Secure and Efficient Software Masking on Superscalar Pipelined Processors

Barbara Gigerl[1], Robert Primas[1], and Stefan Mangard[1,2]

[1] Graz University of Technology, Graz, Austria
{barbara.gigerl, robert.primas, stefan.mangard}@iaik.tugraz.at
[2] Lamarr Security Research, Graz, Austria

Abstract. Physical side-channel attacks like power analysis pose a serious threat to cryptographic devices in real-world applications. Consequently, devices implement algorithmic countermeasures like masking. In the past, works on the design and verification of masked software implementations have mostly focused on simple microprocessors that find usage on smart cards. However, many other applications such as in the automotive industry require side-channel protected cryptographic computations on much more powerful CPUs. In such situations, the security loss due to complex architectural side-effects, the corresponding performance degradation, as well as discussions of suitable probing models and verification techniques are still vastly unexplored research questions.
We answer these questions and perform a comprehensive analysis of more complex processor architectures in the context of masking-related side effects. First, we analyze the RISC-V SweRV core — featuring a 9-stage pipeline, two execution units, and load/store buffers — and point out a significant gap between security in a simple software probing model and practical security on such CPUs. More concretely, we show that architectural side effects of complex CPU architectures can significantly reduce the protection order of masked software, both via formal analysis in the hardware probing model, as well as empirically via gate-level timing simulations. We then discuss the options of fixing these problems in hardware or leaving them as constraints to software. Based on these software constraints, we formulate general rules for the design of masked software on more complex CPUs. Finally, we compare several implementation strategies for masking schemes and present in a case study that designing secure masked software for complex CPUs is still possible with overhead as low as 13%.

Keywords: Masking · Verification · Side-channel analysis · SweRV· Glitches · Application-level processors · Coco · Probing model

1 Introduction

Cryptographic primitives are primarily designed to withstand mathematical attacks in a black-box setting. However, as soon as these primitives are deployed in the real world, they find themselves in a grey-box setting in which an attacker may observe additional physical side-channel information, such as instantaneous power consumption that can be used to extract secrets like cryptographic keys. One particularly powerful example of such a side-channel attack, differential power analysis (DPA), was introduced in 1999 by Kocher et al. [27]. In this type of attack, the adversary observes a device's power consumption while encrypting several known plaintexts, and can then extract sensitive information using statistical analysis.

The typical approach of protecting against these attacks is to implement algorithmic countermeasures, like masking [6, 9, 14, 22, 25, 34]. The main idea of masking is to make computations independent from the actually processed data. For this purpose, masking schemes split input and intermediate variables of cryptographic computations into $d+1$ random shares such that observations of up to d shares do not reveal any information about the native (unmasked) value. The security of such dth-order protected computations relies, amongst others, on the assumption of independent leakage, i.e., independent computations result in independent leakage [33]. However, many academic works in the past have shown that such assumptions are typically not satisfied on ordinary CPUs, for example, memory transitions in the register file or RAM can leak the Hamming distance between two shares [2, 15, 21, 28, 32]. In general, one can work around these problems using two different strategies. Works like [7, 20, 21, 32] show that one can design dedicated masked software implementations that take specific characteristics of the microprocessor into account, e.g., by never processing shares of the same variable in immediate succession. Alternatively, one can follow the *lazy engineering* approach, accept a certain loss of masking protection order due to architecture side-effects and compensate for that by using a protection order that is higher than theoretically required. This strategy was more formally analyzed by Balasch et al. [2] who also formulated the so-called order reduction theorem. This theorem states that, when considering simple register-based CPU architectures, the security of a dth-order masked software implementation reduces to $\lfloor \frac{d}{2} \rfloor$-th order if transition-based leakage is taken into account.

Building efficient and correct masked software implementations is generally difficult since one either needs to (1) carefully patch implementations for specific microprocessors [7, 21, 32], or (2) invest in masking orders that are a lot higher than required [2]. In both cases, the runtime of the resulting masked software implementations is significantly increased and subsequent manual leakage assessments are needed to confirm that the performed modifications have the desired effect, which is a quite labor-intensive and error-prone task. This situation becomes only ever more difficult with increasing processor complexity. For example, the effects of multiple ALU pipeline stages, forwarding logic, superscalar building blocks, caches, and complex logic for handling loads/stores on masked software implementations have not been analyzed in this detail before. One reason for that might be the sheer complexity of application-level processors that usually consist of superscalar building blocks and multi-stage pipelines. On such processors, identifying and understanding masking related side-effects can barely be done manually anymore. Here, automated analysis methods that can give concrete conditions under which masked software implementations can guarantee a certain protection order on such CPUs are more relevant then ever.

In this context, a recent work by Gigerl et al. [20] studies the simple Ibex core with Coco, a tool that can verify the correct execution of masked software implementations on given CPU netlists, while considering all possible architectural side effects. Simply speaking, Coco treats an entire CPU design as a hardware circuit and then tracks all the shares of executed masked software implementations over several cycles using methods that are inspired by Rebecca [13]. One result of their analysis is a slightly modified *secured* Ibex core on which masked software implementations can preserve their theoretic protection order in practice if a few simple software constraints are followed. While this result is certainly interesting for applications like smart cards where low computing power is sufficient, many other IoT or automotive use cases require the usage of significantly more powerful processors. This raises a number of questions about the performance, as well as the theoretic and practical security of masked software on more complex CPUs.

Our contribution We answer these questions by providing the following contributions:

– We generate several generic higher-order masked cryptographic software implementations using Tornado and show with Coco that there is little hope that such implementations can even provide 1st-order protection on more complex CPU cores. We demonstrate this based on the dual-issue 9-stage RISC-V SweRV core.
– In addition to the formal analysis of Coco, we perform gate-level simulations to demonstrate that architecture-based glitch effects are visible in practice and reduce the security of masked software by multiple orders. This points out a significant gap between security in the simple software probing model and practical security, and further motivates the verification of masked software on concrete CPU netlists in a more hardware focused probing model.
– We then further analyze the components of SweRV that do not exist in simpler cores, identify new problems, and discussed possible solutions in software or hardware.
– Based on this analysis, we formulate more general rules for designing masked software that takes into consideration properties such as the pipeline length, the amount of execution units, or architectural buffers. We also present arguments why relying on the *lazy engineering* approach alone, as proposed by [2], does not seem viable anymore in case of more complex CPUs.
– Finally, we present a case study that compares how efficiently our derived software constraints can be met with different implementation strategies. Maybe somewhat surprisingly we show that, with knowledge about a processors netlist, one can build secure and efficient masked software for SweRV-like cores with overhead as low as 13%.

Outline In Section 2 we cover relevant background on masking and the verification of masking, including the basic working principles of Coco and Tornado.

In Section 3, we describe the evaluation setup for the analysis of more complex CPUs with Coco, present some initial verification results and describe the significance of these in a practical evaluation. In Section 4, we present a detailed analysis of SweRV architecture, describe all hardware components that can pose problems to masked software implementations and propose viable solutions. In Section 5, we list the generic software constraints and evaluate their overhead in Section 6. We conclude our work in Section 7.

Open Source We plan to publish both, our modified SweRV core, as well as the corresponding software implementations that are used in this paper on github [3].

2 Background

2.1 Masking

Masking has become one of the first-choice measures to defeat power-analysis attacks on algorithmic level. In general, masking is a secret-sharing technique which splits intermediate values of a computation into $d+1$ shares. The shares are uniformly random, such that an attacker who observes up to d shares cannot infer any information about the underlying native value. A dth-order Boolean masking scheme splits a native variable s into $d+1$ random shares $s_0 \dots s_d$, such that $s = s_0 \oplus \dots \oplus s_d$. The values $s_0 \dots s_{d-1}$ are chosen uniformly at random while $s_d = s_0 \oplus \dots \oplus s_{d-1} \oplus s$. Consequently, each share s_i is uniformly distributed and statistically independent of the native value s.

Implementing linear functions when designing masked cryptographic implementations is trivial, as they can simply be computed on each share individually. However, non-linear functions (S-boxes) are not as simple, since computations involve multiple shares of a native value at the same time, which is more difficult to implement in a secure and correct manner. Therefore, the main interest in literature lays on masked implementations of non-linear functions [6,9,14,22,23,25,34]

2.2 Formal Verification of Masking

In general, masked implementations must ensure that each intermediate value of a computation is statistically independent of any native values. The verification of this property is usually done with the help of a security model that specifies the abilities of an attacker. Typically, it is assumed that the ability of the attacker is to place a certain amount of probes in a computation, that allow monitoring concrete values at those locations.

Formal Verification of Hardware Implementations. The *classical probing model* by Ishai et al. [25] is the most commonly used security model for masked hardware circuits and it's accuracy in modeling real world attacks has been confirmed by many works [18,35]. Here, an attacker is allowed to place up to d probes at any

[3] https://github.com/barbara-gigerl/sw-masking-swerv

location in a circuit, which can be used to observe the corresponding gates/wires permanently. A masked hardware implementation is considered dth-order secure if an attacker cannot learn any information about the native values by combining all d observations. Examples of tools that can verify classical probing security for cryptographic hardware implementations are are REBECCA [13], Silver [26], and maskVerif [3]. These tools are mainly tailored to the verification of masked hardware (ASIC/FPGA) implementations. maskVerif does offer some support for software implementations but (1) can only deal with code that is written in a special intermediate language, and (2) only considers simple CPU side-effects such as register overwrites.

