An Instruction Set Extension to Support Software-Based Masking

Johann Großschädl[1], Ben Marshall[2], Dan Page[2], Thinh Pham[2] and Francesco Regazzoni[3,4]

[1] Department of Computer Science, University of Luxembourg.
johann.groszschaedl@uni.lu

[2] Department of Computer Science, University of Bristol.
{ben.marshall,daniel.page,th.pham}@bristol.ac.uk

[3] University of Amsterdam,
f.regazzoni@uva.nl

[4] ALaRI, University of Lugano.
regazzoni@alari.ch

Abstract. In both hardware *and* software, masking can represent an effective means of hardening an implementation against side-channel attacks such as Differential Power Analysis (DPA). Focusing on software, however, the use of masking can present various challenges: specifically, it often 1) requires significant effort to translate any theoretical security properties into practice, and, even then, 2) imposes a significant overhead in terms of efficiency. To address both challenges, this paper explores use of an Instruction Set Extension (ISE) as a means of supporting masking in software-based implementations of symmetric cryptographic algorithms: we design, implement, and evaluate such an ISE using RISC-V as the base architecture.

Keywords: side-channel attack, masking, ISE

1 Introduction

The threat of implementation attacks. Evolution of the technology landscape, for example improvement in storage, computational, and communication capability, has produced a corresponding evolution in user-facing platforms and applications that we now routinely depend on. Many such cases are now deemed security-critical, as a result of trends such increased connectivity (cf. IoT), outsourced computation (cf. cloud computing), and use of identity-, location-, and finance-related data. Within this setting, cryptography often represents a transparent enabler: cryptographic solutions are routinely tasked with ensuring the secrecy, robustness, and provenience of our data (when communicated and/or while stored), plus the authenticity of interacting parties. While robust theoretical foundations often underpin such solutions, their realisation in practice can remain difficult. For example, per the remit above, cryptographic implementations represent an important component of the attack surface; within an attack landscape of increasing breadth and complexity (where "attacks only get better"), the threat of so-called implementation attacks is particularly acute.

The premise of an implementation attack is that by considering a concrete implementation, vs. an abstract specification say, theoretical security properties (however strong) can potentially be bypassed. At a high level, they are often divided into active (e.g., fault injection) or passive (i.e., side-channel) classes. Differential Power Analysis

(DPA) [KJJ99, MOP07] is a concrete example[1] of a side-channel attack with particular relevance to embedded devices. Following an optional profiling (or characterisation) phase, a typical DPA attack performs an initial, online acquisition phase: (passive) monitoring by the attacker yields traces of power consumption during computation of some target operation by the target device. The underlying assumption is that *both* a) operations (e.g., addition vs. multiplication), *and* b) the operands they process (e.g., higher vs. lower Hamming weight) contribute to features, or leak information, then evident in the traces. Such features are harnessed by a subsequent, offline analysis phase, which attempts to recover security-critical information (e.g., key material) they relate to.

Challenges in realisation of countermeasures. Techniques for mitigating implementation attacks are becoming increasingly well understood. At a high level, examples pertinent to DPA are typically classified as being based on hiding [MOP07, Chapter 7] and/or masking [MOP07, Chapter 10]. The latter, which is our focus, can be described as a low-level realisation of the more typically higher level "computing on encrypted data" concept. For a target operation normally invoked as $r = f(x)$, application of a given masking scheme demands that 1) x is encrypted (resp. masked) to yield $\tilde{x}$, 2) alternative computation is applied to $\tilde{x}$, i.e., $\tilde{r} = \tilde{f}(\tilde{x})$, st. it acts on the *underlying* x in a manner compatible with f, then 3) $\tilde{r}$ is decrypted (resp. unmasked) to yield r; any leakage stemming from the computation of $\tilde{f}$ will now relate to $\tilde{x}$ rather than x, so the latter cannot be directly recovered as would likely be the case using f.

In common with other countermeasure techniques, masking can be utilised at various levels in either hardware and/or software: for example, algorithm-level (e.g., to a block cipher such as AES [Mes01]), system-level (e.g., across the data-path of a processor core [GJM+16, MGH19]), and gate-level (e.g., in secure logic style such as MDPL [PM05]) techniques are all viable. For a concrete implementation that uses such techniques, however, at least two significant challenges must be addressed. First, it must translate theoretically modelled security properties into practice. This challenge is neatly illustrated by the contrast between a theoretically, *provably* secure masking scheme proposed by Rivain and Prouff [RP10], vs. attacks on a practical implementation thereof by Balasch et al. [BGG+14]. Second, it must do so while satisfying other quality metrics such as demand for high-volume, low-latency, high-throughput, low-footprint, and/or low-power.

An ISE-assisted approach to masking. Instruction Set Extensions (ISEs) [GB11, BGM09, RI16] have proved an effective implementation technique within the context of cryptography. The idea is to identify, e.g., through benchmarking, a set of additional instructions that allow the target operation to leverage special-purpose, domain-specific functionality in the resulting ISE, vs. general-purpose functionality in the base Instruction Set Architecture (ISA), and thereby deliver improvement wrt. pertinent quality metrics. ISEs are *particularly* effective for embedded devices, because they afford a compromise improving footprint and latency vs. a software-only option while also improving area and flexibility vs. a hardware-only option.

There is an increasingly accepted argument (see, e.g., [RKL+04, RRKH04, BMT16]) that security should be considered as a first-class metric at design-time, rather than a problem to be addressed in a reactive, post hoc manner. In line with such an argument, this paper explores use of an ISE as a means of supporting masking in software-based implementations of cryptography: we design, implement, and evaluate such an ISE using RISC-V as the base ISA. We suggest there are (at least) three reasons an ISE-based approach may be attractive vs. alternatives (e.g., a dedicated IP module). First, use of masking in software-only implementations will impose a significant overhead, e.g., wrt. a)

[1] Although our focus is specifically on DPA, we note that associated attack and countermeasure techniques apply more generally, e.g., to classes such as EM.

execution latency and b) demand for high-quality randomness; our ISE can help mitigate this problem. Second, an ISE is well positioned to act as an interface wrt. security properties. For example, there is increased evidence (see, e.g., [CGD18, GMPO19]) that secure use of masking in software-only implementations is complicated by the lack of guarantees wrt. leakage stemming from the underlying micro-architecture; our ISE can help mitigate this problem, e.g., by adopting an approach similar to the augmented ISA (or aISA) of Ge et. al [GYH18] and constraining the micro-architecture to meet properties demanded by the ISA. Third, the design of masking schemes is a relatively fast-paced field, with novel designs and techniques appearing regularly. Our ISE mitigates this problem by following a RISC-like ethos: it provides a suite of general-purpose "building block" operations, that can be used to support a wide range of cryptographic constructions (e.g., block ciphers) and higher-level masking schemes.

We note that concurrently with our work, Kiaei and Schaumont [KS20] published a similar proposal: vs. their work, we a) enrich the ISE with a wider set of operations, b) provide an implementation of the ISE within an existing RISC-V compliant micro-architecture, and c) evaluate it, wrt. efficiency and security properties, using a suite of representative kernels.

Organisation. Section 2 surveys related work. Section 3 then presents 1) the ISE design, and 2) an implementation of said design in a RISC-V compliant micro-architecture. The associated efficiency and security properties are then evaluated in Section 4, via a suite of representative software implementations. Finally, Section 5 presents some conclusions and potential directions for future work.

2 Background

2.1 RISC-V

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 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.

We focus wlog. on extending RV32I [RV:19, Section 2], i.e., the 32-bit integer RISC-V base ISA. Let $\mathsf{GPR}[i]$, for $0 \leq i < 32$, denote the i-th entry of the general-purpose register file. RISC-V uses XLEN to denote the word size; we adopt the same approach, but by focusing on RV32I assume a focus on XLEN $= 32$.

2.2 Masking

Masking is a protection technique based on secret-sharing schemes. The key idea is to split the message into parts called shares. The reconstruction of the message become then possible only if a sufficient number of shares is known. In this direction, Chari et al [CJRR99], propose to split the data of the original computation into k shares having two properties: the shares have to be equiprobably distributed and every subset of $k-1$ shares have to be statistically independent from the encoded data, and then perform the computation on the shares. The word masking applied appeared for the first time in 2000 [Mes00], where Messerges described the use of a "random mask to obscure the calculation made by the fundamental operations"of the AES candidates. Thanks to the

mask, which has to be random, the recovery of the secret key using the power consumption should be more difficult.

A given masking scheme specifies a) a non-standard *representation* of data, where each variable x is represented by (or split into) n separate shares, and b) a non-standard *implementation* of functions, which operate on the representations. Use of the scheme *should* resist a t-th order attack (e.g., under the probing model of Ishai, Sahai, and Wagner [ISW03]), where information from at most $t < n$ shares is combined.

2.2.1 Representation

Under Boolean masking this means $x = x_0 \oplus x_1 \oplus \cdots \oplus x_{n-1}$, whereas under arithmetic masking this means $x = x_0 + x_1 + \cdots + x_{n-1} \pmod{2^w}$. Consider the specific case of $n = 2$, and let $\hat{x} = (x_0, x_1)$ denote the representation of some x under Boolean masking, i.e., as two shares x_0 and x_1: this demands that $x = x_0 \oplus x_1$.

2.2.2 Hardware-oriented implementation

Masking is a generic approach for mitigating power analysis attacks, and as such it can be applied to several classes of algorithms (including block ciphers [], but also post quantum algorithms []). Furthermore it can be applied in hardware, in software, or in a combination of both. In this section we will revise the main approaches used in the past for implementing masking in hardware and in software.

Classical Goubin and Patarin [GP99] formalized the idea of replacing each intermediate variable of the computation that is dependent of the inputs or outputs, and thus potentially being used by the attacker to guess the secret key, and depending on the inputs (or the outputs), by a combination of sub-variables, that would allow to retrieve the original variable if combined together. This is secure if the function selected for implementing the combination operation is such that allows to perform the transformation of the algorithm on the sub-variables, without calculating the original variable. The two functions analyzed are the bit-by-bit exclusive or function (which was later called additive masking) and the multiplication mod "n" (multiplicative masking).

Threshold Implementation (TI) Threshold Implementation (TI), presented by Nikova et al.[NRR06], is a countermeasure that is provable secure against first order attacks (also when the circuits implementing the schema cause glitches. TI requires to use shares having three fundamental properties: correctness, incompleteness and uniformity. Correctness means that the computation carried out on the shares, should be correct, namely that composition of the results of the operations carried out on each shares has to be equal to the shared representation of the original result. incompleteness means that each share should be independent from at lest one input share. The security of the schema requires that the masks are uniformly distributed. Uniformity is usually the most difficult property to guarantee, by this can be relaxed by using functions that do not always satisfy the uniformity but their randomness is refreshed frequently.

