No Security Without Time Protection: We Need a New Hardware-Software Contract

Qian Ge
Data61, CSIRO
UNSW Sydney
qian.ge@data61.csiro.au

Yuval Yarom
Data61, CSIRO
University of Adelaide
yval@cs.adelaide.edu.au

Gernot Heiser
Data61, CSIRO
UNSW Sydney
gernot@unsw.edu.au

ABSTRACT

The recent Spectre exploits demonstrated that covert timing channels are a mainstream security threat. Their prevention requires that operating systems provide *time protection*, in addition to the established *memory protection*. We propose OS mechanisms and designs which provide time protection, and define requirements on the hardware to enable them. We demonstrate that present mainstream processors do not meet these requirements, making them *inherently insecure*. We argue the need for a *new security-oriented hardware-software contract*, which we call the aISA as it augments the ISA, in order to enable time protection.

ACM Reference Format:
Qian Ge, Yuval Yarom, and Gernot Heiser. 2018. No Security Without Time Protection:, We Need a New Hardware-Software Contract. In *9th Asia-Pacific Workshop on Systems (APSys '18), August 27–28, 2018, Jeju Island, Republic of Korea.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3265723.3265724

1 INTRODUCTION

Modern processors contain microarchitectural components, such as caches and branch predictors, that exploit temporal or spatial locality for improving average-case performance. Inherently, these components maintain state that depends on recent execution history and affects the performance of subsequent execution. The execution of a previous process can thus affect the timing of the execution of the current.

Timing variations can be exploited as *covert channels* [Lampson 1973] that bypass the security policy of the system. In a typical scenario a *Trojan* program induces timing variations that encode information, and a *spy* program observes these variations to decode the communicated information. Various microarchitectural components have been used to implement covert channels, including stateless hardware of limited bandwidth, such as busses [Ge et al. 2018].

APSys '18, August 27–28, 2018, Jeju Island, Republic of Korea

Compared to *side channels*, where the sender transmits the information inadvertently, covert channels depend on insider help and are traditionally considered a less significant security threat. However, in the recent disclosure of the Spectre attack [Kocher et al. 2019], an attacker uses a covert communication channel from a speculatively executed gadget acting as a Trojan. *Spectre shows that covert channels pose a real security risk to mainstream computing.* Furthermore, any covert-channel mechanism bears the risk of being exploitable as a side channel by an ingenious attacker. Hence, proactive security requires prevention of covert channels.

Preventing unauthorised information flow is a primary duty of the operating system (OS). Traditionally, OSes provide *memory protection*, which prevents unauthorised spatial interference between programs. However, present-day OSes notoriously lack *time protection*, i.e. preventing unauthorised temporal interference. In other words, they have no means for preventing timing channels.

The OS could prevent timing channels exploiting a microarchitectural feature if the OS could ensure that it is

- never shared between security domains, i.e. the feature is strictly *partitioned*, or
- where a feature is time-multiplexed between domains, it is *reset* to a defined state on a domain switch.

In general, on-core state cannot be partitioned on present hardware and must be reset, which raises the question of whether architectures provide appropriate mechanisms to do so. To find out, we examine multiple generations of x86 and Arm processors, with the disappointing result that *each processor studied contains microarchitectural state that can be exploited as a timing channel, but cannot be reset by architected mechanisms* (Section 4). In other words, we find that the OS is powerless to prevent timing channels.

This leads to the paradoxical situation that manufacturers can claim that their hardware operates as specified [Intel 2018a], yet an OS relying on this specification, the instruction-set architecture (ISA), cannot prevent leakage, meaning that *the hardware is inherently insecure.* The inevitable conclusion is that *for security, the ISA is an insufficient specification of the hardware-software contract.*

We propose to remedy this situation by an improved contract, which we call the *augmented ISA*, aISA for short. We introduce the aISA and outline the kind of information it must specify for enabling security (Section 5).

2 BACKGROUND

The existence of a microarchitectural timing channel is often characterised as a flaw in the hardware. However, these channels are a necessary byproduct of measures taken by architects to maximize average-case performance, typically by exploiting the well-established principle of temporal and spatial locality. The inevitable implication is that execution speed depends on such microarchitectural state, and thus on the execution history. This history dependence is a fundamental reason for the existence of timing channels, and cannot be completely avoided without dramatically reducing performance. Hence the solution cannot be to completely remove any potential timing channel from the hardware, but to *prevent their use for unauthorised information flow across security domains*. In other words, the OS must provide *time protection* between processes. It must do so without significantly impacting overall system performance.

Time protection, like the more established memory protection, implies mandatory enforcement by the OS, rather than depending on application cooperation. As the use of gadgets in the Spectre attacks has demonstrated, even "trusted" code may be tricked into leaking. Like its memory counterpart, *time protection is a black-box form of isolation.*

This implies that approaches which provide some amount of "secure" memory, such as Sentry [Colp et al. 2015] or CATalyst [Liu et al. 2016], are not sufficient.