Formal Verification of Software Implementations. On software side, the research community has also published many methods and tools to automatically generate or verify masked software implementations [4, 5, 8, 17, 29, 45]. More recently, Belaïd et al. proposed Tornado [10], a tool that takes a high-level description of an unmasked cryptographic function, generates a corresponding (any-order) masked C implementation, and verifies its probing security. Tornado's verification itself is based on tightPROVE+, an extension of tightPROVE [9]. tightPROVE+ performs the verification of masked software in the *register probing model*. This model allows an attacker to place probes on individual words of a processor's register file, and to use them for one cycle each during the execution of a masked software implementation. Hereby, it is assumed that the probed registers cause independent leakage, in other words, no additional potential side effects of a processors architecture, such as glitches or register overwrites, are considered [33].

More precise verification tools, that e.g. also cover transition leakage have been presented in [1,7,40], while with COCO, Gigerl et al. have recently presented a tool that can verify the correctness of masked software implementations while considering possible architectural side effects of a given CPU netlist [20].

2.3 Coco

COCO is a tool for the co-design and co-verification of masked software implementations on CPU netlists [20]. It formally verifies the security of (any-order) masked assembly implementations that are executed on concrete CPUs, defined by gate-level netlists. COCO's verification strategy is inspired by REBECCA but extended in a way such that the verification of masked software, when running on hardware, is converted into a pure hardware verification problem. This involves not only the addition of control-flow awareness but also several performance improvements since entire CPUs are usually significantly larger designs, when compared to pure hardware implementations of cryptographic functions. COCO does not only capture transition-based effects, but in principle any glitch-related hardware side-effects that can be derived from a CPU netlist. This is also formalized in the so-called *time-constrained probing model*, in which an attacker can use each probe to measure any specific gate/wire for the duration of one clock cycle that can be chosen independently for each probe.

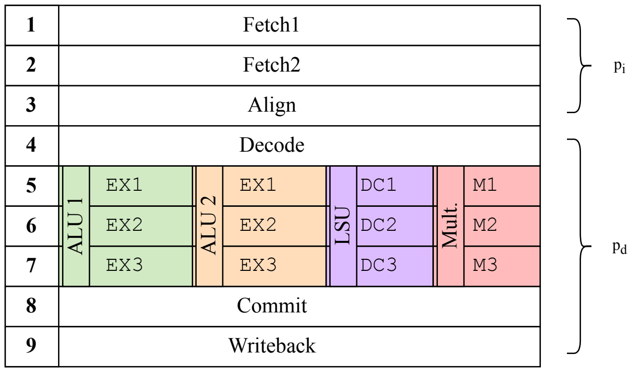

Fig. 1: Pipeline stages of SweRV [42]

Verification Flow In the following, we briefly outline the workflow of Coco, broken into multiple steps. Steps **A** and **B** explain how the execution of software can be combined with an otherwise purely hardware-focused verification method. Step **C** then describes the application of Coco in a bit more detail.

A. Yosys [44] is used to parse the given CPU design into a gate-level netlist. The masked assembly implementation together with the netlist is then given to Verilator [36], which produces a cycle-accurate simulation of the execution in form of an *execution trace*. The execution trace contains concrete values of all CPU control signals during the software execution.

B. Registers or memory locations in the CPU netlist receive annotations (labels) that indicate the location of shares and randomness at the start of the software execution.

C. The CPU netlist, execution trace, initial labeling and desired verification order is given to the verifier, which propagates the labels through the CPU netlist, for as many cycles as the software execution takes. In case Coco detects that a specific gate in the netlist leaks information about a native value (by observing a combination of shares of the same native value), e.g. due to implementation mistakes or architectural side-effects, the exact gate in the netlist and the execution cycle is reported as a leak. For a more detailed description of this verification method we refer to the original publication [20].

2.4 RISC-V SweRV Core

The SweRV processor family [37] was first introduced by Western Digital in 2019 and designed for data-intensive applications like storage controllers and industrial IoT. As of today, there are three different variants of the processor: the EH1, the EH2 and the EL2 [43]. The EH1 features a 32-bit superscalar 9-stage pipeline, while the EH2 basically adds a second thread with a dedicated register file and instruction fetch buffer. The EL2 is a smaller version of the EH1 with only 4 pipeline stages and one execution unit.

In our experiments, we use the SweRV EH1 core[4], which implements the RISC-V RV32IMC instruction set and has nine pipeline stages [42], as sketched in Figure 1. The first three pipeline stages (Fetch1, Fetch2, Align) are responsible for loading instructions from the instruction memory and storing them into the fetch buffer. In the Decode stage, the instructions are decoded and prepared for execution. The execution happens in pipeline stages 5-7, either in the Load-Store unit (DC1,DC2,DC3), the multiplication unit (M1, M2, M3) or the ALUs (EX1, EX2, EX3). The EH1 core has a dual-issue pipeline, which means that in each clock cycle, the processor can decode two instructions and send them to two different ALUs. In the last two pipeline stages, Commit (EX4) and Writeback (EX5), the final result is stored in the register file. There are several peripherals attached to the core via an AXI4 bus, including the SRAM and instruction and data closely-coupled memories. The core operates in-order, except for loads which might get executed earlier when the value is needed in the pipeline.

According to Western Digital, the SweRV EH1 core can be operated at frequencies of up to 1 GHz [41] and its performance can be compared to an ARM Cortex A15, making it outperform other RISC-V processors like the Berkely BOOM core [38]. This makes EH1 an interesting target to analyze the effects of more complex CPU architectures on masked software implementations. Another reason why we chose SweRV EH1 is Coco's current requirement of CPU designs to be written in Verilog or SystemVerilog.

3 Generic Masked Software on SweRV

In this section we perform an initial analysis of generic (higher-order) masked software implementations on the SweRV EH1 core and show that, even after applying the same hardware modifications as proposed for Ibex in [20], a more complex CPU architecture introduces additional problems that can reduce the protection of masked software by serveral orders. In Section 3.1, we describe a few small hardware modifications that we carry over from Gigerl et al.'s *secured* Ibex to SweRV, that would otherwise lead to identical problems on SweRV. In Section 3.2 we describe modifications we made to Coco's verification flow itself so that it can better handle CPU designs that are significantly larger than Ibex. In Section 3.3, we generate generic, up to 4th-order masked software implementations of the Keccak S-box using Tornado, verify their execution on the *secured* SweRV using Coco, and conclude that there is little hope that such implementations can achieve even just 1st-order protection. Finally, we present additional empirical evidence of the impact of architectural glitches on masked software via several gate-level timing simulations in Section 3.4.

3.1 Modifications of SweRV

Gigerl et al. have analyzed the simple 32-bit RISC-V Ibex core in terms of software masking-related side effects. As a result of their analysis, they pointed out

[4] https://github.com/chipsalliance/Cores-SweRV

three hardware components that can cause unintended combinations of shares during the execution of masked software implementations that are completely invisible from software perspective: the register file, the Arithmetic Logic Unit (ALU), and the Load-Store unit (LSU). Not surprisingly, the SweRV core has similar problems, which is why we briefly discuss how we map these proposed hardware fixes from IBEX to the SweRV core in the following. The resulting *secured* SweRV core will then serve as the base of our further analysis. We expect that the total area overhead of the hardware modifications for the SweRV core is very similar to the IBEX core as analyzed in [20], which was about 2 kGe. Since the SweRV core is much larger, this overhead is insignificant.

We use SweRV core commit `499378d0c67ab11965` as the baseline for our modifications. For our analysis, we disable closely-coupled memories for instructions and data, but enable the instruction cache. We do this since (1) the instruction cache is large enough to hold all implementations that we intend to test, (2) we want to analyze the "worst-case" in which the CPU can fetch instructions without delay, thereby achieving the maximal possible amount of instructions (and side-effects) in the pipeline stages. Hence, when running a verification with COCO, we execute each software implementation twice, once to load it into the instruction cache from instruction memory, and once to perform the actual verification.

Register File Ordinary register file implementations consist of a group of register words (32×32bit for RV32IMC) plus addressing logic for reading two words and writing one word within one clock cycle. This addressing logic is usually implemented via multiplexer trees that select source and destination registers depending on the currently decoded instruction. As previously shown for IBEX, these selector signals are usually calculated by combinatorial logic within the same cycle as the actual read/write event. Consequently, within a single clock cycle, differences in signal propagation delays can cause glitches on these selector signals, which in return can cause a read/write port to unreliably switch between multiple register words until the selector signals at all multiplexers are stable[5]. This is problematic for masked software implementations as they hold many shares in the register file that must be kept strictly separated from each other.

The proposed solution for this problem is to replace multiplexer trees with OR trees while introducing a one-hot encoded gating mechanism for each value that is calculated in the previous clock cycle and buffered in a additional register [20]. This mechanism ensures that glitches on a read/write port can only ever happen between the operand of two consecutive instructions. In the SweRV core, we face the same problems and fix these by applying the same register gating concept. The main difference here is the fact that SweRV features four read and three write ports, compared to IBEX's two read and one write port. Gating the read and write ports for SweRV works almost straightforward, except

[5] Even if the selector signals were stable, e.g. by calculating and buffering them in the previous clock cycle, there is still no guarantee that this signal arrives at all multiplexers in stable condition in the next clock cycle due to different wire lengths.

for the third write port, which is used for data from the memory, and requires a dedicated solution (cf. Section 4.2).