Domain-Oriented Masking (DOM) Domain-Oriented Masking (DOM) has been presented by Gross et al. [GMK16]. The main objective of the work was to guarantee security against i-th order attack using i+1 shares, thus reaching the same level of security of TI, but incurring in less area (when implemented in hardware) and less randomness. To achieve this, the authors concentrate their effort in the design of the DOM-dep multiplier, that is a dedicated masked multiplier suitable to implementing the proposed schema in an efficient and secure way. The approach is evaluated using the AES algorithm as a case

of study, which is analyzed in depth presenting several variants of protection up to 15 protection order.

2.2.3 Software-oriented implementation

The main challenge when applying masking in software is to implement the round functions of a block cipher in such a way that the shares can be processed independently from each other, while it still must be possible to recombine them at the end of the execution to get the correct result. This is fairly easy for all linear operations, but can introduce massive overheads for the non-linear parts of a cipher, e.g. S-boxes or modular additions/subtractions. Furthermore, all round transformations need to be executed twice (namely for x_1 and for x_2, where $x = x_1 \oplus x_2$), which entails an extra performance penalty. Another problem is that a basic masking scheme with two shares is vulnerable to a so-called second-order DPA attack where an attacker combines information from two leakage points. Such a second-order DPA attack can, in turn, be thwarted by second-order masking, in which each sensitive variable is concealed with two random masks and, consequently, represented by three shares.

Depending on the algorithmic properties of a cipher, a masking scheme can have to protect Boolean operations (e.g. XOR, shift) or arithmetic operations (e.g. modular addition). When a cipher involves both Boolean and arithmetic operations, it is necessary to convert the masks from one form to the other to obtain the correct ciphertext (or plaintext). Examples of symmetric algorithms that involve arithmetic as well as Boolean operations include the widely-used hash functions SHA-2, Blake and Skein, and any ARX-based block cipher (e.g. Speck). In essence, the basic operations performed by common block ciphers can be divided into three categories depending on how costly they are to mask in software: (i) linear operations (e.g. XOR, NOT, shift, rotation), (ii) non-linear Boolean operations (e.g. and, or), and (iii) non-linear arithmetic operations (e.g. modular addition, inversion in GF(2^8)).

As mentioned before, linear operations like XOR and rotation are fairly easy to mask in software since one just has to apply the operation to each pair of shares individually. The XOR of a constant to a set of shares can be performed by XORing it to a single share. Similarly, the logical NOT operation is masked by applying NOT to one of the shares. Computing a non-linear Boolean function on the shares assuring all variables processed are independent of sensitive variables is more complicated and introduces higher computational overheads. The simplest non-linear Boolean operations is the logical AND, which can be masked in different ways, whereby the different approaches proposed in the literature differ by the amount or randomness and the number of underlying basic operations. The first proposal for a first-order masked AND gate came from Trichina and was published more than 15 years ago [Tri03]. This so-called "Trichina AND-gate" consists of four basic AND operations, four XORs, and requires additional fresh randomness to ensure that the shares are statistically independent of any sensitive variable. Biryukov et al introduced in [BDCU17] an improved expression for masked AND that consists of only seven basic operations does not require an additional random variable since the shares are inherently refreshed. Furthermore, on ARM microcontrollers, the masked AND can be performed using only six basic instructions. Biryukov et al also presented a masked OR operation, which consists of only six basic operations (and six basic instructions on ARM) and does not require fresh randomness.

Highly non-linear arithmetic operations, such as modular addition or inversion in a binary field, are the most costly operations when it comes to masking in software. There are two basic options for implementing a masked addition (or subtraction) in software; the first consists of converting the Boolean shares to Arithmetic shares, then performing the addition on the arithmetic shares, and finally converting the arithmetic shares of the sum back to Boolean shares. The second option is to perform the modular addition

directly on Boolean shares without conversion. Both options have in common that a straightforward software implementation has a complexity that increases linearly with the length of the operands to be added. Coron et al presented in [CGTV15] a recursive formula for arithmetic addition on Boolean shares with logarithmic complexity. This approach is based on the Kogge-Stone adder (a special variant of a carry-lookahead adder) and uses masked AND, masked XOR, and masked shift as sub-operations. Biryukov et al presented an improved Kogge-Stone adder that uses the more efficient masked operations from [BDCU17] and is able to perform a 32-bit addition on Boolean shares between 14% and 19% faster than the Kogge-Stone adder of Coron et al.

2.3 Related work

As mentioned, the masking countermeasures can be applied at hardware level, at software level (or a combination of both) or can be applied at different granularities (at the level of whole core or at minimal portion of the accelerator, while still guaranteeing the security. This is also valid when the approach is applied at the level of CPUs. In that context, in fact, it is possible to modify complete block of the CPU (such as ALU or similarly large components), or the modifications can be applied at much lower level, protecting only small registers and limited portions of the datapath.

Concerning the level of whole core within CPUs, the topic was widely explored in open literature, and a number of masked accelerators have been proposed, designed, implemented and evaluated in the last years. We focus here on the one related with RISC-V architecture.

Gross et al [GJM+16] propose a SCA-protected processor design based on the open-source V-scale RISC-V processor. The starting point is the experience gained with the study of domain oriented masking, which is leveraged to modify the CPU to make it resistant against side channel attacks. To achieve this, the authors split the processor in a protected and an unprotected part, and they propose to realize a protected ALU, which implements basic operations protected using the domain oriented approach. Experimental results show an increased resistance against side channel attacks and a scale of the system almost linear with the order of protection.

Protection against power analysis attacks for the RISC-V processor have been also proposed by De Mulder et al[MGH19]. With the goal of simplify the implementation of software resistant against power analysis attacks, the authors proposed to integrate side channel countermeasures into the RISC-V core. The proposed solution aims at protect against first-order power and electromagnetic attacks. The protection is achieved using a combination of known masking techniques and a masked access to memory. The second mask for access to memory is generated on the fly within the boundary of the CPU, and thus, at least in principle, robust. The leakage reduction is demonstrated by a number of practical experiments.

The most relevant work related to our concerns is probably the one of Kiaei and Schaumont [KS20]. They propose to extend the RISC-V processor with dedicated instructions to mitigate side channel attacks, focusing in particular on Domain-Oriented-Masking. Our paper shares the core idea of extending the instruction set of a processor to achieve side channel resistance, but provides novel contributions. Firstly, our instructions are not limited to the case of Domain-Oriented-Masking, but are suitable for implementing masking countermeasures in general and are suitable to protect a wide range of algorithms. Secondly, our instructions are integrated in the core providing an quantitative analysis of the achieved robustness and of the performance overhead. Lastly, we show that to achieve security of masking using dedicated instructions, there is no need to have a duplication of the datapath for achieving strong separation between secure and insecure zone. To the best or our knowledge, previous works on instruction set extension for accelerating masking and for side channel security in general, have always proposed to have such strong differentiation.

3 Implementation

3.1 ISE design

Concept. Focusing wlog. on use of Boolean masking, the ISE targets inclusion of instructions to support

1. binary masked operations, i.e., $\hat{r} = \hat{x} \ominus \hat{y}$ for some set of $\ominus$,
2. unary masked operations, i.e., $\hat{r} = \oslash \hat{x}$ for some set of $\oslash$, and
3. various auxiliary operations, such as conversion into, from, and between masked representations.

The set of supported operations should be general-purpose in the sense they are useful for a range of cryptographic constructions and masking schemes; they often have an equivalent in, and so represent close to a "drop in" replacement for instructions in the base ISA by including, e.g., $\ominus \in \{\wedge, \vee, \oplus\}$ and $\oslash \in \{\neg\}$ to mirror the unmasked Boolean operations already available. Doing so is complicated, however, by the fact that for $n = 2$ shares we have

$$\hat{r} = \hat{x} \ominus \hat{y} \implies (r_0, r_1) = (x_0, x_1) \ominus (y_0, y_1).$$

That is, doing so increases the number of register indexes required, and, therefore, pressure on instruction encoding: an unmasked binary (resp. unary) operation requires 3 (resp. 2) register indexes, whereas a masked equivalent requires 6 (resp. 4). The same scenario is articulated by Lee et al. [LYS04], who describe use the term Multi-word Operand, Multi-word Result (MOMR) to characterise and thereby distinguish cryptographic operations from the general case. There are various ways to satisfy this requirement: we use an implied approach, where two indexes are encoded as one, i.e., $(i, i+1) \mapsto i$. For example, the even-odd index pair $(2, 3)$ is encoded as the first, even index 2; the second, odd index 3 is then implicit rather than explicit. This is an limited instance of the Register File Extension for Multi-word and Long-word Operation (RFEMLO) approach proposed by Lee and Choi [LC08].

Design. We defer a complete description of the instruction semantics and encoding to Appendix A and Appendix B respectively; the former is specified in terms of a common set of atomic operations, described algorithmically in Appendix C to avoid repetition inline.

In short, however, the ISE includes a suite of instructions that support Boolean masking. They allow masking, unmasking, remasking, and application of operations to (masked) operands: these operations include NOT, AND, OR, XOR, left- and right-shift, and (left-)rotation, addition and subtraction. For example, the instruction

```
mask.b.add (rd1,rd2), (rs1,rs2), (rs3,rs4)
```

uses the inputs $\hat{x} = (x_0, x_1) = (\mathsf{GPR}[\texttt{rs1}], \mathsf{GPR}[\texttt{rs2}])$ and $\hat{y} = (y_0, y_1) = (\mathsf{GPR}[\texttt{rs3}], \mathsf{GPR}[\texttt{rs4}])$ st. $x = x_0 \oplus x_1$ and $y = y_0 \oplus y_1$; it computes $\hat{r} = (r_0, r_1) = (\mathsf{GPR}[\texttt{rd1}], \mathsf{GPR}[\texttt{rd2}])$ st. $r = r_0 \oplus r_1 = x + y$.

