# eLIMInate: a Leakage-focused ISE for Masked Implementation

Hao Cheng[1*] and Daniel Page[2]

[1] DCS and SnT, University of Luxembourg, Esch-sur-Alzette, Luxembourg.
hao.cheng@uni.lu
[2] Department of Computer Science, University of Bristol, Bristol, UK.
daniel.page@bristol.ac.uk

**Abstract.** Even given a state-of-the-art masking scheme, masked software implementation of some cryptography functionality can pose significant challenges stemming, e.g., from simultaneous requirements for efficiency and security. In this paper we design an Instruction Set Extension (ISE) to address a specific element of said challenge, namely the elimination of micro-architectural leakage. Conceptually, the ISE allows a leakage-focused behavioural hint to be communicated from software to the micro-architecture: using it informs how computation is realised when applied to masking-specific data, allowing associated micro-architectural leakage to be eliminated. We develop prototype, latency- and area-optimised implementations of the ISE design based on the RISC-V Ibex core; using them, we demonstrate that use of the ISE can close the gap between assumptions about and actual behaviour of a device and thereby deliver an improved security guarantee.

**Keywords:** side-channel attack, masking, RISC-V, ISE

## 1 Introduction

**Use of masking to mitigate information leakage.** Modern embedded computing devices are increasingly used in applications that can be deemed security-critical in some sense. This role is challenging due to the inherent constraints on storage, computation, and communication, and also because such devices may be deployed in an adversarial environment. Set within this context, implementation attacks, which focus on the concrete implementation rather than abstract specification of some functionality, represent a particularly potent threat. A side-channel attack is a category of implementation attack: the idea is that an attacker passively observes a target device while it executes some target functionality, using the observed behaviour to make inferences about 1) the computation performed and/or 2) the data said computation is performed on. Doing so affords the attacker an advantage with respect to some goal, such as recovery of any security-critical information (e.g., key material) involved; we say such information is leaked via (or is leakage with respect to) the mechanism used for observation (i.e., the side-channel in question).

Although alternatives exist, we focus on Differential Power Analysis (DPA) [KJJ99] and variants thereof. The importance of robust countermeasures against DPA has motivated a significant amount of research activity, with techniques often classified as being based on hiding [MOP07, Chapter 7] and/or masking [MOP07, Chapter 10]. We focus on the latter, and, more specifically, the concept of a $d$-th order Boolean masking scheme. Such a scheme

*This work was done while Hao Cheng was a visiting PhD student at the University of Bristol.

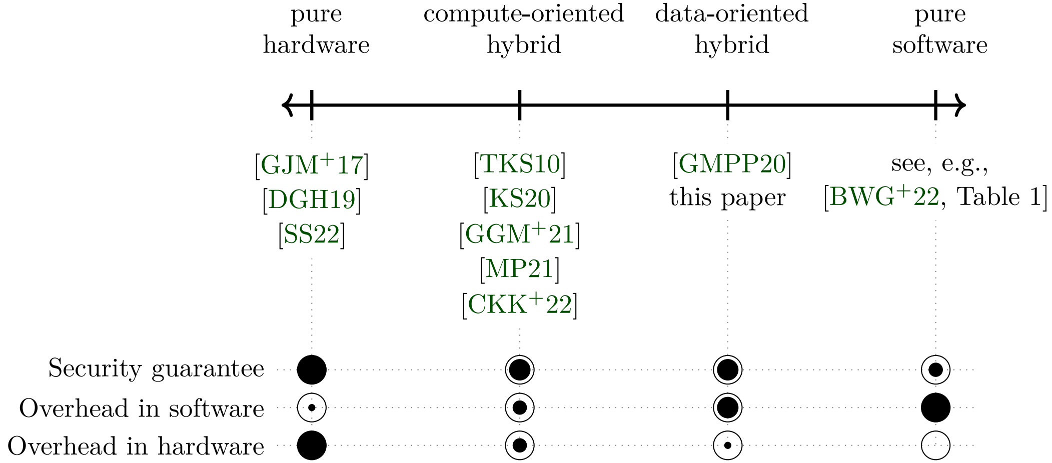


Figure 1: A selective overview of the design space for masked software implementation; indicative assessment of security guarantee and overhead is reflected by zero (◯), low (⊙), and high (●), plus various intermediate points.

represents a variable $x$ as $\hat{x} = \langle \hat{x}_0, \hat{x}_1, \ldots, \hat{x}_d \rangle$, i.e., as $d+1$ statistically independent shares, where

$$x = \bigoplus_{i=0}^{i \leq d} \hat{x}_i.$$

Application of the scheme to some functionality $r = f(x)$ can be described as three high-level steps: 1) $x$ is masked to yield $\hat{x}$, 2) an alternative but compatible functionality $\hat{r} = \hat{f}(\hat{x})$ is executed, then 3) $\hat{r}$ is unmasked to yield $r$. An attacker is now tasked with recovering $\hat{x}_i$ for all $0 \leq i \leq d$ using leakage which stems from $\hat{f}$, because $x$ can no longer be recovered directly (as it might have been using leakage which stems from $f$). Put another way, such a scheme is designed to prevent a $t$-th order attack, in which the attacker is able to combine leakage from $t < d+1$ points of interest. For example, a 1-st order scheme prevents a 1-st order attack but may be vulnerable to a 2-nd order attack.

**Challenges stemming from production of a masked implementation.** Consider a software implementation of some $f$, intended for execution by a micro-processor that supports a given Instruction Set Architecture (ISA), and the task of producing an associated masked implementation, i.e., an implementation of $\hat{f}$. At least two significant challenges stem from this task. The first challenge relates to efficiency, i.e., ensuring the masked implementation is efficient enough to be viable. Doing so is challenging because masking implies a notoriously high overhead due to factors such as computation on shares (i.e., overhead related to each “gadget” which represent the masked version of some non-masked functionality), storage of shares (e.g., register pressure due to the larger working set), and the requirement for generation of randomness; all the above are amplified when scalability to larger $d$ is considered. The second challenge relates to security, i.e., translating theoretical security guarantees related to the masking scheme into practical guarantees related to the masked implementation. There is significant evidence that doing so is challenging (cf. Beckers et al. [BWG+22]), e.g., due to the invalidity of theoretical assumptions on a given device. One common example is the occurrence of micro-architectural leakage (see, e.g., [PV17, MPW22]), which can invalidate 1) the only computation leaks assumption (“*computation, and only computation, leaks information*” [MR04, Section 2, Axiom 1]),

and 2) independent leakage assumption ("*information leakage is local*" [MR04, Section 2, Axiom 4]).

**A design space for masked implementation.** Given the task outlined above, Figure 1 attempts to illustrate the design space of viable implementation strategies; a given strategy within said design space essentially selects whether software and/or hardware is responsible for (resp. aware of) or not responsible for (resp. unaware of) masking-specific properties of instructions and their execution. Toward the right-hand side are pure software or ISA-based implementation strategies, which place responsibility in software alone. These imply zero overhead in hardware, e.g., in relation to metrics such as area, but high overhead in software, e.g., in relation to metrics such as execution latency and memory footprint. Since hardware is unaware of masking, it cannot eliminate micro-architectural leakage; software must address micro-architectural leakage via purely architectural means, e.g., using the ISA-based rewrite rules presented by Shelton et al. [SSB+21, Section V.C]. Toward the left-hand side are pure hardware implementation strategies, which place responsibility in hardware alone (typically via an entirely masked micro-architecture). These imply high overhead in hardware, but close to zero overhead in software. Since hardware is aware of masking, it can eliminate micro-architectural leakage; hardware can address micro-architectural leakage via micro-architectural means, e.g., through careful management of instruction execution. A variety of hybrid, implementation strategies exist between the two extremes. Generalising a little, such strategies will typically share responsibility by 1) adding some limited, hardware-supported functionality for masking, and 2) exposing this functionality to software via an Instruction Set Extension (ISE); an ISE-based implementation strategy of this type naturally implies a compromise, namely some overhead in hardware and some overhead in software. Addressing micro-architectural leakage could be a shared responsibility, although, since hardware is aware of masking, a security guarantee more in line with a pure hardware implementation strategy is at least plausible.

It seems reasonable to claim there is no definitively best implementation strategy. Rather, each strategy will simply offer a different trade-off in terms of the metrics above plus other important examples such as usability (i.e., the burden on a software developer) and invasiveness (i.e., whether alteration of hardware is possible, and the scope and form of said alterations).

**Contributions and organisation.** Within Figure 1, we claim there are (at least) two classes of hybrid, ISE-based implementation strategy:

1. a class of *compute*-oriented ISEs (which are closer to a pure hardware implementation strategy), where software indicates that the micro-architecture should execute masking-specific computation (e.g., a gadget) on masking-specific data (i.e., the shares used to represent a variable), and

2. a class of *data*-oriented ISEs (which are closer to a pure software implementation strategy), where software indicates that the micro-architecture should execute generic computation on masking-specific data.

We note that the data-oriented ISE class is at best less explored than the compute-oriented ISE class.

In this paper we explore a specific instance of it. Conceptually, our ISE allows a leakage-focused behavioural hint[1] to be communicated from software to the micro-architecture;

[1] As an aside, note that the same concept has been harnessed for various non-security use-cases across a range of existing ISAs. For example the ARMv6-M [ARM18, Section A6.6] and ARMv7-M [ARM21, Section A7.6] ISAs include a generic mechanism that can "*provide advance information to memory systems*

doing so informs how existing, generic computation is realised when applied to masking-specific data. After presenting relevant background information in Section 2, we organise the paper content as follows:

- In Section 3 we provide some technical analysis that fixes the scope of (i.e., provides a problem statement for) subsequent content. In short, we aim to support an ISE-based implementation strategy which eliminates leakage stemming from architectural and micro-architectural overwriting.
- In Section 4 we present a concrete ISE design. We stress that although the design is based on RISC-V, or, more specifically, the RV32I [RV:19a, Section 2] base ISA, the concepts involved are more generally applicable.
- In Section 5 we explore prototype, latency- and area-optimised implementations of our ISE design, each based on the open source Ibex[2] base core. We stress that *any* implementation of the ISE will depend inherently on the base core (resp. micro-architecture); our implementations are intended to act as exemplars, therefore, rather than a limit on how the ISE could or should be implemented in general.
- In Section 6 we evaluate our prototype ISE implementations with respect to their impact on area, execution latency, and security guarantee; for the latter, we utilise the Coco [GHP+21, HB21] formal verification framework.