Concurrent ALU Computations Cores like Ibex and SweRV always concurrently calculate simple operations like AND, XOR, ADD, SHIFT in the execution stage and later only forward the result that is actually needed by the currently executed instruction. This is not a problem for most masking techniques, however there do exist some masking techniques that store individual shares of the a native value in the same register word [6]. This is okay as long as all computations keep the individual bits of a register word separate from another, e.g., by performing only bit-wise operations such as AND and XOR. Operations such as ADD or SHIFT on the other side do combine bits of individual operands and can thus create side-channel leakage, even if the results of these computations are ultimately discarded.

The suggested solution for the Ibex core is to implement a gating mechanism that ensures that only the intended computation is performed. This mechanism can also be easily carried over from Ibex to SweRV.

Data Memory Storing shares in the data memory leads to similar problems with glitches in the addressing logic as for the register file. In theory, one could again use the same one-hot encoded gating mechanism as discussed before, however, this approach does not scale well for the large address ranges that are required for data memory. Consequently, Gigerl et al. propose a trade-off that consists of using only partially one-hot encoded addresses for data memory which can be implemented with an area overhead that is indeed negligible when compared to the area of SRAM blocks themselves. The downside of this trade-off is that only memory words within certain address ranges (blocks) are properly separated from each other. This is sufficient as long as a block is large enough to hold all the shares that need to be kept isolated from each other during the execution of masked software implementations.

We apply the same LSB one-hot address encoding to SweRV's data memory. Since the SweRV core reads 64 bit from the memory in one cycle instead of 32-bit, we gate memory words on 64-bit granularity.

3.2 Modifications of Coco

In this section we briefly outline modifications that we have made to Coco's verification workflow so that it can better handle large CPU designs. These modifications first and foremost reduce SweRV's circuit size which in return also significantly reduces Coco's verification runtime.

Removal of Unused Logic As mentioned before, we ensure that instructions can be directly loaded from the instruction cache during Coco's verification. We only ever use the slower instruction memory in a read-only fashion to fill the instruction cache and can, for the pure purpose of Coco's verification, remove any unused logic that would allow writes to data memory, which reduces the circuit size by about 29%.

Control Wire Tagging The initial version of Coco effectively treats each wire of a CPU netlist equally and does not distinguish control from data wires. In reality, only a small fraction of wires can actually affect the data that is processed by a masked software implementation in such a way that side-channel related problems could occur. Therefore, we adapted Coco such that it is possible to tag wires as explicit *control* signals. During the verification, Coco will then simply ignore these wires instead of applying the laborious process of constructing empty SAT equations for them. Clearly, this tagging needs to be done carefully such that we do not later overlook any architecture side-effects during Coco's verification. Since manual tagging of individual wires is infeasible for entire CPU designs, we instead only do this in a course-grained manner and only in cases where we can easily deduce that there will be no consequences for the processed data of software with constant (data-independent) control flow. More precisely, we tag the instruction memory, instruction cache and signals depending solely on those as control signals automatically.

3.3 Initial Analysis of the SweRV Core

In this section we present our initial analysis of several higher-order masked software implementations on the *secured* SweRV core that already includes all hardware modifications that were proposed in the previous analysis of Ibex [20]. First, we use Tornado to generate generic, up to 4th-order masked C implementations of the Keccak S-box that are formally verified in Tornado's register probing model, meaning that an attacker observing up to d intermediate values (of the algorithm) is not allowed to learn information about native values. We then analyze the execution of these implementations on SweRV using Coco to get an impression of how many more issues can be detected in Coco's time-constrained probing model, in which an attacker, able to observe up to d wires/gates in the CPU netlist throughout one clock cycle each, is not allowed to learn information about native values.

Since Coco can only deal with assembly implementations by default, we create an assembly wrapper around the Tornado-generated C functions and adapt the work flow accordingly. We then analyze these implementations using Coco, while targeting the verification of 1st-order protection. Unfortunately, the verification results show that none of the tested implementations can even reach just 1st-order protection. Upon first inspection of the reported problems, we can see that multiple additional issues still exist within SweRV that can significantly reduce the protection order of our tested software implementations. For example, the forwarding logic in SweRV's 9-stage pipline is reported by Coco as one of the main culprits for the loss of multiple protection orders in the time-constrained probing model.

3.4 Empirical Evaluation

In order to empirically confirm the problems identified by Coco in the SweRV core, we perform and analyze gate-level simulations in this section. More con-

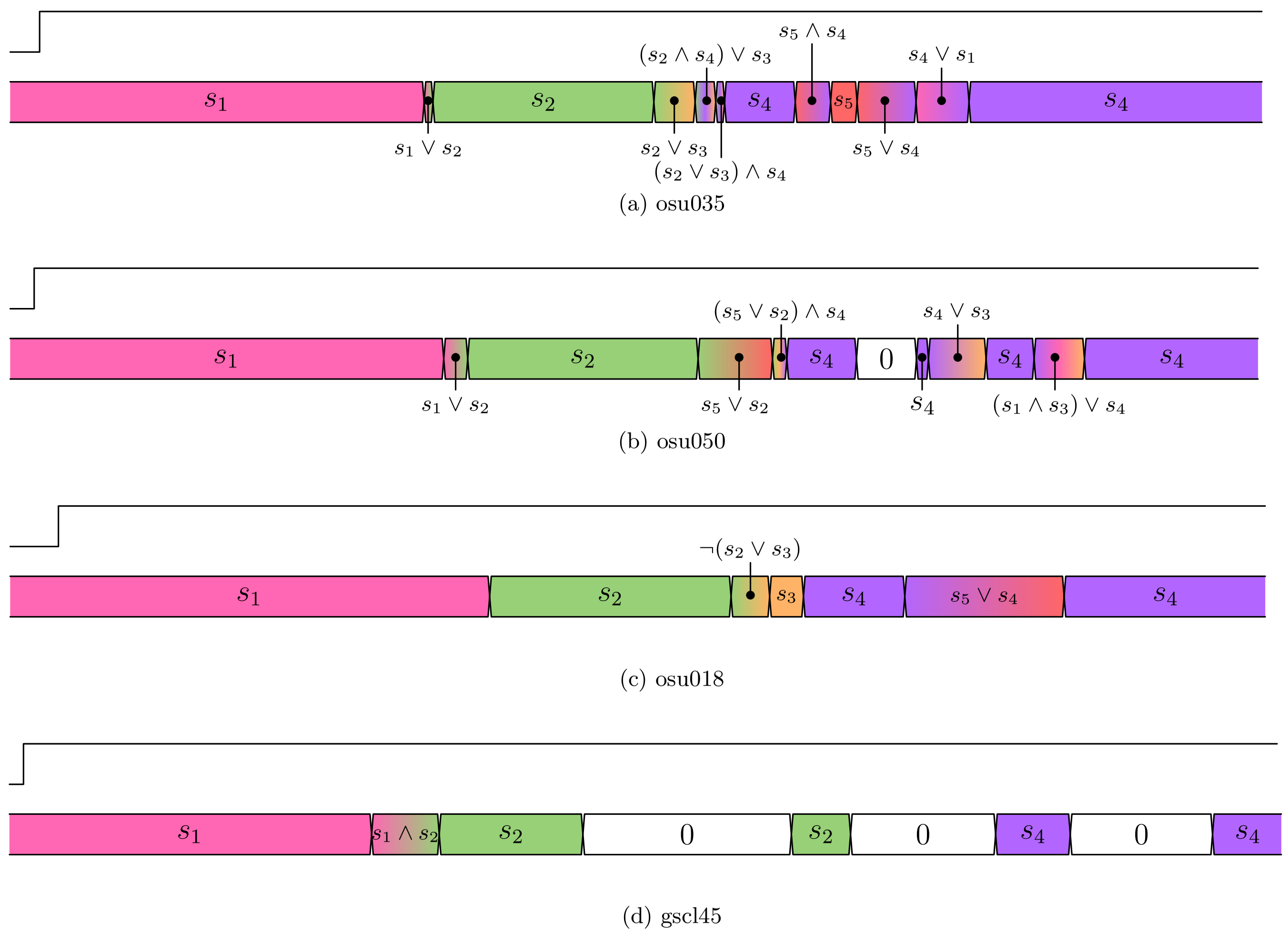

Fig. 2: Value of wire `fwd_data` at the beginning of a clock cycle for four different standard cell libraries. $s_1 \ldots s_5$ denote up to five of the shares that are visible in this experiment due to architectural glitch effects. Since the analyzed time window in each plot is different (due to different propagation delays) we have applied suitable horizontal scalings to improve readability. The time before the rising clock edge is always 10ps.

cretely, we perform gate-level timing simulations of the forwarding logic within SweRV's pipeline (see Figure 3) using multiple cell libraries to better illustrate how problems in the time-constrained probing model can relate to practical problems. Our evaluation reveals that glitches in the forwarding logic can lead to independent occurrences of up to five shares on one wire within one clock cycle, and combined occurrences of up to three shares at the same time. We note that while the exact behaviour of glitches strongly depends on the used standard cell library, all of our tested standard cell libraries report leaks leading to a reduction in the masking order between three and five.