3.2 ISE implementation

The ISE operations are executed by a masked ALU. The masked ALU gets two 2-share-masked operands $(rs1_s0, rs1_s1)$ and $(rs2_s0, rs2_s1)$ to produce one 2-share-masked output (rd_s0, rd_s1). The masked ALU is designed with a compact architecture in which the functional submodules of the ISEs are shared between the instructions as much as possible to reduce area overhead. A block diagram of the masked ALU is

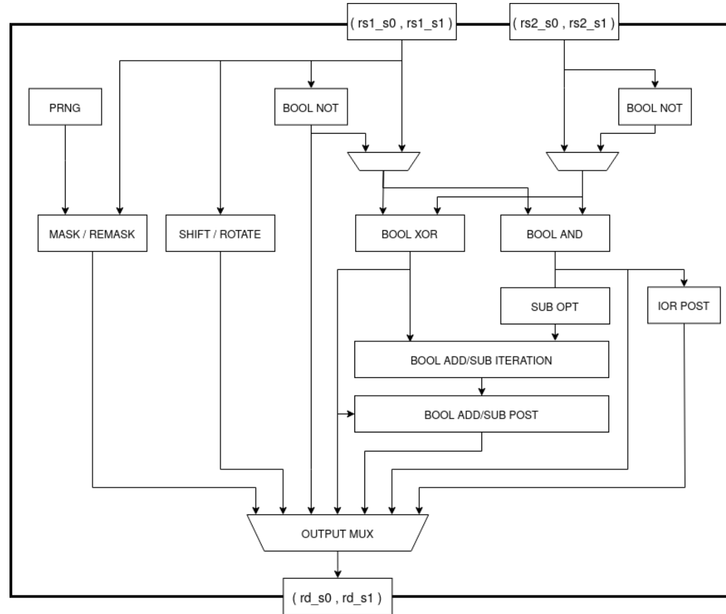


Figure 1: Masked ALU architecture

shown in Figure 1. The PRNG generates pseudo random numbers for masking, re-masking operations. The MASK/REMASK masks and re-masks the operands $rs1_s0$ and $(rs1_s0, rs1_s1)$, respectively. The SHIFT/ROTATE performs left-shift, right-shift, and right-rotate operations on the Boolean masked operand $(rs1_s0, rs1_s1)$. The BOOL XOR and the BOOL AND execute `XOR` and `AND` operations under Boolean masking, respectively. `mask.b.add` and `mask.b.sub` instructions are executed by a shared Add-Sub operation consisting of three steps (i.e., pre-processing, iteration, and post-processing) based on the Kogge-Stone adder's algorithm. These instructions share the BOOL XOR and the BOOL AND with `mask.b.xor` and `mask.b.and` instructions to perform the pre-processing step of the Add-Sub operation. The BOOL ADD/SUB ITERATION and the BOOL ADD/SUB POST perform iterative and post-processing steps, respectively, of the Add-Sub operation. The SUB OPT controls the Add-Sub operation to perform the `mask.b.sub` instruction. Boolean masking `OR` operation is executed by reusing the BOOL AND. The IOR POST calculates the output of the `OR` operation from the return values of the `AND` operation according to the De Morgan's law. All algorithms of the instructions are listed in Appendix C.

3.3 ISE integration

The masked ALU described in Section 3.2 was integrated into the SCARV[2] core, a 5-stage pipelined micro-architecture implementing the RISC-V `rv32i` base instruction set, with the **m**ultiply/divide and **c**ompressed standard extensions. A block diagram of the core is show in Figure 2, with masking ISE related blocks hilighted.

The masking ISE adds reading and writing of adjacent odd/even pairs of general purpose registers to the ISA. This nessesitated a re-structuring of the register file read ports, and a single additional 32-bit pipeline register (`s2_opr_d`) to hold all of the masked ALU operands.

Share 0 (resp. 1) of a variable is stored in the even (resp. odd) register. Share 0 (resp. 1) of `rs1` is stored in `s2_opr_a` (resp. `s2_opr_c`). Likewise, Share 0 (resp. 1) of `rs2` is

[2] https://github.com/scarv/scarv-cpu

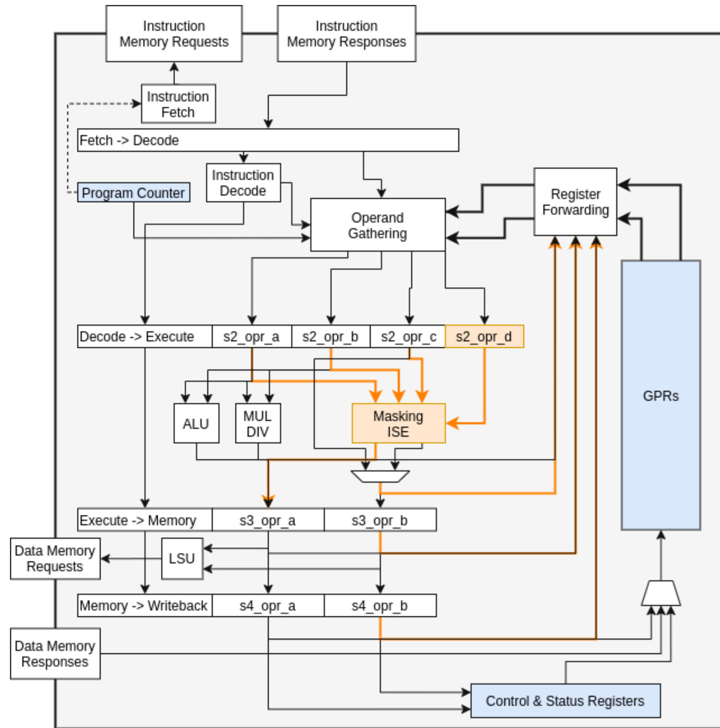


Figure 2: CPU Micro-architecture. Masking ISE components are hilighted in yellow.

stored in `s2_opr_b` (resp. `s2_opr_d`). For base ISA instructions, `rs1` (resp. `rs2`) is stored in `s2_opr_a` (resp. `s2_opr_b`). This makes the masked ISE instruction operands partially resistant to accidental un-masking, since both shares of the same variable cannot enter other functional units in the execute stage.

The odd/even paired nature of the reads meant that the register file could be split into odd and even groups of registers, with each pair element read in parallel. For register-pair reads, the values from each 16-to-1 mux tree is selected. For single-register reads, an additional multiplexer in the decode stage selects between either the odd or even register based on the least significant bit of the register address. This structuring of the mux trees helps avoid the nearest neighbour effect [PV17], which could result in the accidental un-masking of shares stored in adjacent registers. The final selection between an odd/even register (i.e. shares of a variable) is done at the very last mux-tree stage, which reduces the chance of select signal glitches causing the muxes to switch from one share to the other[3].

Further opportunities for accidentally combining shares occur throughout the execution pipeline; particularly in the register forwarding network which must be capable of switching between shares stored in pipeline registers. This leads some to suggest that entirely separate datapaths (and the attendant increase in resources) are needed for masked v.s. base instrucitons [KS20].

We mitigate this problem by transparently storing share 1 of every masked ISE instruction source and result in a bit reversed representaiton, both in the register file, and in pipeline registers. Hence any switching between shares (in the forwarding network, for example) will cause non-corresponding bits of the shares to interract. I.e. gliches would cause a transition from bit 0 of share 0 to bit 31 of share 1, rather than to bit 0 of share 1. This bit reversal is transparent to the software developer, and shares which are read

[3] This technique does not solve the problem entirely; developers may deliberately read first share 0, then share 1. Careful ordering of instructions and register selection is still nessesary.

by non-masking ISE instructions (e.g. loads and stores) are automatically un-reversed. Un-reversal of operands is performed at the last possible moment before any computation. Non-masking ISE instructions have operands un-reversed immediately before the they enter the `s2_opr_a` and `s2_opr_b` pipeline registers. Masking ISE instructions only un-reverse their instructions in the execute stage, immediately before entering the masked ALU. Share 1 is immediately reversed on exiting the masked ALU, before entering the mux tree to decide the next values of the `s3_opr_b` pipeline register.

By utilising this bit-reversed representation, and tracking the reversed-ness of operands across masking ISE and base instrucitons, we were able to share the CPU datapath and save significant resources. The alternative would have required adding two extra 32-bit registers to the `s3` and `s4` pipeline stages, and four extra 32-bit registers to the `s2` pipeline stage. In contrast, the bit-reversal scheme requires 16-bits of register storage in the GPRs, one extra register bit per `s[2,3,4]` pipeline stage, and 1 32-bit multiplexer per `s2_opr_*` pipeline register. This is only for the CPU datapath; it does not include duplicating architectural state (the GPRs) or mechanisms for accessing that state.

3.4 ISE verification

Here we describe how we verified the correctness and security properties of the masking ISE. We then discuss some wider considerations for verifying ISE designs which claim security properties in the context of power side-channels. Verifying the masking ISE was split into two steps: functional correctness and security properties.

Functional correctness captures whether instructions behave as they are specified. There are two ways to check this: 1) given known inputs, expect these exact outputs. 2) given known inputs, this relationship between the outputs must hold. Option 1 gives the best assurance that the instructions are doing exactly what is specified, and enables easy co-simulation with a golden reference model[4] for the CPU, or specification in a formal model.

The masking ISE instructions can also be sources of randomness however. While the relationships between the two output shares is well defined, the exact output values are not, because they involve mixing random values into the outputs. Co-simulation based verification methods hence require hints from the hardware to produce correct results and be kept in sync with the design under test. This becomes complex to maintain in even simple designs.

For this paper, we used formal verification methods[5] to ensure that the relationships between input and output shares was always correct, without needing to know the exact values. The baseline SCARV core was already formally verified using the `riscv-formal`[6] framework. We extended the framework to support new double-width register access idioms. This enabled verification of the base ISA instrucitons, the masking ISE instructions, and their interractions, since we include checks for register read/write consistency.

Functional verification is primarily done on Register Transfer Level (RTL) designs, with no modelling of gate delays or net capacitances. This is possible because synthesis and layout tools must always preserve the functionality of a design, while optimising for power, performance and area. Security properties, particularly those dealing with physical aspects of the manufactured device (power, EM emissions) cannot be verified in the same way, and come with specific challenges.

There are clear trade-offs between speed and accuracy when evaluating side-channel vulnerablity at different levels of abstraction between RTL and post layout or even implemented systems. While some side-channel evaluation can be done at the RTL to provide early feedback to designers [CS09, HPN+19, ZBPF18] (particularly as to which

[4] Such as QEMU or OVPSim.
[5] Specifically, Bounded Model Checking (BMC).
[6] https://github.com/SymbioticEDA/riscv-formal/

modules are causing leakage), this is not sufficient to be sure that the design will be secure at the post-layout level. Indeed, synthesis and layout tools can undermine or even reverse some side-channel countermeasures implemented at the RTL [GBR+19]. In [BDG+13], the authors assert that digital simulation of a post layout design is sufficient for effective side-channel analysis, though we note this was for a hardware implementation of the PRESENT block cipher, not a general purpose CPU. Likewise in [HPN+19], the authors describe an RTL leakage evaluation methodology targeting the AES block ciper.

Because our design targets an FPGA platform, we were able to perform evaluations of the complete design very quickly, and make changes very easily based on leakage we found. We recognise that this would not be possible in an ASIC, or design IP style commercial project, and discuss extra verification steps later.

We created kernels which executed each masking ISE instruction in isolation, surrounded by `nops`. We then evaluated each kernel for leakage using a T-Test, and hamming weight based corrolation analysis on the un-masked inputs and outputs. These baseline tests helped us gain confidence than masking ISE instructions *in isolation* would not leak. For comparison, we executed the same set of kernels on the equivalent baseline ISA instructions. The results of these evaluations and comparisons are listed in Section 4.2.2.

