Your Processor Leaks Information – and There's Nothing You Can Do About It

Qian Ge[1,2], Yuval Yarom[1,3], Frank Li[1,2], and Gernot Heiser[1,2]

[1]Data61, CSIRO
[2]UNSW, Australia
[3]School of Computer Science, The University of Adelaide
{*qian.ge,yuval.yarom,frank.li,gernot.heiser*}*@data61.csiro.au*

Abstract

Timing channels are information flows, encoded in the relative timing of events, that bypass the system's protection mechanisms. Any microarchitectural state that depends on execution history and affects the rate of progress of later executions potentially establishes a timing channel, unless explicit steps are taken to close it. Such state includes CPU caches, TLBs, branch predictors and prefetchers; removing the channels requires that the OS can partition such state or flush it on a switch of security domains. We measure the capacities of channels based on these microarchitectural features on several generations of processors across the two mainstream ISAs, x86 and ARM, and investigate the effectiveness of the flushing mechanisms provided by the respective ISA. We find that in all processors we studied, at least one significant channel remains. This implies that closing all timing channels seems impossible on contemporary mainstream processors.

1 Introduction

Computer hardware is increasingly being shared between multiple, potentially untrusted, programs. Examples of such sharing range from cloud services, where a single computer may share workloads of multiple clients, to mobile phones that run apps authored by different developers, to web browsers executing Javascript code originating from different sites. To protect confidential or private information that some of these programs may access, the system imposes a *security policy* that is designed to prevent information flow between different security domains (eg. VMs, apps or web pages).

One long-established threat to the security of shared systems are *covert channels* [27], which allow colluding programs to bypass the security policy, by transferring information over media that are not controlled by the system. A typical scenario includes two programs: a *Trojan* program as the sender, which has access to sensitive information but is confined [30] by the security policy (i.e. prevented from sending information to arbitrary destinations), and a *spy* program as the receiver, which does not have access to the sensitive information but can communicate with less restrictions.

Traditionally, covert channels were considered mainly in the context of military-grade multi-level secure systems [11, 28]. However, with the spread of cloud platforms and the proliferation of untrusted mobile code, including mobile-phone apps and third-party code running in web browsers, covert channels are becoming a mainstream security problem. Colluding mobile apps present a concrete example of the risk: Consider an app which operates on sensitive private data and is denied network access to prevent it from leaking the data. Such an app (Trojan) could use a covert channel to leak to a separate app (spy), which has unrestricted network access but no direct access to sensitive data. Similarly, a cloud-based web service might contain a Trojan which leaks secret data to a co-located spy VM, circumventing the encryption of client-server traffic.

Covert channels are usually classified as either *storage* or *timing* channels [43]. Storage channels represent information as some system state affected by the sender that can be sensed by the receiver, frequently exploiting operating system (OS) metadata as the storage medium. Past research has demonstrated that storage channels can be eliminated completely [36].

Here we focus on the open problem of timing channels, which exploit timing variations for communication. They are harder to deal with, partially because of the breadth of exploitable mechanisms. For example, secret-dependent code paths or data access patterns can lead to timing variations that can be exploited locally [37, 51] or even remotely [6], and the usual defence is to strive for deterministic execution time via constant-time algorithms [7].

Such timing channels are arguably a problem of user-

level software, although OS mechanisms may be employed to ensure deterministic timing of externally-visible effects [5, 9, 10]. However, there are other classes of timing channels that clearly fall into the responsibility of the OS.[1] This specifically includes timing channels resulting from microarchitectural state, such as caches, that is not explicitly exposed to user-level software. Such channels, from now on simply referred to as *microarchitectural channels*, are the topic of this paper.

Specifically, *we examine the degree to which it is possible to prevent microarchitectural timing channels on contemporary hardware.* Timing channels between concurrent executions on a single core (i.e. simultaneous multithreading, SMT) are well-documented and understood [4], are high-bandwidth, and are probably impossible to close, so we assume a security-conscious OS that disables SMT. We further restrict our examination to intra-core timing channels, i.e. between time-multiplexed users of a single core.

One might assume that elimination of microarchitectural channels in this restricted scenario would be trivially (although expensively) achieved by flushing all microarchitectural state on each switch of security context. However, as we will demonstrate, this assumption is wrong on contemporary mainstream hardware, and significant channels remain despite the best efforts of the OS. In short, **our findings mean that contemporary processors are inherently insecure, as they cannot be prevented from leaking information**.

We make the following contributions:

- We implement covert channels attacking all known caching microarchitectural features (Section 4.1), and identify and implement all known mitigation techniques for those channels that do not depend on undocumented/unreliable information (Section 4.2).
- We measure the capacity of those channels with and without mitigations on multiple generations of recent implementations of the two mainstream architectures, x86 and ARM (Section 5.2).
- We demonstrate that on each investigated processor, there are significant residual channels that cannot be closed with known techniques (Section 5.2).
- We find that an apparent I-cache remnant channel on x86 as well as ARM seems to result from instruction prefetching (Section 5.3).
- We show that the branch target buffer on x86 provides timing channels that cannot be mitigated (Section 5.4).
- We show the branch history buffer provides a channel that cannot be closed on x86 platforms and on more recent ARM platforms (Section 5.5).
- We demonstrate that flushing the TLB is insufficient for mitigating the TLB channel on x86 (Section 5.6)
- We argue the need for a new hardware-software contract, extending the ISA with mechanisms required to enforce confidentiality on time-shared cores (Section 6).

Our results show that, contrary to conventional wisdom, intra-core timing channels remain even when using all hardware-supported flush operations. Our results furthermore show that this is not a one-off defect of a particular implementation, but affects multiple implementations of both mainstream architectures.

2 Background

2.1 Microarchitectural State and Timing Channels

Modern processors contain a number of microarchitectural features that exploit temporal or spatial locality for improving average-case performance. Inherently, these features hold state that depends on recent execution history and affects the performance of subsequent execution. These are the data and instruction caches, the TLB, the branch predictor as well as code and data prefetchers. The branch predictor typically caches state in two places, the branch target buffer (BTB) and the branch history buffer (BHB).

On a context switch, the recent history and immediate future belong to different processes (and thus potentially security domains), which means that the execution of one domain can affect the timing of the execution of another, thus establishing a channel. In fact, timing attacks have been demonstrated via all of these microarchitectural features, see Ge et al. [15] for a survey.

Preventing such channels requires either avoiding any sharing of (or contention for) microarchitectural state, or ensuring that it is reset to a defined state (independent of execution history) on a switch of security domain, i.e. flushing. Partitioning by the OS is generally possible where resources are accessed by physical address (which is under OS control); eg. the L2 and lower-level caches can be partitioned by page colouring [25, 29, 46]. On-core resources, such as the L1 caches, TLB and branch predictors, are accessed by virtual address and thus cannot be partitioned by the OS.[2] In the absence of ex-

[1] We use the term "operating system" in a generalised sense, referring to the most privileged software level that has full control over the hardware. In a cloud scenario this would refer to the hypervisor, and the term "process" would represent a virtual machine.

[2] The L1 caches on some processors are said to be physically addressed. In reality this almost always means that the set-selector bits are a subset of the page offset, meaning that the lookup only uses address bits that are invariant under address translation. ARM seems to be using hardware alias-detection on at least some cores, to make the caches be have as physically indexed.

plicit hardware support for partitioning, channels based on such resources can only be prevented by flushing.

