For Safety's Sake: We Need a New Hardware-Software Contract!

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

***Abstract*—The ISA is the established hardware-software contract. As the ISA hides hardware features that affect timing of execution, it is no longer sufficient for ensuring system security and safety. We argue that a new contract is required, which exposes such features.**

***Index Terms*—C.0.b Hardware/software interfaces, C.3.d Real-time and embedded systems, D.4.6.d Information flow controls**

I. Introduction

Safety as well as security of critical systems requires *temporal isolation*, which means that one computing task cannot affect the execution speed of another, unrelated one. If the affected task is a critical real-time one, its execution might get delayed to the point of missing a deadline. Deadline misses are *integrity violations* that lead to loss of system *safety*. In safety-critical systems, temporal isolation is therefore as important as the more established spatial isolation, i.e. memory safety.

Lack of temporal isolation can also lead to *security* violations: The exact effect the execution of one task has on the execution speed of another can depend on confidential data the former task is processing, and thus can leak secrets to the latter task. Such information leakage is called a timing side channel and can, for example, lead to the theft of encryption keys [Q. Ge, et al., "A Survey of Microarchitectural Timing Attacks Countermeasures on Contemporary Hardware", J. Cryptographic Engin., https://link.springer.com/article/10.1007/s13389-016-0141-6]. Such *confidentiality violations* are a clear threat to system security.

In modern cyberphysical systems, security violations are also a threat to safety, as demonstrated by the recent spate of car hacking, with attackers controlling breaks and engine throttle [A. Greenberg, "Hackers Remotely Kill a Jeep on the Highway—With me in It", July 2015, https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/.]

In summary, temporal isolation, the prevention of such integrity and confidentiality violations, is of critical importance for the safety of future cyberphysical systems. But achieving it is only possible with sufficient understanding and control of the hardware. We will argue that contemporary hardware is deficient in this respect, making it effectively impossible to build safe systems, and that the trend is for the worse.

II. Ensuring temporal isolation

Hard real-time systems, where failure to complete an action by a deadline is disastrous, used to be small control programs running on simple microcontrollers without internal protection. This model has reached its use-by date, with even critical systems becoming complex and rich in functionality. This means that modern real-time systems are increasingly mixed-criticality systems (MCS), where functions of different criticality co-exist on the same processor [J. Barhorst et al., "A Research Agenda for Mixed-Criticality Systems", April 2009, http://www.cse.wustl.edu/~cdgill/CPSWEEK09_MCAR/]. A core property of an MCS is that the ability of a critical task to meet its deadlines must not depend on the correct behaviour of less critical components. In other words, less critical components cannot be trusted, and hence the OS must protect the critical components from interference (in time and space) by less critical ones.

This requires, among others, a worst-case execution time (WCET) analysis of any OS functionality that is required for enforcing isolation. The WCET analysis process is without doubt explained elsewhere in the special issue [*cross-reference to suitable article in this special issue*]. suffice to say here that it requires hard bounds on the maximum execution latency of all instructions of the processor. ARM used to publish these, up to the Cortex-A8 cores. With the introduction of out-of-order (OoO) cores ARM has discontinued this practice, even for the lower-end in-order A-series cores. This makes it impossible to provide safe execution-time bounds on any recent ARM cores. The situation is not at all better in the x86 world, Intel has not published instruction latencies for a long time (if they ever did).

Others will argue that not only the bounds are important, but also that they need to be reasonably tight, to prevent excessive latency. Tight bounds require predictable architectures, but this predictability constrains the architect and precludes many of the optimisations which enable contemporary high-performance architectures; mainstream architectures are highly unlikely to adopt such an approach. Furthermore, this should not be major issue for future MCS, as they allow reusing the scheduling slack resulting from WCET pessimism for less critical activities.

If the safety story was not bad enough, the security situation is worse. One defence against timing-channel attacks, espe-

cially for crypto algorithms, is *constant-time implementations*, where execution time is independent of inputs. However, these are only possible if the implementer understands exactly what the hardware does, and in general they do not have sufficient information about the hardware. The result is frequently that "constant-time" implementations are not constant-time at all, as we have recently demonstrated on the supposedly constant-time implementation of TLS in OpenSSL 1.0.1e [D. Cock et al., "The Last Mile: An Empirical Study of Some Timing Channels on seL4", Computer and Communications Security Conference, Nov. 2014, p 570–581.]