For our purposes, this was enough confidence to begin evaluating entire ciphers. A more rigorous verificaton exercise would need to look at interractions between adjacent and non-adjacent (for pipeline forwarding) masking ISE instructions, and forwarding of values between masking and non-masking instructions. We note that the verification effort needed to ensure side-channel security proeprties for a general purpose CPU is orders of magnitude higher than for a dedicated cryptography module (e.g. an AES or SHA accelerator). This stems from the number of individual functions (instructions) present in a CPU design, and how those functions interract in an execution pipeline.

Ultimately, functional correctness of a single code sequence can usually be verified with a single execution of that code sequence. Verification complexity then increases with the number of code sequences which need to be covered. Side-channel security however must be evaluated statistically, requiring many thousands of executions of the same code path to build any level of confidence. This may make verifying all side-channel security property relevant interractions between instructions infeasible in even a shallow CPU pipeline with reasonable engineering budgets. Note that the verification complexity would not nessesarily be aided by complete separation of datapaths, since it is interractions between masking and non-masking ISE instructions which form the bulk of the verification problem space. Nonetheless this may be interpreted as an implicit argument for separation of masked cryptographic functions from general purpose CPU pipelines, or against general-purpose masking instructions altogether.

3.5 ISE utilisation

Here we detail the ciphers used to demonstrate the masking ISE, and several issues encountered while using the masking ISE instructions to implement side-channel secure software.

Implemented kernels. To demonstrate the effectiveness of the masking ISE, we implemeted the Speck [BSS+13] and Sparx [DPU+16] block-ciphers, plus the ChaCha20 [Ber08] stream cipher. These were chosen because their ARX nature made them good candidates to evaluate the complete set of proposed masking ISE instructions. For Speck and Sparx, the Key-Schedule, Encrypt and Decrypt functions were all implemented in masked and un-masked representations. For ChaCha20, we focus on only the round function. All implementations were written in assembly, using the same policies for loop unrolling to ensure fair comparisons..

Register pressure. By using pairs of general purpose registers to store mask shares, register pressure is inevitably increased. While the same is true for any implementation of masking in software, the requirement for corresponding shares to be stored in adjacent odd/even pairs of registers makes allocation more complex. While this would be a burden on a compiler's register allocator, we note that side-channel secure code must usually be written in assembly anyway. We found that RISC-V had enough registers for all of the Speck and Sparx functions without the need for inner-loop stack spills, though the function epilog/prolog needed to store saved registers to the stack. The ChaCha20 kernel needed 4 additional stack load/store operations due to the number of working variables it requires. Again, this would be the case in any 2-share masked implementation.

The additional register pressure could be alleviated by storing extra shares in coppies of the general purpose register file. While this is a reasonable design choice it comes with considerable extra resource costs. Further, one must either 1) add additional instructions to load/store to/from the new state (i.e. the new state is architectural, and must be saved/restored on a context switch), or 2) existing load/store instructions could automatically create masked versions of values as they enter the register file from memory. We note that point 2 on its own is inadequate, as side-channel protection then only applies within the CPU boundary. Registers in the memory hierarchy would still cause hamming weight leakage of secret values as they are loaded or stored. An additional mechanism (such as the one described in [DMGH19]) would then be required to protect memory values.

Register access scheduling. Loading and storing shares required careful sheduling of instructions. It was nessesary to re-order instruction sequences such that all 0-th shares were loaded in sequence, followed by all 1-st shares. This prevented hamming distance leakage due to share collisions. Where re-ordering of instructions was not possible, fence-like instructions which loaded/stored dummy random values were used to clear micro-architectural resources. The techniques used in this paper were identical to those described in [SRS+]. We note that dedicated micro-architectural leakage fence instructions as described in [GMPP20] also work, with potentially less overhead.

Implicit register access. Before implementing the bit-reversal scheme detailed in Section 3.3, we found that keeping loop-counter or pointer variables in even-numbered registers removed some sources of leakage. We believe this is because all masking ISE instructions store even-numbered register addresses in their encodings, hence only explicitly accessing even-numbered registers using base ISA instructions removes the possibility of glitching between odd and even register values.

We also found that the use of the compressed instruction extension could cause unexpected leakages. We believe that this was due to race-conditions in the register address decoding logic, where decoding of the register address in mixed sequences of 32-bit and 16-bit instructions requires the source register addresses to come from different places in the instruction buffer. This was solved by disabling compressed instruction encodings for the target functions, and aligning the target functions to a 4-byte boundary.

Both of these can be related to the Neighbour Leakage effect detailed in [PV17]. We suggest Implicit Register Access effects as a more general term for these sorts of leakage hazard.

Speculative execution. Despite being a very simple micro-controller, the SCARV-CPU still implements a degree of speculative execution. Specifically, it has no branch-predictor, and all control flow changes are resolved very late in the pipeline[7]. This means that instructions immediately following a taken branch are partially executed before being flushed from the pipeline. In some cases, this led to the unmasking of variables (e.g.

[7] One could describe this as an "assume always not-taken" branch prediction strategy.

the `mask.b.unmask` instruction immediately following the end of a loop) and required re-ordering or padding instructions to avoid speculatively excuting instructions which caused leakage.

We note that most CPUs analysed in the literature do not have pipelines deep enough to exhibit this effect; the ARM M0, M0+, M3 and M4 all have only 3 pipeline stages. As side-channel secure implementations are moved to more powerful CPUs with deeper pipelines, we expect this effect to become more prevelant.

4 Evaluation

4.1 Experimental platform

We used a standard experimental platform for evaluating the masking ISE, outlined here in the interest of reproducibility.

The augmented SCARV core was implemented on a Sasebo GIII [HKSS12] side-channel analysis platform. It contains two FPGAs: a Xilinx Kintex-7 (model `xc7k160tfbg676`) target FPGA, and a Xilinx Spartan-6 (model `xc6slx45`) support FPGA. Our evaluation only used the Kintex-7. The design was synthesised using Xilinx Vivado 2019.2, using the default synthesis and implementaiton strategies. No effort was spent on optimising the synthesis or routing. The Kintex-7 FPGA uses a 200MHz differential external clock source, which is transformed into a 50MHz internal clock used by the entire design.

Trace capture uses a standard pipeline of components, including: a MiniCircuits BLK+89 D/C blocker, an Agilent 8447D amplifier (with a 100 kHz to 1.3 GHz range, and 25 dB gain), and a PicoScope 5000 series oscilloscope. The oscilloscope uses a 250 MHz sample rate, with a 12-bit sample resolution. The capture process is coordinated using a laptop, which is responsible for: 1) configuring the target device with the correct experiment program, 2) uploading and retrieving experiment input and output values, 3) recording oscilloscope trace data and packaging it for storage.

Initial leakage experiments on the baseline core showed the presence of some noise in the capture pipeline. This was removed after trace capture using software filtering. A butterworth low-pass filter with 5 taps and a 8MHz cutoff frequency was used to remove the noise. The 8 MHz cutoff was chosen to maximise detectable leakage in the baseline ISA instructions.

4.2 Results

4.2.1 Hardware overhead

Table 1 shows the hardware implementation costs of the masking ISE, using the un-modified SCARV core as a baseline. We first evaluate the design targeting the FPGA platform described in Section 4.1. We also give ASIC synthesis results using the Yosys [Wol] open Synthesis Suite. For the ASIC results, area is reported in NAND2 gate equivilent cells, and timing as the Longest Topological Path (LTP) through the circuit. Note that these results are not tied to a particular technology library, but an abstract library bundled with Yosys. This makes design comparisons and reproduction easier, as it does not depend on researchers being able to source any technology libraries.

Note the difference in area overheads between the FPGA implementation (1.48x LUTs) and simple ASIC flow (1.22x NAND2 Cells). We believe this is down to the granularity of the FPGA synthesis target cells (i.e. N-input LUTs) v.s 2-input CMOS cells. That is, not all LUTs are fully utilised.

We also note that the masked ALU does not appear on the critical path, but its inclusion still results in a reduction in timing slack. This is likely due to extra logic in

Table 1: Implementation results for the ISE, as integrated into the SCARV core. Note that timing slack is quoted for a frequency of $f = 50\,\text{MHz}$.

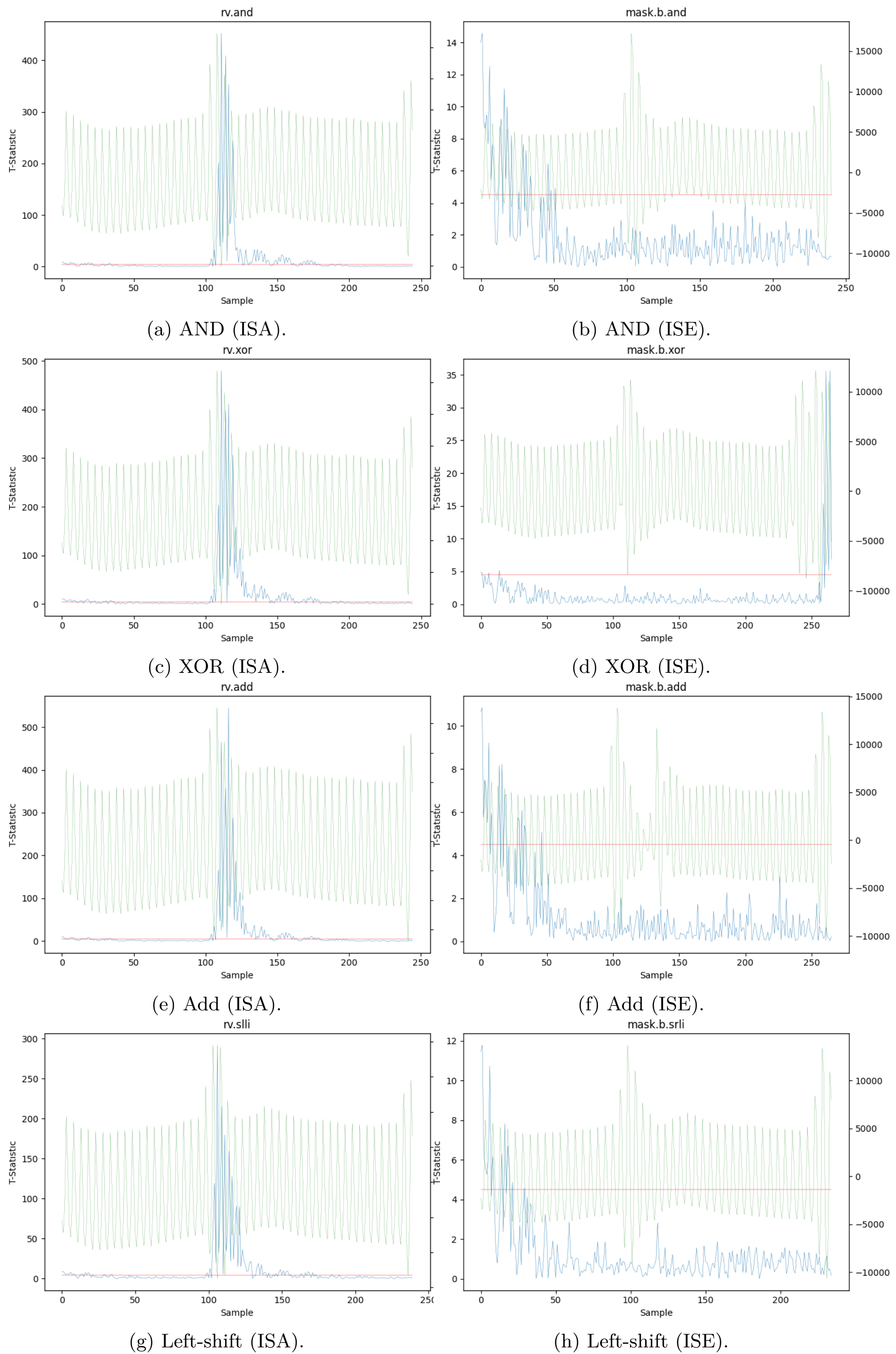
	FPGA			ASIC
Implementation	Slice LUTs	FFs	Timing slack	NAND2 cells
SCARV core	4229	2141	3.417 ns	46750
SCARV core + ISE	6261	2348	2.736 ns	57385