  Among existing[3] work with a similar remit, we view the Rosita tool of Shelton et al. [SSB+21] and FENL design of Gao et al. [GMPP20] as the most closely related; Section 6 offers a comparative evaluation of the ISE relative to such work.

Note that all material associated with the paper, e.g., documentation and source code relating to all hardware and software implementations, is openly available[4] under an open source license.

# 2 Background

## 2.1 RISC-V

RISC-V (see, e.g., [Wat16]) is an ISA specification which emerged from academic roots; it now enjoys a significant role in educational and research activities, and industrial deployment across a range of use-cases and sectors. At least two features make RISC-V an attractive option. First, the design is open in the sense it can be implemented or modified by anyone, with neither licence nor royalty requirements. This fact has contributed to 1) a rich community organised around the RISC-V International non-profit, 2) availability of supporting infrastructure such as compilation tool-chains, and 3) a range of (typically open source) compliant implementations. Second, it adopts strongly RISC-oriented design principles but is highly modular: a sparse, general-purpose base ISA, e.g., RV32I [RV:19a,

*about future memory accesses, without actually loading or storing any data*"; the RISC-V RV32I [RV:19a, Section 2.9] and RV64I [RV:19a, Section 5.4] ISAs include a generic mechanism that can be "*used to communicate performance hints to the microarchitecture*"; the x86 ISA includes various specific mechanisms with applications that span branch prediction (e.g., branch taken and not taken prefixes [X8622, Page 2-2]), pre-fetching (e.g., as in `prefetch` [X8622, Page 4-414]), and non-temporal memory access (e.g., as in `movntdq` [X8622, Page 4-99]).

[2] https://github.com/lowRISC/ibex

[3] We note that the RISC-V Zkt [RV:22, Chapter 5] (meta-)extension is conceptually analogous: we omit it from Figure 1, however, because it focuses on execution latency and so not masking nor micro-architectural leakage per se. Likewise, we omit other fence instructions, e.g., [WSG+20, LHP20], due to the same lack of specificity.

[4] See https://github.com/scarv/eliminate.

Chapter 2] or RV64I [RV:19a, Chapter 5], can be augmented with special-purpose (or even domain-specific), standard and non-standard extensions.

We focus, without loss of generality, on a non-standard extension for RV32I, i.e., the 32-bit integer RISC-V base ISA; although such base ISAs use XLEN to denote the word size abstractly, our focus means we assume XLEN = 32 concretely throughout. We assume that some mechanism is available which supports the generation of randomness and hence fresh masks, so deem this out of scope; such a mechanism might, for example, but without loss of generality, be constructed using the RISC-V Zkr [RV:22, Chapter 4] extension.

## 2.2 Notation

Let $x_{(b)}$ denote $x$ expressed in radix- or base-$b$; if the base is omitted, it is safe to assume use of decimal (i.e., that $b = 10$). Let $x \longleftarrow y$ denote assignment of $y$ to $x$, and $x \xleftarrow{\$} y$ denote selection of $x$ uniformly at random from (e.g., a set) $y$. Let $\neg$, $\wedge$, $\vee$, and $\oplus$, denote the Boolean NOT, AND, (inclusive) OR, and (exclusive OR, or) XOR operators respectively, and $x \ll y$ and $x \lll y$ (resp. $x \gg y$ and $x \ggg y$) denote left-shift and left-rotate (resp. right-shift and right-rotate) of $x$ by $y$ bits respectively. Let $x \parallel y$ denote concatenation of $x$ and $y$. Let $\text{ext}_0^w(x)$ and $\text{ext}_\pm^w(x)$ respectively denote zero- or sign-extension of $x$ to $w$-bits. Let $\mathsf{MEM}[i]^b$ denote a $b$-byte access to some byte-addressable memory, using the address $i$; where $b = 1$, the access granularity may be omitted. Let $\mathsf{GPR}[i]$, for $0 \leq i < r$, denote the $i$-th, $w$-bit entry in the $r$-entry general-purpose register file. Note that our focus on RV32I means $\mathsf{GPR}[0]$ is fixed to 0, in the sense reads from it always yield 0 and writes to it are ignored, $w = \text{XLEN} = 32$, and $r = 32$. We allow reference to Control and Status Registers (CSRs) using either a numeric- or mnemonic-based notation; per [RV:19b, Chapter 2], for example, $\mathsf{CSR}[C00_{(16)}] \equiv$ `cycle` both refer to the cycle counter CSR.

The micro-architectural implementation of instructions may involve one or more steps. For example, the RISC-V load word instruction

$$\texttt{lw rd, imm(rs1)} \quad \mapsto \quad \mathsf{GPR}[\texttt{rd}] \longleftarrow \mathsf{MEM}[\mathsf{GPR}[\texttt{rs1}] + \texttt{imm}]^4$$

might be executed by 1) latching $s = \mathsf{GPR}[\texttt{rs1}] + \texttt{imm}$ in a Memory Address Register (MAR), 2) carrying out a memory access to yield $v = \mathsf{MEM}[s]^4$ then latching $v$ in a Memory Buffer Register (MBR), 3) writing-back MBR into $\mathsf{GPR}[\texttt{rd}]$. When describing the semantics of such an instruction, it can be important to show the cycle a given step is performed in. For example, we could describe the above as

$$\texttt{lw rd, imm(rs1)} \quad \mapsto \quad \begin{cases} 1 : \mathsf{MAR} \longleftarrow \mathsf{GPR}[\texttt{rs1}] + \texttt{imm} \\ 2 : \mathsf{MBR} \longleftarrow \mathsf{MEM}[\mathsf{MAR}]^4 \\ 3 : \mathsf{GPR}[\texttt{rd}] \longleftarrow \mathsf{MBR} \end{cases}$$

to show that the three steps are performed in cycles 1, 2 and 3, within what is therefore a 3-cycle execution stage. Said annotation may include ranges, e.g., $1 \ldots 3$ denotes cycles 1 to 3 inclusive: a step annotated as such is itself multi-cycle therefore. Annotation of multiple steps with the same cycle means they are performed in parallel; if no annotation appears, this means all steps are performed, in parallel, in cycle 1.

## 2.3 Terminology

Modulo details such as access granularity, memory and the register file can both be viewed as addressable forms of storage. As such, transfers between them can be modelled using the operation $\mathcal{T}[t] \longleftarrow \mathcal{S}[s]$ noting that if $\mathcal{S} = \mathsf{GPR}$ and $\mathcal{T} = \mathsf{MEM}$ this models a store instruction type, whereas if $\mathcal{S} = \mathsf{MEM}$ and $\mathcal{T} = \mathsf{GPR}$ this models a load instruction type; in both cases, $s$ and $t$ are the (effective) source and target addresses respectively.

**Terminology 1.** We focus on data, and so, e.g., the MBR throughout: noting that neither use of nor terminology for the MBR is consistent, if $\mathcal{S} = \mathsf{GPR}$ and $\mathcal{T} = \mathsf{MEM}$ we term it the **store buffer**, if $\mathcal{S} = \mathsf{MEM}$ and $\mathcal{T} = \mathsf{GPR}$ we term it the **load buffer**, and if the MBR is bi-directional (i.e., one MBR is used to support both operations) we term it the **load/store buffer**.

**Terminology 2.** We distinguish between resources which are physically internal or external to the micro-architecture: we term such resources **intra-core** or **extra-core** resources respectively.

**Terminology 3.** We distinguish between resources which permit **direct control** (e.g., via specific control signals) or require **indirect control** (i.e., via an abstraction layer or interface).

For example, a load/store buffer might be intra-core or extra-core (e.g., exist within an SRAM module, or bus connecting such a module to the core): the former would permit direct control by the micro-architecture but require indirect control by software, whereas the latter would require indirect control by both the micro-architecture and software.

Various work has identified architectural and micro-architectural leakage effects which relate to unintentional share recombination shown to occur during transfer of shares between forms of storage. For example, using an ST-based ARM Cortex-M0 [Cor09] target device, Shelton et al. [SSB+21, Section IV.E] carry out experiments which identify leakage stemming from overwriting one value with another 1) within $\mathcal{T} = \mathsf{GPR}$ (see [SSB+21, Section IV.E.1]) or $\mathcal{T} = \mathsf{MEM}$ (see [SSB+21, Section IV.E.2]), and 2) within the interface, i.e., a load or store buffer between $\mathcal{S}$ and $\mathcal{T}$ (see [SSB+21, Section IV.E.4]).

**Terminology 4.** We refer to the cases above as **architectural overwriting** and **micro-architectural overwriting**, because they stem from architectural and micro-architectural resources respectively.

# 3 Analysis

**Some leakage-focused requirements for share transfer.** Gaspoz and Dhooghe [GD23] introduce what they term horizontal [GD23, Definition 5] and vertical [GD23, Definition 6] non-completeness requirements on the representation of variables: their goal is to prevent unintentional share recombination that might stem from inter- and intra-register interaction respectively. One could imagine attempting to introduce analogous requirements to guide the transfer of shares between memory and the register file. For example:

**Requirement 1** (Architectural overwriting)**.** Suppose instructions of the form $\mathcal{T}[t] \longleftarrow v_0$ and $\mathcal{T}[t] \longleftarrow v_1$ are executed in cycles $i$ and $j > i$ respectively, and that no intermediate instructions that update $\mathcal{T}[t]$ are executed, i.e., no instruction of the form $\mathcal{T}[t] \longleftarrow v_2$ is executed in cycle $k$ where $i < k < j$. If $v_0$ equals $\hat{x}_p$ for some $0 \leq p \leq d$, one must ensure that $v_1 \neq \hat{x}_q$ for all $0 \leq q \leq d$.