Setup We use signal traces from the post-synthesis simulations of the SweRV core netlist. The synthesis process maps logic gates in the netlist to suitable cells in the standard cell library, which defines the exact behavior and delay of each cell. We investigate and compare four different open-source cell libraries[6], osu035, osu018, osu050, and gscl45nm. The mapping process is performed by Yosys [44], before running the simulation with Modelsim to obtain an execution trace of our test program.

The same test program is used in all four evaluation scenarios. The test program works with a native value split into 10 shares, which corresponds to a 9th-order masked implementation. First, the test program executes 10 instructions, each operating on exactly one share. This effectively stores each share to its own register in a specific pipeline stage. Second, the test program executes an instruction referring to a previously computed result, which sends the shares in the pipeline registers to the bypass logic, which finally forwards the correct share to the ALU. It should be noted that the program is correctly masked on algorithmic level because exactly one share is processed per instruction.

Results Figure 2 shows what information an attacker can observe by probing the wire `fwd_data` in SweRV's forwarding logic for the duration of one clock cycle using different cell libraries. Each plot additionally shows the corresponding clock signal and contains marks that indicate at which point in time a specific share (or combination of shares) is visible until the value of the wire has stabilized. Since the analyzed time window in each plot is different (due to different propagation delays) we have applied suitable horizontal scalings to improve readability.

From these plots we can see that an attacker can always observe at least three shares (Figure 2d), and at most five shares (Figure 2a-c) within one clock cycle when probing the `fwd_data` wire. Sometimes, shares do not appear independently, but also in combination with other shares. For example, in Figure 2a, the attacker first observes s_1, and then s_1 in combination with s_2. Note that both, the occurrence of multiple shares independently within one cycle, or the occurrence of combinations of shares at any point in time breaks the assumption of independent leakage.

Clearly, this evaluation is not exhaustive. Every technology, every cell library, and every different placement of a design, leads to different timing properties

[6] https://github.com/RTimothyEdwards/qflow/tree/master/tech

and differences in the exact leakage. Also concrete ASIC or FGPA prototypes are just instances of particular configurations. The exact quantification of the leakage, i.e. determining the number of traces that are needed for exploitation in a particular configuration, is not in the scope of this paper. In fact, it is also not clear if it would be possible to find a representative configuration and setup that would allow more than making a statement on leakage for one particular realization in one particular setup. A worst case setup would be a library with delay settings that lead to the observation even all 10 shares in a single clock cycle.

Instead of focusing on more specific instances, the focus of our analysis in this section was on showing that problems identified using Coco actually lead to critical signal transitions in the design. Given the empirical confirmation of critical signal transitions, we therefore use Coco as a reference for the identification of critical design elements in the design of SweRV. With Coco we are able to formulate a generalized statement about the security of a masked software implementation in the time-constrained probing model, which is independent of a specific technology or platform.

4 Analysis of Problems on SweRV

As shown in our previous analysis of generic (higher-order) masked software implementations, the hardware components of more complex CPUs can cause a significant reduction in the protection order. In this section, we discuss these problematic components in the *secured* SweRV core that already has the Ibex-patches applied (cf. Section 3.1). We divide these problems into ***big*** and ***small*** problems, based on how many shares may be combined, since, as we show later in Section 5, one can follow different strategies to deal with them. A component causes a ***big*** problem when more than two shares can be potentially combined. A ***small*** problem indicates that a component can combine at most two shares. For each potential leakage source, we discuss the options of making further modifications in hardware or shifting this problem as a constraint to masked software implementations.

4.1 Pipelines and Execution Units

The dual-issue SweRV EH1 core features nine pipeline stages and can process two instructions per clock cycle. Accordingly, the fetch/decode stages (1-4) can handle multiple instructions at the same time, the execution/writeback stages (5-9) exist twice, while the lesser used multiply (5-7) and load/store stages (5-7) exist only once (c.f. Figure 1). The dual issue design also requires a register file with four read ports and three write ports. Since symmetric cryptographic software implementations are usually implemented with constant (data independent) control flow, which is also the case for all our tested software implementations, only the later execution/writeback stages (5-9) get in touch with actual data and can thus cause potential side-channel related problems.

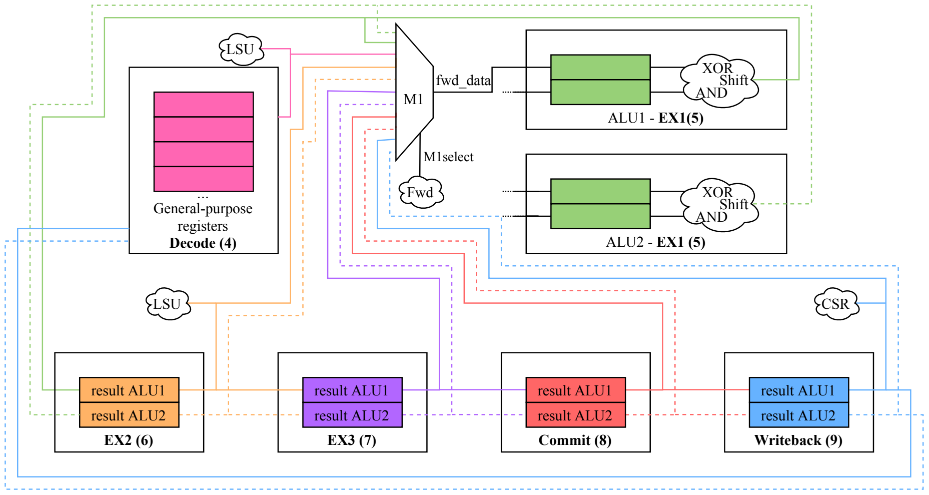

Fig. 3: Pipeline stages 4-9 in SweRV. Shares reside in the register file (🟪), are then sent to the ALU (🟩) before being buffered in pipeline registers (🟧, 🟪, 🟥, 🟦). Forwarding values from the pipeline registers to the ALU is possible in each stage and handled by the multiplexer M1, and the respective select signal M1select.

A typical optimization in pipelined CPU designs is the usage of forwarding logic, also known as bypass-logic, that can redirect the result of an instruction from a later pipeline stage to a previous stage without needing to wait for the result to be written into the register file. Forwarding significantly reduces the occurrence of pipeline stalls in cases where one instruction uses an operand that was only just calculated by the previous instruction. In the context of masking, this architectural design causes problems in two different points.

Figure 3 shows a simplified depiction of SweRV's pipeline stages 4-9. The multiplexer M1 is responsible for selecting which data is used as input for EX1, the first of the execution stages. This data either comes from the register file (GPR), the (LSU), or from any of the later execution stages due to forwarding logic. The select signal of this multiplexer, M1select, is computed in the respective pipeline stage from combinatorial logic and is therefore susceptible for glitches. Consequently, an attacker probing the output of M1, `fwd_data`, could, in the worst case, observe all of M1's possible inputs within one clock cycle until the select signal stabilizes. This means that if multiple shares of the same native value are in different pipeline registers, a combination of those can be observed at `fwd_data` in the same clock cycle. On top of that, since two different instructions are executed by SweRV at the same time, `fwd_data` can also combine data from the other execution unit. Exactly this problem was also seen in our empirical evaluation in Section 3.4.

In software, special care is also needed for control transfer instructions like conditional jumps. The instructions `beq`, `bne`, `blt` and `bge` perform conditional branches on data but are typically not used in (symmetric) cryptographic implementations to avoid potential timing side channels. Still, they can be used together with the unconditional jump instructions `jal` and `jalr` to implement loops or function calls. In the context of masking, these instructions can cause problems which are invisible via e.g. the control flow graph of the software. Since the SweRV core decodes two instructions per cycle, the jump is potentially decoded with the instruction which comes code-wise after it. If this instruction operates on shares, and there are still shares of the same native value in the pipeline, a leak occurs, before the CPU realizes a change in the instruction pointer caused by the branch two cycles later. The instructions in these two cycles must be unrelated instructions, which requires in total four unrelated instructions.

Possible Hardware Solutions One could first consider to solve this problem in hardware by using a trick similar to the one used to prevent unintended glitches in the multiplexer tree of the register file. For example, one could gate the output of each pipeline register with a bit indicating whether the respective value should be forwarded back to the first execution stage (5) or not. This would further require individual gate-bits to be glitch-free, i.e., to be computed in the previous clock cycle and buffered in a register. The problem with pre-computing gate-bits is that those values are typically only available in the same cycle like the forwarding signal. One can overcome this problem by introducing additional pipeline stages in between the execution stage, however, this would significantly impact the overall performance of the core, also in cases where ordinary non-masked software is executed. Since we do not consider such a performance degradation to be a viable option, we next explore if those problems can better be dealt with on software-level.

Possible Software Solutions For a masked software implementation to not be affected by the side-effects of SweRV's forwarding logic, it must ensure that at no time there are two or more shares corresponding to the same native value in any execution stage of either execution unit. For example, if we consider the execution of two instructions, each of which uses a different share of the same native value, then one would need to ensure that there are $2 \times 6 + 1$ unrelated instructions between them. Hereby, an amount of 6 instructions is needed to clear all execution stages (5-9) of one execution unit, that then has to be doubled since SweRV has two execution units in total. Unrelated instructions are instructions processing data unrelated to any share (for example a `nop`), or shares from another native value.

