested page tables. These features do not directly contribute side or covert channels. However, the larger trend of virtualization, and cloud computing that is based on it, means that more and more software runs in a virtualized environment. The attackers can run software within same virtual machine (VM) as the victim, or even run their own VMs along side victim's VM in a public cloud setting. The cloud computing trend, co-location of VMs, and the virtualization give more opportunities for attackers to exploit side and covert channels.

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. To date, more attacks have been presented for non-virtualized systems as these have been around for longer.

4.1 Attacks in Non-virtualized Settings

4.1.1 Caches

Traditional side [40] and covert [76] channels based on timing have been explored since 1990s. One of first theoretical uses of processor cache memory for a side-channel attack was shown in 2002 [52]. Timing attacks due to sharing of data caches have been widely studied in literature, with main focus on attacks on cryptographic protocols and algorithms. Typically, these attacks in numerous ways leverage processor data caches [3, 6, 12–14, 29, 33, 43, 47, 48, 54, 55, 62, 64, 65, 81, 82]. 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 (e.g., attacker's) data gets kicked out of the cache, which can be measured using timing, that means that another program (e.g., victim) has accessed data that maps to the same cache set. The attacker often can make association between the cache hit or miss and the victim's operation because of some known information. For example, typically the attack may be on AES S-box where the attacker knows the S-box and how it is located in the memory.

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

Most of the attacks focus on first-level caches, but recent work [80] utilized shared last-level cache (LLC) attack to extract cryptographic keys. The L1 caches 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.

4.1.2 Data Path Units

Beyond caches, integer and floating-point units have been found to be sources of information leaks [7]. These are classic examples of contention that causes different timing and can leak information. When two or more programs access a shared 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 [69].

In addition, covert channels leveraging exceptions due to shared functional units and speculative load instructions in simultaneously multithreading (SMT) processors have been shown [71]. SMT processors allow multiple processor pipelines to share some hardware, on assumption that on average not all programs will do exactly the same functional units, so fewer units are implemented. However, when the average case does not hold, contention arises, causing timing channels.

4.1.3 Control Path Units

Side channel attacks through branch prediction units [4, 5, 36] 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 predictions or mispredictions 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.

4.1.4 System Bus and Interconnect

Channels have also been presented based on the Quick Path Interconnect (QPI) lock mechanism in Intel's processors [69]. Some channels become rediscovered over time, e.g., with first processor bus contention channel presented in 1995 [31] and recently applied to the modern memory bus [57].

The interconnect and memory bus are a shared resource, and these channels are further examples 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.

4.1.5 On-chip Environmental Sensors

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 are 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.

4.2.1 Caches

One of the first attacks in a virtualized setting was a cross-VM covert channel that exploited the shared cache [56]. Such cross-VM side channels have been used to extract private keys from VMs [83]. Researchers have moreover leveraged L2 caches to create covert channels in virtualized environments [79]. Interesting resource-freeing attacks improve an attacker's VM's performance by forcing interference with a competing VM [67]. Leveraging some hypervisor software features in combination with hardware memory, researchers have also utilized sharing of redundant pages in memory via deduplication mechanisms [46, 60] to create communication channels.

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 read operations to see which memory was kicked out by another program, allowing them to make an educated guess about which data or instructions were executed.

4.2.2 System Bus and Interconnect

In [78], researchers have exploited the memory bus as a high-bandwidth covert channel medium. Specifically, memory controllers have been shown by these researchers to be a source of potential channels and leaks. 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 the 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 explored for attacks in non-virtualize environments, and also the first ones in virtualized 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 is difficult to quantify the contribution of the different units to the performance of the processor as 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 bits or bits of sensitive information for their target application. Nevertheless, most of the rates are in ranges of kilobits per second (kbps) or even megabits per second (Mbps) in optimal or idealized setting.