**Requirement 2** (Micro-architectural overwriting)**.** Suppose instructions of the form $\mathcal{T}[t_0] \longleftarrow \mathcal{S}[s_0]$ and $\mathcal{T}[t_1] \longleftarrow \mathcal{S}[s_1]$ are executed in cycles $i$ and $j > i$ respectively, and that no intermediate instructions of the same type are executed, i.e., no instruction of the form $\mathcal{T}[t_2] \longleftarrow \mathcal{S}[s_2]$ is executed in cycle $k$ where $i < k < j$. If $\mathcal{S}[s_0]$ equals $\hat{x}_p$ for some $0 \leq p \leq d$, one must ensure that $\mathcal{S}[s_1] \neq \hat{x}_q$ for all $0 \leq q \leq d$.

The aim of these requirements is to eliminate leakage stemming from $\hat{x}_p$ being overwritten with some $\hat{x}_q$: put simply, the former requirement does so by preventing architectural overwriting while the latter requirement does so by preventing micro-architectural overwriting. Note that the former requirement is more general than required by the context, in the sense it captures *any* instruction which updates $\mathcal{T}[t]$ (rather than load or store instructions specifically).

**ISA-based requirement satisfaction.** As part of a pure software implementation strategy, both architectural and micro-architectural overwriting must be prevented by using the ISA alone: for architectural resources this fact implies use of direct control, whereas for micro-architectural resources it implies use of indirect control. The Rosita tool of Shelton et al. [SSB+21, Section V.C] offers an excellent example of how to do so concretely. [SSB+21, Section V.A] outlines the main strategy: Rosita reserves a (random) mask register `r7`, and uses this to flush architectural and micro-architectural state, i.e., shares, by rewriting pertinent instructions. For example:

1. Suppose GPR[4] $= \hat{x}_p$. Per [SSB+21, Section V.A], Rosita might rewrite

   `movs r3, r4` $\mapsto$ `movs r3, r7` ; `movs r3, r4`

   to prevent architectural overwriting: doing so randomises GPR[3] before it is overwritten.

2. Suppose MEM[GPR[3]] $= \hat{x}_p$. Per [SSB+21, Section V.E], Rosita might rewrite

   `ldr r2, [ r3 ]` $\mapsto$ `push r7` ; `pop r2` ; `ldr r2, [ r3 ]`

   to prevent architectural and micro-architectural overwriting: doing so randomises GPR[2] and the load buffer before they are overwritten.

3. Suppose MEM[GPR[2]] $= \hat{x}_p$. Per [SSB+21, Section V.E], Rosita might rewrite

   `str r2, [ r3 ]` $\mapsto$ `str r7, [ r3 ]` ; `str r2, [ r3 ]`

   to prevent architectural and micro-architectural overwriting: doing so randomises MEM[GPR[3]] and the store buffer before they are overwritten.

Even given a set of requirements, whose specification is a challenge in and of itself, we make two claims about an ISA-based strategy for their satisfaction along the lines above. First, an ISA-based strategy may be sub-optimal with respect to efficiency. Consider the example above, where Rosita prevents architectural and micro-architectural overwriting related to an `ldr` instruction: the rewrite translates 1 load instruction (resp. memory access) into 3. Although the overhead differs on a case-by-case basis (both per-instruction and per-ISA), it clearly may be significant. Second, an ISA-based strategy may be sub-optimal with respect to security. In particular, there are clear limitations on how effective indirect control of a micro-architectural resource can be. Consider the same example above: the security guarantee offered will depend on validity of assumptions about the micro-architecture, e.g., that the `push` and `pop` instructions use the same data-path and hence store buffer as the `ldr` instruction. In fact, some instructions can prevent either direct or indirect control over pertinent micro-architectural resources. Consider the store instruction variants in ARMv6-M: in contrast to the single-access variant `str` [ARM18, Section A6.7.60], the multi-access variant `stm` [ARM18, Section A6.7.58] "*store[s] multiple registers to consecutive memory locations using an address from a base register*". So if GPR[1] $= \hat{x}_p$ and GPR[2] $= \hat{x}_q$, then while executing `stm r0, { r1, r2 }` one cannot prevent $\hat{x}_q$ from overwriting $\hat{x}_p$ within a store buffer: the instruction semantics mean one cannot control the order registers are accessed in, nor take a Rosita-like approach by randomising the store buffer between accesses (because execution of `stm` is architecturally atomic, i.e., the multiple accesses are captured "inside" execution of a single instruction).

**An argument for ISE-based requirement satisfaction.** We claim the points above stem from the role of an ISA as an abstraction of the micro-architecture, and thus relevant resources. Somewhat aligned with the argument of Ge, Yarom, and Heiser [GYH18] for a

"*new security-oriented hardware/software contract*", we propose to address this fact using an ISE-based strategy. Specifically, we aim to design a data-oriented ISE class which is leakage-focused: the ISE should eliminate leakage stemming from architectural and micro-architectural overwriting. This goal can be described as necessary but not sufficient, in the sense that additional forms of micro-architectural leakage may also need to be considered.

It is important to stress that doing so has inherent limitations, reflecting the idea in Section 1 that it simply offers a different trade-off. For example, relative to a compute-oriented ISE, a data-oriented ISE cannot be competitive in terms of execution latency because it does not add support for masking-specific computation; we focus on comparison with an ISA-based strategy therefore. Likewise, under the conservative assumption that extra-core resources require indirect control by the micro-architecture, neither a compute-oriented nor data-oriented ISEs can deliver an "ideal" security guarantee: again, we focus on comparison with an ISA-based strategy therefore.

# 4 Design

In this Section, we present the ISE design. To explain it at a high level, consider, without loss of generality, the RISC-V load word instruction

$$\texttt{lw rd, imm(rs1)} \quad \mapsto \quad \mathsf{GPR}[\texttt{rd}] \longleftarrow \mathsf{MEM}[\mathsf{GPR}[\texttt{rs1}] + \texttt{imm}]^4$$

and some (abstractly defined) mechanism denoted

$$\mathsf{GPR}[\texttt{rd}] \overset{\triangle}{\longleftarrow} \mathsf{MEM}[\mathsf{GPR}[\texttt{rs1}] + \texttt{imm}]^4$$

which represents a variant of the existing semantics. The existing and variant semantics are functionally identical but may be behaviourally different: the existing semantics are expressed in [RV:19a, Section 2.6] as "*loads a 32-bit value from memory into* `rd`", whereas the variant semantics might be expressed as "*loads a 32-bit value from memory into* `rd`*, preventing any architectural and micro-architectural overwriting while doing so*" by appending a written hint which controls how they are implemented: the hint essentially captures a guarantee that leakage stemming from architectural and micro-architectural overwriting will be eliminated by said implementation. Note that expression of the written hint requires some care, because under-specification means any value it affords is degraded (because the semantics offer too weak a security guarantee), whereas over-specification means the semantics may be unimplementable. We attempt to balance these facts by capturing the goal in Section 3 while also permitting some degree of micro-architectural flexibility, i.e., multiple viable micro-architectural implementations.

Framed as such, the ISE can be viewed as two somewhat orthogonal components. First, a general mechanism by which such a hint can be encoded programmatically: Section 4.1 explores and selects from a range of candidate encoding mechanisms. Second, a specific set[5] of instructions: as summarised by Table 1, we divide this set into instruction classes outlined in Section 4.2 and Section 4.3.

## 4.1 Encoding

RV32I employs a fixed-length, 32-bit instruction encoding with 4 base instruction formats [RV:19a, Figure 2.2] (plus variants for, e.g., immediate operands). Each of the

[5] We stress that the ISE presented is fixed by the scope in Section 1 rather than reflecting a limitation of the concept: a broader scope can be catered for naturally, but extending the set of instructions considered to include 1) additional class-1 instructions, e.g., for addition and subtraction to support arithmetic versus Boolean masking, and 2) additional class-2 instructions, e.g., for memory access with a byte- versus word-sized granularity.

| Class | Syntax | Bits 31–25 | Bits 24–20 | Bits 19–15 | Bits 14–12 | Bits 11–7 | Bits 6–2 | Bits 1–0 | Semantics |
|---|---|---|---|---|---|---|---|---|---|
| Class-1 | `sec.and rd, rs1, rs2` | 0000000 | rs2 | rs1 | 000 | rd | 00010 | 11 | $\mapsto \mathsf{GPR}[\texttt{rd}] \stackrel{\triangle}{\longleftarrow} \mathsf{GPR}[\texttt{rs1}] \wedge \mathsf{GPR}[\texttt{rs2}]$ |
| | `sec.andi rd, rs1, imm` | simm | | rs1 | 001 | rd | 00010 | 11 | $\mapsto \mathsf{GPR}[\texttt{rd}] \stackrel{\triangle}{\longleftarrow} \mathsf{GPR}[\texttt{rs1}] \wedge \mathrm{ext}^{w}_{\pm}(\texttt{simm})$ |
| | `sec.or rd, rs1, rs2` | 0000000 | rs2 | rs1 | 010 | rd | 00010 | 11 | $\mapsto \mathsf{GPR}[\texttt{rd}] \stackrel{\triangle}{\longleftarrow} \mathsf{GPR}[\texttt{rs1}] \vee \mathsf{GPR}[\texttt{rs2}]$ |
| | `sec.ori rd, rs1, imm` | simm | | rs1 | 011 | rd | 00010 | 11 | $\mapsto \mathsf{GPR}[\texttt{rd}] \stackrel{\triangle}{\longleftarrow} \mathsf{GPR}[\texttt{rs1}] \vee \mathrm{ext}^{w}_{\pm}(\texttt{simm})$ |
| | `sec.xor rd, rs1, rs2` | 0000000 | rs2 | rs1 | 100 | rd | 00010 | 11 | $\mapsto \mathsf{GPR}[\texttt{rd}] \stackrel{\triangle}{\longleftarrow} \mathsf{GPR}[\texttt{rs1}] \oplus \mathsf{GPR}[\texttt{rs2}]$ |
| | `sec.xori rd, rs1, imm` | simm | | rs1 | 101 | rd | 00010 | 11 | $\mapsto \mathsf{GPR}[\texttt{rd}] \stackrel{\triangle}{\longleftarrow} \mathsf{GPR}[\texttt{rs1}] \oplus \mathrm{ext}^{w}_{\pm}(\texttt{simm})$ |
| | `sec.slli rd, rs1, imm` | 0000000 | imm | rs1 | 110 | rd | 00010 | 11 | $\mapsto \mathsf{GPR}[\texttt{rd}] \stackrel{\triangle}{\longleftarrow} \mathsf{GPR}[\texttt{rs1}] \ll \texttt{imm}$ |
| | `sec.srli rd, rs1, imm` | 0000000 | imm | rs1 | 111 | rd | 00010 | 11 | $\mapsto \mathsf{GPR}[\texttt{rd}] \stackrel{\triangle}{\longleftarrow} \mathsf{GPR}[\texttt{rs1}] \gg \texttt{imm}$ |
| Class-2 | `sec.lw rd, rs1, imm, ms` | ms (31–30), imm (29–20) | | rs1 | 000 | rd | 01010 | 11 | $\mapsto \mathsf{GPR}[\texttt{rd}] \stackrel{\triangle}{\longleftarrow} \mathsf{MEM}[\mathsf{GPR}[\texttt{rs1}] + \texttt{imm}]^4$ |
| | `sec.sw rs2, rs1, imm, ms` | ms (31–30), imm (29–25) | rs2 | rs1 | 001 | imm | 01010 | 11 | $\mapsto \mathsf{MEM}[\mathsf{GPR}[\texttt{rs1}] + \texttt{imm}]^4 \stackrel{\triangle}{\longleftarrow} \mathsf{GPR}[\texttt{rd}]$ |