Table 2: A summary of cycle count (i.e., execution latency) for each instruction in the ISE, plus, where appropriate, a comparison with an equivalent from the base ISA.

Instruction	Cycle count	
	Base ISA	ISE
`mask.b.mask`		1
`mask.b.unmask`		1
`mask.b.remask`		1
`mask.b.not`	1	1
`mask.b.and`	1	1
`mask.b.ior`	1	1
`mask.b.xor`	1	1
`mask.b.add`	1	6
`mask.b.sub`	1	6
`mask.b.srli`	1	1
`mask.b.slli`	1	1
`mask.b.rori`	1	1

Table 3: Comparison of a) instruction count (i.e., number of instructions executed), b) cycle count (i.e., execution latency), and c) instruction footprint (measured in bytes) for both unmasked (using the base ISA) and masked (using the ISE) implementations of various cryptographic kernels.

Kernel	Instruction count		Cycle count		Instruction footprint	
	Base ISA	ISE	Base ISA	ISE	Base ISA	ISE
Sparx Key Schedule	435	685	561	1131	118	244
Sparx Encrypt	708	935	825	1231	350	604
Sparx Decrypt	707	928	778	1300	358	576
Speck Key Schedule	198	245	219	396	656	946
Speck Encrypt	173	237	207	405	610	920
Speck Decrypt	173	237	204	409	664	896
ChaCha20 Round	1260	1766	1631	3720	468	1406



(a) AND (ISA).

(b) AND (ISE).

(c) XOR (ISA).

(d) XOR (ISE).

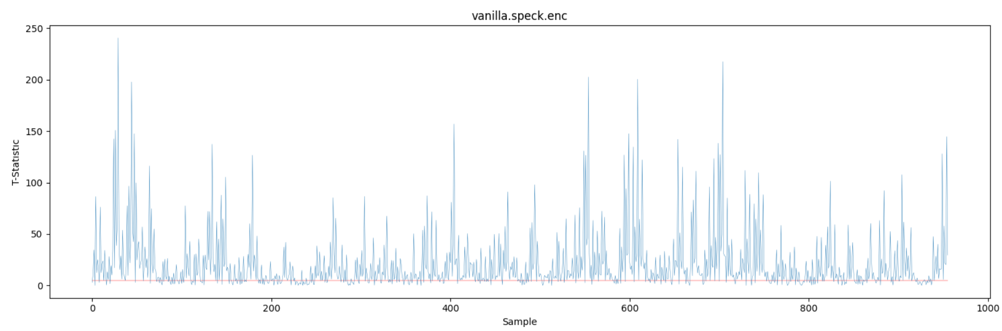
(e) Add (ISA).

(f) Add (ISE).

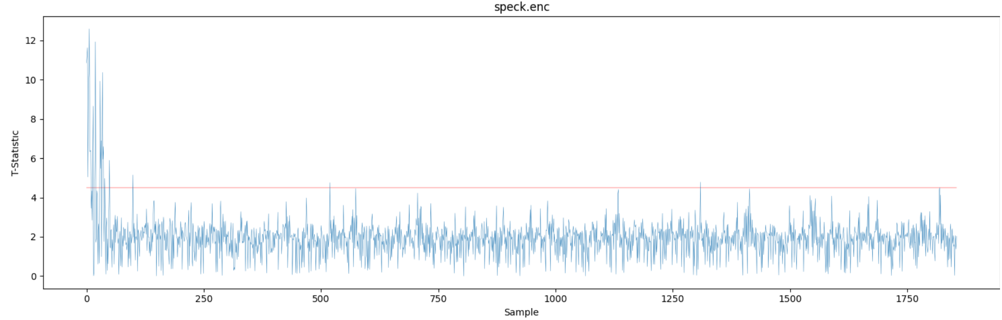
(g) Left-shift (ISA).

(h) Left-shift (ISE).

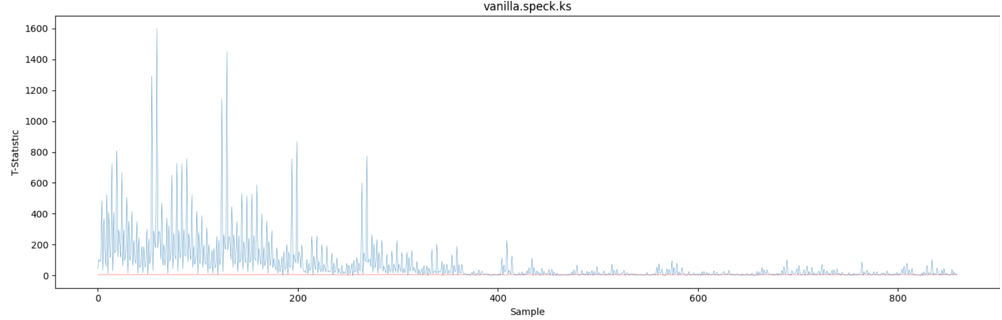
Figure 3: Comparison of leakage for selected instructions as executed on the SCARV core using the base ISA and ISE: each case relates to TVLA-based leakage detection, plotting the (absolute) t-statistic stemming from $100,000$ power consumption traces.



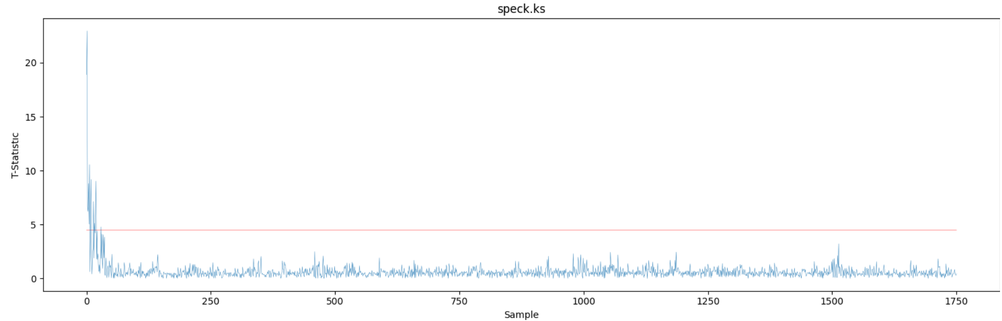
(a) Encrypt (ISA).



(b) Encrypt (ISE).

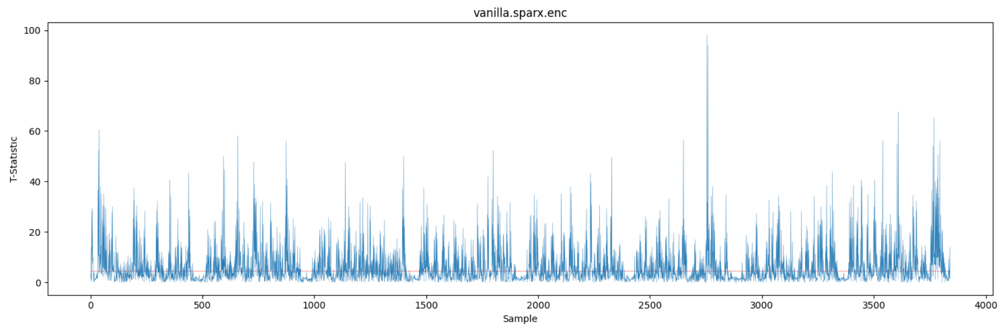


(c) Key schedule (ISA).

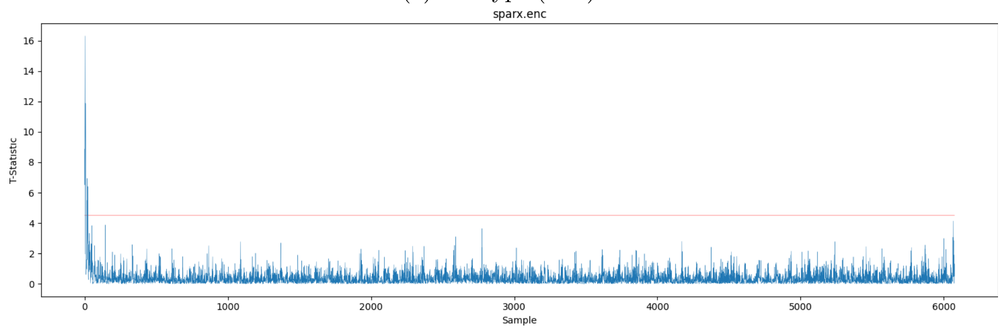


(d) Key schedule (ISE).

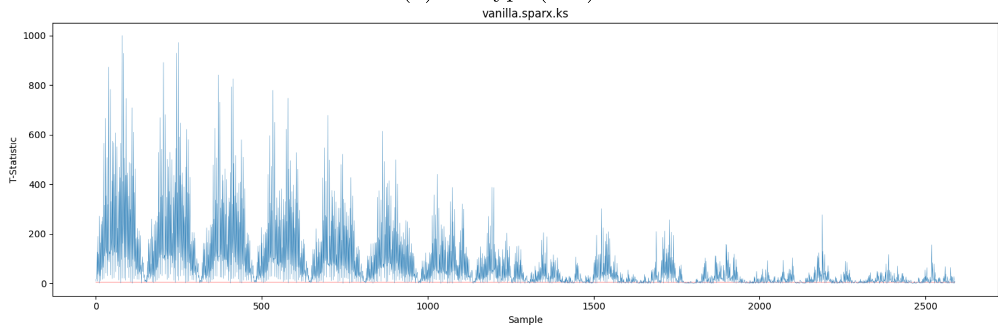
Figure 4: Comparison of leakage for kernels related to the Speck block cipher as executed on the SCARV core: each case relates to TVLA-based leakage detection, plotting the (absolute) t-statistic stemming from $100,000$ power consumption traces.