While such a software constraint can significantly decrease the performance of masked software implementations, it is a solution that does not impact the performance of ordinary non-masked software. Nevertheless, as we will show later in Section 6, it is still possible to implement efficient masked software implementations fulfilling this constraint if the right masking/implementation techniques are used.

Software Constraints for ALU Operations

- *(Pipeline Stages and Execution Units)* Two instructions using different shares of the same native value must be separated by $6 \times 2 + 1$ unrelated instructions.
  *Combination of up to **13** shares possible (**big** problem).*
- *(Control transfer instructions)* Control transfer instructions, which are preceded by instructions processing shares, must be followed by 4 unrelated instructions.
  *Combination of up to 4 shares possible (**big** problem).*

4.2 Management Components of Data Memory

The SweRV core manages communication to the data memory via the Load-Store unit (LSU). The LSU is a component between the CPU and the memory to ensure low memory latency by providing buffers and a dedicated pipeline for store operations. Our analysis shows, that the LSU Bus Buffer, responsible for saving values of recent loads or stores, similar to a data cache, turns out to be a major source of leakage which potentially combines multiple shares (***big*** problem). Furthermore, the dedicated store pipeline, components in the data memory interface, back-to-back memory accesses and the dedicated register file write port for memory accesses potentially combine two shares (***small*** problems). For each of these problems, we discuss possible solutions in hardware and software.

LSU Bus Buffer Since data memory is connected to the SweRV core over an AXI4 bus, which can potentially introduce a considerable amount of latency, the LSU implements the so-called the LSU Bus Buffer, which works in principle like a small data cache. The LSU Bus Buffer consists of eight elements that are used to temporarily store the values of recent load or store events. Each element additionally stores the target address, an age, and a state, since the LSU uses a state machine to manage the buffer entries. Initially, all element states are set to *Idle*, meaning that they are ready to receive data, and their age is set to 0. While executing the memory access, the state and age are updated accordingly, until the memory access is finished and the element enters the *Idle* state again. However, the element is not removed from the buffer until the buffer is full and the oldest element is overwritten.

In the context of masked software implementations two problems arise in the LSU Bus Buffer. First, it is problematic if one share of a native value in the buffer and is overwritten with its counterpart, which might happen, e.g., when loading two shares from the data memory. This is not only a problem for load operations within short succession but can also occur if these operations are far apart since buffer elements are not cleared once their state goes back to *Idle*. Second, if multiple shares of the same native value are stored in the buffer at the same time but at different locations, one can observe similar side-effects as originally described for ordinary register files (c.f. Section 3.1). The second

```
# Reset state/age of buffer elements
fence
# Load share 1 from address 0x20
lw x1, 0x20
# Reset state/age of buffer elements
fence
# Dummy overwrite of buffer element 1
lw x0, (x0)
# Reset state/age of buffer elements
fence
# Load share 2 from address 0x40
lw x2, 0x40
```

Fig. 4: Example of flushing the LSU buffer to clear it from shares

problem could in principle be solved in hardware by applying a similar gating mechanism as for the the register file. However, in case of the LSU buffer, such a solution requires an additional register layer for pre-computing stable one-hot encoded signals, which decreases the performance of all software.

Instead, we can solve this problem on software-level by ensuring that the buffer holds at most one share per native value, which additionally prevents the problem of overwriting shares. When doing so, we could ideally target individual elements of the buffer such that a share can easily be overwritten with dummy data whenever needed. However, the LSU buffer is completely invisible from a progammer's perspective, which is also why there is no easy way to manipulate specific elements from software side. The choice of which buffer element is overwritten is determined by the element age, which depends not only on time, but also instruction dependencies, data addresses and the element state. While one could formulate a software constraint that takes all of these factors into consideration, we do not consider this a worthwhile solution due to complexity. Instead, we propose a software solution utilizing RISC-V's `fence`, that while introducing a 2-5 cycle overhead for each usage, is significantly easier to use in a correct way. In general, a `fence` can be used to ensure a certain order of memory operations by stalling the CPU pipeline unless previous load/store operations are finished. For the LSU Bus Buffer this means that all buffer elements are set to *Idle* with age 0. However, the stored values are not cleared, which must be done manually by executing load/store instructions dealing with unrelated data. The `fence` ensures that these loads and stores are inserted consecutively into the buffer, i.e., starting at the first slot and ending at the last slot, finally overwriting all buffer elements. It is recommended to place a second `fence` after this load/store sequence, before loading or storing further shares. Figure 4 shows a short exemplary code snipped, in which a share is stored in the buffer and later cleared.

Store Pipeline Stages Before being stored to memory, data values pass through three dedicated pipeline stages in the LSU (c.f. Figure 1), which are exclusively updated when a store happens. A share used as an operand in a store instruction will therefore hang in the pipeline until it is overwritten by the data of the next store. This is problematic if the data of the next store is a share from the same native value. In order to avoid this problem, it is sufficient to ensure that at least one unrelated store operation is performed between two stores that transfer two shares of the same native value.

Data Memory Interface SweRV reads 8 bytes from the external data memory module in one cycle, and then selects the parts which are effectively needed according to the load address and load instruction (`lw`, `lh` or `lb`). Glitches in the selection signal can lead to problems if two shares of a native value are stored in the same 8-byte data memory word. A hardware gating mechanism is again not viable since it would increase latency, which is why we suggest to store shares of the same native value not within the same 8-byte word.

Back-to-back Memory Accesses SweRV is able to execute memory accesses in a back-to-back fashion, i.e., in two consecutive cycles. Given a data memory layout that utilizes partially one-hot encoded addresses (c.f. Section 3.1), an additional problem can occur if shares s_i, s_j are stored stored in block $b1$ at indices i, j respectively, and one accesses unrelated data at indices i, j in another block $b2$ in two consecutive cycles. The the output of $b1$, even though ignored, will switch from s_i to s_j in two consecutive cycles. Preventing this kind of leakage can be done by paying special attention to the block indices during each memory access, or reserving one "neutral" index within blocks that never holds any shares and thus can be used for inserting a dummy load.

Register file gating for Data from Memory Register file write ports of the SweRV core need to be gated by a stable gate bit (c.f Section 3.1). Computing the gate bit is straightforward for all write ports except for the one dedicated to data loaded from memory, since it depends on a potentially glitching write enable signal derived from the LSU bus buffer entries. First, we gate the write data with the stable register write address only, which means the preliminary gate bit is set for all registers which have loads pending in the LSU bus buffer. In the next cycle, the write enable value is then used to select the final, correct write register. This solution requires the software to ensure that no other pending load in the LSU bus buffer writes to a register, which contains another share from the same native value.

Software Constraints for Memory Operations

– *(LSU Bus Buffer)* Two memory accesses processing two shares must be separated by a `fence`, followed by a load of unrelated data, followed by a `fence`. *Combination of up to **8** shares possible (**big** problem).*
– *(Store Pipeline Stages)* Two stores storing two shares must be separated by a `fence`, followed by a store of unrelated data, followed by a `fence`. *Combination of up to **2** shares possible (**small** problem).*

– *(Data Memory Interface)* Shares must be stored in the same memory block, but not within an 8-byte word.
  *Combination of up to **2** shares possible (**small** problem).*
– *(Back-to-back memory accesses)* Either one 8-byte region per block at index i is not used to store shares and between any two loads, a load to this region is performed, or if a share s_i is stored at index i and s_j is stored at index j in a block, no back-to-back accesses to any addresses mapping to index i and j are performed.
  *Combination of up to **2** shares possible (**small** problem).*
– *(Write port 2 of the register file)* If a share s_i is stored in register x_i and s_j is stored in memory, then there must not exist another load at the same time which writes to register x_i.
  *Combination of up to **2** shares possible (**small** problem).*

5 Deriving Generic Software Rules

In this section, we propose generic rules for the design of masked software implementations that are intended to run on more complex CPUs like SweRV. These rules take into account features like pipeline length, the number of execution units, and the size of load/store buffers, and are based on the software constraints defined in Section 4. We also discuss the *lazy engineering* approach by Balasch et al. [2] and demonstrate that, while entirely relying on this approach in our setting is not recommended, it can still be a worthwhile trade-off that can eliminate many smaller problems, that would otherwise all need be dealt with in software.

5.1 Generic Rules for Masked Software

A CPU can be described by numerous characteristics, ranging from the architecture width to register file size to cache sizes. Our analysis in Section 4 shows that, when considering the implementation of masked software implementations, the following characteristics are especially important:

– The amount of pipeline stages p
– The amount of execution units e
– The size of data buffers.

Pipelines and Execution Units Forwarding logic, also known as bypass-logic, is a common optimization in pipelined CPUs which we identify to be a big problem for masked software implementations. In the worst case, each pipeline stage forwards its current content to the first stage, where it can be effectively combined with data from all stages due to glitches. Assuming a pipeline length $p = p_i + p_d$, where p_i is the number instruction fetch stages and p_d is the number of decode/execute stages (processing actual data), this problem can be avoided by ensuring that at least $p_d + 1$ unrelated instructions are executed between any two instructions processing the shares of the same native value. We have observed this problem on the SweRV core ($p_i = 3$, $p_d = 6$) but it also affects simpler cores such as the CV32E40P (formerly known as RI5CY) that is roughly comparable to an ARM Cortex M4 [31]. This core features a 4-stage pipeline ($p_i = 1$, $p_d = 3$), and would therefore still require a "padding" with four unrelated instructions.