Table 1: A summary of additional instructions that constitute the ISE, described in terms of assembly language syntax (left), encoding (middle), and semantics (right).

following candidate encoding mechanisms offers advantages and disadvantages given the goal at hand. However, we try remain consistent by aligning with wider RISC-V design principles. For example, we do not consider candidates which define new instruction formats or redefine existing instruction formats (e.g., to go beyond a 3-address format, with at most 2 source registers and 1 destination register) in a significant way.

**Candidate #1.** One could define variant instructions, providing the necessary hint via their use. For example, one could define and then use

$$\texttt{sec.lw rd, imm(rs1)} \quad \mapsto \quad \mathsf{GPR}[\texttt{rd}] \overset{\triangle}{\longleftarrow} \mathsf{MEM}[\mathsf{GPR}[\texttt{rs1}] + \texttt{imm}]^4$$

for security-critical cases.

**Candidate #2.** One could redefine existing instructions, providing the necessary hint via management of a processor mode. For example, given SEC, a Control and Status Register (CSR) for said mode, one could redefine

$$\texttt{lw rd, imm(rs1)} \quad \mapsto \quad \begin{cases} \mathsf{GPR}[\texttt{rd}] \overset{\triangle}{\longleftarrow} \mathsf{MEM}[\mathsf{GPR}[\texttt{rs1}] + \texttt{imm}]^4 & \text{if } \mathsf{SEC} = 1 \\ \mathsf{GPR}[\texttt{rd}] \longleftarrow \mathsf{MEM}[\mathsf{GPR}[\texttt{rs1}] + \texttt{imm}]^4 & \text{otherwise} \end{cases}$$

and then use $\mathsf{SEC} = 1$ for security-critical cases. This approach is conceptually similar to those now employed by ARM via Data Independent Timing (DIT) and by Intel via Data Operand Independent Timing (DOIT). For a capability-enabled ISA (e.g., one that supports CHERI [WMSN19]), it may be possible to control the mode via a capability associated with the program counter: this would allow the variant semantics to be applied while executing the masked implementation, and the existing semantics otherwise. Arguably, fault injection (e.g., skip instructions which update SEC, or corrupt it directly) may be a plausible attack vector against an implementation of this approach. Other practical considerations include the execution latency of instructions which update SEC; depending on the micro-architecture, for example, it may be necessary to flush the pipeline in order to maintain coherency with respect to execution of in-flight instructions.

**Candidate #3.** One could redefine existing instructions, providing the necessary hint via their operands. For example, given SEC, a set of distinguished registers, one could redefine

$$\texttt{lw rd, imm(rs1)} \quad \mapsto \quad \begin{cases} \mathsf{GPR}[\texttt{rd}] \overset{\triangle}{\longleftarrow} \mathsf{MEM}[\mathsf{GPR}[\texttt{rs1}] + \texttt{imm}]^4 & \text{if } \texttt{rd} \in \mathsf{SEC} \\ \mathsf{GPR}[\texttt{rd}] \longleftarrow \mathsf{MEM}[\mathsf{GPR}[\texttt{rs1}] + \texttt{imm}]^4 & \text{otherwise} \end{cases}$$

and then use an $\texttt{rd} \in \mathsf{SEC}$ for security-critical cases. This approach is conceptually similar to that outlined by Escouteloup et al. [EFLL20, Recommendation 1], who apply some security-focused requirements and semantics to a set of general-purpose registers, e.g., $\mathsf{SEC} = \{8, 9, \ldots, 15\}$, deemed confidential. As above, and at least for load and store instructions (where $\mathsf{GPR}[\texttt{rs1}]$ can be viewed as a pointer) within a capability-enabled ISA, an alternative way to designate a register as distinguished might be via a capability.

**Candidate #4.** One could redefine existing instructions, providing the necessary hint via micro-architectural compliance with a suitable specification. For example, a non-compliant micro-architecture could retain

$$\texttt{lw rd, imm(rs1)} \quad \mapsto \quad \mathsf{GPR}[\texttt{rd}] \longleftarrow \mathsf{MEM}[\mathsf{GPR}[\texttt{rs1}] + \texttt{imm}]^4,$$

whereas a compliant micro-architecture could redefine

$$\texttt{lw rd, imm(rs1)} \quad \mapsto \quad \mathsf{GPR}[\texttt{rd}] \overset{\triangle}{\longleftarrow} \mathsf{MEM}[\mathsf{GPR}[\texttt{rs1}] + \texttt{imm}]^4.$$

This approach is conceptually similar to the RISC-V Zkt [RV:22, Chapter 5] (meta-)extension, which, rather than defining functionality per se, simply "*attests that the machine has data-independent execution time for a safe subset of instructions*".

**Summary.** The candidates presented above can be summarised as follows

| | Definition | Invocation |
|---|---|---|
| Candidate #1 | Variant | Unconditionally dynamic |
| Candidate #2 | Existing | Conditionally dynamic |
| Candidate #3 | Existing | Conditionally dynamic |
| Candidate #4 | Existing | Static |

using two properties. First, a given mechanism can either define variant instructions or redefine (or overload) existing instructions. Second, invocation of a given mechanism can either be 1) static, i.e., the variant semantics are "always on" or "always off", 2) conditionally dynamic, i.e., the variant semantics are "opt in" but there is some overhead or constraint, or 3) unconditionally dynamic, i.e., the variant semantics are "opt in" and there is no overhead nor constraint.

We anticipate that a masked implementation will use a limited subset of the ISA and form a limited component of the overall workload. Although all candidates are viable, these factors suggest candidate #1 would be an effective choice: although it consumes encoding space, it permits a targeted, self-contained extension (limiting impact on the ISA as a whole) with no overhead related to invocation (for masking-specific instruction sequences) or non-invocation (for generic instruction sequences).

## 4.2 Class-1 instructions: computation-related

**Concept.** Consider the (optimised) SecAnd and SecOr gadgets for $d = 2$ presented by Biryukov et al. [BDLU17, Table 1], for example, which represent the masked versions of AND and OR respectively:

**function** $\mathsf{SecAND}((\hat{x}_0, \hat{x}_1), (\hat{y}_0, \hat{y}_1))$ **begin**
  $\hat{r}_1 \longleftarrow (\hat{x}_1 \wedge \hat{y}_1) \oplus (\hat{x}_1 \vee \neg\hat{y}_0)$
  $\hat{r}_0 \longleftarrow (\hat{x}_0 \wedge \hat{y}_1) \oplus (\hat{x}_0 \vee \neg\hat{y}_0)$
  **return**$(\hat{r}_0, \hat{r}_1)$
**end**

**function** $\mathsf{SecOR}((\hat{x}_0, \hat{x}_1), (\hat{y}_0, \hat{y}_1))$ **begin**
  $\hat{r}_1 \longleftarrow (\hat{x}_1 \wedge \hat{y}_1) \oplus (\hat{x}_1 \vee \hat{y}_0)$
  $\hat{r}_0 \longleftarrow (\hat{x}_0 \vee \hat{y}_1) \oplus (\hat{x}_0 \wedge \hat{y}_0)$
  **return**$(\hat{r}_0, \hat{r}_1)$
**end**

Although generalisation to other functionality and larger $d$ is clearly important, we claim these gadgets act as exemplars: they are implemented using a (short) sequence of bit-wise logical and shift instructions. As such, the goal of this instruction class is to provide a minimal set of such instructions to support the implementation of a maximal set of gadgets.

**Instructions.** Per Table 1, this instruction class includes `sec.andi` (resp. `sec.and`), `sec.ori` (resp. `sec.or`), `sec.xori` (resp. `sec.xor`), `sec.slli`, and `sec.srli`: these instructions support register-immediate (resp. register-register) variants of AND, OR, XOR, left-shift, and right-shift respectively. In line with the approach taken by the base ISA, note that NOT can be synthesised by using XOR: doing so relies on the fact that $\neg x \equiv x \oplus \mathrm{ext}^w_{\pm}(-1)$.

Table 2: Additional mask seed CSRs which support class-2 instructions.

| Number | Privilege | Name | Description |
|---|---|---|---|
| $800_{(16)}$ | read/write | `ms0` | Mask seed #0 |
| $801_{(16)}$ | read/write | `ms1` | Mask seed #1 |
| $802_{(16)}$ | read/write | `ms2` | Mask seed #2 |
| $803_{(16)}$ | read/write | `ms3` | Mask seed #3 |

### 4.3 Class-2 instructions: storage-related

**Concept.** The goal of this instruction class is to support transfer of shares between memory and the register file using load and store instructions. During execution of the masked implementation, we claim use of such instructions is dominated by spilling, i.e., temporary use of (a larger) memory to deal with pressure on the register file (stemming from the smaller size); this implies some structure, in the sense that loads (to pop, or restore some shares) and stores (to push, or preserve some shares) will be "grouped" into phases rather than used in a more isolated, ad hoc manner.