Partitioning is required where hardware resources are shared between cores, such as the last-level cache (LLC). Such sharing is truly concurrent, and on some resources (interconnects) contention is for bandwidth rather than state, so resetting between accesses is not an option.

Mandatory partitioning of off-core caches (L2–LLC) is possible by *page colouring* [Kessler and Hill 1992; Liedtke et al. 1997]. These caches are physically addressed, and large enough that the associative lookup forces a particular address into a certain subset of the cache. The OS controls physical memory allocation, and can thus prevent domains from competing for the same cache lines by allocating them physical frames of different cache colours. Partitoning stateless features (busses) is not supported on present hardware.

Resetting is needed for stateful on-core resources, such as the L1 caches, the TLB, the branch predictor and L1 prefetchers, which are virtually addressed. As virtual addresses are under application, rather than OS control, the OS cannot partition such resources without explicit hardware support.

Table 1: Evaluation hardware. *Mode* refers to our platform configuration, all support 32- and 64-bit mode.

Architecture	x86	x86	x86	Arm
Microarch.	**Sandy Bridge**	**Haswell**	**Skylake**	**Hikey**
Processor	i7-2600	E3-1220 v3	i7-6700	Kirin 620
Vintage	2011	2013	2015	2015
Clock rate	3.4 GHz	3.1 GHz	3.4 GHz	1.2 GHz
Mode	32-bit	64-bit	64-bit	32-bit
L1-D cache	32 KiB	32 KiB	32 KiB	32 KiB
sets×assoc.	64×8	64×8	64×8	128×4
L1-I cache	32 KiB	32 KiB	32 KiB	32 KiB
sets×assoc.	64×8	64×8	64×8	256×2
I+D-TLB	128+64	128+64	128+64	10+10
Unified TLB	512	1024	1536	512
BTB	???	???	???	256

While such support has been proposed in the past [Domnister et al. 2012; Wang and Lee 2007], it is not available on commodity processors. Hence, such resources must be reset on each domain switch.

Note that hyperthreading leads to concurrent access of on-core resources. Hence, in the absence of specific hardware support, no time protection is possible between hyperthreads [Ge et al. 2018], and therefore hyperthreads should always be allocated to the same security domain.

3 RESETTING STATE

For the remainder of this paper we focus on channels enabled by on-core state, which, as explained earlier, cannot be partitioned without further hardware support. We examine whether such channels can be eliminated by resetting microarchitectural state.

We study multiple processors from the presently dominant ISAs, x86 and Arm; the main characteristics of the processors are shown in Table 1. In this section we investigate what support these processors provide for resetting on-core microarchitectural state.

3.1 x86

The x86 architecture has very limited support for resetting microarchitectural state, and no way of resetting only on-core state. It provides the `wbinvd` instruction, which flushes the complete cache hierarchy [Intel Corporation 2016]. In terms of time protection, this is overkill, since the physically-indexed off-core caches should be partitioned rather than reset, as explained in the previous section. The cost of flushing the whole cache hierarchy is too high to make a full cache flush on each security-partition switch practical; we measure a worst-case cost of this flush of 12 ms on an Intel Sandy Bridge system (including the indirect cost of subsequent misses). Nevertheless, it is the only hammer the ISA gives us, so we use it for now.

There are multiple mechanisms for TLB flushing: We use `invpcid` in 64-bit mode and reload `CR3` and `CR0` (for invalidating both non-global and global mappings) in 32-bit mode.

x86 has no instructions for flushing other on-core state, although this could happen as an (undocumented) side effect of flushing caches or the TLB. The architecture does support disabling the data prefetcher by updating `MSR 0x1A4` [Viswanathan 2014], which we use to be on the safe side, understanding well that this will degrade performance. There is no way to control the instruction prefetcher.

Following disclosure of the Spectre attack, Intel provided a microcode patch that introduced three mechanisms collectively called *indirect branch control* (IBC) [Intel 2018d]. IBC does not claim to clear branch predictor state, but provides a degree of isolation between processes. The initial patch was soon withdrawn due to causing issues with booting machines [Intel 2018c], then updated three months later [Intel 2018b]. As it is a possible mechanism for providing time protection we evaluate it separately.

3.2 Arm

Arm supports a selective flush of the L1 caches, without affecting lower levels in the cache hierarchy; we use the `DCCISW` and `ICIALLU` instructions. We use the `TLBIALL` instruction to flush the TLB and `BPIALL` for the branch predictor.

There are platform-dependent mechanisms for disabling the data prefetcher, the CPU auxiliary control register in case of our Hikey platform. As on x86, there is no way to disable the instruction prefetcher.