On top of that, more powerful CPUs like SweRV often feature a superscalar architecture, including e.g. a dual-issue pipeline, that allows executing two instructions per clock cycle. This is achieved by having e execution units in parallel, all of which have their own fetch/decode/execute stages. In those cases, forwarding is not only possible between stages of the same execution unit but also across them. This additionally increases the required amount of padding to $e \times p_d + 1$.

Data Buffers and Caches Besides pipeline stages, another big problem for masked software implementations is the existence of data buffers that are invisible from a programmers perspective. Defining generic rules for these components is somewhat harder as their exact behavior can differ quite a lot depending on their concrete implementation. However, typically when considering SweRV's LSU buffer or many other cache designs, these components can cause shares to essentially get stuck at certain locations within the CPU where they then represent an additional source of leakage from this time onward. While such problems can be resolved in hardware, e.g. as shown for the register file (c.f. Section 3.1), this is only really a viable option in cases where these hardware modifications do not increase latency, which is also why we need to deal with SweRV's LSU buffer side effects in software. In general one needs to ensure that, whenever a share is transferred over an unmodified buffer, none of the other buffer entries contain shares that correspond to the same native value. How this can be achieved is implementation dependent. In the easier case, a mechanism to clear the buffer contents could be implemented in hardware, which is however not always efficient since it would also affect unmasked data. In the harder case, one has to make use of dummy loads/stores to clear all unwanted values.

Rules Here we summarize the most important rules for masked software on application-level processors. As we explain in Section 5.2, many of the other smaller problems are probably better dealt with using the "lazy engineering" approach.

`R1` Two instructions processing shares from the same native value must be separated by $e \times p_d + 1$ unrelated instructions.
`R2` Whenever a share is transferring through a buffer, none of the other buffer entries must contain shares that correspond to the same native value.

Naturally, at this point one could also ask how these rules would look like on even more complex CPUs with multi-level caches, out-of-order execution, or speculative memory accesses. For example, the 64-bit out-of-order RISC-V BOOM core would be a potential target for further analysis. However, when considering the analysis of such CPUs we currently see quite a few problems that are not necessarily easy to overcome. First, out-of-order execution will violate our assumption of having software with constant control flow, meaning that verifying a program's execution once might not be indicative of future runs. Second, the effects of, e.g., large cache hierarchies will likely cause problems where corresponding software constraints would become too complex to implement with reasonable effort and overhead. Nevertheless, we argue that physical attacks like power analysis are most relevant only for devices in the range from microprocessors to application processors. An attacker having physical access to a desktop/server could anyway use other methods, like cold boot attacks, to compromise a system more efficiently.

5.2 The Cost of Lazily Engineering

Until now, the verification of masked software implementations is mostly done using rather simple security models like the value-based or register-based leakage model. While such models are certainly useful to detect some problems, many other works also show that processors do emit leakage that is not captured by these models [2, 19, 21, 28, 32]. Balasch et al. [2] formalize this behaviour in their order-reduction theorem, which states that on simple CPUs, the security of dth-order masked software in the value-based leakage model reduces to $\lfloor \frac{d}{2} \rfloor$-th order in the transition-based leakage model. In other words, as a rule-of-thumb for a "lazy" software engineer, they suggest to double the security-order of a masked implementation to achieve the desired security-order in a model that more accurately reflects reality.

While these works focus on rather simple microprocessors, our analysis has shown that on more complex application-level processors, the reduction of security order can be significantly higher. When deriving the expected security reduction of *lazy engineering* on application-level processors, the main point to consider is the component that can potentially combine the most shares. In the case of our modified SweRV this component would be the forwarding logic of the CPU pipeline. According to our generic rules, a processor executing algorithmically correct masked software, might combine up to $e \times p_d + 1$ shares in its pipeline. Consequently, without any further assumptions, the CPU could create all combinations of any choice of $e \times p_d + 1$ shares, which corresponds to an order reduction of $\left\lfloor \frac{d}{e \times p_d + 1} \right\rfloor$.

To give a concrete example, when relying entirely on lazy engineering, one would in theory require at least a 13th-order masked implementation for actual 1st-order security on SweRV in the time-constrained probing model. While we do not expect an easily exploitable order reduction this large when performing physical power measurements of SweRV (c.f. Section 3.4), we also want to stress that these architectural side effects should not be underestimated. For example, works like [28] show that, already on simple microprocessors, a generic 2nd-order masked software implementations can very well loose both of its protection orders in practice. If we add to that the fact that SweRV's architecture has the potential to unintentionally combine many more shares at many more locations, one can expect that quite a few masking orders will also be required in case

			Plain Implementations		Secure Implementations			
Name	Input Shares	Fresh Randomness	Cycles	Instr-uctions	Cycles	Instr-uctions	NOPs	Verification Runtime
Tornado-generated Implementations								
ISW Keccak S-box	10×32 bit	5×32 bit	163	330	-			
ISW Keccak S-box, 2nd order	15×32 bit	15×32 bit	1272	810	-			
ISW Keccak S-box, 3rd order	20×32 bit	30×32 bit	2124	1121	-			
ISW Keccak S-box, 4th order	25×32 bit	50×32 bit	4406	3309	-			
AND Gate Implementations								
DOM AND *reg.* [23]	4×32 bit	32 bit	10	8	33	48	40	1.4 m
ISW AND *reg.* [25]	4×32 bit	32 bit	10	8	32	48	40	57 s
TI AND *reg.* [30]	4×32 bit	-	14	15	37	54	39	1.1 m
Trichina AND *reg.* [39]	4×32 bit	32 bit	9	8	34	46	38	1.28 m
DOM AND *reg.*, 2nd order [23]	6×32 bit	3×32 bit	20	21	86	148	127	3.2 m
DOM AND *reg.*, 3rd order [23]	8×32 bit	6×32 bit	33	42	250	295	235	9.6 m
Serial/Parallel Implementations								
DOM Keccak S-box *reg.*, serial [24]	10×32 bit	5×32 bit	83	95	240	418	333	8.4 m
DOM Keccak S-box *reg.*, parallel	10×32 bit	5×32 bit	36	60	81	144	79	3.7 m
DOM Keccak S-box, serial [24]	10×32 bit	5×32 bit	174	140	550	624	464	22.38 m
DOM Keccak S-box, 2nd order, serial	15×32 bit	15×32 bit	283	250	2050	1465	283	1.5 h
Threshold Implementations								
TI Keccak S-box, *reg.*	15×32 bit	-	66	105	72	126	15	3.5 m
TI Ascon (1 round)	15×64 bit	-	721	863	1621	1153	290	1.18 h

Table 1: Runtime comparison of masked software implementations on the SweRV core. Plain implementations do not consider software constraints, and thus lose all protection orders. Secure implementations are handcrafted for SweRV, consider all required constraints, and can thus preserve their claimed protection order. NOPs indicate the required amount of `nop`'s or dummy loads/stores. Testcases marked with *reg.* do not perform any memory accesses, i.e., all data is in the register file at the beginning/end of the computation.

of SweRV. Given that masking imposes a runtime overhead that is quadratic in the masking order, such very high-order implementations might however still not be a desirable solution, especially in automotive applications with real-time requirements. As we show later, in such cases, we recommend utilizing lazy engineering only for eliminating small problems while tackling big problems using more effective implementation/masking strategies that we describe in Section 6.

6 Evaluation

In this section we demonstrate that, despite the fact that cores like SweRV can cause significant problems for masked software implementations in general, one can still design fine-tuned, secure versions with very small overhead. First, we explain how one can use a *parallel* instead of the usual *serial* coding strategy to significantly reduce the performance impact of software constraints that require a separation between processing shares of the same native value. We then explain how one can utilize Threshold masking schemes, and by extension also the core idea of lazy engineering, to design masked software for SweRV that is secure, efficient, and easy to implement. More concretely, we show that the runtime overhead of e.g. a masked Keccak S-box implementation providing 1st-order security on SweRV, when compared to a corresponding implementation ignoring all software constraints, can be as little as 13%.

Evaluation Setup All of our tested implementations are hand-written assembly code, except for the Tornado-generated C implementations that are compiled with the compiler flag `-O1`. For the verification and performance benchmarks we used the cycle accurate simulation of SweRV's netlist within Coco. Coco itself was executed on a 64-bit Linux operating system on an Intel Core i7-7600U CPU with a clock frequency of 2.70 GHz and 16 GB of RAM. We configure SweRV with data memory ranging from 256 byte to 2 KB, adapted as required by the respective testcase. The instruction memory and instruction cache is configured to be 2 KB for each test case.

The SweRV configuration using 256 byte of data memory, after applying the optimizations described in Section 3.2, results in a circuit with 420 000 gates, of which 108 000 are registers and 97 000 are non-linear gates. A detailed breakdown of these numbers can be found in Appendix B. This makes the hardware design of SweRV orders of magnitudes larger than the Ibex design which was studied in [20], and consisted of only 27 000 gates.

