IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2020, No. 2, pp. 73–98. DOI:10.13154/tches.v2020.i2.73-98

FENL: an ISE to mitigate analogue micro-architectural leakage

Si Gao, Ben Marshall, Dan Page and Thinh Pham

Department of Computer Science, University of Bristol,
Merchant Venturers Building, Woodland Road,
Bristol, BS8 1UB, United Kingdom.
{si.gao,ben.marshall,daniel.page,th.pham}@bristol.ac.uk

Abstract. Ge et al. [GYH18] propose the augmented ISA (or aISA), a central tenet of which is the selective exposure of micro-architectural resources via a less opaque abstraction than normal. The aISA proposal is motivated by the need for control over such resources, for example to implement robust countermeasures against micro-architectural attacks. In this paper, we apply an aISA-style approach to challenges stemming from *analogue* micro-architectural leakage; examples include power-based Hamming weight and distance leakage from relatively fine-grained resources (e.g., pipeline registers), which are not exposed in, and so cannot be reliably controlled via, a normal ISA. Specifically, we design, implement, and evaluate an ISE named FENL: the ISE acts as a fence for leakage, preventing interaction between, and hence leakage from, instructions before and after it in program order. We demonstrate that the implementation and use of FENL has relatively low overhead, and represents an effective tool for systematically localising and reducing leakage.

Keywords: Information leakage, side-channel attack, ISA, ISE, micro-architecture

1 Introduction

Micro-architecture as a design principle. In the context of (micro-)processor design, an Instruction Set Architecture (ISA) is conventionally viewed as an abstraction [HP17, Page 12]. By defining 1) a set of accessible resources, or state, and 2) a set of instructions, including their format and semantics (e.g., how they manipulate said resources), it represents an abstract interface to a concrete, hardware implementation; the former constitutes the architecture (i.e., what is "visible" to or accessible by the programmer), whereas the latter constitutes the *micro*-architecture (i.e., what is "*in*visible" to or *in*accessible by the programmer). The same concept can be couched as an interface, or contract: provided that a micro-architecture adheres to a contract specified by the associated ISA, any software written against it will execute as intended. This description is attractive, because it highlights the fact that an ISA is responsible for allowing behavioural diversity while ensuring functional compatibility, or, put another way, maximising flexibility wrt. implementation while retaining transparency wrt. usage: by separating behavioural and functional semantics of instruction execution, *different* micro-architectures can realise the *same* ISA, but, in doing so, employ features that differ in their design and/or implementation. Adopting a positive perspective, Dunham and Beard [DB18] stress the value of this fact in their analysis of early RISC designs (e.g., MIPS [HJBG81]). In essence, they argue that a clean(er) separation is an advantage because it facilitates, e.g., more aggressive optimisation of instruction execution to satisfy a given quality metric, market, or use-case.

From a negative perspective, however, the exact same pursuit of optimised instruction execution is arguably an enabler of micro-architectural attack techniques (see, e.g., [Sze16,

Received: 2019-10-15 Accepted: 2019-12-15 Published: 2020-03-02

Section 4] and [GYCH18, Section 4]). Considering a given micro-architectural resource, such attacks stem from two main underlying issues: either 1) the resource exhibits observable data-dependent behaviour, which may then be correlated with some target value, and/or 2) the resource is shared at a micro-architectural level, which may allow interaction between otherwise isolated instruction streams. Micro-architectural attacks represent a significant threat, and, as such, a range of associated countermeasure techniques have been explored (see, e.g., [Sze16, Section 5] and [GYCH18, Section 5]). Although such techniques are increasingly well understood, challenges remain wrt. their design, implementation, and deployment. For instance, resources in the micro-architecture are deeply ingrained, so not easily avoided (e.g., the penalty for disabling the resource is significant) or altered (e.g., any countermeasure may be extremely invasive), and are transparent to executing instructions as a result of abstraction by the ISA. This implies a) a lack of control over said resource [GYLH17], which makes it hard to eliminate insecure behaviour, and b) the security properties of instructions may differ when executed on two functionally equivalent micro-architectures (i.e., an implemented countermeasure may not "port" across micro-architectures).

Some proposals have attempted to address such challenges by fundamentally re-evaluating what an ISA *is*, and so the roles and structure of both ISA and associated micro-architecture(s). Examples include proposals for ISAs that prevent certain forms of attack by-design (e.g., [YHHF19, ZSM19]), and proposals for ISAs that selectively expose resources in the associated micro-architecture (e.g., [Hei18, GYH18, LPAF+18]). The proposal of Ge et al. [GYH18] for a "new security-oriented hardware/software contract" they dub the augmented ISA (or aISA), is an exemplar of the latter approach. Their argument, in essence, is that normal ISAs are failing to provide an appropriate abstraction because they abstract details that would otherwise allow provision of time protection (i.e., the prevention of unauthorised temporal interaction). Based on the high-level principle that it should be possible to 1) partition, and/or 2) reset resources in the micro-architecture, the aISA demands lower-level properties that render the ISA less opaque, *but*, as a trade-off, able to provide time protection and hence prevent associated micro-architectural attacks.

An aISA-style approach to analogue micro-architectural leakage. One can imagine a general classification including two forms of information leakage: 1) *discrete* (or *digital*) leakage stemming from logical, or functional characteristics of a given micro-architecture (e.g., relating to cache-hit or miss activity), and 2) *analogue* leakage stemming from physical, or behavioural characteristics (e.g., relating to transistors realising said cache); instruction execution latency would be an example of the former, with power consumption [KJJ99, MOP07] an example of the latter. Shifting focus specifically to micro-architecture side-channel attacks then, most instances harness discrete forms of leakage. Despite this, however, it should be clear that instances based on analogue leakage are also plausible. Among an increasing range of such instances, Zoni et al. [ZBPF18] exemplify this fact by systematically evaluating analogue leakage from the micro-architecture of an OpenRISC-based SoC. A central example is Hamming weight and/or distance based leakage related to pipeline registers and forwarding paths, which can be observed via, for example, power-based measurement (or simulation).

Given the focus on discrete forms of leakage by, for example, aISA, an open question would be whether and how a similar approach can be usefully applied in the context of analogue leakage. We attempt to answer this question, introducing a micro-architecture aware Instruction Set Extension (ISE) that acts as a fence for analogue leakage; we use the name FENL to refer to the ISE. Whereas a conventional (memory) fence instruction relates to the *memory ordering* model, for example, FENL relates to the *leakage* model. We position FENL as a means of supporting (at least) three core use-cases or principles:

1. It acts as a mechanism for *localisation* of leakage: using the fence instruction, one can

localise leakage to the micro-architectural resource causing it. In concept, this can be viewed as aligning with principles described by the term design-for-test: it represents a designed-in mechanism to enable leakage testing (or evaluation), rather than assume that doing so is plausible post hoc.

2. It acts as a mechanism for *reduction* of leakage: using the fence instruction, one can reduce the leakage previously identified. In the interests of clarity, we stress two points. First, FENL is not a panacea, nor even a countermeasure in and of itself. It is best understood as a facilitator for more effective implementation (or fine-tuning) of *other* countermeasures (e.g., masking). Second, even given the above we deliberately use the term reduce rather than remove: the degree to which leakage is reduced is inherently limited by the FENL concept, and any implementation of it.
3. It offers the potential for a close(r) connection between theory and practice: it provides a form of anchor (or guarantee) wrt. leakage, with which provably secure (e.g., masking, per [BBD+15]) constructions can be be implemented more robustly.

Abstractly, we claim FENL is general-purpose in the sense it is agnostic to any specific base ISA, micro-architecture, or workload executed on an associated implementation (i.e., processor core). Concretely, however, an implementation of FENL is necessarily specific in terms of form and function. As a result, we limit the paper and temper any claims to a specific remit: we specifically consider 1) micro-controller class processor cores only, which limits the degree of of micro-architectural complexity vs. the wider design space, and 2) leakage stemming from the power consumption of such cores, dismissing various similar (e.g., EM) and dissimilar (e.g., fault injection) attack vectors out of scope.

Organisation. Section 2 presents some motivation for and (abstract) design of FENL. Section 3 then presents 1) a (concrete) FENL design based on use of RISC-V as the base ISA, and 2) an implementation of said design in two different RISC-V compliant micro-architectures. Both the concept and implementations of it are then evaluated in Section 4, via a suite of experiments designed to explore their efficacy. Finally, Section 5 presents some conclusions and potential directions for future work.

2 Design

2.1 Related work

Metrics. In order to detect and/or quantify micro-architectural leakage, a range of metrics and methodologies exist: Side-channel Vulnerability Factor (SVF) [DMWS12, DMWS13], Signal Available to Attacker (SAVAT) [CZP14], and the Welch's t-test [Wel47] based Test Vector Leakage Assessment (TVLA) [GJJR11], can, for example, be used to assess *if* (or *when*) leakage occurs. A central advantage of such approaches is their non-specific nature, which helps them shift leakage detection away from priori-driven[1] principles. However, although leakage detection can be couched as a pass-or-fail process in academic or test laboratories, the situation is more complex for vendors. They are tasked with understanding any leakage detected, then harnessing said understanding to formulate and implement appropriate countermeasures; doing so likely represents an iterative, ongoing process in contrast. So, specifically, the lack of explainability, stemming from the non-specific nature of approaches such as TVLA, represents a challenge: there is no easy way to localise (e.g., to a given resource) or reduce leakage, so address challenges relating to *where* and *why*.

In our experience, this challenge is typically addressed using a somewhat ad hoc "guess-and-determine" strategy which can be demanding wrt. the effort, time, and expertise

[1] Examples include the "test all known attacks" approach, which may fail for reasons such as an invalid leakage model rather than an absence of leakage; such an approach is, in fact, the de facto standard for cases such as Common Criteria (CC) and EMVCo evaluation.

required, while also fragile wrt. the outcome. FENL is (partly) motivated by a need to address such problems, and thus support a more systematic strategy.