One of the first side channel attacks was the 0.001-bps Bernstein's AES cache attack using L1 cache collisions [11]. Bonneau improved the attack to about 1 bps [13]. Around same time, Percival reported attacks with about 3200 kbps using L1 cache-based covert channel and 800 kbps using L2 cache-based covert channel, which reduce to few kilobits per second when they were done in a realistic setting [54]. Besides caches, a 1-bps proof-of-concept channel due to branch predictor [23] was presented. The work has since been updated [24] and latest results show about 120-kbps channel. Further, units inside the processor that have been exploited are the thermal sensors and recent work has shown 300-bps covert channel [10] that leverages these on-chip sensors.

Other set of researchers have focused on virtualization and virtual machines. Ristenpart et al. showed 0.006-bps memory bus contention channel across VMs [56] and also 0.2-bps cross-VM L2 access-driven attack [56]. Xu et al. presented 262-bps L2 cache-based covert channel in a virtualized environment [79]. Zhang et al. show 0.02-bps L1 cache-based covert channel across VMs, using IPIs to force attacker VM to interrupt victim VM [83]. Wu et al. show 100-bps channel on Amazon's EC2 due to shared main memory interface in symmetric multi-processors [78]. Hunger et al. show up to 624-kbps channel when sender and receiver can have a well optimized and have aligned clock signals [36].

4.5 Attack Bandwidth Analysis

The Orange Book, also called the Trusted Computer System Evaluation Criteria (TCSEC), sets the basic requirement for trusted computer systems [50]. The Orange Book specifies that a channel bandwidth exceeding a rate of 100 bps is a high-bandwidth channel. It can be seen that many idealized attacks are well above that rate, but there is also quite a 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 less directly. For example, many branch instructions are needed to re-train the branch predictor, which leads to lesser bandwidth channel.

New attacks do not necessarily have bandwidth better than prior attacks. Usually, 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. Nevertheless, overall bandwidths are getting higher and have reached the bounds set by TCSEC for "high-bandwidth channels.'

When considering idealized attacks which are on the order of 100 s 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 tend to be also synchronous, where the attacker and victim are executing while synchronized (e.g., attacker and victim alternate execution on same processor core). 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. A dedicated attacker can thus come up with clever ways of

improving bandwidth by having more synchronous attacks, for example.

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 defenses, many of which can be deployed today, but at cost of performance.

5.1 Analyzing Side and Covert Channels Susceptibility

Microarchitectural side and covert channels arise because of functional unit sharing and 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 a resulting 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.

5.1.1 Design-Time Approaches

One of the first works on design-time mitigation of timing channels looked at shared resource matrix methodology to identify storage and timing channels [37]. In [49], a method to separate timing flows from other flows of information was presented and researchers have tried to define formal basis for hardware information flow security based on this method. Also, a side channel vulnerability factor has been presented as a metric for measuring information leakage in processors [19, 20].

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—the 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.

5.1.2 Run-Time Approaches

A number of run-time approaches have been proposed. Detection based on entropy-based approach [27] or dynamically tracking conflict patterns over the use of shared processor hardware have been shown [17]. Attempts to detect malware through analyzing existing performance counters have been proposed [21], or by using other hardware supported lower-level processor features [61].

One new innovative run-time approach that stands out uses groups of VMs and L2 cache contention to detect the attackers' VMs [82]. 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 the authors, by analyzing cache usage through memory timings, "friendly" VMs coordinate to avoid accessing specific cache sets and can detect the activity of a co-resident "foe" VM if the cache sets do get accessed.

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 a side or a 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.

5.2 Defending Side and Covert Channels

The analysis of the hardware and software has led 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 at design-time and software approaches at run-time.

5.2.1 Hardware Approaches

To mitigate side and covert channels, hardware architectural changes have been proposed including partitioning or time-multiplexing caches [53, 72], which have since been improved [41]. Cache design which actively reserves cache lines for a running thread and prevents other threads from evicting reserved lines have also been shown [22]. Such approaches essentially reserves 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 the memory it is accessing. In addition, other applications cannot use the reserved cache blocks, effectively cutting down on cache size and the performance.