As well as a destination (resp. source) register address, load (resp. store) instruction provided by this class must specify 1) an effective address (via a base register address, plus an immediate offset), and 2) a mask seed; the former mirrors existing RISC-V load (resp. store) word instructions, whereas the latter is an addition to and so deviation from them. Furthermore, two constraints apply to their use. First, from a functional perspective, we assume load and store instructions operate in pairs. For example, consider two instructions: the first stores $v_0$ at address $a_0$ using mask seed $m_0$, whereas the second loads $v_1$ from address $a_1$ using mask seed $m_1$. These instructions form a load/store pair iff. $a_0 = a_1$ and $m_0 = m_1$; otherwise, there is no guarantee that $v_0 = v_1$. Second, from a behavioural perspective, a security guarantee is offered iff. each load/store pair uses a unique combination of address and mask seed. For example, consider two instructions: the first stores $v_0$ at address $a_0$ using mask seed $m_0$, whereas the second stores $v_1$ at address $a_1$ using mask seed $m_1$. If $a_0 \neq a_1$ or $m_0 \neq m_1$, the guarantee offered is that no leakage will stem from $v_1$ overwriting $v_0$.

As will become more obvious later in Section 5, the design represents an interface that allows several micro-architectural implementations, e.g., enable an approach which randomises (or remasks) shares while storing them into memory then derandomises shares while loading them from memory. Variants of this approach are applied, e.g., De Mulder, Gummalla, and Hutter [DGH19, Section 4], and Stangherlin and Sachdev [SS22, Section G]; one could also view it as a realisation of Escouteloup et al.[EFLL20, Recommendation 2] i.e., to "*encrypt the confidential data in memory, as soon as it leaves the pipeline*".

**State.** Per Table 2, instructions in this class are supported by four additional CSRs: $\mathsf{CSR}[800_{(16)} + i]$ for $0 \leq i < 4$ denotes the $i$-th mask seed. The CSRs must be initialised with fresh randomness *before* execution of the masked implementation (or at least before their first use); we assume the overhead of doing so is amortised by the execution latency of said implementation as a whole. The CSRs may need to be refreshed *during* execution of the masked implementation, e.g., to satisfy the two constraints outlined above.

**Instructions.** Per Table 1, this instruction class includes `sec.lw` and `sec.sw`: these instructions support load word and store word memory access respectively. The encodings for `sec.lw` and `sec.sw` reserve 2 MSBs of `imm` for some meta-data `ms`, which is used to specify the mask seed, i.e., $\mathsf{CSR}[800_{(16)} + \texttt{ms}]$. Note that doing so reduces `imm` from 12 to 10 bits, and thus the range from $2^{12} = 4096$ to $2^{10} = 1024$.

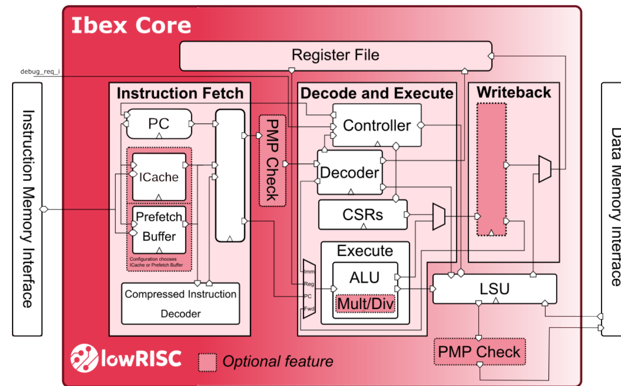


Figure 2: A block diagram describing the Ibex micro-architecture (image source: blockdiagram.svg, obtained from https://github.com/lowRISC/ibex).

# 5 Implementation

In this Section, we present prototype implementations of our ISE design: Section 5.1 introduces the base core, after which Section 5.2 and Section 5.3 then describe latency- and area-optimised implementations of the ISE within it.

We stress (again) that *any* implementation of the ISE will depend on the base core (resp. micro-architecture), meaning certain aspects of them are tailored to suit Ibex specifically. For example, we assume use of a generic SRAM module: we can neither select nor modify the specific SRAM module combined with the core. This suggests a conservative approach, where *potential* leakage (stemming, e.g., from a potential load/store buffer within the SRAM module) is eliminated using indirect control; doing so means greater overhead, but also a more robust security guarantee. However, we note that it is clearly possible and, depending on the context, attractive to do the opposite: in their CocoIbex core, for example, Gigerl et al. [GHP+21, Section 4] use a special-purpose SRAM module that eliminates certain forms of leakage.

## 5.1 Base core: Ibex

**General overview.** Ibex is a 32-bit, RISC-V compliant micro-processor core, which is designed for embedded use-cases; originally developed as part of the PULP[6] platform, the core (and a suite of associated resources) is now maintained by lowRISC. Ibex supports both FPGA- and ASIC-based synthesis targets: the block diagram in Figure 2 describes the micro-architectural design, which is highly configurable. For example, the core can support either the integer (i.e., RV32I) or embedded (i.e., RV32E) RISC-V base ISA; said base ISA can be supplemented by the multiplication [RV:19a, Chapter 7], compressed [RV:19a, Chapter 16], or bit manipulation [RV:19a, Chapter 17] extensions; the micro-architecture can use either a 2- or 3-stage pipeline (by excluding or including a dedicated write-back stage), and supports options relating to the multiplier, branch prediction, and Physical Memory Protection (PMP). Beyond this, implementation of specific units can be specialised to suit the underlying technology; the register file can be implemented using flip-flops, latches, or RAM elements, for example, in order to suit the synthesis target.

**Specific configuration.** We develop the prototype ISE implementation and perform our experiments on Ibex Demo System[7], which is also developed by lowRISC and comprises

[6] https://pulp-platform.org
[7] https://github.com/lowRISC/ibex-demo-system

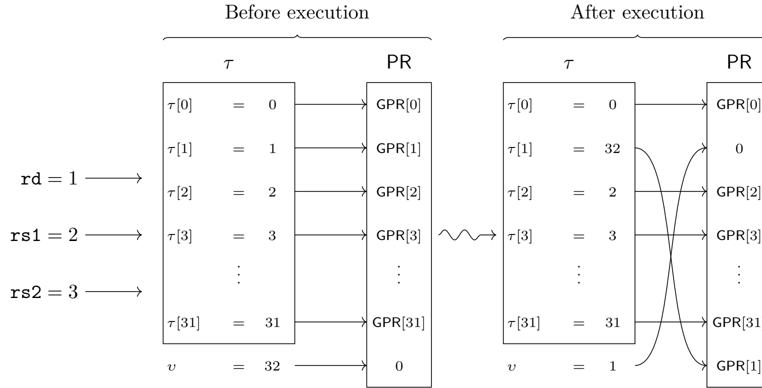


Figure 3: A diagrammatic description of how PR, GPR, $\tau$, and $\upsilon$ are managed during execution of `sec.xor x1, x2, x3` by the latency-optimised implementation.

the Ibex core. We select to use a flip-flop-based register file. For other settings, we adopt the default configuration of Ibex Demo System; the ISA is RV32IMC, i.e., bit manipulation extension is not enabled; a fast multi-cycle multiplier and an iterative divider are used; 2-stage pipeline is used, which includes an Instruction Fetch (IF) stage and an Instruction Decode and Execute (ID+EX) stage, while Write-Back (WB) is not enabled as a dedicated stage; PMP and instruction cache are disabled.

## 5.2 ISE implementation #1: latency-optimised

**Class-1 instructions.** For the implementation of class-1 instructions, we design a new mechanism for indexing the registers. In order to make it clearer, we introduce a term Physical Register, denoted by PR. In the base core, GPR[$i$] and PR[$i$] means exactly the same register; given a register index (or say address) $k$ there is only one map $k \Rightarrow$ PR[$k$] (equally $k \Rightarrow$ GPR[$k$]) used by reading (resp. writing) the data from (resp. to) the target physical register. In our implementation, first of all, GPR and PR are different: the use of GPR is viewable to software in the sense that developers can see what GPRs are being used and can choose what GPRs to use in their code (instructions); however, the use of PR is not viewable to software and is controlled by only micro-architecture. We add a new index look-up table $\tau$ between the two objectives of the original map, i.e., the register index and the target physical register, and change to use two maps to link them such that $k \Rightarrow \tau[k]$ then $\tau[k] \Rightarrow$ PR[$\tau[k]$]. But for GPR, it is still a direct map from $k$ to GPR[$k$]. Furthermore, we add one more physical register PR[32] to the register file. In our setting, in each clock cycle there are 32 general-purpose registers and 1 *idle register* that holds a value 0 all the time. In more detail, there are 32 entries in $\tau$ (see a diagrammatic description in Figure 3), each of which stores the index of a general-purpose register, and there is another variable $\upsilon$ used to hold the index of the idle register. At the beginning (i.e., after a reset signal), each entry of $\tau$ is initialised as $\tau[i] \longleftarrow i$, and $\upsilon$ points to PR[32]. In each instruction executed, we always use the current idle register, i.e., PR[$\upsilon$], as the destination (physical) register, and set PR[$\tau$[`rd`]] to be the new idle register. In this way, the data is always written to a cleared register, which prevents the architectural overwriting in the register file. Of course, the values of entries in $\tau$ and of $\upsilon$ update dynamically according to the different instructions executed. Note if `rd` is 0, we will not update $\tau$ and $\upsilon$. For the software side, there is no difference in the use of GPR. Taking `sec.xor` as an example, a formal definition

is as follows:

$$\texttt{sec.xor rd, rs1, rs2} \;\mapsto\; \begin{cases} \mathsf{PR}[\upsilon] \longleftarrow \mathsf{PR}[\tau[\texttt{rs1}]] \oplus \mathsf{PR}[\tau[\texttt{rs2}]] \\ \mathsf{PR}[\tau[\texttt{rd}]] \longleftarrow 0 \\ \tau[\texttt{rd}] \longleftarrow \upsilon \\ \upsilon \longleftarrow \tau[\texttt{rd}] \end{cases}$$

For easy understanding, in Figure 3 we consider an example that `sec.xor x1, x2, x3` is executed. It reads operands $\mathsf{PR}[\tau[2]]$ and $\mathsf{PR}[\tau[3]]$, i.e., in essence $\mathsf{PR}[2]$ and $\mathsf{PR}[3]$ per current $\tau$, computes the result, writes the result to the idle register $\mathsf{PR}[\upsilon]$, i.e., $\mathsf{PR}[32]$, and clears the register $\mathsf{PR}[\tau[1]]$, i.e., $\mathsf{PR}[1]$. In the meantime, $\upsilon$ changes to be $\tau[1]$, i.e., 1, which indicates the idle register is now $\mathsf{PR}[1]$, and $\tau[1]$ should accordingly update to be 32 as well, i.e., $\mathsf{GPR}[1]$ is now essentially the register $\mathsf{PR}[32]$.