Analysis. Experimental analysis of (embedded) processor cores in the context of energy and power efficiency has a lengthy history: a much cited study of Tiwari et al. [TMW94] is now, for example, 25 years old. Given the threat of analogue micro-architectural leakage, similar approaches have often been re-purposed to provide insight of a security-oriented nature. Examples (see, e.g., [SR15, CGMA+15, CGD18, BP18, ZBPF18, DAK19]) cover a spectrum of processor cores, underlying fabrics (e.g., ASIC, FPGA), and micro-architectural complexity (from non-pipelined through to deep, even super-scalar pipelines). The majority of such examples attempt black-box characterisation of leakage from a given core, for example in terms of source (e.g., architectural, general-purpose registers, or micro-architectural pipeline registers) and/or type (e.g., Hamming weight or distance). Le Corre et al. [CGD18] offer an example of a different type, in identifying a more explicit, attack-oriented problem. By using a HDL-based leakage simulator for Cortex-M3 called MAPS, they demonstrate that analogue micro-architectural leakage (specifically, Hamming distance leakage from a pipeline register) can render masked implementations insecure.

FENL is (partly) motivated by delivery of more explicit, grey- or even white-box analysis, which can offer easier to use and more precise characterisation.

Countermeasures. Seuschek et al. [SSG17] suggest use of leakage aware code generation [SSG17, Section 6]. This involves careful 1) scheduling (i.e., ordering) of instructions, and 2) register allocation informed by a leakage characterisation [SSG17, Section 4] which captures vulnerable instruction sequences. Bayrak et al. [BRN+15] consider what we view as the closest related work[2] to FENL: they described a compiler-driven approach, which operates by 1) identifying [BRN+15, Section 4] vulnerable instructions, then 2) applying [BRN+15, Section 5] a program transformation to mitigate leakage from vulnerable instructions. One of the transformations considered is that of random pre-charge. For an 8-bit AVR processor code, the compiler might, for example, translate `lds Rd, ADDR` into `lds Rr, RND ; mov Rd, Rr ; lds Rd, ADDR`, st. general-purpose register `Rd` is randomised before use as the destination of a vulnerable load from memory.

The advantage of both approaches is their non-invasive nature: they can be realised in software alone. However, by operating at an architectural vs. a micro-architectural level, this also means less control over and so less robust mitigation wrt. leakage from resources of the latter class. This forms a central differentiator from and so motivation for FENL.

2.2 An overview of FENL

2.2.1 Concept

The concept[3] of a fence (or barrier) instruction is common in modern ISAs. When defined wrt. a class of instructions, instances typically guarantee all instructions in said class *before* the fence (in program order) complete execution before any instructions *after* it. For the class of memory access instructions, fences are commonly provided to enforce a specific memory ordering model: they allow control of out-of-order memory access, e.g., to ensure serialisation. Examples of this type are supported by flavours of x86 (e.g., via `mfence` [X8618, Page 4-22], `sfence` [X8618, Page 4-597], and `lfence` [X8618, Page 3-541]), ARM (e.g., via `dmb` [ARM18, Section A6.7.21]), SPARC (e.g., via `membar` [WG03,

[2] We note that the Rosita tool of Shelton et al. [SSB+19], which appeared post-submission, makes use of a similar mechanism.

[3] We note that the terms fence and barrier are overloaded, in the sense they are used for a different purpose elsewhere in the field of hardware security; see, e.g., [KGS+19].

Section 8.4.3]), MIPS (e.g., via `sync` [MIP16, Pages 407–411]), and RISC-V (e.g., via `fence` [RV:19a, Section 2.7]).

By analogy, the central concept in FENL is that of a fence for leakage: it guarantees that instructions *after* the fence cannot logically interact[4] with those *before* it, and thus will not yield associated leakage. To realise this concept, FENL applies Property 1 of the aISA [GYH18, Section 5], which states that "any shared micro-architectural feature can either be partitioned between security domains, or reset when required". We carefully use the term flush rather than reset, because the latter has a particular interpretation in the context of digital logic; by selectively flushing resources as the fence is executed, their previous state (dictated by instructions before the fence) cannot influence their future state or transitions (dictated by instructions after the fence). Note that the remit of FENL is strictly limited to micro-architectural resources, and leakage from them: architectural resources, such as the general-purpose register file and memory, are outside that remit.

One may alternatively use the analogy of security-oriented pre-charge logic styles [MM17]: computing an output using such a cell proceeds in two phases, namely 1) pre-charge using fixed inputs, then 2) evaluation using real inputs, designed to reduce leakage during the evaluation phase. Substituting instruction for cell, resource for input, and flush for pre-charge, FENL can be viewed as (selectively) applying an analogous approach.

2.2.2 Realisation

Consider a set R of micro-architectural resources (or logical collections thereof), and let $\sigma(R_i)$ denote the stage[5] of execution some i-th resource R_i exists or is applied in. The FENL design can be described as comprised of two components:

1. A w-bit configuration register FENL.CR, each i-th bit in which controls (or maps to) R_i, plus a suite of access instructions that allow FENL.CR to be written to and read from: examples could include (but are not limited to) transfer from and to the general-purpose register file per

   $$\begin{array}{lcl} \texttt{fenl.crwr rs} & \mapsto & \mathsf{FENL.CR} \leftarrow \mathsf{GPR}[\texttt{rs}] \\ \texttt{fenl.crrd rd} & \mapsto & \mathsf{GPR}[\texttt{rd}] \leftarrow \mathsf{FENL.CR} \end{array}$$

2. A fence instruction `fenl.fence`. From an architectural perspective, the semantics of said instruction mirror those of a NOP: it has *no* architectural impact. From a *micro*-architectural perspective, however, the semantics act to flush resources as it progresses through stages of execution: when an instance of `fenl.fence` reaches execution stage j, each i-th resource which exists or is applied in said j-th stage (i.e., each $i \in \{i' \mid 0 \leq i' < w,\ \sigma(R_{i'}) = j\}$) is flushed iff. $\mathsf{FENL.CR}_i = 1$.

Beyond this intentionally high-level description, various low-level details demand further discussion. Such details inform our implementation(s), as outlined in Section 3, in line with the paper remit.

Parameterisation. The parameter w limits the number of resources that can be controlled via FENL.CR. Cases where $w < |R|$ are problematic, because any R_i for $i \geq w$ cannot be controlled. Depending on micro-architecture complexity, such cases seem plausible when w matches the word size of the base ISA, e.g., where $w \in \{32, 64\}$. To resolve this problem one can imagine options including 1) selection of a larger w, which implies the need for a richer set of access instructions to set and clear out-of-range bits, or 2) consideration of some $R' \subset R$, which implies some resources cannot be controlled, 3) a more coarse

[4] Using the terminology of secure information flow, this guarantee prevents unintentional transfer of information between said instructions (e.g., via values used as input, or produced as output).

[5] A precise definition is inherently dependent on the micro-architecture, but it is reasonable to consider either cycle (for a non-pipelined case) or pipeline stage (for a pipelined case) as representative.

mapping of bits from FENL.CR onto resources in R, e.g., by formation of logical resource groups.

Immediate variants. An immediate version(s) of the fence instruction is an obvious extension of the description as is. The goal would be to directly, albeit statically, specify the set of resources to flush as an immediate within the encoding. Doing so would reduce the overhead related to repeated (re-)configuration of FENL.CR. The only complication of such an approach is that the immediate will likely permit specification of less than w bits; this implies a need to consider a restriction to some $R' \subset R$ per the above.

Hazards and latency. In a conventional sense, structural, data, and control hazards [HP17, Section C.2] capture situations where pipelined execution of an instruction stream would be incorrect (vs. non-pipelined, which would not). We define a leakage hazard in an analogous manner: it captures any situation where pipelined execution of an instruction stream would cause leakage (vs. non-pipelined, which would not).

Such a situation may occur wrt. pipelined execution of a fence instruction; pipeline forwarding paths may allow interaction between instructions before and after a fence, for example, violating the guarantees it should provide. To resolve a leakage hazard of this type, forwarding paths can be nullified by stalling earlier stages of the pipeline until all instructions after the fence complete execution. As a result, however, the execution latency of a fence instruction may (depending on the micro-architecture) be larger than ideal.

Invasiveness. In terms of implementation, FENL is invasive because of the requirement to act on micro-architectural resources at a fine-grained level (meaning it cannot be realised by using a "drop-in" IP module, for example); in terms of design, FENL is invasive because it exposes features in the micro-architecture.

Both facts are inherent in any aISA-style approach, but the latter demands some discussion. First, one might argue that the exposure of the micro-architecture renders software using FENL non-portable. This is true, but, because portability wrt. leakage is at least non-trivial anyway, the "portability overhead" is in fact less than one might assume. Second, there is some room for interpretation about how invasive FENL is. For example, one can imagine scenarios[6] where the mapping implied by FENL.CR is proprietary and hence unknown. Clearly there are advantages for it be public, but, equally, the knowledge that FENL.CR$_i$ controls some *unknown* resource vs. a *specific* resource can still be leveraged.

Privilege. As with components in the base ISA, the semantics of components in FENL must be accompanied by a specification of the mode(s) in which access is (dis)allowed. It seems clear that user mode access would be preferable. However, this is only viable *if* FENL.CR is restored and preserved across context switches. Not doing so implies processes are not isolated (or protected) from each other: one process could influence the semantics of a fence instruction executed by, and so associated leakage of, another, for example. This requirement forms a motivation for including access instructions to *both* write to *and* read from FENL.CR (vs. the former alone).

Flush semantics. In order to implement FENL, the semantics of flushing need specification in concrete terms. Doing so demands considering features such as:

Value: it is possible to flush a given resource via 1) zeroisation, 2) randomisation using a PRNG, 3) randomisation using a TRNG, or 4) some hybrid of the latter two choices.

Source: given $|R| > 1$ resources, it is possible, for example, to flush R_i and R_j for $i \neq j$ using the same (i.e., shared) or different (i.e., unique) source.

[6] Plausible examples include requirements for IP protection.

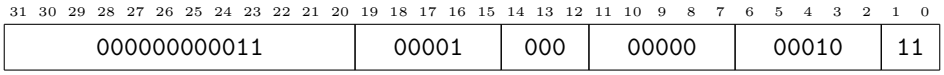


Figure 1: Encoding of a RISC-V `fenl.fence` instruction.

In brief, the (relative) area and latency overhead of these options seems to be clear. It is less clear, however, how they trade-off wrt. their security properties. Imagine R_i is a pipeline register, for example. Intuitively, and overhead aside, randomisation of R_i seems more attractive; this would avoid the leakage of Hamming weight implied by zeroisation. It does, but a counterargument would be that said Hamming weight would likely *already* have been leaked *before* the flush. As a result, zeroisation of R_i may amplify existing leakage (e.g., offer a second point of interest within a higher-order attack), but, equally, may be reasonable given the much reduced overhead.