Software Implementation Package To measure the overhead imposed by different software constraints, we construct a comprehensive set of masked software implementations. First, we take a look at several examples of masked And gates, which represent the simplest non-linear function (degree 2). More concretely, we analyze 1st-order implementations of the Ishai-Sahai-Wagner (ISW) And [25], the Trichina And [39], the Threshold Implementation (TI) And [30], and up to 3rd-order masked variants of the Domain Oriented Masking (DOM) And [23].

We then investigate masked S-box implementations which represent the non-linear layer within symmetric cryptographic computations, and use masked And-gates as basic building blocks. Here, we focus on 1st- and 2nd-order masked implementations of the Keccak S-box, which has a prominent use in the SHA-3 hash function. Furthermore, we provide TI variants of the Keccak S-box [11], as well as one complete round (linear + non-linear layer) of the Ascon cipher [16].

In Table 1 we list *plain* implementations, which are correct in the value-based leakage model, but do not consider any of SweRV's software constraints, and are thus also not secure on this core when verified by Coco in the time-constrained probing model. In contrast, *secure* implementations fulfill all required constraints can thus be verified successfully for their claimed protection order on SweRV. For each implementation, we report SweRV's execution runtime in cycles, as well as the number of executed instructions. Additionally, for secure implementations, we report the number of unrelated instructions (NOPs), that are needed to achieve the required amount of time separation between the processing of shares of the same native value, as stated by the individual software constraints.

6.1 Serial vs. Parallel Implementations

Many modern symmetric cryptographic primitives have a mathematical description based on simple Boolean functions that can be easily mapped into a corresponding software/hardware implementation. For example, the Keccak S-box (as used in SHA-3) operates on a state consisting of five *lanes*, each of which is combined with two other lanes using a sequence of simple AND, XOR, and NOT operations to compute the corresponding output lane. The most straightforward way of implementing this S-box in software is to take a set of three lanes, processing them, storing the resulting output lane, and repeating these steps until the computation of all five output lanes is finished.

If we now consider a masked implementation, where each input/output lane is represented by two (or more) shares, the same implementation strategy can be used, except for the fact that the sequence of Boolean operations needs to be adapted such that (1) shares of the same native value (lane) are never directly combined, (2) the (native) output is still the same. If we further consider a software constraint that requires a certain amount of unrelated instructions between processing shares of the same native value, one can imagine that additional `nop` instructions will need to be introduced for this purpose. Alternatively, one could consider a *parallel* implementation, where one interleaves the computation of the five output lanes such that `nop`'s can be replaced with computations on shares of other lanes. We give an example that illustrates the runtime difference between serial and parallel implementations in Appendix A. This runtime difference is also quite visible in Table 1. For example, the parallel DOM Keccak S-box implementation (81 cycles) is three times faster than its serial counterpart (240 cycles).

One potential downside of parallel implementations is the fact that they increase the maximum amount of intermediate values that need to be kept track of. Especially in case of higher-order masked implementations, a processor's register file might not be large enough to hold this increased amount of intermediate values. The resulting register spilling then requires additional load/store operations that also need to comply with software constraints and can thus eliminate any potential gain of this approach. To illustrate the overhead of memory operations, we have included a serial implementation of the Keccak S-box that initially loads all shares from memory and computes the S-box. If we compare the runtime of this implementation (550 cycles) to the serial implementation that performs computations without intial memory operations (240 cycles), we can observe a runtime overhead of about factor two.

6.2 Threshold Implementations

Threshold implementations (TI) [30], is a provable secure masking scheme that splits non-linear functions into multiple incomplete *component functions*. More concretely, in TI, each component function fulfills the non-completeness property, meaning that its computation is independent of at least one of its input shares. One consequence of incompleteness is that TI schemes require computations with at least three shares in order to provide 1st-order security. At the same time, this incompleteness guarantees that any combination of intermediate values during the computation of one component function can combine at most two out of three shares of any native value, therefore leaving 1st-order security intact.

In the context of implementing secure and efficient masked software implementations for SweRV, TI turns out to be beneficial in two ways. First, the "lazy" characteristic of TI allows us to ignore all ***small*** problems that can combine at most two shares. Second, a TI description of Keccak, for example as shown in [12], also gives a description of three S-box component functions, each of which only contain instructions that operate on an incomplete set of shares. Hence, when implementing TI Keccak in software, one can calculate the linear layer in sequence for each share, and the non-linear layer in sequence for each component function. Then, one only really needs to pay attention to ***big*** problems when switching the calculation from one component functions to another. This significantly simplifies the software development process as ***big*** problems can only really occur twice per Sbox computation.

In Table 1, we show a TI implementation of the Keccak S-box (72 cycles) which has almost no overhead compared to the corresponding plain implementation (66 cycles). Compared to a plain parallel DOM implementation, the overhead of a secure TI implementation is still only a about a factor of two, while being at lot easier to implement. With TI Ascon, we also present runtimes of implementations that compute an entire cipher round (linear + non-linear layer). The choice of using Ascon for this comparison is motivated by the fact that Ascon uses a S-box very similar to Keccak, and a linear layer that is significantly easier to implement in assembly than in case of Keccak. From the reported numbers we can see that only 290 additional `nops` are needed to make this implementation conform to the required software constraints. While the cycle count of the secure implementation is still about twice as large as in the plain case, we want to stress that most of this overhead (≈ 900 cycles) is due to software constraints for data memory since three shares of Ascon's state do not quite fit into the register file anymore.

7 Conclusion

In this work, we have performed a comprehensive analysis of more complex CPU architectures in the context of masking-related side effects. First, we showed that on cores like SweRV, there exists a significant gap between security in a simple software probing model and practical security for masked software. We underlined this point both via a formal analysis in the hardware probing model and via empirical analysis based on gate-level timing simulations. We then further analyzed the components of SweRV in the hardware probing model, identified new problems, and discuss possible solutions in terms of software constraints. Ultimately, while there exist many hardware components that can reduce the security of masked software due to architectural side-effects, we show that there only exist a few components that could reduce the security of masking schemes by multiple orders. Hence, when considering the implementation of efficient masked software for such CPUs, we recommend to use a combination of TI/lazy engineering to deal with small problems while only addressing the few large problems directly in the software implementation. In that case, the performance overhead of software constraints can be as low as 13% while the resulting implementation can be fully formally verified on our *secured* SweRV in the hardware probing model. If 2nd-order protection is desired, one could again rely on TI/lazy engineering for small problems, here however the additional cost of this approach might not justify this convenience anymore. When aiming for even higher protection orders, one likely needs to consider all software constraints directly in the implementation to keep the runtime overhead manageable.

Acknowledgements

This work was supported by the TU Graz LEAD project ”Dependable Internet of Things in Adverse Environments”, and the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia, and via the FERMION project (grant nr 867542).

References

1. Athanasiou, K., Wahl, T., Ding, A.A., Fei, Y.: Automatic Detection and Repair of Transition-Based Leakage in Software Binaries. In: Christakis, M., Polikarpova, N., Duggirala, P.S., Schrammel, P. (eds.) Software Verification - 12th International Conference, VSTTE 2020, and 13th International Workshop, NSV 2020, Los Angeles, CA, USA, July 20-21, 2020, Revised Selected Papers. Lecture Notes in Computer Science, vol. 12549, pp. 50–67. Springer (2020)
2. Balasch, J., Gierlichs, B., Grosso, V., Reparaz, O., Standaert, F.: On the cost of lazy engineering for masked software implementations. In: Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers. Lecture Notes in Computer Science, vol. 8968, pp. 64–81. Springer (2014)
3. Barthe, G., Belaïd, S., Cassiers, G., Fouque, P., Grégoire, B., Standaert, F.: maskverif: Automated verification of higher-order masking in presence of physical defaults. In: Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11735, pp. 300–318. Springer (2019)
4. Barthe, G., Belaïd, S., Dupressoir, F., Fouque, P., Grégoire, B., Strub, P.: Verified proofs of higher-order masking. In: Oswald, E., Fischlin, M. (eds.) Advances in

Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9056, pp. 457–485. Springer (2015)