One of the best proposals focuses on a new type of randomized caches [73]. Today's commodity hardware has caches where the mapping between memory and cache sets is fixed and same for all applications. Randomized caches in effect change this mapping for each application, e.g., application A has address 0×0 mapped to set 1 and application B has address 0×0 mapped to set 0. This can help thwart external timing, but internal timing may still be an issue. While not designed with security as a goal, the Z-cache [58] may have some of the similar properties where it searches among many cache sets to find least recently used block for replacement. Thus, effective set size is larger, reducing applications' contention for the same set.

Most recently, work has turned to on-chip networks and ensuring non-interference [74] or providing timing channel protection [70]. In [74], authors redesign the interconnect to allow precise scheduling of the packets that flow across the interconnect. Data from each processor are grouped and carried together in "waves" while strictly non-interfering with other data transfers. In [70] observe, as we do, that due to shared resources, 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 has also been added for enforcing strong non-interference. One research proposal has included "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 [63]. This temporal partitioning again focuses on un-doing the original design where resources are shared at very fine granularity, and instead making it coarser, leading to small potential leaks. The tradeoff is the performance impact of locking parts of a processor for exclusive use of an application. The longer the application can keep the resources, the better the leak protection, but also more negative impact on performance of other applications.

Processor architects have also proposed the addition of random noise to hardware counters [45]. The key to any timing attacks is to be able to obtain a timing reference, either within the application or somewhere from outside. In [45] 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 the inside and outside of a computer system needs to be considered even when focusing on microarchitectural channels.

A number of processors have also introduced trusted execution environments (TEEs) which aim to protect software applications from untrusted operating systems or even virtual machines. They focus on isolation of the protected software from other, untrusted software. These processor architectures include Intel's SGX [18] or ARM's TrustZone [75]. While they aim to protect the software, numerous attacks, including side and covert channel attacks, have been found in these architectures (e.g., [68]). Focusing on just isolation on the logical level is not sufficient, and TEEs need to consider side and covert channels.

It should be noted, however, that some defense approaches can actually contribute to new problems. For example, Cache Allocation Technology in Intel processors can be used to protect against denial of service attacks and it has been proposed to use it for defending some cache timing channels. Meanwhile, these protective features can actually accelerate another type of attack, the Rowhammer attack [8]. When defending against side channels, it should be considered if other types of attacks may be aided or even new types of attacks become possible due to the new defensive features.

5.2.2 Software Approaches

Researchers have suggested clearing out leftover state in caches through frequent cache flushes [51]. This is clearly a performance degrading technique; nevertheless, it is able to help prevent side and covert channels, as all application data and code are flushed from memory on context switch and when application runs again, it observes the long time 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 mostly prevented, and internal timing can also be potentially 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, periodic 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 a good branch predictor hit rate.

Addition of noise has also been suggested. For example, [34] explores reducing channels by introducing fuzzy time. In the work, a collection of techniques is introduced that reduce the bandwidths of covert timing channels by adding noise to all clock sources available to a process; this

includes system time stamp counters and inputs from disk drives or network cards. In [30], authors introduce similar fuzzy time noise to help defeat bus contention channel.

Other works focus on making the time and all events deterministic, such as in deterministic OSes designs [9, 77]. 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 precise 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 [35]. 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 hardware 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. As long as software uses instructions where 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 already beyond the 100-bps threshold.

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 other users.

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

Section 3 listed a variety of functional units and how performance optimizations embedded in these units are unlikely to be removed, 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, which could become future side and covert channels.

Analysis of the processor hardware reveals units such as the prefetcher that keep internal state based 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 being 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 the last two decades, side and covert channel research has shown a variety of, often very clever, ways of exfiltrating information from a computer system. Processor timing-based 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 timing channels is critical as users have less and less control over environment where the software runs.