3 Implementation

RISC-V (see, e.g., [AP14, Wat16]) is an open ISA specification. It adopts *strongly* RISC-oriented design principles (so is similar to MIPS) and can be implemented, modified, or extended by anyone with neither licence nor royalty requirements (so is dissimilar to MIPS, ARM, and x86). A central tenet of the ISA is modularity: a general-purpose base ISA (e.g., RV32I [RV:19a, Section 2]) can be augmented with some set of special-purpose, standard or non-standard (i.e., custom) extensions. As a result of these features, coupled with the surrounding community and availability of supporting infrastructure such as compilation tool-chains, a range of (typically open-source) RISC-V implementations exist.

In this section, we describe 1) our experimental hardware platform, and then 2) how said implementation is realised both in the RISC-V ISA, and two different RISC-V compliant micro-architectures that constitute two different processor cores.

3.1 Experimental platform

Our implementation and evaluation of FENL-enabled cores is supported by a standard experimental platform, which, in the interests of reproducibility, we outline briefly here.

The central component is a SASEBO-GIII [HKSS12] side-channel analysis platform, which houses two FPGAs: a Xilinx Kintex-7 (model `xc7k160tfbg676`) target FPGA, and a Xilinx Spartan-6 (model `xc6slx45`) support FPGA. We use the former exclusively, synthesising stand-alone designs for it using Xilinx Vivado 2019.1; default synthesis settings are used, with no effort invested in synthesis or post-implementation optimisation. The FPGA uses a 200 MHz external clock input, which is adjusted into a 25 MHz internal clock signal for use by the core itself.

A standard pipeline of components is attached to the SASEBO-GIII, allowing acquisition of power consumption traces. These components include a MiniCircuits BLK+89 D/C blocker, a MiniCircuits SLP-30+ 32 MHz low-pass filter, an Agilent 8447D amplifier (with a 100 kHz to 1.3 GHz range, and 25 dB gain), and a PicoScope 5000 series oscilloscope; the latter is configured to use a 250 MHz sample frequency, and 12-bit sampling resolution. Coordination of the acquisition process is managed by a workstation connected to each component: it is tasked with 1) configuration of the FPGA with a synthesised bit-stream, 2) upload of software, as generated by a RISC-V capable instance of the GNU tool-chain[7] including GCC 8.2.0, to the core via a simple boot-loader, 3) communication of input and output to and from the core via a UART-based connection, and 4) configuration and download of traces from the oscilloscope.

[7] See, e.g., https://github.com/riscv/riscv-gnu-toolchain

Table 1: A table, for each core, capturing R, the set of micro-architectural resources controlled by FENL.CR.

i	R_i	$\sigma(R_i)$	Description
0	`mem_wdata`	Operand Read	Memory write data register
1	`reg_op1`	Operand Read	Register read data 1 (RS1)
2	`reg_op2`	Operand Read	Register read data 2 (RS2)
3	`reg_out`	Operand Read	Register write data
4	`alu_out_q`	Operand Read	ALU output register
5	`uncore_0`	Operand Read	Un-core resource 0
6	`uncore_1`	Operand Read	Un-core resource 1

(a) Core 1: PicoRV32.

i	R_i	$\sigma(R_i)$	Description
0	`s2_opr_a`	Decode	Decode ⇒ Execute pipeline register A
1	`s2_opr_b`	Decode	Decode ⇒ Execute pipeline register B
2	`s2_opr_c`	Decode	Decode ⇒ Execute pipeline register C
3	`s3_opr_a`	Execute	Execute ⇒ Write memory result pipeline register A
4	`s3_opr_b`	Execute	Execute ⇒ Write memory result pipeline register B
5	`fu_mult`	Execute	Multiply-accumulate intermediate state registers
6	`fu_aessub`	Execute	AES `SubBytes` intermediate state registers
7	`fu_aesmix`	Execute	AES `MixColumns` intermediate state registers
8	`s4_opr_a`	Memory	Memory ⇒ Write-back result pipeline register A
9	`s4_opr_b`	Memory	Memory ⇒ Write-back result pipeline register B
10	`uncore_0`	Memory	Un-core resource 0
11	`uncore_1`	Memory	Un-core resource 1
12	`uncore_2`	Memory	Un-core resource 2

(b) Core 2: SCARV.

3.2 Implementing FENL in the ISA

We focus wlog. on the 32-bit RV32I [RV:19a, Section 2] base ISA, extending it with a FENL-compliant ISE by implementing the two required components as follows:

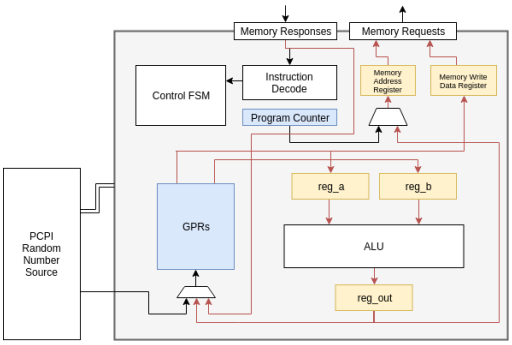
1. We realise FENL.CR using a Control and Status Register (CSR) [RV:19b, Chapter 2]. More specifically, we use the user-mode, read/write CSR $800_{(16)}$ [RV:19b, Table 2.1] which is reserved for non-standard extensions. Per [RV:19a, Section 9.1], the associated access (pseudo-)instructions are then realised as follows:

```
fenl.crwr rs  ↦  csrw 0x800, rs
fenl.crrd rd  ↦  csrr rd, 0x800
```

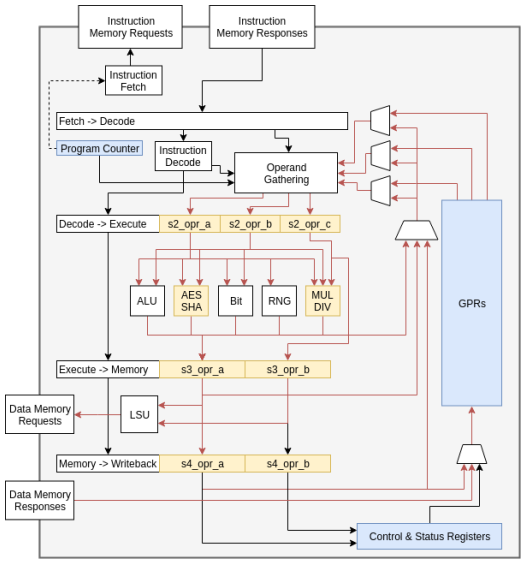
2. We realise the fence instruction using the *custom-0* encoding space [RV:19a, Table 25.1], which is reserved for non-standard extensions. Specifically, such instructions are encoded per Figure 1.

3.3 Implementing FENL in the micro-architecture(s)

To stress the generality of FENL as a concept, we opt to implement it in two micro-architectures constituting two different baseline processor cores. Both cores implement RV32IMC, i.e., the RV32I base ISA plus the M(ultiply) [RV:19a, Chapter 7] and



(a) Core 1: PicoRV32.



(b) Core 2: SCARV.

Figure 2: A block diagram of each core: architectural resources are highlighted in blue, micro-architectural resources are highlighted in yellow, and data-paths that may carry security critical data (from a side-channel perspective) are highlighted in red.

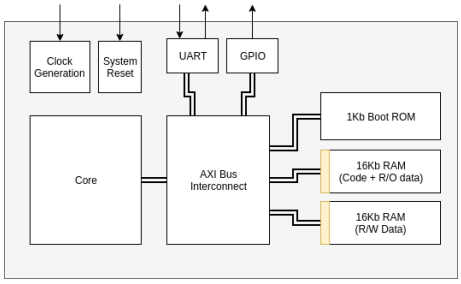


Figure 3: A block diagram of the SoC, allowing a uniform interface to either the PicoRV32 or SCARV core.

Table 2: A table, for each core, capturing some implementation results (including overhead vs. the baseline); timing slack is quoted wrt. a 25 MHz clock.

	Baseline	Baseline + FENL with zeroisation	Baseline + FENL with randomisation
Slice LUTs	1977	2002 (+1.3%)	2005 (+1.4%)
Slice FFs	1098	1100 (+0.1%)	1139 (+3.7%)
Timing Slack	25.45 ns	24.33 ns	25.06 ns

(a) Core 1: PicoRV32.

	Baseline	Baseline + FENL with zeroisation	Baseline + FENL with randomisation
Slice LUTs	5952	6014 (+1.0%)	6173 (+3.1%)
Slice FFs	2147	2163 (+0.7%)	2193 (+2.1%)
Timing Slack	23.44 ns	23.05 ns	24.38 ns

(b) Core 2: SCARV.

C(ompressed) [RV:19a, Chapter 7] standard extensions, plus selected components of the non-standard XCrypto [MPP19] extension to support implementation of cryptographic kernels. Note that neither core supports either a data or instruction cache, and so use of FENL to control the memory hierarchy is considered out of scope in this paper.

3.3.1 Baseline core 1: PicoRV32

The PicoRV32[8] core implements a non-pipelined, multi-cycle micro-architecture, a block diagram of which is shown in Figure 2a. This means each stage of execution (namely fetch, decode/operand read, execute, and write-back) occurs in *sequence* for each instruction.

Note there is one memory interface, shared between (instruction) fetch and (data) memory access stages. The core implements various performance counters, but no elements of the RISC-V Privileged Resource Architecture (PRA) [RV:19b, Chapter 3]. An RNG to support the XCrypto randomness class [MPP19, Section 2.5.2], e.g., `xc.rngsamp`, is external to the core, as realised using the so-called Pico Co-Processor Interface (PCPI).

3.3.2 Baseline core 2: SCARV

The SCARV[9] core implements a pipelined micro-architecture, a block diagram of which is shown in Figure 2b. A standard 5-stage, in-order pipeline is used, which means that each stage of execution (namely fetch, decode, execute, memory access, and write-back) occurs in *parallel* for multiple different in-flight instructions.