2.2 Covert channels

Where microarchitectural hardware state is not partitioned or flushed, a Trojan can, through its own execution, force the hardware into a particular state, and a spy can probe this state by observing its own progress against real time. This will constitute a covert channel, i.e. an information flow bypassing the system's security policy [27]. For example, the Trojan can modulate its cache footprint, encoding data into the number of cache lines accessed. The spy can read the data by observing the time taken to access each cache line. Or the Trojan can force the branch predictor state machine into a particular state, which the spy can sense by observing the latency of branch instructions (i.e. whether they are predicted correctly).

The actual implementations of covert channels depend on the details of the particular microarchitectural feature they exploit. A large number of such implementations have been described, as surveyed by Ge et al. [15].More such channels have been implemented and evaluated recently [12, 14, 34].

The threat of microarchitectural channels is not restricted to environments compromised by Trojans. A *side channel* is a special case of a covert channel, which does not depend on a colluding Trojan, but instead allows a spy program to recover sensitive information from a non-colluding *victim*. Where they exist, side channels pose a serious threat to privacy and can be used to break encryption [15].

Collusion allows better utilisation of the underlying hardware mechanism and hence covert channels tend to have much higher bandwidth than side channels based on the same mechanism; the capacity of the covert channel is the upper bound of the corresponding side channel capacity.

We focus on covert channels, as the existence of a covert channel means that there is a potential side channel as well, particularly where the timing channel allows the receiver to obtain address information from the sender, as is the case in cache-based channels. Furthermore, closing a covert channel implicitly eliminates side channels. For that reason we focus on covert channels in this work, as we aim to establish the degree to which microarchitectural timing channels can be eliminated.

Historically, covert channels were mostly discussed within the scope of multilevel security (MLS) systems [11, 28]. Such systems have users with different classification levels and the system is required to ensure that a user with a high security clearance, e.g. a Top Secret classification, does not leak information to users with a lower clearance. The security evaluation requirements for military-style separation kernels [41] require evaluating covert channels and limiting their capacity [23].

The advent of modern software deployment paradigms, including cloud computing, app stores and browsers executing downloaded code, increases the risk of covert channels. Consequently, channels in such environment have received recent attention [31, 34, 49]. Covert channels break the isolation guarantees of cloud environments, where workloads of several users are deployed on the same hardware. Similarly, mobile devices rely on the system's security policy to ensure privacy whilst executing software of multiple, possibly untrustworthy developers.

As indicated in the Introduction, we focus on channels that can be exploited by a Trojan and a spy who time-share a processor core. This scenario implicitly eliminates *transient-state* channels [4, 53], i.e. those that exploit the limited bandwidth of processor components, such as busses. These rely on concurrent execution of Trojan and spy, and hence do not exist in a time-sharing scenario. We thus only need to handle *persistent-state* channels, which rely on competing for storage capacity of processor elements.

2.3 Signalling techniques

Like any communication, a covert channel requires not only a communication medium (the microarchitectural state in our case) but also a protocol, i.e. exploitation technique.

In this work we use the Prime+Probe technique [37, 38], which is commonly used for exploiting timing channels in set-associative caching elements. It has been applied to the L1 D-cache [37, 38], L1 I-cache [3], and the LLC [32]. Several other techniques for exploiting cache channels have been suggested [18, 20, 37, 51], some of which demonstrate higher capacity. Our focus is not the maximum potential channel capacity nor the specific mechanism used. Instead we focus on establishing the existence of channels and whether they can be closed; and choose Prime+Probe as a generic approach for exploiting multiple channels.

In Prime+Probe, the spy primes the cache by filling some of the cache sets with its own data. The Trojan uses each of the sets that the spy primes to transmit one bit, leaving the corresponding cache line untouched for sending a zero, or replacing it with its own data for sending a one. The spy then probes the cache state to receive the information, measuring the time taken to access the data it primed; a long access time indicates that the Trojan replaced cache content, representing a one, a short access time a zero. We use a simplified implementation:

the input symbol is the total number of cache lines accessed by the Trojan (i.e. for simplicity we use a unary instead of a more efficient binary encoding).

3 Threat Model

We assume that the adversary manages to execute a Trojan program within a restricted security domain. For example, the adversary may compromise a service within the restricted domain and inject the Trojan's code, or she may be a corrupt developer that inserts malicious code into a software product used in the restricted domain. Executing within the restricted domain gives the Trojan access to sensitive data which the adversary wants, however the security policies confine the secure environment to prevent data exfiltration by the Trojan.

Additionally, the adversary controls a spy program which executes on the same computer as the Trojan, for example, in a different virtual machine or app. The spy is executing outside the restricted domain and consequently can communicate freely with the adversary, but it does not have access to the sensitive data. The adversary's aim is to exploit a microarchitectural covert channel in the shared hardware. If such a channel exists and is not controlled by the system, the Trojan can use the channel to send the sensitive data to the spy, which can then send the data to the adversary. In this work we investigate the degree to which the system can prevent the adversary from exploiting such covert channels.

We are focusing on time-shared use of processors, and thus ignore transient channels. As we are exploring microarchitectural channels, we exclude timing channels that are controlled by software. For example, the Trojan could create a timing channel by varying its execution time before yielding the processor. We note that the system can protect against such channels by padding the execution time of the Trojan following a yield [5, 9, 10]. Moreover, because we investigate the processor's ability to close the channel, we only investigate channels within the processor itself. External channels, such as the DRAM open rows channel [40], are outside the scope of this work.

4 Methodology

In this work we examine the level of support that manufacturers provide for eliminating microarchitectural timing channels in their processors. To this purpose, we implement multiple covert channels, identify the processor instructions and available information that can be used for mitigating the channels, and measure the capacity of the channels with and without the mitigation techniques. These steps are described in greater details below.

4.1 Channels

Cock et al. [10] investigates efficient software mitigation techniques against microarchitectural channels on ARM. We adopt their techniques, viewing a channel as a pipe into which a sender (the Trojan) places *inputs* drawn from some set I and which a receiver (the spy) observes as *outputs* from a set O. The inputs and output sets depend on the specific covert channel used.

We implement five channels, each targeting a different microarchitectural component. But we note that the microarchitectural features interact, so our attacks cannot be orthogonal. For example, an attack targeting the I-cache channel will implicitly probe the BTB, and thus may show a channel even if the I-cache does not actually carry state across security domains.

Except where noted we do not make any assumptions on the virtual-address-space layout of either the Trojan or the spy; the memory used for Prime+Probe attack may be allocated anywhere in the virtual address space.

We target the channels exploiting the L1 I-cache, L1 D-cache, TLB, branch target buffer and branch history buffer.

L1 data cache For attacking the D-cache [37, 38] we use the Prime+Probe implementation from the Mastik toolkit [50]. The input symbols enumerate the cache sets; to send a symbol s, the Trojan reads enough data to fill all ways of cache sets $0, 1, \ldots, s-1$. The spy first fills the whole cache with its own data, waits for a context switch (to the Trojan), and then measures the total time taken to read a data item from each cache set; the output symbol is the recorded time.

Note that we could use a more sophisticated encoding of the input symbols to increase capacity. However, the point is not to establish the maximum channel capacity, but to investigate the degree to which it can be mitigated. We therefore keep things as simple as possible.

L1 instruction cache The attack on the I-cache [1, 2] is identical to the L1 D-cache channel, except that instead of reading data, the programs execute code in memory locations that map to specific cache sets. The implementation, also taken from the Mastik code, uses a series of jumps.