Acknowledgments The author would like to thank Bryan Ford and Dmitry Evtyushkin for suggesting recent work to add this survey. The author would also like to thank the anonymous reviewers for their feedback and comments.

Funding Information This work has been supported in part by grants 1651945, 1716541, and 1524680 from the United States' National Science Foundation. This work has further been supported in part through grant by Semiconductor Research Corporation (SRC).

References

1. Aciiçmez O. (2007) Yet another microarchitectural attack: exploiting i-cache. In: Proceedings of the Workshop on Computer Security Architecture. ACM, pp 11–18
2. Acıiçmez O, Brumley BB, Grabher P (2010) New results on instruction cache attacks. In: Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems. Springer, pp 110–124
3. Acıiçmez O, Koç ÇK (2006) Trace-driven cache attacks on AES (short paper). In: Information and Communications Security. Springer, pp 112–121
4. Acıiçmez O, Koç ÇK, Seifert JP (2006) Predicting secret keys via branch prediction. In: Topics in Sryptology–CT-RSA 2007. Springer, pp 225–242
5. Aciiçmez O, Koç CK, Seifert JP (2007) On the power of simple branch prediction analysis. In: Proceedings of the ACM Symposium on Information, Computer and Communications Security. ACM, pp 312–320
6. Acıiçmez O, Schindler W, Koç ÇK (2006) Cache based remote timing attack on the aes. In: Topics in Cryptology–CT-RSA 2007. Springer, pp 271–286
7. Aciicmez O, Seifert JP (2007) Cheap hardware parallelism implies cheap security. In: Proceedings of the Workshop on Fault Diagnosis and Tolerance in Cryptography. IEEE, pp 80–91
8. Aga MT, Aweke ZB, Austin T (2017) When good protections go bad: exploiting anti-DoS measures to accelerate rowhammer attacks. In: Proceedings of the International Symposium on Hardware Oriented Security and Trust. IEEE, pp 8–13
9. Aviram A, Hu S, Ford B, Gummadi R (2010) Determining timing channels in compute clouds. In: Proceedings of the workshop on cloud computing security, CCSW '10. ACM, New York, pp 103–108. https://doi.org/10.1145/1866835.1866854
10. Bartolini DB, Miedl P, Thiele L (2016) On the capacity of thermal covert channels in multicores. In: Proceedings of the European conference on computer systems. ACM, p 24
11. Bernstein DJ (2005) Cache-timing attacks on aes
12. Bogdanov A, Eisenbarth T, Paar C, Wienecke M (2010) Differential cache-collision timing attacks on aes with applications to embedded cpus. In: Topics in cryptology–CT-RSA, vol 10. Springer, pp 235–251
13. Bonneau J, Mironov I (2006) Cache-collision timing attacks against aes. In: Proceedings of the workshop on cryptographic hardware and embedded systems. Springer, pp 201–215
14. Brumley BB, Hakala RM (2009) Cache-timing template attacks. In: Advances in cryptology–ASIACRYPT 2009. Springer, pp 667–684
15. Burns J, Gaudiot JL (2002) Smt layout overhead and scalability. IEEE Trans Parallel Distrib Syst 13(2):142–155. https://doi.org/10.1109/71.983942
16. Championship branch prediction (2014). http://www.jilp.org/cbp2014/, accessed August 2015
17. Chen J, Venkataramani G (2014) Cc-hunter: uncovering covert timing channels on shared processor hardware. In: Proceedings of the international symposium on microarchitecture. IEEE Computer Society, pp 216–228
18. Costan V, Devadas S (2016) Intel sgx explained. IACR Cryptology ePrint Archive 2016(086):1–118
19. Demme J, Martin R, Waksman A, Sethumadhavan S (2012) Side-channel vulnerability factor: a metric for measuring information leakage. In: ACM SIGARCH computer architecture news, vol 40. IEEE Computer Society, pp 106–117