Note there are two memory interfaces, one for (instruction) fetch and one for (data) memory access stages. The core implements various performance counters, and elements of the RISC-V Privileged Resource Architecture (PRA) [RV:19b, Chapter 3] related to exception and interrupt handling. An RNG to support the XCrypto randomness class [MPP19, Section 2.5.2], e.g., `xc.rngsamp`, is internal to the core. Support for other XCrypto classes motivates use of a general-purpose register file with three read and two write ports, although this fact is not specifically salient from here on.

[8] https://github.com/cliffordwolf/picorv32
[9] https://github.com/scarv/scarv

3.3.3 Implementing FENL

Each block diagram in Figure 2 includes annotation of architectural resources (highlighted in blue) and micro-architectural resources (highlighted in yellow); the later describes R, the mapping of which wrt. FENL.CR is captured in Table 1. Based on this mapping, implementation of FENL is surprisingly simple. Consider a pipeline register, for example. Assuming the flush semantics are to randomise (resp. zeroise) the register, the implementation is realised by adding a suitably controlled multiplexer at the input (resp. reset). The associated control logic is no more complex than any other instructions supported by the core: the same requirement to transfer a result into an appropriate register still applies.

Hazards and latency. To support Section 4.5, two variants of the SCARV core can be selected between: the bubbling variant resolves leakage hazards (cf. Section 2.2.2) by stalling the decode stage and inserting pipeline bubbles, whereas the non-bubbling variant ignores them.

For the PicoRV32 core, the fence instruction has a 3 cycle latency; this is the same as a typical ALU-based instruction. For the SCARV core, the fence instruction has a 1 or 4 cycle latency. More specifically, for the non-bubbling variant the instruction always has a 1 cycle latency, but for the bubbling variant the instruction has a 1 cycle latency iff. no pipeline resources (i.e. bits 0 to 9 in Table 1b) are being flushed; otherwise, if any pipeline resource is being flushed, the instruction effectively takes 4 cycles to execute because the three preceding instructions must complete execution before the fence instruction can exit the decode stage. The mechanism to resolve existing (e.g., data) hazards is reused for the bubbling variant, meaning the associated overhead is not significant.

Flush semantics. In cases that demand randomisation-based flush semantics, each core makes use of a resource-constrained (P)RNG implementation. At a high level, one could view this as a decision favouring reduction in area overhead over randomness quality. More specifically, a *single* 32-bit randomness source is provided by using a Linear Feedback Shift Register (LFSR); the LFSR is updated after being sampled (vs. each clock cycle), and, if the resources flushed within a given execution stage demand *more* than 32 bits, reused across them. Note that the PRNG state is not exposed to instructions executing on the core, i.e., it cannot be read from nor written to.

SoC implementation. In order to present a consistent, uniform interface to either core, we embed them in a simple System-on-Chip (SoC). A block diagram of said SoC is shown in Figure 3, highlighting 1) system reset and clock management modules, 2) a Xilinx GPIO IP module [Xil16] (e.g., to manage the trigger signal), 3) a Xilinx UART Lite IP module [Xil17b] (e.g., to manage serial communication), and 4) various memories (e.g., a ROM for the boot-loader, and RAM and/or ROM for instructions and data), and 5) an Xilinx Advanced eXtensible Interface (AXI) interconnect IP module [Xil17a]. Note that use of off-the-shelf IP modules implies some inefficiencies. For example, the AXI interconnect introduces a 4 cycle load penalty and a 6 cycle store penalty for all load and store instructions; optimisation wrt. similar inefficiencies may alter the relative overhead of FENL.

The SoC comprises components other than the core, and, as a result, we must also consider leakage from them. This fact leads to Table 1 including in-core *and* un-core (i.e., at the SoC level) resources. We allow flushing of resources associated with the memories, for example, as a way to control undesired behaviour[10] they exhibit wrt. leakage. We revisit the rationale for doing so in Section 4.4.

[10] Conventional behaviour for a BRAM is, for example, to retain the last value read in a register (or buffer) until the next read request.

```
1  kernel1: fenl.fence          // fence
2           addi    t1, a1, 16  // t1 = &x + 16
3  .L0:     lbu     t2, 0(a1)   // t2 = *x
4           fenl.fence          // fence
5           add     t2, t2, a0  // t2 =    &sbox + *x
6           lbu     t3, 0(t2)   // t3 = *( &sbox + *x )
7           fenl.fence          // fence
8           sb      t3, 0(a2)   // *r = t3
9           addi    a1, a1, 1   // x++
10          addi    a2, a2, 1   // r++
11          bltu    a1, t1, .L0 // if x < t1, goto .L0
12          ret                 // return
```

Figure 4: Target kernel for experiment 1: masked AES `SubBytes` (assuming pre-computation of a masked S-box). Note that the kernel is described *post*-insertion of fence instructions; the original, *pre*-insertion version can be inferred by ignoring said instructions (i.e., lines 1, 4, and 7).

Implementation results. Table 2 presents a set of implementation results for each core, including cases for the 1) baseline, 2) baseline plus FENL with zeroisation-based flush semantics, and 3) baseline plus FENL with randomisation-based flush semantics. A broad interpretation of these results would be that the overhead of FENL wrt. area is marginal in both cores. Although the results show a difference wrt. timing slack, we found that, in both cores, FENL did not interact with the existing critical path of the baseline core. The reported timing slack does not behave consistently when additional logic is added to a given core; this indicates the results step from variance due to the synthesis tool-chain, since the cores operate far below their maximum frequency.

4 Evaluation

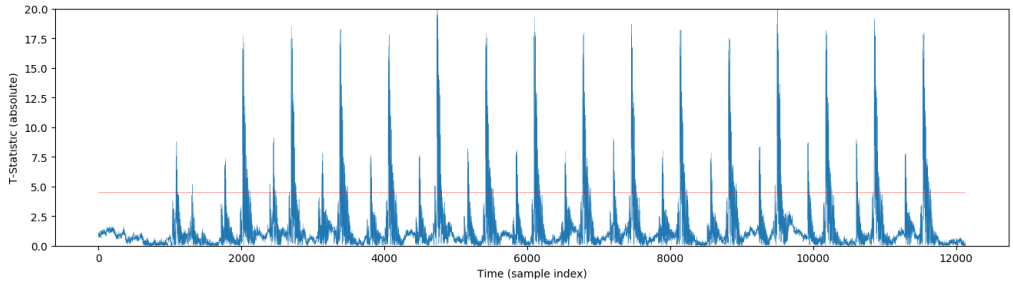
In this Section, we present a suite of experiments designed to explore the efficacy of the FENL concept and our implementations of it. Each experiment is selected to evaluate a specific aspect or articulate a specific point, so we are non-exhaustive wrt. presentation by selecting a specific workload (i.e., kernel) and core to suit.

Note that we use the *relative* magnitude of t-statistic peaks (e.g., whether and how significantly the use of FENL decreases them) as an intuitive metric for the efficacy of our implementations; doing so is in line with the remit of FENL. One must obviously then consider the *absolute* value (e.g., whether said peaks are above or below some threshold), and indeed other metrics, in any broader security evaluation.

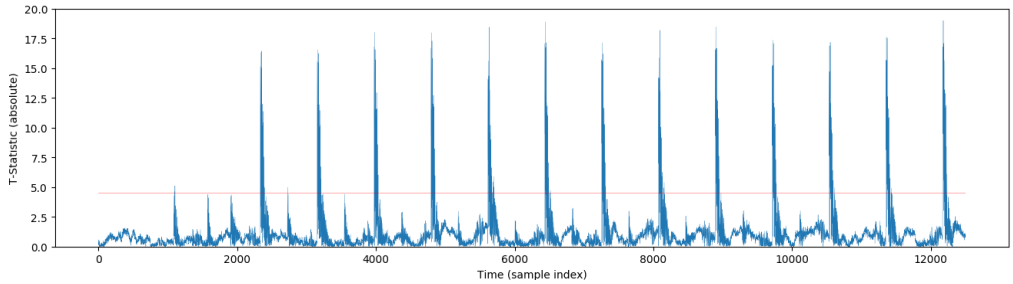
4.1 Experiment 1: intuitive localisation and reduction of leakage

Motivation. In certain situations, an intuitive approach can inform 1) where leakage stems from, and/or 2) how to reduce said leakage. As an example of the latter, one can identify cases using NOP instruction(s) as "padding" in an attempt to separate, and so weakly isolate instructions from each other wrt. leakage. For RISC-V, this makes some sense: rather than define a specific encoding, `nop` is a pseudo-instruction translated [RV:19a, Page 20] into `addi x0, x0, 0`. Although such a NOP has no architectural impact, as a valid instruction one might assume it is executed as normal; this implies it would have an impact on micro-architectural registers, e.g., setting them to zero.

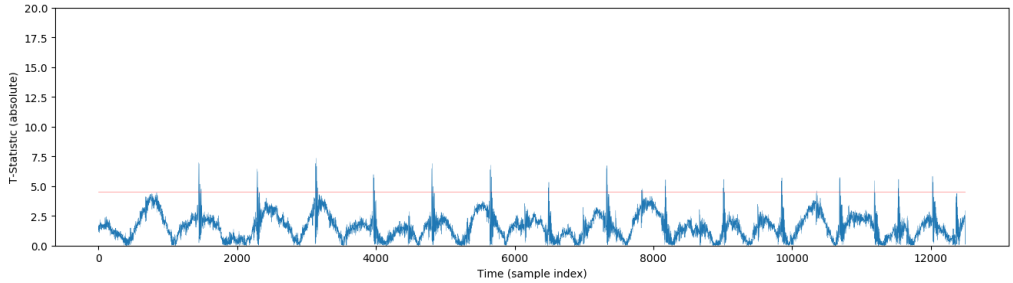
Experiment. Figure 5 captures the result of t-test based leakage detection, applied to $100,000$ power consumption traces; these stem from execution of (variants of) the masked AES `SubBytes` kernel shown in Figure 4 on the PicoRV32 (randomisation-based flush semantics) core. Insertion of the fence instructions follows an intuitive approach: line 1



(a) Fence instructions removed.
FENL.CR ignored, latency = 1224 cycles.



(b) Fence instructions replaced with `nop`.
FENL.CR ignored, latency = 1461 cycles.

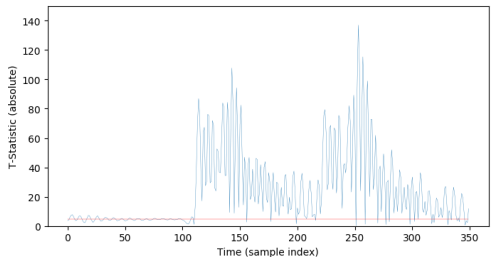


(c) Fence instructions used as is.
FENL.CR = $7F_{(16)}$, latency = 1497 cycles.