Translation lookaside buffer For the TLB channel, the input set enumerates the TLB entries. The Trojan sends an input symbol, s, by reading a single integer from each of s consecutive pages. The spy measures the time to access a number of pages. In order to reduce self-contention in the spy, it only accesses half of the TLB.

A more sophisticated design would take into account the structure of the TLB and aim to target the individual associative sets, and exclude only the minimal set of pages the spy needs for its own execution. As before, we opt for simplicity rather than capacity.

```
1   #define X_4(a) a; a; a; a
2   #define X_16(a) X_4(X_4(a))
3   #define X_256(a) X_16(X_16(a))
4
5   #define JMP     jnc 1f; .align 16; 1:
6
7     xorl %eax, %eax
8     X_256(JMP)
9
10    rdtscp
11    movl %eax, %esi
12    and $1, %edi
13    jz  2f
14    X_256(nop)
15  2:
16    rdtscp
17    subl %esi, %eax
```

Figure 1: BPU channel code for the x86 architecture.

The only prior implementation of a TLB-based channel is that of Hund et al. [22], which uses an intra-process TLB side channel to bypass the protection of kernel address space layout randomisation (KASLR). We are not aware of any prior implementation of inter-process TLB channels and past work considers such channels infeasible because the x86 TLB used to be flushed on context switch [54]. However, recent x86 processors feature a tagged TLB when operating in 64-bit mode, and ARM processor TLBs have been tagged for a long time. As a result, TLB channels are feasible on modern hardware.

Branch history buffer For exploiting the BHB we use an approach similar to the residual state-based covert channel of Evtyushkin et al. [13]. In each time slice, the Trojan sends a single-bit input symbol. Trojan and spy use the same channel code for sending and receiving. The code, the x86 version of which is shown in Figure 1, consists of a sequence of conditional forward branches that are always taken (Line 8) which set the history to a known state. The next code segment (Lines 10–17) measures the time it takes to perform a branch (Line 13) that conditionally skips over 256 `nop` instructions (Line 14). The branch outcome depends on the least significant bit of register `%edi`. The return value of the code (register `%eax`) is the measured time. The ARM implementation is similar.

Because the code takes different paths depending on whether the input bit is set, there is a timing difference between these two paths. Another source of timing difference is the processor's prediction on whether the branch in Line 13 is taken. The channel exploits this timing difference.

To implement this channel, the Trojan and the spy map the code at the same virtual address, however each uses its own copy of the code (i.e. we do not rely on shared memory). To send an input symbol, the Trojan sets the least significant bit of the input register to the input symbol and repeatedly calls the code throughout the time slice. This sets a strong prediction of the branch's outcome. The spy, in its time slice, calls the code with the input bit cleared (branch taken) and uses the measured time as the output symbol.

Branch target buffer To our knowledge, no BTB-based covert channel has been demonstrated to date. To build the channel, we chain branch instructions into a probing buffer. The Trojan probes on the first s instructions, the input symbol, while the spy measures the time taken for probing the entire buffer. On ARM platforms, the number of branch instructions equals the known size of BTB. On x86 platforms, details of BTB are generally not specified by manufacturers, but can frequently be reverse-engineered [35]. According to Godbolt [16], the BTB contains 4096 entries on the Ivy Bridge microarchitecture. In our test, the Trojan probes from 3072 to 5120 `jmp` instructions, whereas the spy probes on 4096 `jmp` instructions. The `jmp` instructions are 16-byte aligned, therefore the probing buffers are bigger than the L1 I-cache (32 KiB).

4.2 Mitigations

The microarchitectural channels we examine in this work exploit timing variations due to the internal state of components of the processor core. The OS cannot mitigate by partitioning the components, as this is not supported by the hardware. Furthermore, the OS cannot force partitioning through the allocation of physical addresses as the resources are indexed virtually. The only mitigation strategy available is through hardware cache-flushing mechanisms or completely disabling features.

Architectural support on x86

The `wbinvd` instruction invalidates all of the entries in the processor caches [24]. Consequently, after the instruction executes, the caches are in a defined state, all lines invalid, and subsequent accesses to data and instructions are served from memory, until program locality leads to use of recently cached data/instructions.

When executing in 32-bit mode, reloading the page-table pointer register, `CR3`, invalidates all of the entries in the TLB and paging-structure caches except global mappings. For flushing the entries for global mappings, we reload the `CR0` register. In 64-bit mode the TLBs are tagged with PCID, we use `invpcid` to flush and invalidate both tagged and global TLB entries, including paging structure caches.

The architecture offers no instruction for flushing the state of the prefetcher. Instead, it allows disabling the data prefetcher by updating `MSR 0x1A4` [47], which avoids any shared state in this unit. However, there is no way to disable instruction prefetching.

In summary, on x86 we can flush all caches and the TLBs and disable the data prefetcher. We cannot clean or disable the instruction prefetcher or the branch predictor.

Architectural support on ARM

We use the `DCCISW` cache maintenance operation for invalidating the data cache and `ICIALLU` for the instruction cache. Because the L2 cache is normally implemented as a core-external cache, operations on that cache are not part of the ARM ISA, and we therefore use the appropriate manufacturer-specified operations on external cache controllers to invalidate cache ways. On the Sabre (A9) platform, we use the *clean and invalidate set* operations. On the Hikey (A53) and TX1 (A57) platforms, we use `DCCISW` and `ICIALLU` for flushing the L2 caches, which are the same operations as used for flushing the L1 caches, but targeted at a different cache level.

On all ARM processors we use `TLBIALL` to invalidate the entire unified TLB and `BPIALL` to invalidate all entries from branch predictors. On ARMv7, we disable branch prediction by clearing the Z-field in the `SCTLR` register; this operation is not supported on ARMv8. On the A9 we disable the L1-data prefetch by setting the `ACTLR` register. On the A53, we disable the data prefetcher by setting the CPU auxiliary control register.

In summary, on ARM we can flush all caches, branch prediction entries, and TLBs and disable the data prefetcher (except on the A57). On the A9 we disable the branch prediction. None of the ARM platforms allow us to flush or disable the instruction prefetcher.

Scrubbing state without architectural support

Where the architecture does not support flushing or disabling a stateful microarchitectural feature, the operating system can attempt to programmatically reset its state during a domain switch. For example, the context switch code can fill the L1 D-cache with kernel data, ensuring that the contents of the cache does not depend on prior computation. This approach is severely limited by the insufficient documentation of relevant microarchitectural features. While researchers have reverse-engineered some of these features [33, 35, 39, 52] or the algorithm used [8, 19, 48], the available information is still insufficient. Hence, any defences that depend on such undocumented microarchitectural properties are inherently brittle and may fail to work as soon as a new version of the processor hits the market. We therefore do not use such techniques, as our aim is to investigate the degree to which microarchitectural timing channels can be reliably closed.

4.3 Measuring the channel capacity

To establish whether there is a channel, we configure the Trojan to send a pseudo-random sequence of input symbols. The spy collects the output symbols; a channel exists if the distribution of output symbols depends on the input symbol.