4 MEASURING RESET EFFICACY

4.1 Implementing timing channels

We implement channels targeting several microarchitectural features. Here we only report on the interesting cases, i.e. the channels that cannot be closed making full use of any reset mechanisms provided by the architecture. Specifically, we look at channels exploiting the L1 I-cache, and two components of the branch predictor: the branch target buffer (BTB) and the branch history buffer (BHB). Complete results are presented on our web site [Data61, CSIRO 2018].

Our attacks are based on the Prime+Probe technique [Osvik et al. 2006; Percival 2005] to implement communication between Trojan and spy. Whereas past implementations focus on mechanisms and channel capacity [Evtyushkin and Ponomarev 2016; Gruss et al. 2016; Liu et al. 2015; Maurice et al. 2017], our focus is on establishing the existence of channels and whether they can be closed.

In Prime+Probe, the spy *primes* the cache by filling cache sets with its own data, then waits for the Trojan to replace some of the cache lines based on the *input symbol* it transmits. Lastly, the spy *probes* the cache sets by measuring the access time to the previously cached data, thus measuring the Trojan's cache footprint. The output symbol is the total probing time of the spy.

L1 D-cache. The spy fills the whole cache with its own data, waits for a context switch (to the Trojan), and then measures the time it takes to access the cache. To send symbol s, the Trojan reads enough data to fill all of the ways of cache sets $0, 1, \ldots, s-1$.

L1 I-cache. To send symbol, s, the Trojan executes a series of jumps in memory locations that map to specific cache sets [Acıiçmez 2007; Acıiçmez et al. 2010]. We use the implementation provided by the Mastik toolkit [Yarom 2016].

BTB. Our implementation of a BTB-based channel uses n chained branch instructions as a probing buffer. The Trojan probes on the first s, while the spy measures time taken for probing the entire buffer. On the Arm, n is the size of the BTB (Table 1). On x86 platforms, the BTB is not documented, but can frequently be reverse-engineered [Milenkovic et al. 2004] and found to have 4096 entries on Ivy Bridge [Godbolt 2016]. We let the Trojan execute from 3072 to 5120 `jmp`s, whereas the spy executes 4096. We align the `jmp`s to 16 bytes to force the buffers to exceed the L1-I cache capacity.

BHB. We implement the residual state-based covert channel [Evtyushkin et al. 2016]. The Trojan and spy use the same channel code for sending and receiving, transmitting a single bit per iteration. The attack includes a sequence of conditional branches that are always taken, to set the history to a known state. This is followed by a branch that conditionally skips over 256 `nop` instructions. The Trojan encodes a single bit by training the prediction for the last branch. The spy measures the cost of executing the `nop` instructions, a mis-predict increases latency.

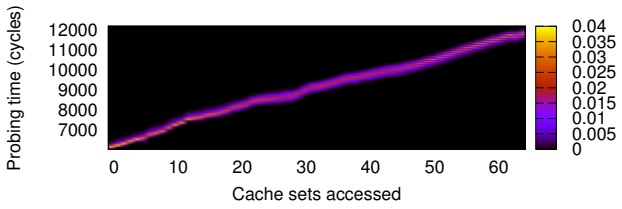

Figure 1: Channel matrix for the unmitigated L1 I-cache channel on the Sandy Bridge platform.

4.2 Visualising channels

We use the *channel matrix* for visualising channels [Cock et al. 2014]. It gives the conditional probability of observing a certain output symbol (spy probing time, y axis) given an input symbol (cache lines accessed by the Trojan, x axis)

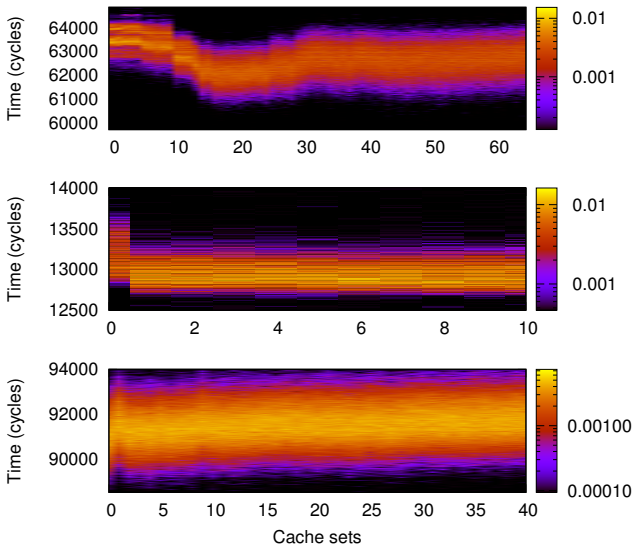

Figure 2: Channel matrix for the mitigated L1 I-cache channels on (top to bottom) Sandy Bridge, Haswell and Hikey.

and is represented as a heat map, where a brighter colour represents a higher probability. With no channel, outputs are independent of inputs and the graph will show no horizontal variation, any horizontal variation can be exploited as a channel. For example, in Figure 1, the spy measures a probing time (output) that is highly correlated with the cache footprint of the Trojan, a clear dependency.