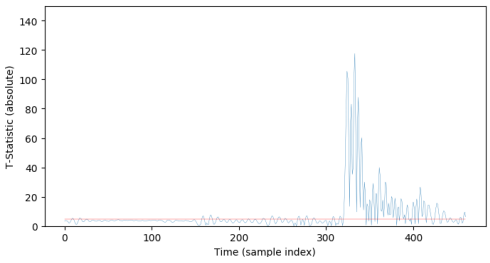
Figure 5: Results for experiment 1, relating to leakage from Figure 4 as executed on the PicoRV32 core (randomisation-based flush semantics). Note that the t-statistic is plotted in absolute form; the plots are cropped wrt. the target kernel, which has a longer latency *post*-insertion of fence instructions.

```
1  kernel2: lw     t0, 0(a0)  // t0   =  a[0]
2           lw     t2, 0(a1)  // t2   =        b[0]
3           lw     t1, 4(a0)  // t1   =  a[1]
4           lw     t3, 4(a1)  // t3   =        b[1]
5           and    t4, t0, t2 // t4   =  a[0] & b[0]
6           fenl.fence        // fence
7           and    t5, t1, t3 // t5   =  a[1] & b[1]
8           fenl.fence        // fence
9           xc.rngsamp t6     // t6   = random mask
10          xor    t6, t4, t6 // t6   = (a[0] & b[0]) ^ random mask
11          sw     t6, 0(a2)  // c[0] = t6
12          and    t0, t0, t3 // t0   =  a[0] & b[1]
13          fenl.fence        // fence
14          and    t1, t1, t2 // t1   =  a[1] & b[0]
15          fenl.fence        // fence
16          xor    t0, t0, t6 // t0   = (a[0] & b[1]) ^ c[0]
17          xor    t0, t0, t1 // t0   = (a[0] & b[1]) ^ c[0] ^ (a[1] & b[0])
18          xor    t5, t5, t0 // t5   = (a[0] & b[1]) ^ c[0] ^ (a[1] & b[0]) ^ (a[1] & b[1])
19          sw     t5, 4(a2)  // c[1] = t5
20          ret               // return
```

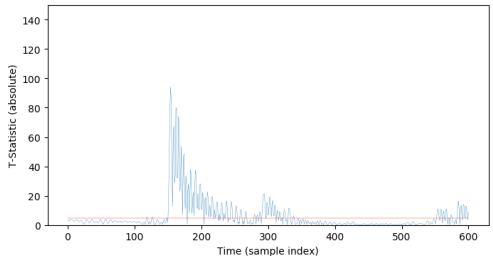
Figure 6: Target kernel for experiment 2: 2-share ISW [ISW03] multiplication. Note that the kernel is described *post*-insertion of fence instructions; the original, *pre*-insertion version can be inferred by ignoring said instructions (i.e., lines 6, 8, 13, and 15). This kernel is based on the implementation of Goudarzi et al. [GJRS18], adapted for RISC-V from ARMv7.



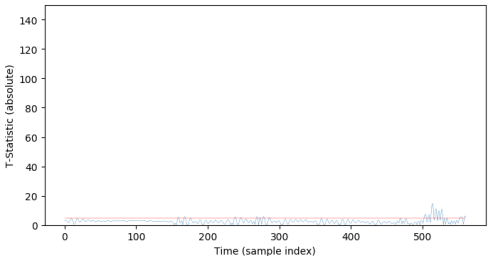
(a) Fence instructions removed from lines 6 and 8, 13, and 15; FENL.CR ignored, latency = 74 cycles.



(b) Fence instructions removed from lines 13 and 15; FENL.CR = $1FFF_{(16)}$, latency = 82 cycles.



(c) Fence instructions used as is; FENL.CR = $1C14_{(16)}$, latency = 88 cycles.



(d) Fence instructions used as is; FENL.CR = $1FFF_{(16)}$, latency = 93 cycles.

Figure 7: Results for experiment 2, relating to leakage from Figure 6 as executed on the SCARV core (randomisation-based flush semantics, bubbling variant). Note that the t-statistic is plotted in absolute form; the plots are cropped wrt. the target kernel, which has a longer latency *post*-insertion of fence instructions.

isolates the kernel from, e.g., previous, mask-related pre-computation, and lines 4 and 7 isolate load, look-up, and store steps within the kernel itself.

- Figure 5a removes the flush instructions, acting as a baseline; the latency is 1224 cycles. One can identify two significant leakage peaks per iteration of the loop, namely a smaller and a larger peak.
- Figure 5b replaces the flush instructions with `nop`; the latency is 1461 (+19.4%) cycles. Doing so significantly reduces the smaller peaks, but has no impact on the larger peaks. Intuitively, this is explained by the smaller peaks stemming from update of a micro-architectural register which the NOPs *implicitly* flush by updating it with zero (per the above).
- Figure 5c retains the flush instructions as is; the latency is 1497 (+22.3%) cycles. By using FENL.CR $= 7F_{(16)} \mapsto R =$ { `mem_wdata`, `reg_op1`, `reg_op2`, `reg_out`, `alu_out_q`, `uncore_0`, `uncore_1` }, both register *and* memory resources are *explicitly* flushed, and, as a result, the smaller *and* larger peaks are significantly reduced.

Discussion. Although the use of NOP *can* provide *some* reduction of leakage, this experiment demonstrates two limitations. First, it will have an impact on a fixed subset of micro-architectural resources, which may not cover those from which the leakage stems; in the above, this is illustrated by the lack of impact on the larger peaks in Figure 5b. Second, any impact it does have is not robust, meaning it cannot be guaranteed. As an example, consider ARMv6-M: this ISA defines a specific encoding for NOP, but the ISA specification includes the note "*[t]he timing effects of including a NOP instruction in code are not guaranteed. It can increase execution time, leave it unchanged, or even reduce it*" [ARM18, Section A6.7.47, Page 146], with the compiler documentation going further by stating "*[t]he processor might remove it from the pipeline before it reaches the execution stage*" [ARM13, Section 13.75, Page 449]. This suggests such a NOP may not be executed in the sense expected, and so may not have the intended isolating effect. Both (security) limitations are addressed by FENL, albeit with some overhead wrt. latency. To assess the latter at a broader scale, we applied the same approach to a complete masked AES implementation: the initial implementation executed in 18609 cycles, whereas reduction of residual (first order) leakage peaks through use of FENL increased this to 22024 (+18.3%) cycles.

4.2 Experiment 2: systematic localisation and reduction of leakage

Motivation. In certain situations, an intuitive approach per Section 4.1 is *not* possible: it may be *un*clear 1) where leakage stems from, and/or 2) how to reduce said leakage. This is problematic, in that one cannot typically give up: a practitioner is likely still tasked with producing a secure implementation, for example, and so needs an alternative approach. We posit that a systematic alternative is made possible by leveraging FENL in a two-phase process:

1. In the first phase, we attempt to identify which instructions cause leakage. To do so, we proceed as follows: a) set the insertion point to the start of the kernel, b) insert a fence instruction at the current insertion point, with FENL.CR configured to flush *all* resources, c) move the fence instruction through the kernel until a leakage peak is reduced or removed, d) set the insertion point to after the fence instruction, e) goto b) unless all leakage points are dealt with, or the insertion point reaches the end of the kernel.
2. In the second phase, we attempt to identify which resources, as used by the previously identified instructions, cause leakage. For each previously inserted fence instruction, we configure FENL.CR so as to (de-)activate flushing for specific resources; when there is a

(resp. is no) change in the leakage peak(s), we infer the associated leakage does (resp. does not) stem from that resource.

Note that such a process assumes independence between the fence instructions inserted. One could interpret any dependence between said instructions as a violation of the guarantees FENL hopes to enforce, and, as such, validating the assumption demands associated evaluation of the implementation.

Experiment. Figure 7 captures the result of t-test based leakage detection, applied to 100,000 power consumption traces; these stem from execution of (variants of) the 2-share ISW multiplication kernel shown in Figure 6 on the SCARV (randomisation-based flush semantics, bubbling variant) core.

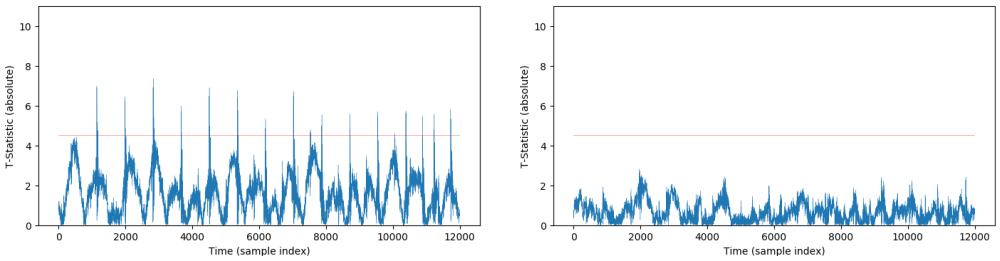
- Figure 7a removes the flush instructions, acting as a baseline; the latency is 74 cycles. This case is *before* the process is applied: one can identify two significant leakage peaks per iteration of the loop, namely a left-hand and a right-hand peak, the reduction of which forms the goal.
- Figure 7b and Figure 7c capture cases *during* the process being applied. The former retains lines 6 and 8, removes lines 13 and 15, uses $\mathsf{FENL.CR} = 1FFF_{(16)} \mapsto R = \{$ `s2_opr_a`, `s2_opr_b`, `s2_opr_c`, `s3_opr_a`, `s3_opr_b`, `fu_mult`, `fu_aessub`, `fu_aesmix`, `s4_opr_a`, `s4_opr_b`, `uncore_0`, `uncore_1`, `uncore_2` $\}$; the latency is 82 (+10.8%) cycles. The latter retains lines 6 and 8, 13 and 15, uses $\mathsf{FENL.CR} = 1C14_{(16)} \mapsto R = \{$ `s2_opr_c`, `s3_opr_b`, `s4_opr_b`, `uncore_0`, `uncore_1`, `uncore_2` $\}$; the latency is 88 (+18.9%) cycles. The cases demonstrate independent isolation and reduction of the left-hand *or* right-hand peak respectively. In short, the insertion and configuration of flush instructions allows us to infer that the a) the left-hand peak stems from pipeline registers, and b) the right-hand peak stems from memory access.
- Figure 7d retains the flush instructions as is, and uses $\mathsf{FENL.CR} = 1FFF_{(16)} \mapsto R = \{$ `s2_opr_a`, `s2_opr_b`, `s2_opr_c`, `s3_opr_a`, `s3_opr_b`, `fu_mult`, `fu_aessub`, `fu_aesmix`, `s4_opr_a`, `s4_opr_b`, `uncore_0`, `uncore_1`, `uncore_2` $\}$; the latency is 93 (+25.7%) cycles. This case is *after* the process is applied: through systematic insertion and configuration of fence instructions, both register *and* memory resources are *explicitly* flushed, and, as a result, the left-hand *and* right-hand peaks are significantly reduced.