20. Demme J, Martin R, Waksman A, Sethumadhavan S (2013) A quantitative, experimental approach to measuring processor side-channel security. IEEE Micro 33:68–77
21. Demme J, Maycock M, Schmitz J, Tang A, Waksman A, Sethumadhavan S, Stolfo S (2013) On the feasibility of online malware detection with performance counters. ACM SIGARCH Computer Architecture News 41:559–570
22. Domnitser L, Jaleel A, Loew J, Abu-Ghazaleh N, Ponomarev D (2012) Non-monopolizable caches: low-complexity mitigation of cache side channel attacks. ACM Transactions on Architecture and Code Optimization (TACO) 8(4):35
23. Evtyushkin D, Ponomarev D, Abu-Ghazaleh N (2015) Covert channels through branch predictors: a feasibility study. In: Proceedings of the workshop on hardware and architectural support for security and privacy. ACM, p 5
24. Evtyushkin D, Ponomarev D, Abu-Ghazaleh N (2016) Understanding and mitigating covert channels through branch predictors. ACM Transactions on Architecture and Code Optimization 13(1):10
25. Freiling FC, Schinzel S (2011) Future challenges in security and privacy for academia and industry. In: Proceedings of the international information security conference. pp 41–55
26. Gandolfi K, Mourtel C, Olivier F (2001) Electromagnetic analysis: concrete results. In: Proceedings of the workshop on cryptographic hardware and embedded systems. Springer, pp 251–261
27. Gianvecchio S, Wang H (2007) Detecting covert timing channels: an entropy-based approach. In: Proceedings of the conference on computer and communications security. ACM, pp 307–316
28. Gold B, Linde R, Cudney P (1984) Kvm/370 in retrospect. In: Proceedings of the symposium on security and privacy. IEEE, pp 13–13
29. Grabher P, Großschädl J, Page D (2007) Cryptographic side-channels from low-power cache memory. In: Cryptography and coding. Springer, pp 170–184
30. Gray III JW (1993) On introducing noise into the bus-contention channel. In: Proceedings of the symposium on research in security and privacy. IEEE, pp 90–98
31. Gray III JW (1994) Countermeasures and tradeoffs for a class of covert timing channels. Hong Kong University of Science and Technology
32. Gullasch D, Bangerter E, Krenn S (2011) Cache games–bringing access-based cache attacks on aes to practice. In: Proceedings of the symposium on security and privacy. IEEE, pp 490–505
33. Henricksen M, Yap WS, Yian CH, Kiyomoto S, Tanaka T (2010) Side-channel analysis of the k2 stream cipher. In: Proceedings of the information security and privacy. Springer, pp 53–73
34. Hu WM (1991) Reducing timing channels with fuzzy time. In: Proceedings of the symposium on research in security and privacy. IEEE, pp 8–20
35. Hu WM (1992) Lattice scheduling and covert channels. In: Proceedings of the symposium on research in security and privacy. IEEE, pp 52–61
36. Hunger C, Kazdagli M, Rawat A, Dimakis A, Vishwanath S, Tiwari M (2015) Understanding contention-based channels and using them for defense. In: Proceedings of the international symposium on high performance computer architecture. IEEE, pp 639–650
37. Kemmerer RA (1983) Shared resource matrix methodology: an approach to identifying storage and timing channels. ACM Transactions on Computer Systems (TOCS) 1(3):256–277
38. Kocher P, Genkin D, Gruss D, Haas W, Hamburg M, Lipp M, Mangard S, Prescher T, Schwarz M, Yarom Y (2018) Spectre attacks: exploiting speculative execution arXiv e-prints