At the core of many timing channels are microarchitectural features, which architects use to improve average-case performance. These are mostly caches that utilise temporal or spatial locality of programs, and include the translation-lookaside buffer (TLB) and branch prediction units, or state machines used in prefetchers. They carry information about execution history and affect the timing of subsequent execution. If such state is left intact across a *context switch*, i.e. when the OS switches between tasks, they can reveal secrets.

The problem is that present hardware does not provide sufficient (documented) mechanisms that allow the OS to sanitise such state, in order to prevents its use as an information channel [Q. Ge et al., "Your Processor Leaks Information – and There's Nothing You Can Do About It", Sep 2017, https://arxiv.org/pdf/1612.04474.pdf. This means that it is not possible to close such timing channels.

III. The root of the problem: The ISA

The two aspects of temporal isolation, and the challenges they face, seem very different from each other. However, the cause of both is the same: lack of information on the temporal aspects of execution of programs on a processor. Ultimately, the cause is the increasing ineffectiveness of the instruction-set architecture (ISA) as the *hardware-software contract*.

The ISA describes the *functional* interface of the hardware to software. Specifically, it describes all you need to know for writing a *functionally correct* program, i.e. a program which is adequately described as a mathematical function that transforms some state and inputs into modified state and outputs. But safety and security, as argued above, requires more than functional correctness, it must account for time as well. However, the ISA hides all such information.

This *hiding of temporal information by the ISA is considered a feature* by processor architects and many software writers alike. For software it has the advantage that much of the complexities of the hardware are abstracted away into a simplified programming model, consisting mostly of instructions, a relatively small set of registers, and a memory-management unit (MMU). This eases programming and helps software portability.

For architects, the ISA provides freedom to change the hardware implementation without (functionally) affecting software, e.g. by adding new caches for improving (average case) performance, or for picking a particular point on a scale of performance vs. energy-efficiency tradeoffs.

However, once temporal behaviour of program execution becomes important, *the abstraction provided by the ISA is no longer a feature, it is a bug*. And, as with any bug, we need to work on fixing it. In our case, this means *we need a new hardware-software contract*, one which allows us to write software that is safe and secure.

IV. AISA: A new hardware-software contract

What would such a new contract look like?

Obviously we want to retain as many of the ISA's desirable properties as possible. In particular, we want to

1) keep the HW-SW interface as simple as possible,
2) minimise the additional hardware features the software writer must understand in order to write correct software, ideally keeping this additional complexity to zero where only functional correctness is needed, and
3) minimise constraints on the hardware architect's ability to optimise and innovate.

This set of requirements speaks in favor of a minimal addition to the ISA, let's call it the *augmented ISA*, short AISA (rhymes with Elisa). It *must* expose some of the microarchitectural features the architects put into the hardware for performance reasons.

Exposing microarchitectural features constrains the architect. We acknowledge the need to minimise such constraints by accepting that some of these AISA details will change between implementations of the underlying ISA, meaning that some software must be adapted to the microarchitecture. We consider this an acceptable burden for the software developer, primarily as most software will remain unaffected – in most case only WCET analysis tools, crypto libraries and the temporal-security enforcement modules of the OS are affected, an acceptable prize to pay for safety.

In fact, it would be permissible if the AISA-specified hardware behaviour is enabled by an OS-controlled switch, provided enabling it does not significantly reduce the processor's performance or energy efficiency. However, it is not clear that such a switch would really help the architects.

V. What does the AISA contain?

One feature the AISA must expose are instruction and data caches, including their associativity, line size, hit and miss latencies, and replacement policy. All these properties are needed for determining cache residency and access costs during WCET analysis.