Discussion. Above and beyond Section 4.1, this experiment demonstrates that FENL enables a *systematic* approach to the localisation and reduction of leakage: using appropriately inserted, appropriately configured, fence instructions one can a) determine which instructions and resources leakage stems from, and, in doing so, b) significantly reduce said leakage, albeit with some overhead wrt. latency.

Note that some sources of leakage identified wrt. Figure 6 were far from trivial. It seems reasonable, for example, to claim that the offset between store instructions on lines 11 and 19 acts to obfuscate understanding of the leakage (wrt. update of `s2_opr_c`) they produce; it therefore *also* seems reasonable to use this fact in support of the systematic approach enabled by FENL.

4.3 Experiment 3: flush semantics

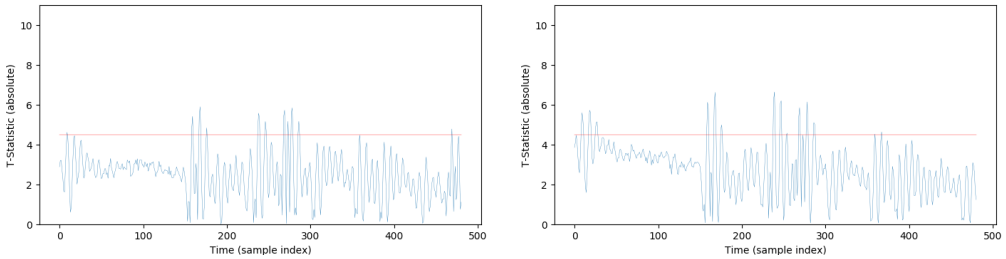
Motivation. In Section 2.2.2 we stressed the optionality associated with flush semantics, including use of either zeroisation or randomisation. Per Section 3.3, *both* options are supported by *both* cores; the latter implies a higher area overhead, even with the resource-constrained implementation adopted. However, we offered only an intuitive argument wrt. the security properties of either option: this experiment provides a more concrete comparison.



(a) Randomisation-based flush semantics. FENL.CR $= 7F_{(16)}$, latency $= 1497$ cycles.

(b) Zeroisation-based flush semantics. FENL.CR $= 7F_{(16)}$, latency $= 1497$ cycles.

Figure 8: Results for experiment 3, relating to leakage from Figure 4 as executed on the PicoRV32. Note that the t-statistic is plotted in absolute form; the plots are cropped wrt. the target kernel, which has a longer latency *post*-insertion of fence instructions.



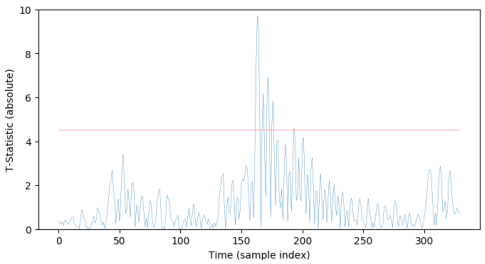
(a) Randomisation-based flush semantics. FENL.CR $= 1FFF_{(16)}$, latency $= 93$ cycles.

(b) Zeroisation-based flush semantics. FENL.CR $= 1FFF_{(16)}$, latency $= 93$ cycles.

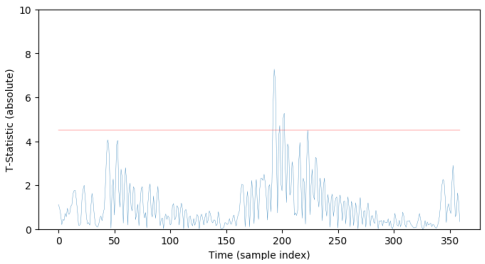
Figure 9: Results for experiment 3, relating to leakage from Figure 6 as executed on the SCARV core (bubbling variant). Note that the t-statistic is plotted in absolute form; the plots are cropped wrt. the target kernel, which has a longer latency *post*-insertion of fence instructions.

```
1 kernel3: lw      a3, 0(a0)   // a3 = x[0]
2          fenl.fence          // fence
3          lw      a4, 4(a0)   // a4 = x[1]
4          ret                 // return
```

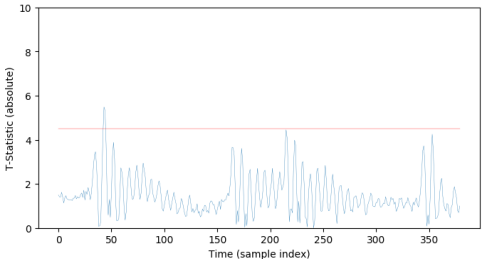
Figure 10: Target kernel for experiment 4: sequential load. Note that the kernel is described *post*-insertion of fence instructions; the original, *pre*-insertion version can be inferred by ignoring said instructions (i.e., line 2).



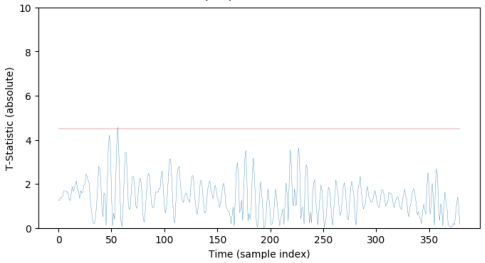
(a) Fence instructions removed; no resources flushed.
FENL.CR ignored, latency = 102 cycles.



(b) Fence instructions used as is; in-core resources flushed.
FENL.CR = $03FF_{(16)}$, latency = 109 cycles.

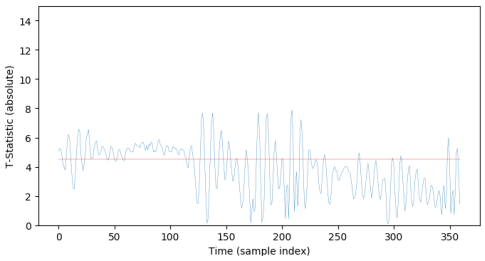


(c) Fence instructions used as is; un-core resources flushed, with modified BRAM.
FENL.CR = $1C00_{(16)}$, latency = 109 cycles.

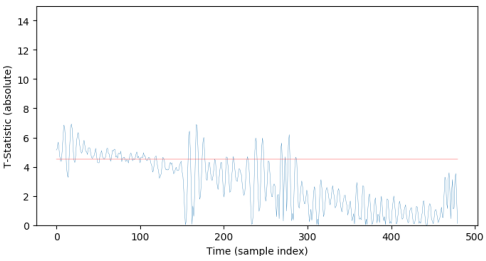


(d) Fence instructions used as is; all resources flushed, with modified BRAM.
FENL.CR = $1FFF_{(16)}$, latency = 109 cycles.

Figure 11: Results for experiment 4, relating to leakage from Figure 10 as executed on the SCARV core (randomisation-based flush semantics, bubbling variant). Note that the t-statistic is plotted in absolute form; the plots are cropped wrt. the target kernel, which has a longer latency *post*-insertion of fence instructions.



(a) Non-bubbling variant.
FENL.CR = $1FFF_{(16)}$, latency = 81 cycles.



(b) Bubbling varient.
FENL.CR = $1FFF_{(16)}$, latency = 93 cycles.

Figure 12: Results for experiment 5, relating to leakage from Figure 6 as executed on the SCARV core (randomisation-based flush semantics). Note that the t-statistic is plotted in absolute form; the plots are cropped wrt. the target kernel, which has a longer latency *post*-insertion of fence instructions.

Experiment. Figure 8 and Figure 9 capture the result of t-test based leakage detection, applied to 100,000 power consumption traces:

- Figure 8a and Figure 8b stem from execution of the masked AES `SubBytes` kernel shown in Figure 4 on the PicoRV32 core. More significant leakage peaks are evident in the left-hand, randomisation-based case.
- Figure 9a and Figure 9b stem from execution of the 2-share ISW multiplication kernel shown in Figure 6 on the SCARV (bubbling variant) core. Slightly more significant leakage peaks are evident in the right-hand, zeroisation-based case.

Discussion. More so than elsewhere, our results for this experiment are inconclusive: it seems important to a) better understand and b) reduce the design space for flush semantics to avoid implementation mistakes that may degrade the guarantees FENL is intended to provide. Doing so demands a (more) rigorous approach, ideally with a formal basis; we view this as non-trivial, so delegate it to future work.

4.4 Experiment 4: leakage from un-core resources

Motivation. The use of (third-party) IP modules (or cores) can be hugely beneficial when developing hardware: such modules help tame design complexity, for example, and promote engineering best-practices such as reuse. Particularly if the IP module is available under an open-source[11] license, use of it can be viewed as cost effective. However, this must be considered in the context of arguments wrt. trust and security (see, e.g., [MBT17]). For example, reusing the functionality offered by an IP module implies some trust relationship with it; the module forms part of the attack surface, so any *in*security introduced by it will plausibly impact the security of the design.

Per Section 3.3 we make use of such IP modules in our SoC, and, in developing our implementation of FENL, we encountered exactly an instance of the scenario above. Specifically, we identified kernels whose leakage, despite all our efforts, could not be influenced by FENL. Concluding that said leakage stemmed from an *un*-core (or external) vs. an *in*-core (or internal) resource, we investigated the IP modules outlined in Figure 3.

Experiment. Having narrowed the leakage source to data load operations, we focused on the minimal test kernel in Figure 10: the kernel simply loads two words from memory into two different registers. By inspecting the RTL for the SCARV core, we were able to verify the values never interact internally; as a result, one would not expect leakage relating to the Hamming distance between them.