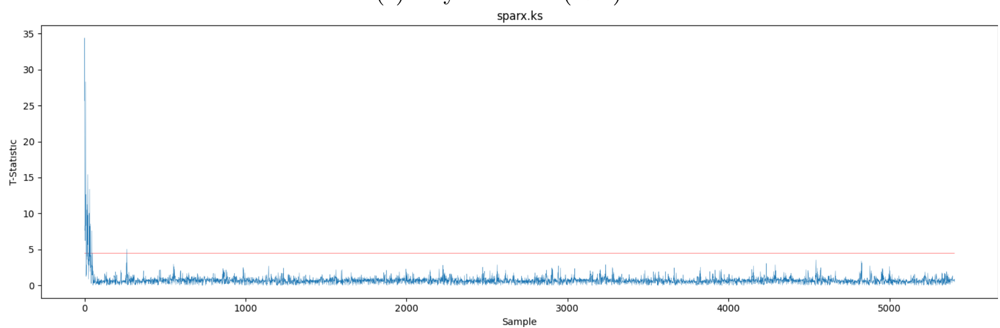
(a) Encrypt (ISA).



(b) Encrypt (ISE).

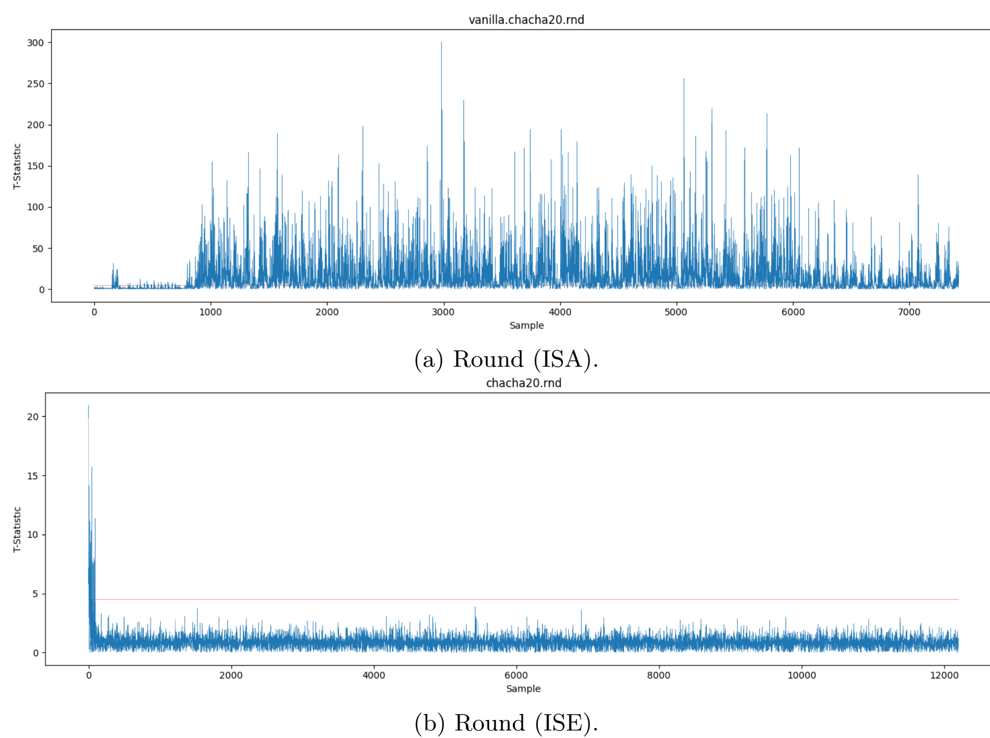


(c) Key schedule (ISA).



(d) Key schedule (ISE).

Figure 5: Comparison of leakage for kernels related to the Sparx block cipher as executed on the SCARV core: each case relates to TVLA-based leakage detection, plotting the (absolute) t-statistic stemming from $100,000$ power consumption traces.



(a) Round (ISA).

(b) Round (ISE).

Figure 6: Comparison of leakage for kernels related to the ChaCha20 stream cipher as executed on the SCARV core: each cases relates to TVLA-based leakage detection, plotting the (absolute) t-statistic stemming from $100,000$ power consumption traces.

forcing signals to be routed along longer paths, rather than the circuits having any extra logical depth.

4.2.2 Leakage and execution latency: single instruction

Each masking ISE instruction was compared against the relevant baseline ISA instruction in terms of execution latency (Table 2) and leakage (Figure 3). We can see from Figure 3 that, as expected, the base ISA instructions leak strongly, and that the masking ISE instructions do not.

4.2.3 Leakage and execution latency: whole kernel

We evaluated several block cipher algorithms using the masking ISE. Table 3 shows the execution latency and static code size for each masked and un-masked implementation of the ciphers.

5 Conclusion

Summary. In this paper, we presented the design, implementation, and evaluation of an ISE to support software-based masking; we pitch it as an option positioned, and so, to some extent, a compromise between software-only and hardware-only alternatives. Accepting the inherent overhead in hardware, our evaluation suggests that the ISE can support secure first-order masked implementations of various kernels; the associated overhead wrt. execution latency and memory footprint is modest.

That said, however, implementing the ISE highlighted various challenges which demand care. For example, vs. whole-core masking [GJM+16, MGH19] it is challenging to ensure non-interaction between shares: even if the masked ALU is modular and so physically separate, the unmasked datapath may require some changes to ensure this property. It is challenging to both a) identify when and where change is required, and b) implement such change, because the ISE becomes more invasive as a result. Likewise, utilising the ISE is not as “drop in” a process as one may expect intuitively. For example, one must contend with increased register pressure while simultaneously considering the security properties of loading/storing shares from/to memory; doing so is not trivial.

Future work. Beyond any more incremental improvements, such as increasing the number and diversity of kernels we evaluate, various higher-level directions represent either important and/or interesting future work:

- Our ISE directly supports masked operations for $n = 2$ shares. The question remains, however, how such support generalises: for example, 1) how to bootstrap higher-order masking schemes using an ISE, such as ours, that includes first-order oriented building blocks, and/or 2) how such an ISE might be alter the ISA to enhance generality wrt. said order.

- It is plausible to consider extending the ISE to assist with various challenges wrt. utilisation. For example, one can imagine enforcing secure register access scheduling via ISE-supported, and so “share aware” memory access instructions.

- Belaïd et al. [BDM+20] present Tornado, a tool capable of deriving secure masked implementations from a high-level description. [BDM+20, Section 3.1] states the model of computation assumed, which includes a number of masked “gadgets” whose form implies that Tornado could be retargeted to benefit from an ISE-enabled platform.

- Instructions such as `mask.b.add (rd1,rd2), (rs1,rs2), (rs3,rs4)` perform arithmetic with an assumed word size of XLEN. However, consider a case where XLEN = 32 but we implement a kernel which demands masked arithmetic modulo 2^w for some $w \neq 32$, e.g., $w = 16$ or $w = 64$. Doing so may be problematic, because the masked operation lacks any direct support for, e.g., management of carries.
- Ge et. al [GYH18] pitch their aISA as "a new hardware-software contract", and, as a result, weaken the abstraction afforded by an ISA of some associated micro-architecture. To meet the required security properties, our ISE demands care wrt. implementation (in hardware) *and* utilisation (in software). Per the aISA, one could therefore question how much the ISE can (or if it even *should*) dictate details of an associated micro-architectural implementation. Exploration of this question, e.g., what requirements are required, and how to specify them, seems an interesting challenge.

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.

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.

[BDCU17] 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.

[BDG+13] Shivam Bhasin, Jean-Luc Danger, Tarik Graba, Yves Mathieu, Daisuke Fujimoto, and Makoto Nagata. Physical security evaluation at an early design-phase: A side-channel aware simulation methodology. In *Proceedings of International Workshop on Engineering Simulations for Cyber-Physical Systems*, pages 13–20, 2013.

[BDM+20] S. Belaïd, P.-É. Dagand, D. Mercadier, M. Rivain, and R. Wintersdorff. Tornado: Automatic generation of probing-secure masked bitsliced implementations. In *Advances in Cryptology (EUROCRYPT)*, LNCS 12107, pages 311–341. Springer-Verlag, 2020.

[Ber08] Daniel J Bernstein. Chacha, a variant of salsa20. In *Workshop Record of SASC*, volume 8, pages 3–5, 2008.

[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.

[BGM09] S. Bartolini, R. Giorgi, and E. Martinelli. Instruction set extensions for cryptographic applications. In Ç.K. Koç, editor, *Cryptographic Engineering*, chapter 9, pages 191–233. Springer, 2009.

[BMT16] W. Burleson, O. Mutlu, and M. Tiwari. Who is the major threat to tomorrow's security? you, the hardware designer. In *Design Automation Conference (DAC)*, pages 145:1–145:5, 2016.

[BSS+13] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The simon and speck families of lightweight block ciphers. *IACR Cryptology ePrint Archive*, 2013(1):404–449, 2013.

[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.

[CGTV15] J.-S. Coron, J. Großschädl, M. Tibouchi, and P.K. Vadnala. Conversion from arithmetic to boolean masking with logarithmic complexity. In *Fast Software Encryption (FSE)*, LNCS 9054, pages 130–149. Springer-Verlag, 2015.

[CJRR99] Suresh Chari, Charanjit S Jutla, Josyula R Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *Annual International Cryptology Conference*, pages 398–412. Springer, 1999.

[CS09] Zhimin Chen and Patrick Schaumont. Early feedback on side-channel risks with accelerated toggle-counting. In *2009 IEEE International Workshop on Hardware-Oriented Security and Trust*, pages 90–95. IEEE, 2009.

[DMGH19] Elke De Mulder, Samatha Gummalla, and Michael Hutter. Protecting risc-v against side-channel attacks. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC âĂŹ19, page 1âĂŞ4. Association for Computing Machinery, Jun 2019.

[DPU+16] Daniel Dinu, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, Johann Großschädl, and Alex Biryukov. Design strategies for arx with provable bounds: Sparx and lax. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 484–513. Springer, 2016.

[GB11] C. Galuzzi and K. Bertels. The instruction-set extension problem: A survey. *ACM Transactions on Reconfigurable Technology and Systems*, 4(2):18:1–18:28, 2011.