4.3 Results

We measure the channel matrix without mitigations, as well as with all reset operations applied (and the data prefetcher disabled). While this effectively removes the D-cache channel on all platforms, this is not the case for the I-cache, as shown in Figure 2. The channel is most pronounced on the (somewhat dated) Sandy Bridge processor, where there is a definite horizontal variation, although nowhere near as pronounced as in the unmitigated case of Figure 1.

On the Haswell, the channel is much less pronounced, only input value zero is special, resulting in a small channel. Similarly, the Hikey shows a small but visible channel, evident at inputs 0–2 and the slightly rising slope of the peak of the channel heatmap.

We have a similar story for the branch-target buffer. A clear channel seems to be closed by applying the flush operation on the Sandy Bridge, while on the Haswell there is a very clear residual channel, indicated by the curiously shaped pattern in Figure 3. On the Skylake there is a drop in the heatmap at input values around 3250 in a graph that has otherwise little horizontal variation, but this is enough to establish a channel.

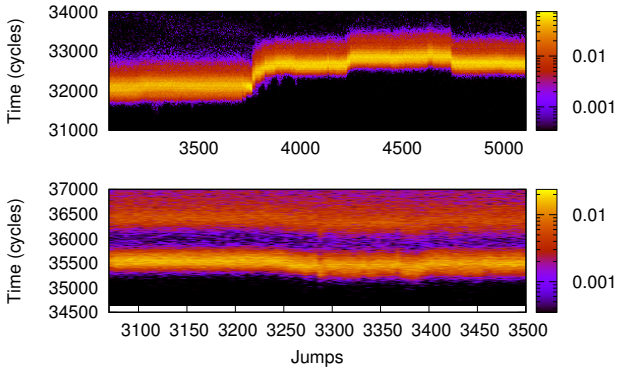

Figure 3: Channel matrix for the mitigated BTB channels on Haswell (top) and Skylake.

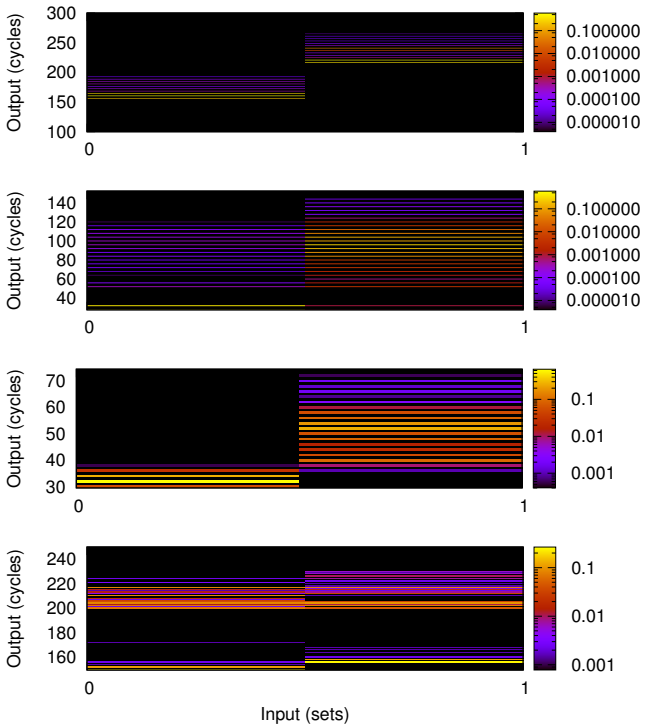

Figure 4: Channel matrix for the mitigated BHB channels on (top to bottom) Sandy Bridge, Haswell, Skylake and Hikey.

The branch-history buffer is a single-bit channel (branch taken or not taken). As per Figure 4, flushing is ineffective on all four platforms, the probe time clearly distinguishes between the two input values.

We have now seen that on all our platforms there is at least one channel that cannot be removed by architected reset operations, on several platforms there are multiple channels.

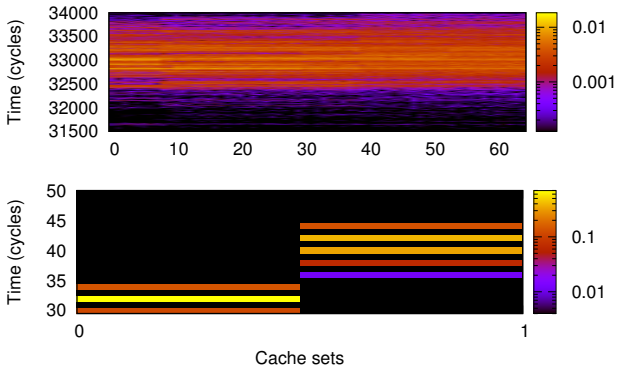

Figure 5: Channel matrix for adding Intel's *original* Spectre mitigation to the Haswell L1-I channel (top) and Skylake BHB channel (bottom).