**Class-2 instructions.** We implement the class-2 instructions based on the re-masking method of De Mulder et al. [DGH19, Section 4]. When storing (resp. loading) a share to (resp. from) the memory, the share is always masked with a Load/Store Mask (LSM), which eliminates both the architectural overwriting in the memory and the micro-architectural overwriting in the MBR. Plus our new mechanism for indexing the registers, both `sec.sw` and `sec.lw` prevent the architectural and micro-architectural overwriting. As described in [DGH19], it suggests to use 2 or 3 rounds of Keccak-f100 permutation [BDP+12] to generate an LSM, where the state is formed with the memory address and a mask seed from a specific CSR. In our implementation, we use 2 rounds of Keccak-f100 and define the generation of an LSM as (given `rs1`, `imm`, and `ms` from a class-2 instruction):

$$\mathsf{LSM} := \textsc{kecck-f}100\text{-}2\textsc{rounds}(0\cdots 0 \;\|\; (\mathsf{PR}[\tau[\texttt{rs1}]] + \texttt{imm}) \;\|\; \mathsf{CSR}[800_{(16)} + \texttt{ms}])$$

We implement the 2 rounds of Keccak-f100 in an unrolled way to ensure a single-cycle execution. As indicated in Table 2, we add four mask seed CSRs with the addresses of $800_{(16)}$ to $803_{(16)}$, which, defined in [RV:19b, Table 2.1], are preserved for the use of custom read/write. `ms` selects which CSR to be used. In formal, `sec.sw` and `sec.lw` are defined as:

$$\texttt{sec.sw rs2, rs1, imm, ms} \;\mapsto\; \mathsf{MEM}[\mathsf{PR}[\tau[\texttt{rs1}]] + \texttt{imm}]^4 \longleftarrow \mathsf{PR}[\tau[\texttt{rs2}]] \oplus \mathsf{LSM}$$

$$\texttt{sec.lw rd, rs1, imm, ms} \;\mapsto\; \begin{cases} \mathsf{PR}[\upsilon] \longleftarrow \mathsf{MEM}[\mathsf{PR}[\tau[\texttt{rs1}]] + \texttt{imm}]^4 \oplus \mathsf{LSM} \\ \mathsf{PR}[\tau[\texttt{rd}]] \longleftarrow 0 \\ \tau[\texttt{rd}] \longleftarrow \upsilon \\ \upsilon \longleftarrow \tau[\texttt{rd}] \end{cases}$$

## 5.3 ISE implementation #2: area-optimised

**Class-1 instructions.** We again take the `sec.xor` as an example to elaborate the implementation details of class-1 instructions. Note that we do not import and use the concept of PR in area-optimised implementation. At the high-level operation viewpoint, the `sec.xor` instruction is composed of two steps, namely:

$$\texttt{sec.xor rd, rs1, rs2} \;\mapsto\; \begin{cases} 1 : \mathsf{GPR}[\texttt{rd}] \longleftarrow 0 \\ 2 : \mathsf{GPR}[\texttt{rd}] \longleftarrow \mathsf{GPR}[\texttt{rs1}] \oplus \mathsf{GPR}[\texttt{rs2}] \end{cases}$$

For the low-level hardware implementation, in the decoder, a dedicated variable named `sec_bwlogic` (secure bitwise logical instruction) will be set to 1 when the class-1 instruction is decoded, and to 0 otherwise. When `sec_bwlogic` is 1, the ID stage stalls in the first clock cycle, and at the same time the signal of `sec_bwlogic` is transmitted to register file and drives to clear the destination register. In the next clock cycle, it just works

Table 3: Comparison of area, stemming from synthesis of the base core plus implementation #1 (latency-optimised, per Section 5.2) and implementation #2 (area-optimised, per Section 5.3) of the ISE; note that cumulative support for instruction classes is presented to highlight their individual contribution.

| | Registers | LUTs |
|---|---|---|
| Base core | 2363 (1.00×) | 3602 (1.00×) |
| Base core + latency-optimised class-1 | 2585 (1.09×) | 4829 (1.34×) |
| Base core + latency-optimised class-1+2 | 2713 (1.15×) | 5000 (1.39×) |
| Base core + area-optimised class-1 | 2365 (1.00×) | 3565 (0.99×) |
| Base core + area-optimised class-1+2 | 2365 (1.00×) | 3847 (1.07×) |

the same as the case of a normal `xor`, i.e., computing the result and writing it to the destination register. The computation of class-1 instructions is realised by simply (re-)using the hardware implementation of normal bitwise logical instructions (in ALU), hence its hardware cost is negligible.

**Class-2 instructions.** Essentially, the memory access instructions are implemented based on a pure-software implementation strategy used in Rosita [SSB+21]. Therefore, the LSM as well as the mask seed are not needed in this implementation; we also do not add four mask seed CSRs to further save the area overhead, and `ms` has no impact on the action of instruction. In detail, `sec.sw` consists of two steps whereas `sec.lw` needs three steps, and they are shown as follows:

$$\texttt{sec.sw rs2, rs1, imm, ms} \mapsto \begin{cases} 1\text{–}2 : \mathsf{MEM}[\mathsf{GPR}[\texttt{rs1}] + \texttt{imm}]^4 \longleftarrow 0 \\ 3\text{–}4 : \mathsf{MEM}[\mathsf{GPR}[\texttt{rs1}] + \texttt{imm}]^4 \longleftarrow \mathsf{GPR}[\texttt{rs2}] \end{cases}$$

$$\texttt{sec.lw rd, rs1, imm, ms} \mapsto \begin{cases} 1\text{–}2 : \mathsf{MEM}[\mathsf{GPR}[\texttt{sp}] + \texttt{(-4)}]^4 \longleftarrow 0 \\ 3\text{–}4 : \mathsf{GPR}[\texttt{rd}] \longleftarrow \mathsf{MEM}[\mathsf{GPR}[\texttt{sp}] + \texttt{(-4)}]^4 \\ 5\text{–}6 : \mathsf{GPR}[\texttt{rd}] \longleftarrow \mathsf{MEM}[\mathsf{GPR}[\texttt{rs1}] + \texttt{imm}]^4 \end{cases}$$

In the low-level hardware implementation, in order to make class-2 instructions work correctly in each of their different steps, it is required to introduce some new states to the finite state machine (FSM) of the ID stage and of the load-store unit respectively, which constitute the most of additional hardware overhead of class-2. Similar to class-1 instructions, we add two dedicated variables `sec_store` (secure store) and `sec_load` (secure load), whose values get updated in the decoder, and they are used by the ID-stage FSM and the load-store unit FSM. Furthermore, some other modifications in the decoder are also needed; e.g., in the first two steps of `sec.lw`, it reads the data of stack pointer register `sp` instead of the source register `rs1`.

# 6 Evaluation

To produce an experimental platform which permits evaluation of area and cycle-accurate execution latency, we use an Arty-100T board[8], which hosts a Xilinx Artix-7 (model `XC7A100TCSG324`) FPGA device. We synthesise the stand-alone design for our implementations using Xilinx Vivado 2019.1; default synthesis settings are used, with no effort invested in synthesis or post-implementation optimisation.

## 6.1 Area

Table 3 summarises the ISE overhead in terms of area: it lists the resource utilisation for base core and base core plus ISE (for both latency and area-optimised implementation

[8] https://digilent.com/reference/programmable-logic/arty-a7/start

Table 4: Comparison of execution latency (measured in clock cycles), stemming from use of the base core plus implementation #1 (latency-optimised, per Section 5.2) and implementation #2 (area-optimised, per Section 5.3) of the ISE; note that functionally comparable instructions are included in the ISE-based (e.g., `sec.and`) and ISA- (e.g., `and`) cases respectively.

| | Class-1 | | | | | | | | Class-2 | |
|---|---|---|---|---|---|---|---|---|---|---|
| | `[sec.]and` | `[sec.]andi` | `[sec.]or` | `[sec.]ori` | `[sec.]xor` | `[sec.]xori` | `[sec.]slli` | `[sec.]srli` | `[sec.]lw` | `[sec.]sw` |
| Base core | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 |
| Base core + latency-optimised | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 |
| Base core + area-optimised | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 6 | 4 |

variants), using an incremental approach to demonstrate the overhead of each instruction class. For the case of latency-optimised implementation, the extra area overhead of class-1 instructions mostly comes from our new mechanism of register indexing, e.g., the cost brought by index look-up table $\tau$ and an extra register $\mathsf{PR}[32]$; the overhead of class-2 is obviously due to the generation of LSM, e.g., additional hardware resources for four mask seed CSRs and for the associated 2 rounds of keccak-f100 permutation. The total overhead of class-1 and class-2 amounts to 15% more Regs and 39% more LUTs compared to the base core. For the area-optimised variant, compared to the base core, the class-1 plus class-2 instructions take nearly no extra Regs and only 7% more LUTs.

## 6.2 Latency

Table 4 summarises the ISE overhead in terms of execution latency: it lists the number of cycles required to execute each instruction on base core and base core plus ISE (for both latency and area-optimised implementation variants). The latency of the class-1 and class-2 instructions in latency-optimised implementation is the same as the latency of their counterparts on the base core, which is as designed and as expected. The latency of instructions in area-optimised implementation is the same as the latency of the ISA-based strategy, e.g., the latency of a `sec.xor x1, x2, x3` equals the latency of a `mv x1, x0` plus an `xor x1, x2, x3`. This translates to, when using area-optimised variant of our ISE for a masked implementation, it might take a similar execution time as an ISA-based strategy. This is less attractive for a use case where the execution time is the priority. However, apart from the timing, the efficiency of a software also includes other metrics, e.g., instruction footprint. When using the area-optimised version of our ISE, the instruction footprint of a masked implementation can get significantly reduced compared to an ISA-based strategy, which is important for the deployment of masked implementation on memory-constrained devices, e.g., C0 and C1 devices defined in RFC 7228 [BEK14, Table 1].