[GBR+19] Vinod Ganesan, Rahul Bodduna, Chester Rebeiro, et al. Param: A microprocessor hardened for power side-channel attack resistance. *arXiv preprint arXiv:1911.08813*, 2019.

[GJM+16] 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, 2016.

[GMK16] H. Gross, S. Mangard, and T. Korak. Domain-Oriented Masking: Compact masked hardware implementations with arbitrary protection order. In *Theory of Implementation Security (TIS)*, page 3, 2016.

[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.

[GMPP20] Si Gao, Ben Marshall, Dan Page, and Thinh Pham. Fenl: an ise to mitigate analogue micro-architectural leakage. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 73–98, 2020.

[GP99] L. Goubin and J. Patarin. DES and differential power analysis the duplication method. In *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 1717, pages 158–172. Springer-Verlag, 1999.

[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.

[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.

[HPN+19] Miao Tony He, Jungmin Park, Adib Nahiyan, Apostol Vassilev, Yier Jin, and Mark Tehranipoor. Rtl-psc: automated power side-channel leakage assessment at register-transfer level. In *2019 IEEE 37th VLSI Test Symposium (VTS)*, pages 1–6. IEEE, 2019.

[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.

[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.

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

[LC08] S.H. Lee and L. Choi. Accelerating symmetric and asymmetric ciphers with register file extension for multi-word and long-word operation. In *International Conference on Information Science and Security (ICISS)*, pages 102–107, 2008.

[LYS04] R.B. Lee, X. Yang, and Z. Shi. Validating word-oriented processors for bit and multi-word operations. In *Annual Computer Security Applications Conference (ACSAC)*, pages 473–488, 2004.

[Mes00] Thomas S Messerges. Securing the aes finalists against power analysis attacks. In *International Workshop on Fast Software Encryption*, pages 150–164. Springer, 2000.

[Mes01] T.S. Messerges. Securing the AES finalists against power analysis attacks. In *Fast Software Encryption (FSE)*, LNCS 1978, pages 150–164. Springer-Verlag, 2001.

[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.

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

[NRR06] S. Nikova, C. Rechberger, and V. Rijmen. Threshold implementations against side-channel attacks and glitches. In *Information and Communications Security (ICICS)*, LNCS 4307, pages 529–545. Springer-Verlag, 2006.

[PM05] T. Popp and S. Mangard. Masked dual-rail pre-charge logic: DPA-resistance without routing constraints. In *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 3659, pages 172–186. Springer-Verlag, 2005.

[PV17] Kostas Papagiannopoulos and Nikita Veshchikov. Mind the gap: towards secure 1st-order masking in software. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 282–297. Springer, 2017.

[RI16] F. Regazzoni and P. Ienne. Instruction set extensions for secure applications. In *Design, Automation, and Test in Europe (DATE)*, pages 1529–1534, 2016.

[RKL+04] S. Ravi, P.C. Kocher, R.B. Lee, G. McGraw, and A. Raghunathan. Security as a new dimension in embedded system design. In *Design Automation Conference (DAC)*, pages 753–760, 2004.

[RP10] M. Rivain and E. Prouff. Provably secure higher-order masking of AES. In *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 6225, pages 413–427. Springer-Verlag, 2010.

[RRKH04] S. Ravi, A. Raghunathan, P.C. Kocher, and S. Hattangady. Security in embedded systems: Design challenges. *ACM Transactions on Embedded Computer Systems*, 3(3):461–491, 2004.

[RV:19] The RISC-V instruction set manual. Technical Report Volume I: User-Level ISA (Version 20191213), 2019.

[SRS+] Madura A Shelton, Francesco Regazzoni, Niels Samwel, Markus Wagner, Lejla Batina, and Yuval Yarom. Rosita: Towards automatic elimination of power-analysis leakage in ciphers.

[Tri03] E. Trichina. Combinational logic design for AES SubByte transformation on masked data. Cryptology ePrint Archive, Report 2003/236, 2003.

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

[Wol] Clifford Wolf. Yosys open synthesis suite. `http://www.clifford.at/yosys/`.

[ZBPF18] Davide Zoni, Alessandro Barenghi, Gerardo Pelosi, and William Fornaciari. A comprehensive side-channel information leakage analysis of an in-order risc cpu microarchitecture. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 23(5):1–30, 2018.

A Instruction semantics

- `mask.b.mask (rd1,rd2), rs1`

  begin
  $x \leftarrow \mathsf{GPR}[\texttt{rs1}]$
  $(r_0, r_1) \leftarrow \textsc{BoolMask}(x)$
  $\mathsf{GPR}[\texttt{rd1}] \leftarrow r_0, \mathsf{GPR}[\texttt{rd2}] \leftarrow r_1$
  end

- `mask.b.unmask rd1, (rs1,rs2)`

  begin
  $x_0 \leftarrow \mathsf{GPR}[\texttt{rs1}], x_1 \leftarrow \mathsf{GPR}[\texttt{rs2}]$
  $r \leftarrow \textsc{BoolUnmask}((x_0, x_1))$
  $\mathsf{GPR}[\texttt{rd1}] \leftarrow r$
  end

- `mask.b.remask (rd1,rd2), (rs1,rs2)`

  begin
  $x_0 \leftarrow \mathsf{GPR}[\texttt{rs1}], x_1 \leftarrow \mathsf{GPR}[\texttt{rs2}]$
  $(r_0, r_1) \leftarrow \textsc{BoolRemask}((x_0, x_1))$
  $\mathsf{GPR}[\texttt{rd1}] \leftarrow r_0, \mathsf{GPR}[\texttt{rd2}] \leftarrow r_1$
  end

- `mask.b.not (rd1,rd2), (rs1,rs2)`

  begin
  $x_0 \leftarrow \mathsf{GPR}[\texttt{rs1}], x_1 \leftarrow \mathsf{GPR}[\texttt{rs2}]$
  $(r_0, r_1) \leftarrow \textsc{BoolNOT}((x_0, x_1))$
  $\mathsf{GPR}[\texttt{rd1}] \leftarrow r_0, \mathsf{GPR}[\texttt{rd2}] \leftarrow r_1$
  end

- `mask.b.and (rd1,rd2), (rs1,rs2), (rs3,rs4)`

  begin
  $x_0 \leftarrow \mathsf{GPR}[\texttt{rs1}], x_1 \leftarrow \mathsf{GPR}[\texttt{rs2}],\ y_0 \leftarrow \mathsf{GPR}[\texttt{rs3}], y_1 \leftarrow \mathsf{GPR}[\texttt{rs4}]$
  $(r_0, r_1) \leftarrow \textsc{BoolAND}((x_0, x_1), (y_0, y_1))$
  $\mathsf{GPR}[\texttt{rd1}] \leftarrow r_0, \mathsf{GPR}[\texttt{rd2}] \leftarrow r_1$
  end

- `mask.b.ior (rd1,rd2), (rs1,rs2), (rs3,rs4)`

  begin
  $x_0 \leftarrow \mathsf{GPR}[\texttt{rs1}], x_1 \leftarrow \mathsf{GPR}[\texttt{rs2}],\ y_0 \leftarrow \mathsf{GPR}[\texttt{rs3}], y_1 \leftarrow \mathsf{GPR}[\texttt{rs4}]$
  $(r_0, r_1) \leftarrow \textsc{BoolIOR}((x_0, x_1), (y_0, y_1))$
  $\mathsf{GPR}[\texttt{rd1}] \leftarrow r_0, \mathsf{GPR}[\texttt{rd2}] \leftarrow r_1$
  end

- `mask.b.xor (rd1,rd2), (rs1,rs2), (rs3,rs4)`

  begin
  $x_0 \leftarrow \mathsf{GPR}[\texttt{rs1}], x_1 \leftarrow \mathsf{GPR}[\texttt{rs2}],\ y_0 \leftarrow \mathsf{GPR}[\texttt{rs3}], y_1 \leftarrow \mathsf{GPR}[\texttt{rs4}]$
  $(r_0, r_1) \leftarrow \textsc{BoolXOR}((x_0, x_1), (y_0, y_1))$
  $\mathsf{GPR}[\texttt{rd1}] \leftarrow r_0, \mathsf{GPR}[\texttt{rd2}] \leftarrow r_1$
  end

- `mask.b.add (rd1,rd2), (rs1,rs2), (rs3,rs4)`

  begin
  $x_0 \leftarrow \mathsf{GPR}[\texttt{rs1}], x_1 \leftarrow \mathsf{GPR}[\texttt{rs2}],\ y_0 \leftarrow \mathsf{GPR}[\texttt{rs3}], y_1 \leftarrow \mathsf{GPR}[\texttt{rs4}]$
  $(r_0, r_1) \leftarrow \textsc{BoolAdd}((x_0, x_1), (y_0, y_1))$
  $\mathsf{GPR}[\texttt{rd1}] \leftarrow r_0, \mathsf{GPR}[\texttt{rd2}] \leftarrow r_1$
  end

- `mask.b.sub (rd1,rd2), (rs1,rs2), (rs3,rs4)`

  begin
  $x_0 \leftarrow \mathsf{GPR}[\texttt{rs1}], x_1 \leftarrow \mathsf{GPR}[\texttt{rs2}],\ y_0 \leftarrow \mathsf{GPR}[\texttt{rs3}], y_1 \leftarrow \mathsf{GPR}[\texttt{rs4}]$
  $(r_0, r_1) \leftarrow \textsc{BoolSub}((x_0, x_1), (y_0, y_1))$
  $\mathsf{GPR}[\texttt{rd1}] \leftarrow r_0, \mathsf{GPR}[\texttt{rd2}] \leftarrow r_1$
  end

- `mask.b.slli (rd1,rd2), (rs1,rs2), imm`

  begin
  $x_0 \leftarrow \mathsf{GPR}[\texttt{rs1}], x_1 \leftarrow \mathsf{GPR}[\texttt{rs2}],$
  $(r_0, r_1) \leftarrow \textsc{BoolSLL}((x_0, x_1), imm)$
  $\mathsf{GPR}[\texttt{rd1}] \leftarrow r_0, \mathsf{GPR}[\texttt{rd2}] \leftarrow r_1$
  end

- `mask.b.srli (rd1,rd2), (rs1,rs2), imm`

  begin
  $x_0 \leftarrow \mathsf{GPR}[\texttt{rs1}], x_1 \leftarrow \mathsf{GPR}[\texttt{rs2}],$
  $(r_0, r_1) \leftarrow \textsc{BoolSRL}((x_0, x_1), imm)$
  $\mathsf{GPR}[\texttt{rd1}] \leftarrow r_0, \mathsf{GPR}[\texttt{rd2}] \leftarrow r_1$
  end

- `mask.b.rori (rd1,rd2), (rs1,rs2), imm`

  begin
  $x_0 \leftarrow \mathsf{GPR}[\texttt{rs1}], x_1 \leftarrow \mathsf{GPR}[\texttt{rs2}],$
  $(r_0, r_1) \leftarrow \textsc{BoolROR}((x_0, x_1), imm)$
  $\mathsf{GPR}[\texttt{rd1}] \leftarrow r_0, \mathsf{GPR}[\texttt{rd2}] \leftarrow r_1$
  end

B Instruction encodings

Table 4: An overview of instruction encodings (operations under Boolean masking).