To explore the scope for improved architected reset mechanisms, we apply Intel's originally (later withdrawn) IBC operation (see Section 3.1) on top of the other resets. We find that this is partially effective. Besides only supporting the newer Haswell and Skylake platform, it seems to remove most of the residual channels we observed. However, it fails to close the L1-I cache channel on the Haswell[1] (top graph in Figure 5), which shows a slight discontinuity at eight cache sets. Furthermore, it leaves a clear BHB channel on the Skylake (bottom graph in Figure 5).

In contrast, with the most recent microcode update [Intel 2018b], IBC is effective in removing all our residual channels (even without disabling the prefetcher, but still requiring the prohibitively expensive full cache flush).

These results show that it is clearly possible to improve the present situation, manufacturers are able to provide more effective reset operations, but it also shows that they really need to reset all microarchitectural state.

5 THE AUGMENTED ISA: A NEW HARDWARE-SOFTWARE CONTRACT

The purpose of the hardware-software contract is to allow independent development of hardware and software, with the expectation that everything works correctly as long as everyone observes the contract. Clearly, the ISA is not the right contract, as it abstracts away information that is essential for ensuring time protection as a prerequisite for security. In other words, *we need a new hardware-software contract.*

The new contract must satisfy several requirements:

(1) It must provide the OS with sufficient mechanisms for supporting time protection
(2) it must be as simple as possible
(3) it must not reveal more architectural details than absolutely necessary
(4) it must minimise restrictions imposed on architects.

The points 2–4 can be viewed as different aspects of the same principle, and are important in practice. Simplicity helps software as well as hardware developers. Manufacturers will resist revealing critical IP, and require the freedom to innovate at the microarchitecture level. Together these points argue for a minimal augmentation, rather than wholesale replacement of the ISA. We therefore call the new contract the *augmented ISA* (aISA).

In order to minimise restrictions on architects, we accept that the aISA will be less stable than the ISA, and may change between processor versions. This is acceptable for software developers, as the difference between the ISA and the aISA only affects small amounts of code, the part of the OS responsible for time protection. To limit the cost of adaptation we desire that these changes are restricted to a small number of parameters of a highly abstracted model of the microarchitecture, and that the hardware provides a mechanism for software to query the values of those parameters.

We assert the core property of the aISA:

> **Property 1: Security-enforcement**
> Any shared microarchitectural feature can either be partitioned between security domains, or reset when required by the OS.

Furthermore, resetting cannot help where the respective hardware unit is shared by concurrent executions (on different cores or hardware threads within a core), e.g. a shared cache. This implies a second property:

> **Property 2: Secure concurrent sharing**
> Microarchitectural features accessed by concurrent execution streams must be partitionable; partitions must be completely static or OS-controlled.

OS control of changes is essential: In terms of information flow, dynamic partitioning by hardware is no different from a normal data or instruction cache, the dynamics can be exploited as a timing channel.

As the OS cannot partition state accessed solely by virtual address (Section 2), we require:

> **Property 3: Secure virtually indexed state**
> Hardware state indexed solely by virtual address must not be concurrently accessible and must be resettable.

Note that if state is accessed by a combination of virtual address and (temporary unique) thread ID, rather than virtual address only, this condition does not apply.

[1]The microcode is for a desktop version of Haswell with the same unmitigated channel, so the result should be representative.

For partitionable hardware, the aISA must provide sufficient information to allow the OS to do the partitioning. For example, physically addressed caches of sufficient size can be partitioned by page colouring, as long as the OS can determine the number of available colours (which is usually determined by the cache stride, but may be larger on some processors [Yarom et al. 2015]).

Similarly, the reset mechanisms must be clearly defined. As the latency of the reset may itself depend on execution history (e.g. the number of dirty D-cache lines), it could be used as a channel too. Hence, the reset must either be a constant-time operation, or the OS must be able to pad it to its worst-case latency.

Property 4: Specified mechanisms
The aISA must completely specify the mechanisms the OS must use to partition or reset microarchitectural features. Reset operations must be constant time or have a specified worst-case latency.

For resettable state, the OS needs to know the kind of information that is cached (data, instructions or addresses of data or instructions):

Property 5: State provenance
The aISA must specify whether a reset operation acts on state derived from data, instructions, data addresses or instruction addresses.

Over-approximations are legal, e.g. it is acceptable to lump all virtually-addressed microarchitectural state (caches, TLB, branch predictor and prefetcher) into a single abstraction with a single reset operation, or an abstraction can be specified to hold data and instructions when in fact it only is affected by instructions. However, a somewhat more detailed abstraction may have performance advantages.