39. Kocher P, Jaffe J, Jun B (1999) Differential power analysis. In: Advances in cryptology–CRYPTO-99. Springer, pp 388–397
40. Kocher PC (1996) Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In: Advances in cryptology–CRYPTO-96. Springer, pp 104–113
41. Kong J, Aciiçmez O, Seifert JP, Zhou H (2009) Hardware-software integrated approaches to defend against software cache-based side channel attacks. In: Proceedings of the international symposium on high performance computer architecture. IEEE, pp 393–404
42. Lampson BW (1973) A note on the confinement problem. Commun ACM 16(10):613–615
43. Leander G, Zenner E, Hawkes P (2009) Cache timing analysis of lfsr-based stream ciphers. In: Cryptography and coding. Springer, pp 433–445
44. Lipp M, Schwarz M, Gruss D, Prescher T, Haas W, Mangard S, Kocher P, Genkin D, Yarom Y, Hamburg M (2018) Meltdown arXiv e-prints
45. Martin R, Demme J, Sethumadhavan S (2012) Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In: ACM SIGARCH computer architecture news, vol 40. IEEE Computer Society, pp 118–129
46. Miłós G, Murray DG, Hand S, Fetterman MA (2009) Satori: enlightened page sharing. In: Proceedings of the USENIX annual technical conference. pp 1–1
47. Neve M, Seifert JP (2007) Advances on access-driven cache attacks on aes. In: Proceedings of the selected areas in cryptography. Springer, pp 147–162
48. Neve M, Seifert JP, Wang Z (2006) A refined look at Bernstein's aes side-channel analysis. In: Proceedings of the ACM symposium on information, computer and communications security. ACM, pp 369–369
49. Oberg J, Meiklejohn S, Sherwood T, Kastner R (2013) A practical testing framework for isolating hardware timing channels. In: Proceedings of the conference on design, automation and test in Europe. EDA Consortium, pp 1281–1284
50. DoD 5200.28-STD (1983) Department of Defense Trusted Computer System Evaluation Criteria (DoD 5200.28-STD), 1983. Available online http://csrc.nist.gov/publications/history/dod85.pdf
51. Osvik DA, Shamir A, Tromer E (2006) Cache attacks and countermeasures: the case of aes. In: Topics in cryptology–CT-RSA 2006. Springer, pp 1–20
52. Page D (2002) Theoretical use of cache memory as a cryptanalytic side-channel. IACR Cryptology ePrint Archive 2002:169
53. Page D (2005) Partitioned cache architecture as a side-channel defence mechanism. IACR Cryptology ePrint Archive 2005:280
54. Percival C (2005) Cache missing for fun and profit
55. Rebeiro C, Mukhopadhyay D, Takahashi J, Fukunaga T (2009) Cache timing attacks on clefia. In: Progress in cryptology–INDOCRYPT 2009. Springer, pp 104–118
56. Ristenpart T, Tromer E, Shacham H, Savage S (2009) Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In: Proceedings of the conference on computer and communications security. ACM, pp 199–212
57. Saltaformaggio B, Xu D, Zhang X (2013) Busmonitor: a hypervisor-based solution for memory bus covert channels. In: Proceedings of EuroSec
58. Sanchez D, Kozyrakis C (2010) The zcache: decoupling ways and associativity. In: Proceedings of the international symposium on microarchitecture. IEEE, pp 187–198
59. Shen JP, Lipasti MH (2013) Modern processor design: fundamentals of superscalar processors. Waveland Press Inc., Long Grove IL, USA