## 6.3 Security

We use Coco [GHP+21, HB21] to evaluate the security (i.e., no leakage stemming from overwriting) of our ISE. In [GHP+21, Section 3], it states when using base Ibex core there are two constraints that a masked implementation should fulfill:

**Constraint 1.** Shares of the same secret must not be accessed within two successive instructions.

**Constraint 2.** A register or memory location which contains one share must not be overwritten with its counterparts.

**Architectural overwriting.** Recall that our ISE aims at eliminating the architectural and micro-architectural overwriting, which means, when using our secure instructions the constraint 2 should be no longer required. We then use Coco to perform the evaluation, where we label GPR[5] and GPR[12] to hold two *shares of the same secret* while we label GPR[6] and GPR[7] to hold *the static random values*. We use the following micro-benchmarks[9] to evaluate the class-1 instructions (using [sec.]xor as an example):

```
1 # base core
2 xor      x5,  x5, x7
3 and      x6,  x6, x6
4 xor      x12, x5, x7
5 # leakage captured
```

```
1 # latency-optimised
2 sec.xor  x5,  x5, x7
3 and      x6,  x6, x6
4 sec.xor  x12, x5, x7
5 # no leakage
```

```
1 # area-optimised
2 sec.xor  x5,  x5, x7
3 and      x6,  x6, x6
4 sec.xor  x12, x5, x7
5 # no leakage
```

Line 4 checks if there is a leakage when writing a share to a register that already contains another share of the same secret. This leakage is captured in the base core whereas it does not exist in the core extended with our ISE, which proves our class-1 instructions are secure in the sense of eliminating the architectural overwriting. We use the following micro-benchmarks to evaluate the store:

```
1 # base core
2 li       x13, 0x20
3 sw       x5,  0(x13)
4 and      x6,  x6, x6
5 sw       x12, 0(x13)
6 # leakage captured
```

```
1 # latency-optimised
2 li       x13, 0x20
3 sec.sw   x5,  x13, 0, 0
4 and      x6,  x6, x6
5 sec.sw   x12, x13, 0, 1
6 # no leakage
```

```
1 # area-optimised
2 li       x13, 0x20
3 sec.sw   x5,  x13, 0, 0
4 and      x6,  x6, x6
5 sec.sw   x12, x13, 0, 0
6 # no leakage
```

A leakage caused by overwriting in memory is expected in the base core since we write two shares to the same memory address, and it is successfully captured by Coco. When using our secure store instructions, this leakage does not exist, i.e., sec.sw eliminates this leakage as expected. Note that when evaluating the latency-optimised implementation, the initialisation of mask seed CSRs is already done before the micro-benchmark. The micro-benchmarks used for evaluating the load are shown as follows:

```
1 # base core
2 li       x13, 0x20
3 sw       x5,  0(x13)
4 and      x6,  x6, x6
5 lw       x12, 0(x13)
6 # leakage captured
```

```
1 # latency-optimised
2 li       x13, 0x20
3 sec.sw   x5,  x13, 0, 0
4 and      x6,  x6, x6
5 sec.lw   x12, x13, 0, 0
6 # no leakage
```

```
1 # area-optimised
2 li       x13, 0x20
3 sw       x5,  0(x13)
4 and      x6,  x6, x6
5 sec.lw   x12, x13, 0, 0
6 # no leakage
```

The same results are also obtained from the evaluation of load instructions, i.e., no leakage is captured in the micro-benchmarks of our secure load.

**Micro-architectural overwriting.** One thing must be noted is that there is no MBR in the Ibex core, which means the above security evaluation can only *straightforwardly* prove our ISE is capable to eliminate the architectural overwriting (i.e., in the register file and in the memory). As we mentioned before, the location of MBR can be intra-core or extra-core, and our ISE is designed to be capable to work in both cases. It is not trivial to directly perform the security evaluation in this situation, especially when extra-core MBR is used. However, it is possible and easy to conclude that (if it is correctly implemented) our sec.sw and sec.lw can eliminate the micro-architectural overwriting (i.e., in MBR) based on the above security evaluation for architectural overwriting; 1) in the case of latency-optimised implementation, what sec.sw and sec.lw act in MBR is completely the same as what sec.sw acts in the memory, i.e., overwriting with a re-masked share;

[9] For the case of latency-optimised ISE implementation, we make sure that the register indices do not get updated before execution of the micro-benchmarks.

Table 5: A somewhat quantitative, somewhat qualitative comparison versus Rosita [SSB+21] and FENL [GMPP20] (the two closest alternatives). Note that $+$, $-$, and $\approx$ suggest the comparison respectively positive, negative, and approximately equal versus Rosita or FENL.

| | | Security | Usability | Footprint | Latency | Area | Invasiveness |
|---|---|---|---|---|---|---|---|
| Base core + latency-optimised versus | Rosita | $+$ | $-$ | $+$ | $+$ | $-$ | $-$ |
| | FENL | $+$ | $-$ | $+$ | $+$ | $-$ | $+$ |
| Base core + area-optimised versus | Rosita | $+$ | $-$ | $+$ | $\approx$ | $-$ | $-$ |
| | FENL | $+$ | $\approx$ | $+$ | $\approx$ | $\approx$ | $+$ |

2) as for area-optimised implementation, `sec.sw` also acts the same in both MBR and the memory. What the last two steps (i.e., steps that interact with the load buffer) of `sec.lw` act in MBR is the same as what `sec.sw` acts in the memory. In other words, for both latency-optmised and area-optmised implementations, if there is no architectural overwriting leakage captured in the micro-benchmarks of `sec.sw`, then we can claim `sec.sw` and `sec.lw` can eliminate the micro-architectural overwriting.

## 6.4 Usability

The latency-optimised implementation demands a software developer correctly manages use of the mask seed CSRs to eliminate architectural and micro-architectural overwriting; in contrast, the area-optimised implementation does so transparently. This implies a clear difference in terms of their usability.

## 6.5 Comparison with related work

The two closest alternatives, and hence most natural comparison points, are 1) the ISA-based strategy provided by Rosita [SSB+21], and 2) the ISE-based strategy provided by FENL [GMPP20]; we focus on the most similar, zeroisation-based variant of FENL. Use of all three can be framed as rewriting instructions within an existing software implementation, with the goal of eliminating leakage. Rosita and FENL introduce additional instructions; eLIMInate replaces existing instructions (from ISA- to ISE-based, e.g., `xor` to `sec.xor`). FENL and eLIMInate use ISE-based instructions, so require support from hardware; Rosita uses ISA-based instructions, so requires no support from hardware. Note that Rosita, FENL, and eLIMInate are all largely agnostic to properties of the masking scheme or attacks on them. For example, Rosita++ [SCS+21] addresses the challenge of higher-order leakage elimination using the same set of rewrite rules as Rosita [SSB+21].

A direct comparison is difficult, because the ISAs, cores, and indeed stated remits differ. However, Table 5 attempts to offer a somewhat quantitative, somewhat qualitative summary that is derived from the analysis below:

- **Security.** The scope of Rosita addresses both architectural and micro-architectural leakage. As an ISA-based strategy, it uses indirect control of extra- and intra-core resources; per Section 3, doing so offers a weaker guarantee than, e.g., direct control. The scope of FENL addresses only micro-architectural leakage with any extra-core resources deemed out of scope. As an ISE-based strategy, it uses direct control of intra-core resources.
- **Usability.** A careful security analysis is required to identify where Rosita rewrite rules are applied. They can be described as local, in the sense they can be applied by using

"peephole-like" translation which has no global impact (and so does not require any global functional analysis). The difficulty of doing so is significantly reduced by the associated, automated tooling. A similar argument to that above for can be applied to FENL, in the sense one needs to analyse 1) how to configure and 2) where to place fence instructions. However, although it is plausible to use Rosita-like automation, a tool to do so for FENL currently does not exist. Application of eLIMInate is local for the area-optimised implementation, but is local (for class-1 instructions) and global (for class-2 instructions) for the latency-optimised implementation.

- **Footprint.** Both Rosita and FENL imply marginal overhead in memory footprint, since their application demands at least one additional instruction; all else being equal, the additional memory access required to fetch said instructions could plausibly contribute to greater energy consumption.
- **Latency.** For both Rosita and FENL, the global impact on execution latency depends where the mechanism is or is not applied, so we focus only on local instances where it is. Translating the ARM-based Rosita rewrite rules to RISC-V yields a similar outcome: as suggested by Section 3, this means a 2-cycle latency for class-1 instructions, a 6-cycle latency for the class-2 instruction `lw`, and a 4-cycle latency for the class-2 instruction `sw` when executed on the base core. For FENL, comparison is more difficult. For the case most similar to the base core, [GMPP20, Section 3.3.3] lists two options in which `fenl.fence` has 1) a 1 cycle (non-bubbling) or 2) a 1 or 4 cycle execution latency (bubbling: depending whether or not a pipeline stall is required to deliver the security guarantee). Using the former, this *suggests* a 2-cycle latency for class-1 instructions, a 3-cycle latency for the class-2 instruction `lw`, and a 3-cycle latency for the class-2 instruction `sw`. However, note that FENL deems extra-core resources such as SRAM out of scope; the comparison is only reasonable for class-1 instructions, therefore.
- **Area.** Rosita implies no overhead in hardware area. FENL implies modest overhead in hardware area: for the core most similar to Ibex, [GMPP20, Table 2] cites 0.7% additional flip-flops plus 1.0% additional LUTs.
- **Invasivness.** Rosita is an ISA-base strategy, so is not invasive. FENL is an ISE-base strategy, so is somewhat invasive: assuming existing instructions to manage CSRs, it adds 1 instruction and 1 CSR. Implementation of that instruction could be viewed as invasive, however, because it 1) has a global impact, potentially throughout the micro-architecture, and 2) intentionally exposes micro-architectural detail to software.