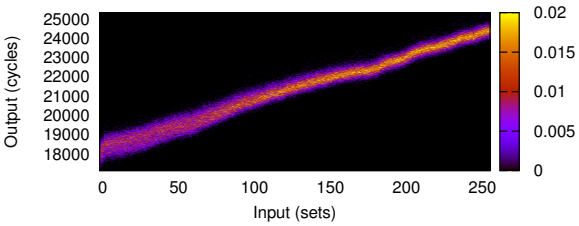

Figure 2: Channel matrix for the unmitigated L1-I covert channel on an ARM Cortex-A9. Colours indicate the probability of observing a particular output (total time in cycles for probing the buffer) for a particular input symbol (the number of cache sets the Trojan's code executes), as per the scale on the right. This example uses 909 samples for each input symbol and shows a channel capacity of 4 bits, we summarise this as $n = 909, \mathscr{C} = 4.0\,\mathrm{b}$.

4.3.1 The channel matrix

We use the technique of Cock et al. [10] to measure the information capacity of a channel. That is, we first create a *channel matrix*, which specifies the conditional probability of an observed output symbol given an input symbol. For example, Figure 2 shows the channel matrix we observe from the L1 I-cache of an ARM Cortex-A9 processor without any countermeasures (see Section 5.1 for description of the hardware platform). As we can see, there is a strong correlation between the input symbol (number of cache sets that the Trojan occupies after each run) and the output symbol (total time to jump through every cache line in a cache-sized buffer), resulting in a strong channel. In the absence of a channel the graph would show no horizontal variation.

4.3.2 Channel capacity

As a measure of the channel capacity, we calculate the *Shannon capacity* [45], denoted by $\mathscr{C}$, from the channel matrix. $\mathscr{C}$ indicates the average number of bits of information that a computationally unbounded receiver can learn from each input symbol. For the channel matrix in Figure 2 we find that $\mathscr{C} = 4.0\,\mathrm{b}$. Because the Trojan sends one of 257 possible values ($0 \cdots 256$), the maximum capacity expected is theoretically 8.0 bits per symbol. The observed capacity is smaller than the maximum due to sampling error, which leads to a distribution of output values even if there was a unique mapping from input to output symbols. Observable capacity could

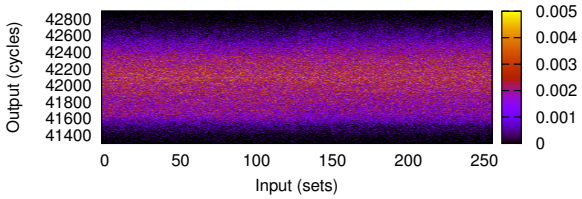

Figure 3: Channel matrix for the L1-I channel on the A9 with cache invalidation. $n = 3823, \mathscr{C} = 0.70\,\text{b}, \mathscr{C}_0 = 0.38\,\text{b}$.

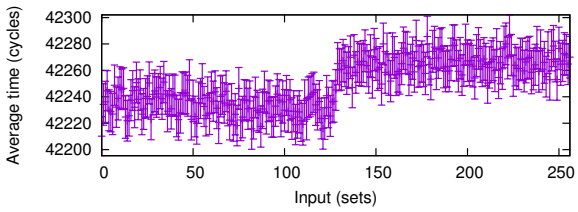

Figure 4: Average probe times, with 95% confidence intervals, for the mitigated L1-I covert channel on ARM Cortex-A9.

also be reduced through undocumented interference by other hardware features, such as replacement policy or prefetching.

4.3.3 Channel bandwidth

The channel bandwidth can now be calculated by multiplying the capacity by the input symbol rate, i.e. the frequency with which the Trojan can send different symbols. As the processor is time-shared between the Trojan and the spy, the Trojan can send at most one symbol per time slice. In the case of Figure 2 we use a time slice of 1 ms, meaning that per second the Trojan gets 500 opportunities to send a symbol, as the two processes run in alternating time slices. The potential bandwidth of the channel is therefore $500 \times 4 = 2,000$ b/s. We use different time slices on the various platforms, as specified in Table 1.

Much higher bandwidths have been demonstrated in the literature [12, 20, 32, 34]. One reason for our relatively low bandwidth is that our channel is between programs that time-share a processor, whereas the other channels were observed between concurrently executing processes, allowing for a much higher symbol transfer rate. Furthermore, as indicated in Section 4.1, we use an unsophisticated encoding, where the input is simply the number of cache sets accessed, resulting in 257 possible values. Instead we could map each bit of the input symbol's binary representation onto a cache set, resulting in 2^{256} values and a theoretical maximum capacity of 256 bits per symbol (although the actually achievable capacity would be much smaller, given the spread of the distribution). Instead of aiming for maximum capacity, our channels are designed to facilitate analysis.

4.3.4 Visualising low-capacity channels

Visualising the channel matrix as a heat map is useful for presenting the wealth of information that the matrix contains. However, for low-capacity channels, such a representation may fail to highlight the channel. For example, when mitigating the L1-I-cache channel on the A9 using methods for flushing and invalidating all caches (Section 4.2), we obtain the channel matrix of Figure 3. There is no channel evident, but the picture is fuzzy enough to hide a small channel of 0.7 b, about a sixth of the original capacity.

For better visualisation we compress the information of the channel matrix by averaging the output symbols for each input set, which yields the graph of Figure 4. We now clearly see the residual channel, with a clear difference between the average output symbol for small vs. large input values.

4.3.5 Verifying low-capacity channels

A challenge for our channel capacity measurement is that it is based on a sampled distribution. Sampling errors may result in an apparent non-zero channel, even when in reality there is no channel. Consequently, when the computed channel capacity $\mathscr{C}$ is low, we need to determine whether we observe a low-capacity channel or just sampling noise. For example, while Figure 4 clearly shows that there is a residual channel, even in the averaged graph it might not be obvious whether there is a channel or just sampling noise; we need a systematic way of detecting a statistically significant channel without relying on visual inspection. Specifically, we need to distinguish between the null hypothesis, that states that there is no channel, i.e. the distribution of output symbols is independent of the input symbols, and the alternative hypothesis that there is a channel.

Following Cock et al. [10], we bound the effect of sampling error by simulated sampling of a true zero-capacity channel. Specifically, we randomly distribute the output symbols collected across all input symbols and measure the capacity of this simulated channel. We repeat the process 1000 times, generating 1000 simulated capacities. If our measured sample (Figure 4) is drawn from a single distribution (i.e. no channel), there is a high likelihood that some of the simulated capacities would be higher than the observed $\mathscr{C}$. Thus, if $\mathscr{C}$ is bigger than the maximum, $\mathscr{C}_0$, of the 1000 simulated capacities, the probability that the sample is drawn from a single distribution is less than 0.1% and we reject the null hypothesis.

Table 1: Experimental hardware, covering 2–3 generations of microarchitectures across x86 and ARM. I-TLB and D-TLB represent first-level TLBs, L2 TLB represents the second-level (unified) TLB. "?" indicates unknown values.