Figure 11 captures the result of t-test based leakage detection, applied to 100,000 power consumption traces; these stem from execution of (variants of) the kernel on the SCARV (randomisation-based flush semantics, bubbling variant) core.

- Figure 11a removes the flush instructions, acting as a baseline; the latency is 102 cycles.
- Figure 11b retains the flush instructions as are, and uses $\mathsf{FENL.CR} = 03FF_{(16)} \mapsto R = \{$ `s2_opr_a`, `s2_opr_b`, `s2_opr_c`, `s3_opr_a`, `s3_opr_b`, `fu_mult`, `fu_aessub`, `fu_aesmix`, `s4_opr_a`, `s4_opr_b` $\}$ (i.e., all in-core resources); the latency is 109 (+6.9%) cycles.
- Figure 11c retains the flush instructions as are, and uses $\mathsf{FENL.CR} = 1FFF_{(16)} \mapsto R = \{$ `uncore_0`, `uncore_1`, `uncore_2` $\}$ (i.e., all un-core resources); the latency is 109 (+6.9%) cycles.
- Figure 11c retains the flush instructions as is, and uses $\mathsf{FENL.CR} = 1FFF_{(16)} \mapsto R = \{$ `s2_opr_a`, `s2_opr_b`, `s2_opr_c`, `s3_opr_a`, `s3_opr_b`, `fu_mult`, `fu_aessub`,

[11] https://opencores.org, https://www.librecores.org

`fu_aesmix`, `s4_opr_a`, `s4_opr_b`, `uncore_0`, `uncore_1`, `uncore_2` } (i.e., all resources); the latency is 109 (+6.9%) cycles.

Despite the latter flushing *all* in-core resources, comparing Figure 11a and Figure 11b shows a significant leakage peak. Through careful analysis of the SoC and constituent IP modules, we identified two potential sources:

1. The Xilinx AXI interconnect IP module [Xil17a]. Leakage stemming from this module was (relatively) less significant; it stemmed from the module buffering data before making it available to the core. Closer analysis revealed that the same register was shared between all attached devices, meaning, e.g., two adjacent data load operations are interspersed with an instruction fetch; although this prevents Hamming distance leakage between the values loaded, it unexpectedly yields Hamming weight leakage.
2. The Xilinx BRAM IP module [Xil19b, Xil19a]. Leakage stemming from this module was (relatively) more significant; it stemmed from the module buffering the most recently loaded value. When using separate BRAMs for instructions and data, a given data load operations would therefore yield Hamming distance leakage between the current and previous value.

To mitigate this problem, we developed a custom BRAM module; it supports the flushing of internal resources, e.g., the read data register, which can then be controlled via un-core bit(s) in FENL.CR. Figure 11c and Figure 11d demonstrate the outcome of executing the same kernel on the same core, using the custom BRAM module. Note that the leakage peak is significantly reduced by flushing all un-core resources, with no significant further reduction as the result of also flushing all in-core resources.

Discussion. This experiment demonstrates that robust implementation of FENL *must* adopt a system-wide approach that considers in-core and un-core resources: focusing on the former alone is not sufficient. In a sense, this fact simply shifts principles outlined by the aISA [GYH18, Section 5], forcing the same reasoning to be applied to un-core resources. However, it also implies some challenges wrt. the form and function of IP modules. For example, vs. the state-of-the-art, one could argue that use of FENL would demand IP modules provide 1) a compatible interface (i.e., which enables flushing by FENL), and even 2) some mechanism to reason about leakage-related design features (e.g., the presence or absence of internal registers).

4.5 Experiment 5: leakage hazard resolution

Motivation. Pipelining [HP17, Section C] is a micro-architectural optimisation that focuses on increasing instruction throughput, i.e., the number of instructions executed in a given time unit; it works by overlapping the execution of multiple in-flight instructions in time. Although the SCARV core uses exactly this approach, the implementation of FENL in such a micro-architecture demands care. As outlined in Section 2.2.2, for example, naive use of forwarding to resolve data hazards (stemming from dependencies between instructions) has the effect of allowing interaction that FENL *should* disallow. Per Section 3.3.2 we therefore support two variants of the SCARV core, which demand some comparative evaluation.

Experiment. Figure 12 captures the result of t-test based leakage detection, applied to 100,000 power consumption traces; these stem from execution of (variants of) the 2-share ISW multiplication kernel shown in Figure 6 on the SCARV (randomisation-based flush semantics) core. Note that the PicoRV32 core uses a non-pipelined micro-architecture, so the same considerations do not apply.

- Figure 12a retains the flush instructions as is, using the non-bubbling variant of the SCARV core with FENL.CR = $1FFF_{(16)} \mapsto R = \{$ `s2_opr_a`, `s2_opr_b`, `s2_opr_c`, `s3_opr_a`, `s3_opr_b`, `fu_mult`, `fu_aessub`, `fu_aesmix`, `s4_opr_a`, `s4_opr_b`, `uncore_0`, `uncore_1`, `uncore_2` $\}$; the latency is 81 cycles.
- Figure 12b retains the flush instructions as is, using the bubbling variant of the SCARV core with FENL.CR = $1FFF_{(16)} \mapsto R = \{$ `s2_opr_a`, `s2_opr_b`, `s2_opr_c`, `s3_opr_a`, `s3_opr_b`, `fu_mult`, `fu_aessub`, `fu_aesmix`, `s4_opr_a`, `s4_opr_b`, `uncore_0`, `uncore_1`, `uncore_2` $\}$; the latency is 93 (+14.8%) cycles.

A broad interpretation of these results would mirror the obvious trade-off our leakage hazard resolution approach implies: for a given kernel, the bubbling variant of the SCARV core slightly increases the latency but will yield slightly less significant leakage.

Discussion. We note that although Zoni et al. [ZBPF18] investigate leakage from forwarding paths, their analysis was performed on a simulated and thereby noiseless platform. The less significant leakage from forwarding paths in the SCARV core is likely due, in part, to a lower SNR. It still exists, however, so we view the resolution of leakage hazards as a necessary but conservative step: the latency overhead is worthwhile, because it ensures more robust isolation and so security properties.

5 Conclusion

Summary. The augmented ISA (or aISA) proposal of Ge at al. [GYH18] fundamentally re-evaluates the role and structure of both ISA and associated micro-architecture(s). Although exposing micro-architectural details in the ISA is controversial from some perspectives, it seems plausible that such an approach will be *required* to robustly mitigate certain attack vectors. Assuming such a premise, this paper has explored an aISA-style approach to the significant challenge of analogue micro-architectural leakage. Specifically, we designed, implemented, then evaluated FENL, an ISE that, by analogy, acts as a fence for analogue leakage: it affords a configurable guarantee wrt. interaction between and hence leakage from instruction before and after the fence.

Our evaluation demonstrates applicability to different micro-architectures, and efficacy wrt. reduction in leakage (so improvement in security level). such advantages are achievable with low overhead in both hardware (e.g., area and critical path) *and* software (e.g., execution latency and memory footprint); in a sense it acts as a compromise solution vs. more heavy-weight approaches (see, e.g., [MGH19] or [FGBR19]) which apply hardware masking or encryption to the data processed by a core, implying some increased latency but significantly lower area.

Future work. Although our results are largely positive, we view FENL as a first step vs. a complete solution. As such, various directions represent either important and/or interesting future work:

1. The micro-architectural design space is large: although the FENL concept may apply, a range of interesting and important questions apply for more complex instances than considered here. For example, it is not clear if FENL could (or should) be employed to control resources related to more complex memory hierarchies (i.e., caches) or execution pipelines (e.g., out-of-order execution).

   That said, existing forms of micro-architectural control, such as the x86 `clflush` [X8618, Page 3-139] or "cache line flush" instruction, are *already* somewhat analogous. Using FENL to realise a similar form of control seems plausible but, potentially demands a higher degree of configuration; an example would be the specification of a set or line

to flush (vs. the whole cache). Defining a suitably extended set of FENL-like fence instructions for RISC-V seems useful.

2. In Section 2.2, we briefly discussed the specification of privilege wrt. components in FENL. We neither considered nor evaluated whether FENL could be used in an adversarial manner, as is the case for `clflush` [X8618, Page 3-139], which could inform whether or not kernel mode access is preferable to user mode access. Exploring and resolving this question is clearly important.
3. It seems clear that FENL can support a (semi-)automatic methodology, where the compilation tool-chain is tasked with injection of suitably configured fence instructions; doing so could be guided via either a) static analysis, and/or b) dynamic, profile-guided approaches. We view the latter approach as aligning with related work such as that of Bayrak et al. [BRN+15]; the Rosita tool of Shelton et al. [SSB+19], which appeared post-submission, already goes some way to realising such a methodology. Likewise, it also seems plausible to (semi-)automate the implementation of FENL within a given micro-architecture. This is particularly true for approaches such as that of Rocket Chip[12], which harness a rich, high-level HDL and associated synthesis tools.
4. In Section 1, one cited motivation for FENL was as an anchor for proof techniques. Intuitively, one could view a fence instruction as a way to reduce transition-based leakage [BGG+14, Definition 1] so (ideally) leave only value-based leakage [BGG+14, Definition 2]; a proof can then confidently consider the latter alone. Whether this is the *right* anchor, e.g., whether it is general or strong enough, is an interesting question. In a sense, exploring an answer requires some form of hardware-software-*proof* co-design, since the requirements or guarantees of one element will impact on the others.
5. Verification of a FENL implementation poses some challenges. On one hand, verification engineers must show that the implementation operates correctly at both architectural and micro-architectural levels; an example would be to show it does not incorrectly influence any architectural state. We believe that doing so would be (relatively) simple, because existing verification techniques can ensure both correct behaviour *and* the absence of incorrect behaviour. On the other hand, however, cryptographic engineers must also have confidence the implementation works in context (i.e., as used within their software). This is more challenging, because it requires evaluation via gate-level and post-layout power consumption simulation and modelling. Although Sijacic et al. [SBY+18], for example, discuss how such design-time side-channel evaluations can be integrated into existing verification flows, doing so in a more concrete, FENL-specific manner would be an interesting task.

Acknowledgements

We would like to thank the anonymous reviewers for their helpful and constructive comments. This work has been supported in part by EPSRC via grant EP/R012288/1, under the RISE (http://www.ukrise.org) programme, and by the European Commission through the H2020 project 731591 (acronym REASSURE).