# 7 Conclusion

**Summary.** In this paper, we presented a functionally light-weight, leakage-focused ISE with the aim of supporting masked software implementation. By developing two concrete, prototype implementations of an underlying design concept, we demonstrate that use of the ISE can close the gap between assumptions about and actual behaviour of a device and thereby deliver an improved security guarantee.

In our view, it is important to stress that use of our ISE enables a subtle shift in how masked implementations can be developed. Currently, the starting point is a masked implementation consisting of instructions from the ISA this is functionally correct, insecure but efficient, implying a need to improve security (e.g., by identifying and eliminating micro-architectural leakage). Anecdotal evidence suggests that doing so is both conceptually difficult (and thus error-prone), and labour-intensive; the impact of failure can be catastrophic, in the sense it can render the implementation insecure. Now, the starting point is a masked implementation consisting of instructions from the ISE: this is functionally correct, secure but inefficient, implying a need to improve efficiency (e.g., by selectively replacing ISE instructions, with ISA alternatives). We claim that doing so is

conceptually easier, and the impact of failure is lessened; it aligns with a more general secure-by-default ethos.

**Future work.** Given the scope of this paper, and work presented within it, the following points seem to represent either useful or interesting future work:

1. Section 3 highlights an inherent limitation of the ISE, namely that extra-core resources require indirect control; improvement beyond this requires a change to the resource interface. For example, consider an SRAM module whose interface supports direct control via a "flush state" control signal: by removing the need for assumptions around indirect control, securing access to the SRAM can be more efficient and yield a more robust security guarantee. Realising such a systemic change is of course non-trivial, not least because of trade-offs between security and other metrics, but seems an important long-term goal.
2. Section 3 is clear about insufficiency of the ISE, in the sense that additional forms of micro-architectural leakage may also need to be considered. Doing so by extending the scope is somewhat open ended, but, for example, Section 4.2 includes `sec.slli` and `sec.srli` for left- and right-shift; it would be plausible to extend the variant semantics for these instructions to, e.g., address the observation by Gao et al. [GMPO19] that bit-interaction within a barrel shifter can produce leakage.
3. For the latency-optimised implementation, Section 6 highlights a challenge with respect to usability: a software developer must correctly manage use of the mask seed CSRs. Alongside generation of ISE-based instructions rather than their ISA-based instructions analogue, this aspect seems ripe for automation within an appropriate compilation tool-chain.

## Acknowledgements

This work has been supported in part by EPSRC via grant EP/R012288/1, under the RISE (http://www.ukrise.org) programme.

## References

[ARM18] ARMv6-M Architecture Reference Manual. Technical Report DDI-0419E, ARM Ltd., 2018. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0419e/index.html.

[ARM21] ARMv7-M Architecture Reference Manual. Technical Report DDI-0403E.e, ARM Ltd., 2021. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0403e.e/index.html.

[BDLU17] A. Biryukov, D. Dinu, Y. Le Corre, and A. Udovenko. Optimal first-order Boolean masking for embedded IoT devices. In *Smart Card Research and Advanced Applications (CARDIS)*, LNCS 10728, pages 22–41. Springer-Verlag, 2017. https://doi.org/10.1007/978-3-319-75208-2_2.

[BDP+12] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer. Keccak implementation overview. Technical report, 2012. https://keccak.team/files/Keccak-implementation-3.2.pdf.

[BEK14] C. Bormann, M. Ersue, and A. Keranen. Terminology for constrained-node networks. Internet Engineering Task Force (IETF) Request for Comments (RFC) 7228, 2014. http://tools.ietf.org/html/rfc7228.

[BWG+22] A. Beckers, L. Wouters, B. Gierlichs, B. Preneel, and I. Verbauwhede. Provable secure software masking in the real-world. In *Constructive Side-Channel Analysis and Secure Design (COSADE)*, LNCS 13211, pages 215–235. Springer-Verlag, 2022. https://doi.org/10.1007/978-3-030-99766-3_10.

[CKK+22] P. Choi, W. Kong, J.-H. Kim, M.-K. Lee, and D.K. Kim. Architectural supports for block ciphers in a RISC CPU core by instruction overloading. *IEEE Transactions on Computers*, 71(11):2844–2857, 2022. https://doi.org/10.1109/TC.2021.3050515.

[Cor09] Cortex-M0 Technical Reference Manual. Technical Report DDI-0432C, ARM Ltd., 2009. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0432c/index.html.

[DGH19] 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. https://doi.org/10.1145/3316781.3323485.

[EFLL20] M. Escouteloup, J.J.A. Fournier, J.-L. Lanet, and R. Lashermes. Recommendations for a radically secure ISA. In *Computer Architecture Research with RISC-V (CARRV)*, 2020. https://carrv.github.io/2020.

[GD23] J. Gaspoz and S. Dhooghe. Threshold implementations in software: Micro-architectural leakages in algorithms. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2023(2):155–179, 2023. https://doi.org/10.46586/tches.v2023.i2.155-179.

[GGM+21] S. Gao, J. Großschädl, B. Marshall, D. Page, T.H. Pham, and F. Regazzoni. An instruction set extension to support software-based masking. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2021(4):283–325, 2021. https://doi.org/10.46586/tches.v2021.i4.283-325.

[GHP+21] B. Gigerl, V. Hadzic, R. Primas, S. Mangard, and R. Bloem. Coco: Co-design and co-verification of masked software implementations on CPUs. In *USENIX Security Symposium*, pages 1469–1468, 2021. https://www.usenix.org/conference/usenixsecurity21/presentation/gigerl.

[GJM+17] H. Gross, M. Jelinek, S. Mangard, T. Unterluggauer, and M. Werner. Concealing secrets in embedded processors designs. In *Smart Card Research and Advanced Applications (CARDIS)*, LNCS 10146, pages 89–104. Springer-Verlag, 2017. https://doi.org/10.1007/978-3-319-54669-8_6.

[GMPO19] S. Gao, B. Marshall, D. Page, and E. Oswald. Share slicing: friend or foe? *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2020(1):152–174, 2019. https://doi.org/10.13154/tches.v2020.i1.152-174.

[GMPP20] S. Gao, B. Marshall, D. Page, and T.H. Pham. FENL: an ISE to mitigate analogue micro-architectural leakage. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2020(2):73–98, 2020. https://doi.org/10.13154/tches.v2020.i2.73-98.

[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)*, pages 1:1–1:9, 2018. https://doi.org/10.1145/3265723.3265724.

[HB21] V. Hadzic and R. Bloem. COCOALMA: A versatile masking verifier. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 1–10. IEEE, 2021. https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_9.

[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. https://doi.org/10.1007/3-540-48405-1_25.

[KS20] P. Kiaei and P. Schaumont. Domain-oriented masked instruction set architecture for RISC-V. Cryptology ePrint Archive, Report 2020/465, 2020. https://eprint.iacr.org/2020/465.

[LHP20] T. Li, B. Hopkins, and S. Parameswaran. SIMF: Single-instruction multiple-flush mechanism for processor temporal isolation. *CoRR*, abs/2011.10249, 2020. https://arxiv.org/abs/2011.10249.

[MOP07] S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007. https://doi.org/10.1007/978-0-387-38162-6.

[MP21] B. Marshall and D. Page. SME: Scalable masking extensions. Cryptology ePrint Archive, Report 2021/1416, 2021. https://eprint.iacr.org/2021/1416.

[MPW22] B. Marshall, D. Page, and J. Webb. MIRACLE: MIcRo-ArChitectural Leakage Evaluation. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2022(1):175–220, 2022. https://doi.org/10.46586/tches.v2022.i1.175-220.

[MR04] S. Micali and L. Reyzin. Physically observable cryptography. In *Theory of Cryptography (TCC)*, LNCS 2951, pages 278–296. Springer-Verlag, 2004. https://doi.org/10.1007/978-3-540-24638-1_16.

[PV17] K. Papagiannopoulos and N. Veshchikov. Mind the gap: Towards secure 1st-order masking in software. In *Constructive Side-Channel Analysis and Secure Design (COSADE)*, LNCS 10348, pages 282–297. Springer-Verlag, 2017. https://doi.org/10.1007/978-3-319-64647-3_17.

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

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

[RV:22] RISC-V cryptographic extension proposals. Technical Report Volume I: Scalar & Entropy Source Instructions (version 1.0.1), 2022. https://github.com/riscv/riscv-crypto.

[SCS+21] M.A. Shelton, L. Chmielewski, N. Samwel, M. Wagner, L. Batina, and Y. Yarom. Rosita++: Automatic higher-order leakage elimination from cryptographic code. In *Computer and Communications Security (CCS)*, pages 685–699, 2021. https://doi.org/10.1145/3460120.3485380.

[SS22] K. Stangherlin and M. Sachdev. Design and implementation of a secure RISC-V microprocessor. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 30(11):1705–1715, 2022. https://doi.org/10.1109/TVLSI.2022.3203307.

[SSB+21] M.A. Shelton, N. Samwel, L. Batina, F. Regazzoni, M. Wagner, and Y. Yarom. Rosita: Towards automatic elimination of power-analysis leakage in ciphers. In *Network and Distributed System Security Symposium (NDSS)*, 2021. https://doi.org/10.14722/ndss.2021.23137.

[TKS10] S. Tillich, M. Kirschbaum, and A. Szekely. SCA-resistant embedded processors: The next generation. In *Annual Computer Security Applications Conference (ACSAC)*, pages 211–220, 2010. https://doi.org/10.1145/1920261.1920293.

[Wat16] A. Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, University of California at Berkeley, 2016. https://people.eecs.berkeley.edu/~krste/papers/EECS-2016-1.pdf.

[WMSN19] R.N.M. Watson, S.W. Moore, P. Sewell, and P.G. Neumann. An introduction to CHERI. Technical Report UCAM-CL-TR-941, University of Cambridge, 2019. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-941.pdf.

[WSG+20] N. Wistoff, M. Schneider, F.K. Gürkaynak, L. Benini, and G. Heiser. Prevention of microarchitectural covert channels on an open-source 64-bit RISC-V core. In *Computer Architecture Research with RISC-V (CARRV)*, 2020. https://carrv.github.io/2020.

[X8622] Intel 64 and IA-32 architectures – software developer's manual (volume 2: Instruction set reference a-z). Technical Report 325383-078US, Intel Corp., 2022. http://software.intel.com/en-us/articles/intel-sdm.