This requirement is not revolutionary, of course. In fact, most of this is routinely documented in reference manuals for the particular processor implementation, meaning there is no additional burden on the architect, and the software folks already use this information and adapt to changing details. The main change would be to remove the utterly unhelpful "random" cache replacement policy. Of course, the real hardware is not random anyway, but uses some undocumented state to determine the victim line that needs replacing. So the real requirement here is that the policy be fully documented (and that the controlling state be analysable).

An additional requirement for WCET analysis is exposition of the pipeline structure, and especially worst-case instruction

latencies. These will, on average, be quite pessimistic on out-of-order processors, but they should not be unbounded. On in-order processors, they should be much tighter.

Again, this is nothing new, ARM used to publish this kind of information until they introduced OoO cores.

More is needed for security. The information on caches, discussed above, is sufficient to let the OS partition physically-addressed caches (this includes the L2 and all caches further down the hierarchy) using *page colouring* [J. Liedtke et al., "OS-controlled cache predictability for real-time systems", IEEE Real-Time Technology and Applications Symposium, Jun 1997, p 213–223].

There is plenty of other microarchitectural state that affects timing, and thus provides potential timing channels, that cannot be partitioned by the OS, or at least it is not obvious how the OS could do it. This includes all forms of on-core cache, which are virtually addressed, and frequently too small for colouring. This kind of state includes the L1 I- and D-caches, the TLB and the branch predictor. But it also includes other stateful accelerators, such as prefetchers and DRAM row buffers.

The AISA must identify each of these, suitably abstracted, and, importantly, must also identify how this state can be either partitioned or completely flushed by the OS. This part of our proposal is probably the one architects will be least comfortable with, but it is a real necessity for security. Any microarchitectural state that is not specified in this way by the AISA will result in a timing channel which the OS is powerless to close, and thus render the processor inherently insecure.

Again, this requirement is not really revolutionary, for example, all processors support an instruction to flush the complete cache hierarchy. Unfortunately, on many contemporary processors, this is incomplete (despite statements in the manual to the contrary). Such incompleteness of the flush functionality is not acceptable, the AISA contract must be strictly obeyed. And it must extend to features that are not considered caches but nevertheless maintain state that depends on execution history, the prefetcher state machine is an example.

Off-chip features, such as DRAM row buffers, are not covered by the ISA but must nevertheless included in the AISA. This should not cause any particular difficulty, as no new functionality is required. As long as the addressing structure of DRAM is sufficiently well specified, the OS can force the row buffers into a defined state (independent of prior execution history) by accessing certain addresses.

VI. Will architects accept it?

There are two main reasons why manufacturers are reluctant to disclose more details about their (micro-)architectures: They want to avoid constraining their design space, and they want to protect their intellectual property.

We have already argued that, unlike the ISA, the AISA does not lose its value if it changes frequently, e.g. with each new processor implementation. This avoids constraining the design.

In many cases such changes would be updates to the values of parameters, such as latencies of multi-cycle instructions or properties of the cache, which are easily adapted in hardware models.

Most of these required properties can be reverse-engineered anyway, although often with substantial effort, and certainly not with the dependability that is required for safety-critical systems. And for most of the relevant hardware features, say branch-predictors, a relatively high-level abstraction, that glosses over many details, is sufficient for a sound model of the hardware, since there is no strong reason for tight bounds, as argued earlier.

And for ensuring confidentiality we really just need dependable mechanisms for scrubbing shared hardware of residual state. This itself does not expose anything about the internals of the architecture. In summary, IP protection should not stand in the way of providing the specifications needed for building safe systems.

The proposed AISA is much less drastic than that of the precision-timed (PRET) machine (Lee, "Computing Needs Time", CACM, May 2009). While PRET certainly greatly simplifies temporal safety, it does strongly constrain the architects, and they seem very reluctant to accept this approach. The AISA specifically aims to reduce that barrier to uptake. Still, can obviously not be sure that manufacturers will accept this recommendation, but for safety's sake, let us hope they will!

VII. Conclusions

In summary, we observe that the existing hardware-software contract, the ISA, is insufficient for enforcing security and safety in computer systems, as it hides important temporal behaviour of the hardware. It must be replaced by a new contract, in the form of an augmented ISA that exposes the relevant detail and the mechanisms the OS can use to manage microarchitectural state safely and securely.