6 DISCUSSION

It might seem that the aISA is nothing more than a flush instruction (and as such really an extension to the ISA). Yet this is not true, as the aISA provides more than a functional specification of instructions. This is most obvious for Property 4, but Property 3 as well as the combination of Property 1 and Property 2 put obligations on the architect that go beyond an ISA. Let us look at some implications.

6.1 Implications

Firstly, we note that Property 1 implies that the two options, partitioning and reset, must not interfere with each other. For example, the LLC may be trivially partitionable using memory colouring. However, an instruction that flushes the complete LLC would break partitioning, as an attacker (or Trojan) could affect contents not only of its own cache partition but also others. If such an instruction exists, it must be privileged, i.e. reserved for use by the OS, else the hardware fails to observe Property 1.

We further observe that Property 2 seems to imply that a multi-threaded core cannot be concurrently shared, but all hardware threads must be allocated to the same security partition. To enable secure sharing of a core (other than by time multiplexing), the microarchitecture would have to partition all state that is shared between hardware threads, including L1 caches, TLB, branch predictors and prefetchers; this would seem to turn the threads into full-blown cores.

Finally, the requirement for resets to be constant time or have a defined latency bound might seem restrictive at first glance. We do not think that this is the case: Resetting will really apply to on-core resources and core-private caches such as a private L2. Almost all on-core state is read-only data (derived from instructions or addresses). Resetting those is most likely a constant-time operation anyway, taking a small number of cycles (no more than the pipeline depth), so guaranteeing constant-time execution should be trivial.

Resetting data caches requires flushing modified data down the memory hierarchy, the latency of which will depend on the number of dirty lines. It is unreasonable to force this instruction to always execute with the same latency, particularly since an L1-D flush has uses other than time protection. For example, it might be used to ensure coherency between I- and D-cache in the case of just-in-time compilation or other instances of self-modifying code. Instead, it is sufficient to specify the maximum latency of the flush, so the OS can pad to the worst case if needed. Alternatively, the flush instruction could be parameterised, to provide the OS with the choice between minimal and constant latency.

Stateless but bandwidth-limited hardware (interconnects) are a real challenge. Partitioning is not supported on present hardware, and it would seem to require partitioning bandwidth (e.g. TDMA), which would likely have a significant performance impact. As it is hard to see how such hardware could be exploited as side channels, a solution might be to accept the possible covert channel but prevent its use by an attacker. Preventing speculative execution from being visible beyond a core, i.e. prohibiting speculative loads from a shared cache, might work, but more research is needed to say whether this would is sufficient for enforcing security.

6.2 Cost

Resetting microarchitectural state has a cost. We have argued above that the *direct* cost is low, except for data caches, which must flush dirty data down the memory hierarchy. In addition there is the indirect cost of starting with cold caches/predictors. However, it must be noted that *such flushes are only needed when switching security domains*, not when switching between threads of the same domain. In many

cases, security domains are heavyweight, e.g. virtual machines in a cloud scenario, which are typically switched at a rate of 10–100 Hz. Such a long execution time implies that caches are cold anyway after a switch, which means that the cost of flushing dirty lines and missing in the cache is paid anyway.

There are scenarios where domains are switched more frequently, such as when a browser executes untrusted JavaScript code. Our experiments show that the worst-case cost of an L1-D flush is about a microsecond, which seems a bearable cost for security. However, a more detailed evaluation is clearly needed.

The performance implications of partitioning caches are well-researched in the architecture community. For example, Sanchez and Kozyrakis [2011] find that the average performance cost of statically partitioning the LLC between cores is 7%. This is far less than the cost of the Spectre or Meltdown defences that are presently being deployed.

7 RELATED WORK

Microarchitectural timing channels have been an active field of research for considerable time, with work on attacks and defences accelerating over the past decade, see the survey by Ge et al. [2018]. In particular, the recent Meltdown [Lipp et al. 2018] and Spectre [Kocher et al. 2019] attacks brought awareness of these threats to a mainstream audience. Spectre is particularly relevant in our context as it demonstrates the importance of covert channels, while most recent work focusses on side channels.

Mitigations are mostly based on partitioning [Liedtke et al. 1997; Lynch et al. 1992; Shi et al. 2011] or flushing [Godfrey and Zulkernine 2013; Guanciale et al. 2016; Osvik et al. 2006; Zhang and Reiter 2013]. Partitioning of on-core state is only possible with non-standard hardware support [Domnister et al. 2012; Wang and Lee 2007]. Our work shows that flushing is not sufficiently supported on commodity processors.