Architecture		**x86**	**x86**	**x86**	**ARMv7**	**ARMv8**	**ARMv8**
Microarchitecture		**Sandy Br**	**Haswell**	**Skylake**	**A9**	**A53**	**A57**
Manufacturer		Intel	Intel	Intel	Freescale	HiSilicon	NVIDIA
Processor		i7-2600	E3-1220 v3	i7-6700	i.MX6	Kirin 620	Jetson TX1
Clock rate (GHz)		3.4	3.1	3.4	0.8	1.2	1.91
Year		2012	2013	2015	2012	2014	2015
Address size (bit)		32 or 64	32 or 64	32 or 64	32	32 or 64	32 or 64
Execution order		OoO	OoO	OoO	OoO	InO	OoO
Time slice (ms)		1	1	1	2	3	3
L1-D	size (KiB)	32	32	32	32	32	32
	line (B)	64	64	64	32	64	64
	sets×assoc.	64×8	64×8	64×8	256×4	128×4	256×2
L1-I	size (KiB)	32	32	32	32	32	48
	line (B)	64	64	64	32	64	64
	sets×assoc.	64×8	64×8	64×8	256×4	256×2	256×3
L2	size (KiB)	256	256	256	1024	512	2048
	line (B)	64	64	64	32	64	64
	sets×assoc.	512×8	512×8	512×8	2048×16	512×16	2048×16
L3	size (MiB)	8	8	8	N×A	N×A	N×A
	line (B)	64	64	64	N×A	N×A	N×A
	sets×assoc.	8192×16	8192×16	8192×16	N×A	N×A	N×A
BTB	size	?	?	?	512	256	256
	sets×assoc.	?	?	?	256×2	?	?
I-TLB	size	128	128	128	32	10	48
	sets×assoc.	32×4	16×8	16×8	32×1	10×1	48×1
D-TLB	size	64	64	64	32	10	32
	sets×assoc.	16×4	16×4	16×4	32×1	10×1	32×1
L2 TLB	size	512	1024	1536	128	512	1024
	sets×assoc.	128×4	128×8	128×12	64×2	128×4	256×4

Otherwise, we conclude that we cannot reject the null hypothesis and that the test is inconclusive (i.e. consistent with no channel).

In the case of Figure 3, we find that of the observed capacity $\mathscr{C} = 0.7$ b, at most $\mathscr{C}_0 = 0.38$ b can be attributed to sampling error, meaning that we have a true residual channel of at least $\mathscr{C} = 0.3\,b$. Given the 1 kHz context-switching rate, this implies a residual bandwidth of at least 150 b/s, enough to leak a typical RSA key in less than half a minute.

5 Results

5.1 Evaluation platforms

We examine processors from the two most widely-used architectures, x86 and ARM. For x86 these are the Sandy Bridge, Haswell and Skylake microarchitectures. For ARM we use a Cortex-A9, an implementation of ARMv7, and a Cortex-A53 and A57, in-order (InO) and out-of-order (OoO) implementations of ARMv8, the latest version of the architecture. We summarise their relevant features in Table 1. Note that the information on the branch predictors are incomplete, as there is very limited information available.

As we are probing hardware properties, the results should be independent of the OS or hypervisor used. However, the more complex the code base, the more complicated and error-prone it is to implement mitigations. Furthermore, with a large system, such as Linux or Xen, it becomes harder to preclude interference from various software components. We therefore use the seL4 microkernel [26, 44], which is a small (about 10,000 lines of code) and simple system. Furthermore, seL4's comprehensive formal verification includes proof of freedom from storage channels [36], which means any remaining channels must be timing channels, simplifying analysis of results.

Specifically, we use the separation kernel [42] configuration of seL4, where we run our Trojan and spy code "bare metal", with no actual OS services. This means that we are working in an almost noise-free environment, which helps measuring small-capacity channels.

5.2 Overview of results

Raw channels

The "none" rows in Table 2 show the unmitigated channel capacities of all the channels we test across our six processors. We see that most of the channels leak several bits per input symbol. The main exceptions are the I-cache channel on the recent x86 processors and the

Table 2: Observed unmitigated ("none") and maximally mitigated ("full") channel capacities $\mathscr{C}$ and maximum apparent capacities for empty channel $\mathscr{C}_0$ in bits. Channels that cannot be mitigated with all hardware-provided operations are marked in red.

Processor		Sandy Bridge	Haswell		Skylake	A9	A53	A57
Chan.	Mitig.	32-bit	32-bit	64-bit	64-bit	32-bit	32-bit	32-bit
L1-D	none	4.0	4.1	4.7	3.3	5.0	2.8	N/M
	full	0.038 (0.025)	0.083 (0.050)	0.43 (0.24)	0.18 (0.01)	0.11 (0.046)	0.15 (0.079)	N/M
L1-I	none	3.7	0.65	0.46	0.37	4.0	4.5	6
	full	0.85 (0.05)	0.25 (0.04)	0.36 (0.1)	0.18 (0.09)	1.0 (0.72)	0.5 (0.24)	2.8 (0.34)
TLB	none	3.2	3.1	3.2	2.5	0.33	3.4	N/M
	full	0.47 (0.29)	0.37 (0.23)	0.18 (0.1)	0.11 (0.06)	0.16 (0.087)	0.14 (0.08)	N/M
BHB	none	1.0	1.0	1.0	1.0	1.0	1.0	N/M
	full	1.0 (0.023)	1.0 (0.009)	1.0 (0.002)	1.0 (0.002)	0.009 (0.008)	0.5 (0.02)	N/M
BTB	none	2.0	4.6	4.1	1.8	1.1	1.3	N/M
	full	1.7 (1.2)	1.3 (0.4)	1.6 (0.5)	1.9 (1.2)	0.068 (0.032)	0.15 (0.08)	N/M

TLB channel on the A9, which have capacities in the range of 0.33–0.65 bits. We do not know the reasons for these lower capacities, but suspect that replacement policies might reduce the capacity seen by our simple attack, which works best with true LRU. A more sophisticated implementation, taking into account the (reverse-engineered) replacement policy, might show a bigger channel.

The main take-away from these results is that **all** microarchitectural features that cache recent execution history can be used for timing channels. This unsurprising result confirms that all such features must be taken into account when trying to stop timing-channel leakage.

Mitigated channels

The "full" rows in Table 2 show the observed channel capacities with all mitigations enabled. This goes beyond just flushing caches, but using all mitigations discussed in Section 4.2; we will discuss specific examples below.

From the table we make the surprising observation that **none of the channels are completely closed by employing all available hardware functions for removing the microarchitectural state**. All observed channels are larger than the statistical bound allowed by the null hypothesis. In fact, for each processor we find at least one channel that leaks over half a bit per symbol (which translates into a bandwidth of hundreds of bits per second).

While all residual channels are statistically significant, we highlight (in red font) the cases where the observed capacity is well above the sampling accuracy or where we can see from the visual representation that there is a definite channel.

In some cases the explanation for the remaining channels is straightforward. For example, Intel architectures do not support any method of clearing the state of the BPU. Consequently, the branch prediction channel remains unchanged even when we enable all of the protection provided by the processor (Section 5.5). In other cases the story is more intricate. We will explore some of them in more depth.

5.3 Finding 1: I-cache channel persists on all platforms

In particular, the L1-I cache channel remains even with all mitigation method enabled on all the testing platforms. This seems particularly surprising, as one would expect that a full cache flush should be sufficient to remove this channel. We pick one microarchitecture from each ISA to explore further.

5.3.1 A9 I-cache channel

We have already done a partial examination of the A9's I-cache channel, when we used it as an example in Section 4.3. Recall that Figure 2 showed a clear linear correlation between input and output symbols. Recall further that a clear channel remained when invalidating the cache on context switches, as evidenced by the clear transition between two distinct distribuions in Figure 4.

On the A9, the distance between two addresses that map to the same cache set, known as the cache *stride* (and equal to the cache size divided by associativity), is 8 KiB. Clearly, the transition in Figure 4 occurs at 4 KiB, which matches the page size. This may indicate that the channel originates not from the cache itself, but from some part of the virtual memory unit, for example, from the TLB. Hence, clearing the TLB might eliminate this channel.