References

[AP14] K. Asanović and D.A. Patterson. Instruction sets should be free: The case for RISC-V. Technical Report UCB/EECS-2014-146, 2014.

[ARM13] ARM. *ARM Compiler: armasm User Guide*, DUI0473J (issue J) edition, 2013.

[12] https://bar.eecs.berkeley.edu/projects/rocket_chip.html

[ARM18] ARM. *ARMv6-M Architecture Reference Manual*, DDI0419E (issue E) edition, 2018.

[BBD+15] G. Barthe, S. Belaïd, F. Dupressoir, P.-A. Fouque, B. Grégoire, and P.-Y. Strub. Verified proofs of higher-order masking. In *Advances in Cryptology (EUROCRYPT)*, LNCS 9056, pages 457–485. Springer-Verlag, 2015.

[BGG+14] J. Balasch, B. Gierlichs, V. Grosso, O. Reparaz, and F.-X. Standaert. On the cost of lazy engineering for masked software implementations. In *Smart Card Research and Advanced Applications (CARDIS)*, LNCS 8968, pages 64–81. Springer-Verlag, 2014.

[BP18] A. Barenghi and G. Pelosi. Side-channel security of superscalar CPUs: evaluating the impact of micro-architectural features. In *Design Automation Conference (DAC)*, pages 120:1–120:6, 2018.

[BRN+15] A.G. Bayrak, F. Regazzoni, D. Novo, P. Brisk, F.-X. Standaert, and P. Ienne. Automatic application of power analysis countermeasures. *IEEE Transactions on Computers*, 64(2):329–341, 2015.

[CGD18] Y. Le Corre, J. Großschädl, and D. Dinu. Micro-architectural power simulator for leakage assessment of cryptographic software on ARM Cortex-M3 processors. In *Constructive Side-Channel Analysis and Secure Design (COSADE)*, LNCS 10815, pages 82–98. Springer-Verlag, 2018.

[CGMA+15] C. Cernazanu-Glavan, M. Marcu, A. Amaricai, S. Fedeac, M. Ghenea, Z. Wang, A. Chattopadhyay, J. Weinstock, and R. Leupers. Direct FPGA-based power profiling for a RISC processor. In *IEEE International Instrumentation and Measurement Technology Conference (I2MTC)*, pages 1578–1583, 2015.

[CZP14] R.L. Callan, A.G. Zajić, and M. Prvulovic. A practical methodology for measuring the side-channel signal available to the attacker for instruction-level events. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 242–254, 2014.

[DAK19] W. Diehl, A. Abdulgadir, and J.-P. Kaps. Vulnerability analysis of a soft core processor through fine-grain power profiling. Cryptology ePrint Archive, Report 2019/742, 2019.

[DB18] C. Dunham and J. Beard. This architecture tastes like microarchitecture. In *Workshop on Pioneering Processor Paradigms (WP3)*, 2018.

[DMWS12] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan. Side-channel vulnerability factor: a metric for measuring information leakage. In *International Symposium on Computer Architecture (ISCA)*, pages 106–117, 2012.

[DMWS13] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan. A quantitative, experimental approach to measuring processor side-channel security. *IEEE Micro*, 33:68–77, 2013.

[FGBR19] M. Arsath K F, V. Ganesan, R. Bodduna, and C. Rebeiro. PARAM: A microprocessor hardened for power side-channel attack resistance. arXiv preprint arXiv:1911.08813, 2019.

[GJJR11] G. Goodwill, B. Jun, J. Jaffe, and P. Rohatgi. A testing methodology for side-channel resistance validation. In *NIST non-invasive attack testing workshop*, volume 7, pages 115–136, 2011.

[GJRS18] D. Goudarzi, A. Journault, M. Rivain, and F.-X. Standaert. Secure multiplication for bitslice higher-order masking: Optimisation and comparison. In *Constructive Side-Channel Analysis and Secure Design (COSADE)*, LNCS 10815, pages 3–22. Springer-Verlag, 2018.

[GYCH18] Q. Ge, Y. Yarom, D. Cock, and G. Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering (JCEN)*, 8:1–27, 2018.

[GYH18] Q. Ge, Y. Yarom, and G. Heiser. No security without time protection: we need a new hardware-software contract. In *Asia-Pacific Workshop on Systems (APSys)*, 2018.

[GYLH17] Q. Ge, Y. Yarom, F. Li, and G. Heiser. Your processor leaks information – and there's nothing you can do about it. *CoRR*, abs/1612.04474, 2017.

[Hei18] G. Heiser. For safety's sake: We need a new hardware-software contract! *IEEE Design & Test*, 35(2):27–30, 2018.

[HJBG81] J. Hennessy, N. Jouppi, F. Baskett, and J. Gill. MIPS: A VLSI processor architecture. In H.T. Kung, R. Sproull, and G. Steele, editors, *VLSI Systems and Computations*, chapter 37, pages 337–346. Springer, 1981.

[HKSS12] Y. Hori, T. Katashita, A. Sasaki, and A. Satoh. SASEBO-GIII: A hardware security evaluation board equipped with a 28-nm FPGA. In *IEEE Global Conference on Consumer Electronics*, pages 657–660, 2012.

[HP17] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 6th edition, 2017.

[ISW03] Y. Ishai, A. Sahai, and D. Wagner. Private circuits: Securing hardware against probing attacks. In *Advances in Cryptology (CRYPTO)*, LNCS 2729, pages 463–481. Springer-Verlag, 2003.

[KGS+19] J. Krautter, D.R.E. Gnad, F. Schellenberg, A. Moradi, and M.B. Tahoori. Active fences against voltage-based side channels in multi-tenant FPGAs. Cryptology ePrint Archive, Report 2019/1152, 2019.

[KJJ99] P.C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in Cryptology (CRYPTO)*, LNCS 1666, pages 388–397. Springer-Verlag, 1999.

[LPAF+18] J. Lowe-Power, V. Akella, M.K. Farrens, S.T. King, and C.J. Nitta. A case for exposing extra-architectural state in the ISA. In *Hardware and Architectural Support for Security and Privacy (HASP)*, pages 8:1–8:6, 2018.

[MBT17] P. Mishra, S. Bhunia, and M. Tehranipoor, editors. *Hardware IP Security and Trust*. Springer, 2017.

[MGH19] E. De Mulder, S. Gummalla, and M. Hutter. Protecting RISC-V against side-channel attacks. In *Design Automation Conference (DAC)*, pages 45:1–45:4, 2019.

[MIP16] MIPS architecture for programmers Volume II-A: The MIPS32 instruction set manual. Technical Report MD00086 (rev. 6.06), MIPS, 2016.

[MM17] M. Mayhew and R. Muresan. An overview of hardware-level statistical power analysis attack countermeasures. *Journal of Cryptographic Engineering*, 7:213–244, 2017.

[MOP07] S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007.

[MPP19] B. Marshall, D. Page, and T. Pham. XCrypto: a cryptographic ISE for RISC-V. Technical Report 1.0.0, 2019.

[RV:19a] The RISC-V instruction set manual. Technical Report Volume I: User-Level ISA (Version 20190608-Base-Ratified), 2019.

[RV:19b] The RISC-V instruction set manual. Technical Report Volume II: Privileged Architecture (Version 20190608-Priv-MSU-Ratified), 2019.

[SBY+18] D. Sijacic, J. Balasch, B. Yang, S. Ghosh, and I. Verbauwhede. Towards efficient and automated side channel evaluations at design time. *Kalpa Publications in Computing*, pages 16–31, 2018.

[SR15] H. Seuschek and S. Rass. Side-channel leakage models for RISC instruction set architectures from empirical data. In *Euromicro Conference on Digital System Design*, pages 423–430, 2015.

[SSB+19] M.A. Shelton, N. Samwel, L. Batina, F. Regazzoni, M. Wagner, and Y. Yarom. Rosita: Towards automatic elimination of power-analysis leakage in ciphers. Cryptology ePrint Archive, Report 2019/1445, 2019.

[SSG17] H. Seuschek, F. De Santis, and O.M. Guillen. Side-channel leakage aware instruction scheduling. In *Cryptography and Security in Computing Systems (CS2)*, pages 7–12, 2017.

[Sze16] J. Szefer. Survey of microarchitectural side and covert channels, attacks, and defences. Cryptology ePrint Archive, Report 2016/479, 2016.

[TMW94] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 384–390, 1994.

[Wat16] A. Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, University of California at Berkeley, 2016.

[Wel47] B.L. Welch. The generalization of "student's" problem when several different population variances are involved. *Biometrika*, 34(1-2):28–35, 1947.

[WG03] D.L. Weaver and T. Germond, editors. *The SPARC Architecture Manual: Version* 9. Prentice-Hall, 2003.

[X8618] Intel 64 and IA-32 architectures – software developer's manual (Volume 2: Instruction set reference A-Z). Technical Report 325383-067US, Intel Corp., 2018.

[Xil16] Xilinx Inc. *LogiCORE IP Product Guide: AXI GPIO*, v2.0 (PG144) edition, 2016.

[Xil17a] Xilinx Inc. *LogiCORE IP Product Guide: AXI Interconnect*, v2.1 (PG059) edition, 2017.

[Xil17b] Xilinx Inc. *LogiCORE IP Product Guide: AXI UART Lite*, v2.0 (PG142) edition, 2017.

[Xil19a] Xilinx Inc. *7 Series FPGAs Memory Resources User Guide*, v1.14 (UG473) edition, 2019.

[Xil19b] Xilinx Inc. *LogiCORE IP Product Guide: AXI Block RAM (BRAM) Controller*, v4.1 (PG078) edition, 2019.

[YHHF19] J. Yu, L. Hsiung, M. El Hajj, and C.W. Fletcher. Data oblivious ISA extensions for side channel-resistant and high performance computing. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[ZBPF18] D. Zoni, A. Barenghi, G. Pelosi, and W. Fornaciari. A comprehensive side-channel information leakage analysis of an in-order RISC CPU microarchitecture. *Transactions on Design Automation of Electronic Systems (TODAES)*, 23(5):57:1–57:30, 2018.

[ZSM19] D. Zagieboylo, G. Suh, and A. Myers. Using information flow to design an ISA that controls timing channels. In *Computer Security Foundations Symposium (CSF)*, 2019.