31 30 29 28 27 26 25	24 23 22 21 20	19 18 17 16 15	14 13 12	11 10 9 8 7	6 5 4 3 2 1 0	
0000000	00010	rs1	000	rdm	1011011	mask.b.mask
0000000	00011	rsm1	000	rd	1011011	mask.b.unmask
0000000	00011	rsm1	000	rdm	1011011	mask.b.remask
0000001	00000	rsm1	010	rdm	1011011	mask.b.not
0000001	rsm2	rsm1	111	rdm	1011011	mask.b.and
0000001	rsm2	rsm1	110	rdm	1011011	mask.b.ior
0000001	rsm2	rsm1	100	rdm	1011011	mask.b.xor
0000001	rsm2	rsm1	000	rdm	1011011	mask.b.add
0000001	rsm2	rsm1	001	rdm	1011011	mask.b.sub
000010 (bits 31–26)	shamt (bits 25–20)	rsm1	000	rdm	1011011	mask.b.slli
000010 (bits 31–26)	shamt (bits 25–20)	rsm1	001	rdm	1011011	mask.b.srli
000010 (bits 31–26)	shamt (bits 25–20)	rsm1	010	rdm	1011011	mask.b.rori

C Additional algorithms

Data: The value x.
Result: The masked value $\hat{r} = (r_0, r_1)$ st. $r = r_0 \oplus r_1 = x$.

function BoolMask((x_0, x_1)) **begin**
  $t \xleftarrow{\$} \{0,1\}^w$
  $r_1 \leftarrow t$
  $r_0 \leftarrow x \oplus t$
  return(r_0, r_1)
end

Algorithm 1: BoolMask: apply mask operation (under Boolean masking).

Data: The masked value $\hat{x} = (x_0, x_1)$.
Result: The value $r = r_0 \oplus r_1 = x$.

function BoolUnmask((x_0, x_1)) **begin**
  $r \leftarrow x_0 \oplus x_1$
  return r
end

Algorithm 2: BoolUnmask: apply unmask operation (under Boolean masking).

Data: The masked value $\hat{x} = (x_0, x_1)$.
Result: The masked value $\hat{r} = (r_0, r_1)$ st. $r = r_0 \oplus r_1 = x$.

function BoolRemask((x_0, x_1)) **begin**
  $t \xleftarrow{\$} \{0,1\}^w$
  $r_1 \leftarrow x_1 \oplus t$
  $r_0 \leftarrow x_0 \oplus t$
  return(r_0, r_1)
end

Algorithm 3: BoolRemask: apply remask operation (under Boolean masking).

Data: The masked value $\hat{x} = (x_0, x_1)$.
Result: The masked value $\hat{r} = (r_0, r_1)$ st. $r = r_0 \oplus r_1 = \neg x$.

function BOOLNOT$((x_0, x_1))$ **begin**
  $t \xleftarrow{\$} \{0,1\}^w$
  $r_1 \leftarrow t \oplus (\neg x_1)$
  $r_0 \leftarrow t \oplus x_0$
  return(r_0, r_1)
end

Algorithm 4: BOOLNOT: apply NOT operation (under Boolean masking).

Data: The masked values $\hat{x} = (x_0, x_1)$ and $\hat{y} = (y_0, y_1)$.
Result: The masked value $\hat{r} = (r_0, r_1)$ st. $r = r_0 \oplus r_1 = x \wedge y$.

function BOOLAND$((x_0, x_1), (y_0, y_1))$ **begin**
  $t \xleftarrow{\$} \{0,1\}^w$
  $r_1 \leftarrow t \oplus (x_1 \wedge y_1) \oplus (x_1 \vee \neg y_0)$
  $r_0 \leftarrow t \oplus (x_0 \wedge y_1) \oplus (x_0 \vee \neg y_0)$
  return(r_0, r_1)
end

Algorithm 5: BOOLAND: apply AND operation (under Boolean masking).

Data: The masked values $\hat{x} = (x_0, x_1)$ and $\hat{y} = (y_0, y_1)$.
Result: The masked value $\hat{r} = (r_0, r_1)$ st. $r = r_0 \oplus r_1 = x \vee y$.

function BOOLIOR$((x_0, x_1), (y_0, y_1))$ **begin**
  $(s_0, s_1) \leftarrow$ BOOLAND$((x_0, \neg x_1), (y_0, \neg y_1))$
  $r_1 \leftarrow \neg s_1$
  $r_0 \leftarrow s_0$
  return(r_0, r_1)
end

Algorithm 6: BOOLIOR: apply OR operation (under Boolean masking).

Data: The masked values $\hat{x} = (x_0, x_1)$ and $\hat{y} = (y_0, y_1)$.
Result: The masked value $\hat{r} = (r_0, r_1)$ st. $r = r_0 \oplus r_1 = x \oplus y$.

function BOOLXOR$((x_0, x_1), (y_0, y_1))$ **begin**
  $t \xleftarrow{\$} \{0,1\}^w$
  $r_0 \leftarrow t \oplus x_0 \oplus y_0$
  $r_1 \leftarrow t \oplus x_1 \oplus y_1$
  return(r_0, r_1)
end

Algorithm 7: BOOLXOR: apply XOR operation (under Boolean masking).

Data: The maked values $\hat{x} = (x_0, x_1)$ and $\hat{y} = (y_0, y_1)$
Result: The masked value $\hat{r} = (r_0, r_1)$ st. $r = r_0 \oplus r_1 = x + y$.

function BoolAdd($(x_0, x_1), (y_0, y_1)$) **begin**
  $(a_0, a_1) \leftarrow$ BoolXor($(x_0, x_1), (y_0, y_1)$)
  $(p_0, p_1) \leftarrow (a_0, a_1)$
  $(g_0, g_1) \leftarrow$ BoolAnd($(x_0, x_1), (y_0, y_1)$)
  for $i \leftarrow 1$ to $\log_2 w$ **do**
    $(h_0, h_1) \leftarrow$ BoolSHL($(g_0, g_1), 2^{i-1}$)
    $(u_0, u_1) \leftarrow$ BoolSHL($(p_0, p_1), 2^{i-1}$)
    $(h_0, h_1) \leftarrow$ BoolAnd($(p_0, p_1), (h_0, h_1)$)
    $(g_0, g_1) \leftarrow$ BoolXor($(g_0, g_1), (h_0, h_1)$)
    $(p_0, p_1) \leftarrow$ BoolAnd($(p_0, p_1), (u_0, u_1)$)
  end
  $(h_0, h_1) \leftarrow$ BoolSHL($(g_0, g_1), 1$)
  $(r_0, r_1) \leftarrow$ BoolXor($(a_0, a_1), (h_0, h_1)$)
  return(r_0, r_1)
end

Algorithm 8: BoolAdd: apply addition operation (under Boolean masking).

Data: The maked values $\hat{x} = (x_0, x_1)$ and $\hat{y} = (y_0, y_1)$
Result: The masked value $\hat{r} = (r_0, r_1)$ st. $r = r_0 \oplus r_1 = x - y$

function BoolSub($(x_0, x_1), (y_0, y_1)$) **begin**
  $(a_0, a_1) \leftarrow$ BoolXor($(x_0, x_1), (y_0, y_1)$)
  $(p_0, p_1) \leftarrow (a_0, a_1)$
  $(g_0, g_1) \leftarrow$ BoolAnd($(x_0, x_1), (y_0, y_1)$)
  $(u_0, u_1) \leftarrow$ BoolAnd($(p_0, p_1), (0, 1)$)
  $(g_0, g_1) \leftarrow$ BoolXor($(g_0, g_1), (u_0, u_1)$)
  for $i \leftarrow 1$ to $\log_2 w$ **do**
    $(h_0, h_1) \leftarrow$ BoolSHL($(g_0, g_1), 2^{i-1}$)
    $(u_0, u_1) \leftarrow$ BoolSHL($(p_0, p_1), 2^{i-1}$)
    $(h_0, h_1) \leftarrow$ BoolAnd($(p_0, p_1), (h_0, h_1)$)
    $(g_0, g_1) \leftarrow$ BoolXor($(g_0, g_1), (h_0, h_1)$)
    $(p_0, p_1) \leftarrow$ BoolAnd($(p_0, p_1), (u_0, u_1)$)
  end
  $(h_0, h_1) \leftarrow (g_0 \ll 1 \parallel 0, g_1 \ll 1 \parallel 1)$
  $(r_0, r_1) \leftarrow$ BoolXor($(a_0, a_1), (h_0, h_1)$)
  return(r_0, r_1)
end

Algorithm 9: BoolSub: apply subtraction operation (under Boolean masking).

Data: The masked value $\hat{x} = (x_0, x_1)$, and an integer $0 \leq i < w$.
Result: The masked value $\hat{r} = (r_0, r_1)$ st. $r = r_0 \oplus r_1 = x \ll i$.

function BoolSHL($(x_0, x_1), i$) **begin**
  $t \xleftarrow{\$} \{0, 1\}^i$
  $r_1 \leftarrow (x_1 \ll i) \parallel t$
  $r_0 \leftarrow (x_0 \ll i) \parallel t$
  return(r_0, r_1)
end

Algorithm 10: BoolSHL: apply left-shift operation (under Boolean masking).

Data: The masked value $\hat{x} = (x_0, x_1)$, and an integer $0 \leq i < w$.
Result: The masked value $\hat{r} = (r_0, r_1)$ st. $r = r_0 \oplus r_1 = x \gg i$.

function BoolSHR$((x_0, x_1), i)$ **begin**
  $t \xleftarrow{\$} \{0,1\}^i$
  $r_1 \leftarrow t \parallel (x_1 \gg i)$
  $r_0 \leftarrow t \parallel (x_0 \gg i)$
  return(r_0, r_1)
end

Algorithm 11: BoolSHR: apply right-shift operation (under Boolean masking).

Data: The masked value $\hat{x} = (x_0, x_1)$, and an integer $0 \leq i < w$.
Result: The masked value $\hat{r} = (r_0, r_1)$ st. $r = r_0 \oplus r_1 = x \ggg i$.

function BoolROR$((x_0, x_1), i)$ **begin**
  $r_1 \leftarrow x_1 \ggg i$
  $r_0 \leftarrow x_0 \ggg i$
  return(r_0, r_1)
end

Algorithm 12: BoolROR: apply right-rotate operation (under Boolean masking).