As exploiting timing channels requires timing of events, a defence is to deny attackers access to a time source. Completely virtualising time [Aviram et al. 2010a,b; Li et al. 2013] in principle eliminates timing channels, but comes with high overheads and is infeasible in many real-world situations. In particular, it does not apply to Spectre attacks. The same holds for injecting noise, as with fuzzy time [Hu 1991], the AES implementation proposed by Brickell et al. [2006] which randomises the cache footprint of lookup tables, or the random fill cache of Liu and Lee [2014], which randomises the address loaded to the cache in case of a cache miss.

Tiwari et al. [2009] suggested a leasing approach to share hardware resources between threads, which guarantees bounds on resource usage and side effects. Tiwari et al. [2011] later proposed a solution for top-to-bottom information flow guarantees, including a Star-CPU, a microkernel, and an I/O protocol. This is a fairly radical departure from mainstream computing architectures and manufacturers of commodity hardware will be hard to convince to go down this route.

Language-based approaches, such as language semantics with deterministic execution latencies [Zhang et al. 2012], are not black-box approaches and therefore not suitable for OS-provided time protection.

There are proposals in the real-time space for improving the accuracy of worst-case execution-time analysis, including PRET [Edwards and Lee 2007], PREM [Pellizzoni et al. 2011] and our recent proposal for exposing more of the microarchitecture [Heiser 2018]. These proposals are primarily concerned with ensuring the safety of critical real-time systems, as opposed to information leakage, and are more intrusive on architects.

8 CONCLUSIONS

Recent security exploits, such as Spectre, have demonstrated that covert timing channels (and not just side channels) are a mainstream threat to information security. As security enforcement is a core responsibility of the OS, this means that the well-established OS enforcement of memory protection is insufficient, and the OS must also provide time protection.

However, our analysis demonstrates that mainstream computer architectures do not provide sufficient mechanisms to allow the OS to enforce this kind of isolation: As we demonstrate on multiple generation of processors across the two main ISAs, there exist timing channels that cannot be closed by architected means, meaning *the OS has no chance of providing time protection on present mainstream architectures.*

We trace this problem to the fact that the ISA, which constitutes the standard hardware-software contract, is too abstract: It is a purely functional specification that cannot capture the requirements for timing-channel freedom.

We infer that *security requires a refined hardware-software contract*, the aISA. We specify the core properties the aISA must satisfy to allow the OS to enforce real security. We believe that this is a minimally-intrusive proposal that does not limit architects' ability to innovate, and can be implemented and used without significant performance impact. We also note that it is compatible with our recent aISA proposal [Heiser 2018] aimed at improving real-time predictability. In fact, achieving real-time safety requires exposing much more of the microarchitecture than is needed for security, which can get away with an opaque representation of microarchitectural state.

Operating systems built on hardware that offers an aISA will be able to enforce time protection, which, among others, would defeat Spectre attacks. For the sake of security, we can only hope that architects and manufacturers will listen.

ACKNOWLEDGMENTS

Part of this work was funded by the Australian Department of Defence's Next Generation Technology Fund.

REFERENCES

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

Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. 2010. New Results on Instruction Cache Attacks. In *Workshop on Cryptographic Hardware and Embedded Systems*. Santa Barbara, CA, US.

Amittai Aviram, Sen Hu, Bryan Ford, and Ramakrishna Gummadi. 2010a. Determinating timing channels in compute clouds. In *ACM Workshop on Cloud Computing Security*. Chicago, IL, US, 103–108.

Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. 2010b. Efficient system-enforced deterministic parallelism. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*. Vancouver, BC, 1–16.

Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. 2006. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. *IACR Cryptology ePrint Archive* 2006 (2006), 52.

David Cock, Qian Ge, Toby Murray, and Gernot Heiser. 2014. The Last Mile: An Empirical Study of Some Timing Channels on seL4. In *ACM Conference on Computer and Communications Security*. Scottsdale, AZ, USA, 570–581.

Patrick J. Colp, Jiawen Zhang, James Gleeson, Sahil Suneja, Eyal de Lara, Himanshu Raj, Stefan Saroiu, and Alec Wolman. 2015. Protecting Data on Smartphones and Tablets from Memory Attacks. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. Istambul, TK.

Data61, CSIRO. 2018. Timing Channel Mitigations. https://ts.data61.csiro.au/projects/TS/timingchannels/arch-mitigation.pml.

Leonid Domnister, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2012. Non-Monopolizable Caches: Low-Complexity Mitigation of Cache Side Channel Attacks. *ACM Transactions on Architecture and Code Optimization* 8, 4 (Jan. 2012).

Stephen A. Edwards and Edward A. Lee. 2007. The Case for the Precision Timed (PRET) Machine. In *Design Automation Conference (DAC)*.

Dmitry Evtyushkin and Dmitry Ponomarev. 2016. Covert Channels through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*. Vienna, AT, 843–857.

Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Understanding and Mitigating Covert Channels Through Branch Predictors. *ACM Transactions on Architecture and Code Optimization* 13, 1 (April 2016), 10.

Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *Journal of Cryptographic Engineering* 8 (April 2018), 1–27.

Matt Godbolt. 2016. The BTB in contemporary Intel chips. http://xania.org/201602/bpu-part-three

Michael Godfrey and Mohammad Zulkernine. 2013. 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.

Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. 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.

Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. 2016. Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures. San Jose, CA, US, 38–55.

Gernot Heiser. 2018. For Safety's Sake: We Need a New Hardware-Software Contract! *IEEE Design and Test* 35 (March 2018), 27–30.

Wei-Ming Hu. 1991. Reducing timing channels with fuzzy time. In *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy*. IEEE Computer Society, Oakland, CA, US, 8–20.

Intel. 2018a. Intel Responds to Security Research Findings. https://newsroom.intel.com/news/intel-responds-to-security-research-findings/

Intel. 2018b. Microcode Revision Guidance. https://www.intel.com/content/dam/www/public/us/en/documents/sa00115-microcode-update-guidance.pdf

Intel. 2018c. Root Cause of Reboot Issue Identified; Updated Guidance for Customers and Partners. https://newsroom.intel.com/news/root-cause-of-reboot-issue-identified-updated-guidance-for-customers-and-partners/

Intel. 2018d. Speculative Execution Side Channel Mitigations. https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf

Intel Corporation 2016. *Intel 64 and IA-32 Architecture Software Developer's Manual Volume 2: Instruction Set Reference, A-Z*. Intel Corporation. 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.

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

Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Haburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwartz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy*. IEEE, San Francisco, 19–37.

Butler W. Lampson. 1973. A Note on the Confinement Problem. *Commun. ACM* 16 (1973), 613–615.

Peng Li, Debin Gao, and Michael K Reiter. 2013. Mitigating access-driven timing channels in clouds using StopWatch. In *Proceedings of the 43rd International Conference on Dependable Systems and Networks (DSN)*. Budapest, HU, 1–12.

Jochen Liedtke, Hermann Härtig, and Michael Hohmuth. 1997. OS-controlled cache predictability for real-time systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, Montreal, CA, 213–223.

Moritz Lipp, Michael Schwartz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*. USENIX, Baltimore, MD, USA, –.

Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. 2016. CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing. In *IEEE Symposium on High-Performance Computer Architecture*. Barcelona, Spain, 406–418.

Fangfei Liu and Ruby B Lee. 2014. Random fill cache architecture. In *Proceedings of the 47th ACM/IEE International Symposium on Microarchitecture*. Cambridge, UK.

Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE Symposium on Security and Privacy*. San Jose, CA, US, 605–622.

William L. Lynch, Brian K. Bray, and M. J. Flynn. 1992. The effect of page allocation on caches. In *ACM/IEE International Symposium on Microarchitecture*. 222–225.

Clémentine Maurice, Manuel Weber, Michael Schwartz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Kay Römer, and Stefan Mangard. 2017. 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.

Milena Milenkovic, Aleksandar Milenkovic, and Jeffrey Kulick. 2004. Microbenchmarks for Determining Branch Predictor Organization. *Software: Practice and Experience* 34, 5 (April 2004), 465–487.

Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Proceedings of the 2006 Crytographers' track at the RSA Conference on Topics in Cryptology*.

Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. 2011. A Predictble Execution Model for COTS-based Embedded Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 269–279.

Colin Percival. 2005. Cache Missing for Fun and Profit. In *BSDCon 2005*. Ottawa, CA.

Daniel Sanchez and Christos Kozyrakis. 2011. Vantage: Scalable and Efficient Fine-Grain Cache Partitioning. In *International Symposium on Computer Architecture*. 57–68.

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

Mohit Tiwari, Xun Li, Hassan M. G. Wassel, Frederic T. Chong, and Timothy Sherwood. 2009. Execution Leases: A Hardware-supported Mechanism for Enforcing Strong Non-interference. In *Proceedings of the 42nd ACM/IEE International Symposium on Microarchitecture*. New York, NY, US.

Mohit Tiwari, Jason K Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T Chong, and Timothy Sherwood. 2011. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. In *Proceedings of the 38th International Symposium on Computer Architecture*. San Jose, CA, US.

Vish Viswanathan. 2014. 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

Zhenghong Wang and Ruby B. Lee. 2007. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *Proceedings of the 34th International Symposium on Computer Architecture*. San Diego, CA, US.

Yuval Yarom. 2016. Mastik: A Micro-Architectural Side-Channel Toolkit. http://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf

Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. 2015. Mapping the Intel Last-Level Cache. http://eprint.iacr.org/.

Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. 2012. Language-based control and mitigation of timing channels. In *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Beijing, CN, 99–110.

Yinqian Zhang and Michael K. Reiter. 2013. 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*. Berlin, DE, 827–838.