60. Suzaki K, Iijima K, Yagi T, Artho C (2011) Software side channel attack on memory deduplication. SOSP Poster
61. Tang A, Sethumadhavan S, Stolfo SJ (2014) Unsupervised anomaly-based malware detection using hardware features. In: Research in attacks, intrusions and defenses. Springer, pp 109–129
62. Tiri K, Acıiçmez O, Neve M, Andersen F (2007) An analytical model for time-driven cache attacks. In: Proceedings of the fast software encryption. Springer, pp 399–413
63. Tiwari M, Li X, Wassel HM, Chong FT, Sherwood T (2009) Execution leases: a hardware-supported mechanism for enforcing strong non-interference. In: Proceedings of the international symposium on microarchitecture. ACM, pp 493–504
64. Tromer E, Osvik DA, Shamir A (2010) Efficient cache attacks on aes, and countermeasures. J Cryptol 23(1):37–71
65. Tsunoo Y, Saito T, Suzaki T, Shigeri M, Miyauchi H (2003) Cryptanalysis of des implemented on computers with cache. In: Proceedings of the workshop on cryptographic hardware and embedded systems. Springer, pp 62–76
66. Uhlig R, Neiger G, Rodgers D, Santoni AL, Martins FC, Anderson AV, Bennett SM, Kagi A, Leung FH, Smith L (2005) Intel virtualization technology. Computer 38(5):48–56
67. Varadarajan V, Kooburat T, Farley B, Ristenpart T, Swift MM (2012) Resource-freeing attacks: improve your cloud performance (at your neighbor's expense). In: Proceedings of the conference on computer and communications security. ACM, pp 281–292
68. Wang W, Chen G, Pan X, Zhang Y, Wang X, Bindschaedler V, Tang H, Gunter CA (2017) Leaky cauldron on the dark land: understanding memory side-channel hazards in sgx. In: Proceedings of the conference on computer and communications security. ACM, pp 2421–2434
69. Wang Y, Ferraiuolo A, Suh GE (2014) Timing channel protection for a shared memory controller. In: Proceedings of the international symposium on high performance computer architecture. IEEE, pp 225–236
70. Wang Y, Suh GE (2012) Efficient timing channel protection for on-chip networks. In: Proceedings of the international symposium on networks on chip. IEEE, pp 142–151
71. Wang Z, Lee RB (2006) Covert and side channels due to processor architecture. In: Proceedings of the annual computer security applications conference. IEEE, pp 473–482
72. Wang Z, Lee RB (2007) New cache designs for thwarting software cache-based side channel attacks. In: ACM SIGARCH computer architecture news, vol 35. ACM, pp 494–505
73. Wang Z, Lee RB (2008) A novel cache architecture with enhanced performance and security. In: Proceedings of the international symposium on microarchitecture. IEEE, pp 83–93
74. Wassel HM, Gao Y, Oberg JK, Huffmire T, Kastner R, Chong FT, Sherwood T (2013) Surfnoc: a low latency and provably non-interfering approach to secure networks-on-chip. ACM SIGARCH Computer Architecture News 41(3):583–594
75. Winter J (2012) Experimenting with arm trustzone–or: how i met friendly piece of trusted hardware. In: Proceedings of the international conference on trust, security and privacy in computing and communications. IEEE, pp 1161–1166
76. Wray JC (1991) An analysis of covert timing channels. In: Proceedings of the symposium on research in security and privacy. IEEE, pp 2–7
77. Wu W, Zhai E, Jackowitz D, Wolinsky DI, Gu L, Ford B (2015) Warding off timing attacks in deterland. arXiv:1504.07070
78. Wu Z, Xu Z, Wang H (2012) Whispers in the hyper-space: high-speed covert channel attacks in the cloud. In: Proceedings of the USENIX security symposium. pp 159–173
79. Xu Y, Bailey M, Jahanian F, Joshi K, Hiltunen M, Schlichting R (2011) An exploration of l2 cache covert channels in virtualized environments. In: Proceedings of the workshop on cloud computing security. ACM, pp 29–40
80. Yarom Y, Falkner KE (2013) Flush+ reload: a high resolution, low noise, l3 cache side-channel attack. IACR Cryptology ePrint Archive 2013:448
81. Zenner E (2009) A cache timing analysis of hc-256. In: Proceedings of the selected areas in cryptography. Springer, pp 199–213
82. Zhang Y, Juels A, Oprea A, Reiter MK (2011) Homealone: co-residency detection in the cloud via side-channel analysis. In: Proceedings of the symposium on security and privacy. IEEE, pp 313–328
83. Zhang Y, Juels A, Reiter MK, Ristenpart T (2012) Cross-vm side channels and their use to extract private keys. In: Proceedings of the conference on computer and communications security. ACM, pp 305–316