We next apply all of the countermeasures available on the A9, including TLB flush, which yields the average probing cost shown in Figure 5. Clearly, despite using everything the hardware gives us, there is still a significant channel: Input values smaller than 14 result in below-average output symbols, whereas symbols in the range 15-50 produce above-average output.

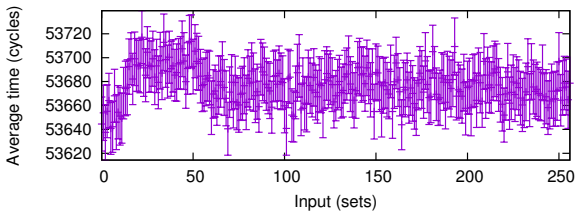

Figure 5: Average probe times, with 95% confidence intervals, for the L1-I I covert channel on A9 with all mitigations. $n = 909, \mathscr{C} = 1.0\,\text{b}, \mathscr{C}_0 = 0.72\,\text{b}$.

One possible explanation for the channel is that the processor maintains some state that is not cleared by flushing the caches and is not deactivated when disabling the data prefetcher. An alternative explanation is that the state is maintained outside the processor, for example resulting from a DRAM open-row channel [40].

To investigate further, we use the performance monitoring unit (PMU) to count the instruction-cache-dependent stall cycles (Figure 6). We observe that the stall cycles show the same trend for small input values as the probe cost of Figure 5. As these instruction dependent stalls are triggered by internal processor state, we can conclude that the channel is not (solely) caused by core-external features.

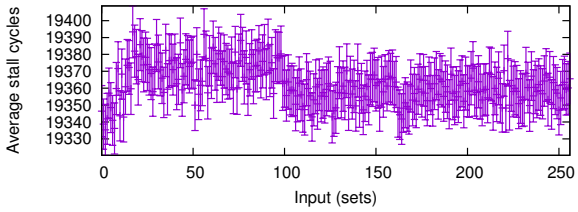

Figure 6: Average instruction cache dependent stall cycles with 95% confidence intervals for A9 with all mitigations.

A53 We similarly use the PMU to analyse the cause of the residual timing channel on the A53. We find that the number of L1 I-cache refills increases as the probing timing decreases (graph omitted for space reasons).

A possible explanation is that the state of instruction prefetcher, which cannot be disabled, might be affected by the Trojan. Because there is no official documentation on the implementation of the instruction prefetcher, we cannot investigate further.

5.3.2 Sandy Bridge I-cache channel

Figure 7 shows that without countermeasure this channel behaves like expected (and qualitatively similar to the A9). Again, flushing the cache is only moderately effective, reducing the capacity by not even 65%, although the channel matrix looks quite different, as shown in Figure 8.

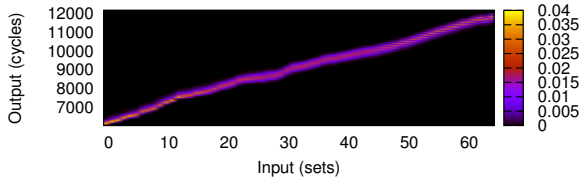

Figure 7: Channel matrix for the L1-I covert channel on Sandy Bridge without countermeasures. $n = 15420, \mathscr{C} = 3.7\,\text{b}$.

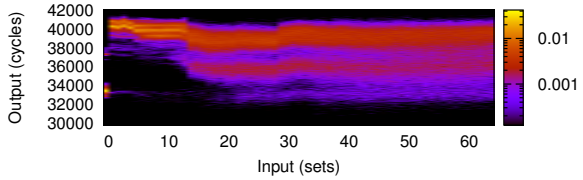

Figure 8: Channel matrix for the L1-I covert channel on Intel Sandy Bridge with cache invalidation. $n = 15415, \mathscr{C} = 1.4\,\text{b}$.

If we apply our remaining countermeasures, flushing the TLB and disabling the data prefetcher, a distinct channel remains, with a capacity of just under a quarter of the original, as shown in Figure 9. The PMU shows that the number of instruction cache, streaming buffer and victim cache misses (ICACHE.MISSES) increases as the probing time decreases. Again, this rules out core-external effects and makes us suspect contention between Trojan and spy on some unknown instruction prefetcher.

A potential alternative explanation is that the cache-flush operation does not operate as advertised. As instructions are normally read only, the I-cache is normally coherent with memory and does not need flushing. Could it be that the hardware is taking a shortcut and does not actually invalidate the I-cache?

To test this hypothesis we change our exploit code to re-write the probing buffer (a chain of jumps) before flushing the cache (using `clflush`). This makes the I-cache incoherent with memory, leaving the hardware no choice but perform the actual flush when requested. However, the residual timing channel remains on both 64-bit Haswell ($\mathscr{C} = 0.2\,\text{b}, \mathscr{C}_0 = 0.03\,\text{b}$) and Skylake machines ($\mathscr{C} = 0.2\,\text{b}$, $\mathscr{C}_0 = 0.06\,\text{b}$); graphs not shown for space reasons.

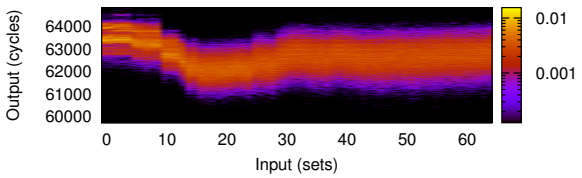

Figure 9: Channel matrix for the L1-I channel on Sandy Bridge with all mitigations. $n = 15415, \mathscr{C} = 0.85\,\text{b}, \mathscr{C}_\text{o} = 0.05\,\text{b}$.

5.4 Finding 2: BTB channel cannot be mitigated on x86

The I-cache results in Table 2 were obtained in a setup where the Trojan and spy had their buffers of chained `jmp` instructions allocated at the same virtual address. As far as the I-cache is concerned, the actual virtual addresses should not matter, as long as the buffers are contiguous in the virtual address space.

To investigate further, we change the setup to use different virtual address ranges for the two buffers (0x2c94000 in the Trojan, 0x2c82000 in the spy), resulting in a *stronger* channel, as visualised in Figure 10. This could be an effect caused by contention in the branch predictor, or some prefetcher, as discussed in Section 5.3.2.

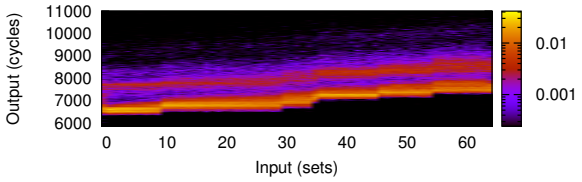

Figure 10: Channel matrix for the L1-I channel using different virtual addresses on Skylake with all mitigations. $n = 7438, \mathscr{C} = 1.2\,\text{b}, \mathscr{C}_\text{o} = 0.2\,\text{b}$.

In fact, Table 2 shows clear BTB channels on all platforms. They are particularly strong on the x86 processors, despite this being a black-box attack, given the complete lack of documentation of the x86 BTBs. An attack based on some understanding of the BTB implementations would likely show even bigger channels. Also, there is no architectural support for flushing these channels. Figure 11 shows an example (on Haswell) of this very definite channel with all mitigations deployed.

5.5 Finding 3: Branch-history-buffer channel persists

We now turn our attention to the BHB channel. Recall from Section 4.1 that the this channel only has two input values, 0 and 1, corresponding to a branch taken and not taken. Clearly, the maximun possible capacity of this channel is 1 b.