5. Barthe, G., Belaïd, S., Dupressoir, F., Fouque, P., Grégoire, B., Strub, P., Zucchini, R.: Strong non-interference and type-directed higher-order masking. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 116–129. ACM (2016)
6. Barthe, G., Dupressoir, F., Faust, S., Grégoire, B., Standaert, F., Strub, P.: Parallel implementations of masking schemes and the bounded moment leakage model. In: Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10210, pp. 535–566 (2017)
7. Barthe, G., Gourjon, M., Grégoire, B., Orlt, M., Paglialonga, C., Porth, L.: Masking in fine-grained leakage models: Construction, implementation and verification. IACR Cryptol. ePrint Arch. **2020**, 603 (2020)
8. Bayrak, A.G., Regazzoni, F., Novo, D., Ienne, P.: Sleuth: Automated verification of software power analysis countermeasures. In: Bertoni, G., Coron, J. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2013 - 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8086, pp. 293–310. Springer (2013)
9. Belaïd, S., Benhamouda, F., Passelègue, A., Prouff, E., Thillard, A., Vergnaud, D.: Private multiplication over finite fields. In: Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part III. Lecture Notes in Computer Science, vol. 10403, pp. 397–426. Springer (2017)
10. Belaïd, S., Dagand, P., Mercadier, D., Rivain, M., Wintersdorff, R.: Tornado: Automatic generation of probing-secure masked bitsliced implementations. In: EUROCRYPT (3). Lecture Notes in Computer Science, vol. 12107, pp. 311–341. Springer (2020)
11. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: The keccak reference (2011)
12. Bilgin, B., Daemen, J., Nikov, V., Nikova, S., Rijmen, V., Assche, G.V.: Efficient and first-order DPA resistant implementations of keccak. In: CARDIS. Lecture Notes in Computer Science, vol. 8419, pp. 187–199. Springer (2013)
13. Bloem, R., Groß, H., Iusupov, R., Könighofer, B., Mangard, S., Winter, J.: Formal verification of masked hardware implementations in the presence of glitches. In: Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II. Lecture Notes in Computer Science, vol. 10821, pp. 321–353. Springer (2018)
14. Cnudde, T.D., Reparaz, O., Bilgin, B., Nikova, S., Nikov, V., Rijmen, V.: Masking AES with d+1 shares in hardware. In: Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9813, pp. 194–212. Springer (2016)
15. Coron, J., Giraud, C., Prouff, E., Renner, S., Rivain, M., Vadnala, P.K.: Conversion of security proofs from one leakage model to another: A new issue. In: Constructive Side-Channel Analysis and Secure Design - Third International Workshop,

COSADE 2012, Darmstadt, Germany, May 3-4, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7275, pp. 69–81. Springer (2012)

16. Dobraunig, C., Eichlseder, M., Mendel, F., Schläffer, M.: Ascon v1.2. submission to the ceasar competition (2016), https://ascon.iaik.tugraz.at/files/asconv12.pd. Retrieved on February 4th, 2021
17. Eldib, H., Wang, C., Schaumont, P.: Formal verification of software countermeasures against side-channel attacks. ACM Trans. Softw. Eng. Methodol. **24**(2), 11:1–11:24 (2014)
18. Faust, S., Rabin, T., Reyzin, L., Tromer, E., Vaikuntanathan, V.: Protecting circuits from leakage: the computationally-bounded and noisy cases. In: Gilbert, H. (ed.) Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Monaco / French Riviera, May 30 - June 3, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6110, pp. 135–156. Springer (2010)
19. Gao, S., Marshall, B., Page, D., Oswald, E.: Share-slicing: Friend or foe? IACR Trans. Cryptogr. Hardw. Embed. Syst. **2020**(1), 152–174 (2020)
20. Gigerl, B., Hadzic, V., Primas, R., Mangard, S., Bloem, R.: Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs. 30th USENIX Security Symposium, USENIX Security 2021 (2021)
21. de Groot, W., Papagiannopoulos, K., de la Piedra, A., Schneider, E., Batina, L.: Bitsliced masking and ARM: friends or foes? In: Lightweight Cryptography for Security and Privacy - 5th International Workshop, LightSec 2016, Aksaray, Turkey, September 21-22, 2016, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10098, pp. 91–109. Springer (2016)
22. Groß, H., Mangard, S.: Reconciling d+1 masking in hardware and software. In: Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10529, pp. 115–136. Springer (2017)
23. Groß, H., Mangard, S., Korak, T.: Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. In: Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016 Vienna, Austria, October, 2016. p. 3. ACM (2016)
24. Groß, H., Schaffenrath, D., Mangard, S.: Higher-order side-channel protected implementations of KECCAK. In: Euromicro Conference on Digital System Design, DSD 2017, Vienna, Austria, August 30 - Sept. 1, 2017. pp. 205–212. IEEE Computer Society (2017)
25. Ishai, Y., Sahai, A., Wagner, D.A.: Private circuits: Securing hardware against probing attacks. In: Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2729, pp. 463–481. Springer (2003)
26. Knichel, D., Sasdrich, P., Moradi, A.: SILVER - statistical independence and leakage verification. In: Moriai, S., Wang, H. (eds.) Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12491, pp. 787–816. Springer (2020)
27. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: CRYPTO. Lecture Notes in Computer Science, vol. 1666, pp. 388–397. Springer (1999)

28. Meyer, L.D., Mulder, E.D., Tunstall, M.: On the effect of the (micro)architecture on the development of side-channel resistant software. IACR Cryptol. ePrint Arch. **2020**, 1297 (2020)
29. Moss, A., Oswald, E., Page, D., Tunstall, M.: Compiler assisted masking. In: Prouff, E., Schaumont, P. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7428, pp. 58–75. Springer (2012)
30. Nikova, S., Rechberger, C., Rijmen, V.: Threshold implementations against side-channel attacks and glitches. In: Information and Communications Security, 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4307, pp. 529–545. Springer (2006)
31. OpenHW Group: Openhw group cv32e40p user manual: Pipeline details, https://cv32e40p.readthedocs.io/en/latest/pipeline/, Retrieved on January 26th, 2021
32. Papagiannopoulos, K., Veshchikov, N.: Mind the gap: Towards secure 1st-order masking in software. In: Constructive Side-Channel Analysis and Secure Design - 8th International Workshop, COSADE 2017, Paris, France, April 13-14, 2017, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10348, pp. 282–297. Springer (2017)
33. Renauld, M., Standaert, F., Veyrat-Charvillon, N., Kamel, D., Flandre, D.: A formal study of power variability issues and side-channel attacks for nanoscale devices. In: Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6632, pp. 109–128. Springer (2011)
34. Reparaz, O., Bilgin, B., Nikova, S., Gierlichs, B., Verbauwhede, I.: Consolidating masking schemes. In: Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9215, pp. 764–783. Springer (2015)
35. Rivain, M., Prouff, E.: Provably secure higher-order masking of AES. In: Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6225, pp. 413–427. Springer (2010)
36. Snyder, W.: Verilator, https://www.veripool.org/wiki/verilator. Retrieved on February 2nd, 2021
37. Ted Marena, Western Digital: The journey of risc-v implementation (2019), https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/collateral/white-paper/article-journey-of-RISC-V-implementation.pdf, Retrieved on January 16th, 2021
38. The Regents of the University of California: Riscv-boom: The load/store unit (lsu), https://docs.boom-core.org/en/latest/sections/load-store-unit.html, Retrieved on January 27th, 2021
39. Trichina, E.: Combinational logic design for AES subbyte transformation on masked data. IACR Cryptol. ePrint Arch. **2003**, 236 (2003)
40. Wang, J., Sung, C., Wang, C.: Mitigating power side channels during compilation. In: Dumas, M., Pfahl, D., Apel, S., Russo, A. (eds.) Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019. pp. 590–601. ACM (2019)

41. Western Digital: Risc-v: high performance embedded swerv core microarchitecture, performance and chips alliance, https://riscv.org/wp-content/uploads/2019/04/RISC-V_SweRV_Roadshow-.pdf, Retrieved on January 16th, 2021
42. Western Digital: Risc-v swerv eh1 programer's reference manual, https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf, Retrieved on January 16th, 2021
43. Western Digital: Risc-v and open source hardware address new compute requirements (2019), https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/collateral/tech-brief/tech-brief-western-digital-risc-v.pdf, Retrieved on January 16th, 2021
44. Wolf, C.: Yosys open synthesis suite, http://www.clifford.at/yosys/. Retrieved on February 2nd, 2021
45. Zhang, J., Gao, P., Song, F., Wang, C.: Scinfer: Refinement-based verification of software countermeasures against side-channel attacks. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10982, pp. 157–177. Springer (2018)

Appendix A

```
# Shares lane 0: x2, x3
# Shares lane 1: x4, x5
# ...
# Randomness: x12, x13, x14, x15, x1
# Lane 0
not x17, x2
and x24, x17, x5
xor x24, x24, x12
and x25, x3, x4
xor x25, x25, x12
and x27, x17, x4
xor x27, x27, x24
and x28, x3, x5
xor x28, x28, x25
xor x27, x27, x10
xor x28, x28, x11
# Lane 1
not x17, x4
and x24, x17, x7
xor x24, x24, x13
and x25, x5, x6
...
#Lane 2
...
```

```
# Shares lane 0: x2, x3
# Shares lane 1: x4, x5
# ...
# Randomness: x12, x13, x14, x15, x16
# NOT
not x17, x2
not x18, x4
not x19, x6
not x20, x8
not x21, x10
#DOM-AND - Instr 1
and x22, x17, x5
and x23, x18, x7
and x24, x19, x9
and x25, x20, x11
and x26, x21, x3
#DOM-AND - Instr 2
xor x22, x22, x12
xor x23, x23, x13
xor x24, x24, x14
xor x25, x25, x15
xor x26, x26, x16
#DOM-AND - Instr 3
...
```

Fig. 5: Comparison between serial and parallel DOM Keccak S-box

Appendix B

	Raw circuit	Optimized circuit
Registers	108 129	108 043
Linear Gates	8 828	8 708
Non-linear Gates	133 415	97 222
Not-Gates	3 518	3 248
Multiplexers	335 294	203 107
Total	**589 188**	**420 332**

Table 2: Circuit size of the SweRV core (256 byte of data memory, 2 KB of instruction memory / cache) before and after optimization (Removal of unused instruction memory logic)