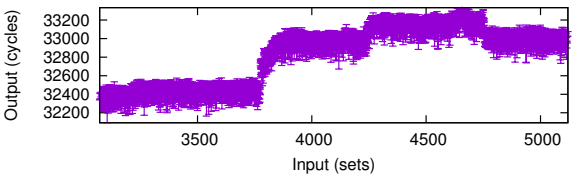

Figure 11: Average probing times, with 95% confidence intervals for the BTB channel on 64-bit Haswell with all mitigations. $n = 893, \mathscr{C} = 1.6\,\text{b}, \mathscr{C}_\text{o} = 0.5\,\text{b}$.

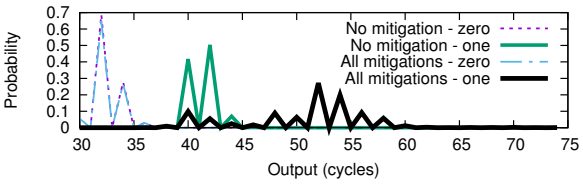

Figure 12: Distribution of output values for input values in the branch prediction channel on Skylake, with and without mitigations. The zero curves are indistinguishable.

5.5.1 Skylake BHB channel

Figure 12 shows the distribution of output values for these inputs without and with mitigations on Skylake.

We first observe that for both cases, the distribution of the output symbols for inputs 0 and 1 are clearly distinct. For the non-mitigated case, the median output value for input 0 is 32 cycles, whereas the median output for input 1 is 42. Applying all mitigation (none of which target the BHB specifically, as the architecture provides no mechanism) changes access times. However, the output values for inputs 0 and 1 are now even more separated than in the case of no mitigation, with median output values being 32 and 52, respectively. In short, our mitigation attempts are completely ineffective on the BHB channel.

5.5.2 A53 BHB channel

Figure 13 shows for the A53 that without mitigation, the median outputs are 18 cycles and 24 cycles for inputs 0 and 1, respectively. With mitigation, including invalidating branch prediction entries (`BPIALL`), the median outputs are 202 cycles for both inputs, but the well-defined minima of the distributions are 150 vs. 156 cycles, producing a clear residual channel.

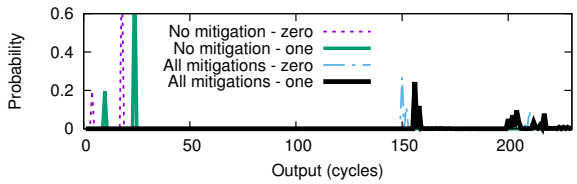

Figure 13: Distribution of output values for input values in the BHB channel on the A53, with and without mitigations.

5.6 Finding 4: TLB flushing is insufficient for removing the x86 TLB channel

We finally turn our attention to the TLB, which, according to Table 2, has a very distinct channel (2.5 bits). Because the TLB entries are tagged with address space ID, the timing reveals the L1 TLB contention. For testing the effectiveness of TLB flush by using `invpcid`, we explicitly flush all the TLB entries including paging caches. As Figure 14 shows, a strong channel remains, in fact, the capacity is almost unaffected by the flush.

The reason is presumably that the x86 architecture caches TLB entries in the data cache. When applying all mitigations listed in Section 4.2, the channel is mostly closed as shown in Figure 15. While $\mathscr{C}$ is still about twice $\mathscr{C}_0$, both are quite small, and from Figure 15 it seems that the remnant channel would be hard to exploit.

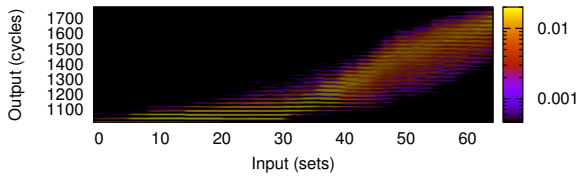

Figure 14: Channel matrix for the TLB covert channel on Skylake with TLB flushing. $n = 3818, \mathscr{C} = 2.2\,\text{b}$.

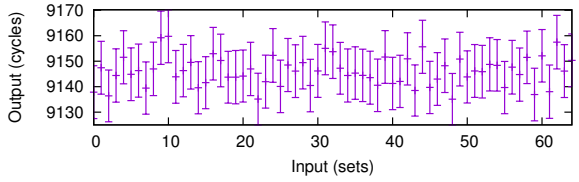

Figure 15: Average probing times, with 95% confidence intervals for the TLB covert channel on Skylake with all mitigations. $n = 19335, \mathscr{C} = 0.11\,\text{b}, \mathscr{C}_0 = 0.06\,\text{b}$.

6 Discussion

Our results clearly show that many microarchitectural features aimed at improving average-case performance produce high-capacity timing channels that cannot be closed with any available countermeasures, including those often suggested for mitigating the exact channels we explore [17, 37, 54], and despite some of them being prohibitively expensive [17, 21]. As we indicated earlier, we do not build the channels for maximum capacity, and further engineering is likely result in even higher capacities. Moreover, for high-security systems, even channels with capacities below one bit per second may pose a threat. For example, the Orange Book [11] recommends that channels with a bandwidth above 0.1 bits per second are audited.

The root of the problem is that many microarchitectural features either explicitly cache recent execution history (e.g. branch target buffers), or accumulate state based on such history (prefetcher state machines). Whenever such state is preserved across a context switch, a timing channel will result.

The OS has no mechanisms to mitigate such channels without hardware support. Specifically, to prevent timing channels, the OS must be given the opportunity to either cleanly partition such state between security domains, or flush on every switch of security domain. Partitioning is easily achieved with large physically-addressed caches through page colouring [25, 29, 46], but is probably not feasible for the virtually-addressed on-core resources. Hence, *there must be architected mechanisms for flushing all history-dependent on-core state*.

Obviously, such mechanisms should not affect performance where they are not needed, and the performance impact should be minimised where they are deployed. For example, the x86 architecture provides no architected mechanism for selectively flushing the L1 caches, the only cache-flush operation, `wbinvd`, flushes the complete cache hierarchy. From the security point of view, this is complete overkill, as the OS can easily partition caches other than the L1. In contrast, flushing just the L1 caches on a partition switch, i.e. at a rate of no more than 1000 Hz, will have no appreciable performance impact, as the direct and indirect costs of the flush should be in the microsecond range, and no L1 content is likely to be hot after a context switch.

Similarly, direct and indirect cost of flushing branch predictors and prefetchers should be negligible, provided efficient hardware support is available.

One way to look at our results is to argue that the ISA, the traditional hardware-software contract, is insufficient for building truly secure systems. *The ISA is sufficient to build software that is functionally correct but it is insufficient for building software that is secure, i.e. able to pre-*

serve confidentiality in the presence of untrusted code. Arguably, for security we need a new hardware-software contract that contains enough information about microarchitecture to ensure secure partitioning or time-sharing.

7 Conclusions

We investigated intra-core covert timing channels in multiple generations of x86 and ARM processors, and found that all investigated platforms exhibited high-capacity channels that cannot be closed with any known mechanism, irrespective of cost.

We therefore have to conclude that *modern mainstream processors are not suitable for security-critical uses where confidentiality must be preserved on a processor core that is time-multiplexed between different security domains*.

This work only explores the tip of the iceberg. We have limited ourselves to intra-core channels in a time-sharing scenario. In doing that we ignored all transient-state covert channels attacks and all attacks that rely on state outside the processor. Furthermore, the hardware contention caused by shared buses remain a serious security risk for threads sharing a platform.

The inevitable conclusion is that *security is a losing game until the hardware manufacturers get serious about it* and provide the right mechanisms for securely managing shared processor state. This will require additions to the ISA that allow any shared state to be either partitioned or flushed.

Acknowledgements

We would like to thank Dr Stephen Checkoway who helped uncovering documentation on processor functionality.

References

[1] Onur Acıiçmez. Yet another microarchitectural attack: exploiting I-cache. In *ACM Computer Security Architecture Workshop (CSAW)*, Fairfax, VA, US, 2007.

[2] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *Workshop on Cryptographic Hardware and Embedded Systems*, Santa Barbara, CA, US, 2010.

[3] Onur Acıiçmez and Werner Schindler. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In *Crytographers' track at the RSA Conference on Topics in Cryptology*, pages 256–273, San Francisco, CA, US, 2008.

[4] Onur Acıiçmez and Jean-Pierre Seifert. Cheap hardware parallelism implies cheap security. In *Fourth International Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 80–91, Vienna, AT, 2007.

[5] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 520–538, Chicago, IL, US, 2010.

[6] Daniel J. Bernstein. Cache-timing attacks on AES, 2005. Preprint available at `http://cr.yp.to/papers.html#cachetiming`.

[7] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *Proceedings of the 2nd Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*, pages 159–176, Santiago, CL, October 2012.

[8] Sarani Bhattacharya and Debdeep Mukhopadhyay. Who watches the watchmen? Utilizing performance monitors for compromising keys of RSA on Intel platforms. In *Workshop on Cryptographic Hardware and Embedded Systems*, pages 248–266, Saint Malo, FR, September 2015.

[9] Benjamin A. Braun, Suman Jana, and Dan Boneh. Robust and efficient elimination of cache and timing side channels. *arXiv preprint arXiv:1506.00189*, 2015.

[10] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. The last mile: An empirical study of some timing channels on seL4. In *ACM Conference on Computer and Communications Security*, pages 570–581, Scottsdale, AZ, USA, November 2014.

[11] DoD. *Trusted Computer System Evaluation Criteria*. Department of Defence, 1986. DoD 5200.28-STD.

[12] Dmitry Evtyushkin and Dmitry Ponomarev. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, pages 843–857, Vienna, AT, October 2016.

[13] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Understanding and mitigating covert channels through branch predictors. *ACM Transactions on Architecture and Code Optimization*, 13(1):10, April 16.

[14] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. Taipei, Taiwan, October 2016.

[15] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptographic Engin.*, 2017. (to appear).

[16] Matt Godbolt. The BTB in contemporary Intel chips. `http://xania.org/201602/bpu-part-three`, February 2016.

[17] Michael Godfrey and Mohammad Zulkernine. A server-side solution to cache-based side-channel attacks in the cloud. In *Proceedings of the 6th IEEEInternational Conference on Cloud Computing*, Santa Clara, CA, US, 2013.

[18] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, Vienna, Austria, October 2016.

[19] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in JavaScript. In *Proceedings of the 13th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, pages 300–321, San Sebastián, ES, July 2016.

[20] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A fast and stealthy cache attack. In *Proceedings of the 13th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, San Sebastián, Spain, July 2016.

[21] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. Cache storage channels: Alias-driven attacks and verified countermeasures. pages 38–55, San Jose, CA, US, May 2016.

[22] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *IEEE Symposium on Security and Privacy*, pages 191–205, San Francisco, CA, May 2013.

[23] IAD. *U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness*. Information Assurance Directorate, June 2007. Version 1.03. http://www.niap-ccevs.org/cc-scheme/pp/pp.cfm/id/pp_skpp_hr_v1.03/.

[24] Intel 64 & IA-32 ASDM. *Intel 64 and IA-32 Architecture Software Developer's Manual Volume 2: Instruction Set Reference, A-Z*. Intel Corporation, September 2016. http://www.intel.com.au/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf.

[25] R. E. Kessler and Mark D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, 10:338–359, 1992.

[26] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, February 2014.

[27] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16:613–615, 1973.

[28] Carl E. Landwehr. Formal models for computer security. *ACM Computing Surveys*, 13(3):247–278, September 1981.

[29] Jochen Liedtke, Hermann Härtig, and Michael Hohmuth. OS-controlled cache predictability for real-time systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Montreal, CA, June 1997.

[30] Steven. B. Lipner. A comment on the confinement problem. In *ACM Symposium on Operating Systems Principles*, pages 192–196. ACM, 1975.

[31] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache attacks on mobile devices. In *Proceedings of the 25th USENIX Security Symposium*, pages 549–564, Austin, TX, US, August 2016.

[32] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*, pages 605–622, San Jose, CA, US, May 2015.

[33] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering Intel last-level cache complex addressing using performance counters. In *Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Kyoto, Japan, November 2015.

[34] Clémentine Maurice, Manuel Weber, Michael Schwartz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Kay Römer, and Stefan Mangard. Hello from the other side: SSH over robust cache covert channels in the cloud. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, US, February 2017.

[35] Milena Milenkovic, Aleksandar Milenkovic, and Jeffrey Kulick. Microbenchmarks for determining branch predictor organization. *Software: Practice and Experience*, 34(5):465–487, April 2004.

[36] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: from general purpose to a proof of information flow enforcement. In *IEEE Symposium on Security and Privacy*, pages 415–429, San Francisco, CA, May 2013.

[37] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Proceedings of the 2006 Crytographers' track at the RSA Conference on Topics in Cryptology*, 2006.

[38] Colin Percival. Cache missing for fun and profit. In *BSDCan 2005*, Ottawa, CA, 2005.

[39] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. Reverse engineering Intel DRAM addressing and exploitation. *arXiv preprint arXiv:1511.08756*, 2015.

[40] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for cross-CPU attacks. In *Proceedings of the 25th USENIX Security Symposium*, Austin, TX, US, August 2016.

[41] John Rushby. Design and verification of secure systems. In *ACM Symposium on Operating Systems Principles*, pages 12–21, Pacific Grove, CA, USA, December 1981.

[42] John Rushby. A trusted computing base for embedded systems. In *Proceedings of 7th DoD/NBS Computer Security Conference*, pages 294–311, September 1984.

[43] Marvin Schaefer, Barry Gold, Richard Linde, and John Scheid. Program confinement in KVM/370. In *Proceedings of the Annual ACM Conference*, pages 404–410, Atlanta, GA, US, 1977.

[44] seL4. seL4 microkernel web site. sel4.systems.

[45] Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 1948. Reprinted in SIGMOBILE Mobile Computing and Communications Review, 5(1):3–55, 2001.

[46] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 194–199, HK, June 2011.

[47] Vish Viswanathan. Disclosure of H/W prefetcher control on some Intel processors. https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors, September 2014.

[48] Henry Wong. Intel Ivy Bridge cache replacement policy. http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/, January 2013.

[49] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *Proceedings of the 21st USENIX Security Symposium*, Bellevue, WA, US, 2012.

[50] Yuval Yarom. Mastik: A micro-architectural side-channel toolkit. http://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf, 2016.

[51] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium*, pages 719–732, San Diego, CA, US, 2014.

[52] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. Mapping the Intel last-level cache. http://eprint.iacr.org/, September 2015.

[53] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A timing attack on OpenSSL constant time RSA. In *Conference on Cryptographic Hardware and Embedded Systems 2016 (CHES 2016)*, pages 346–367, Santa Barbara, CA, US, August 2016.

[54] Yinqian Zhang and Michael K. Reiter. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, pages 827–838, Berlin, DE, November 2013.