David Schaffenrath, BSc

Hardening the PULP Architecture against Side-Channel Attacks

MASTER'S THESIS

to achieve the university degree of

Master of Science

Master's degree programme: Mechanical Engineering and Business Economics

submitted to

Graz University of Technology

Supervisor

Prof. Dr. Stefan Mangard

Institute of Applied Information Processing and Communications

Mario Werner / Institute of Applied Information Processing and Communications
Frank Kagan Gürkaynak, Beat Muheim, Germain Haugou / Integrated Systems Laboratory ETH Zürich

Graz, April 2017

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

15.05.2017
Date

Schaffenrath
Signature

Abstract

During the last decades, securing embedded systems against attacks has been the topic of continuous research on the software, as well as on the hardware level. In particular, fault attacks on hardware implementations of cryptographic functions and their possible countermeasures have received a lot of attention. However, the equally pressing issue of countermeasures against fault attacks on processor cores received only moderate attention. It has been shown that adversaries are able to introduce faults which lead to the skipping or modification of instructions. This allows for example to circumvent critical security checks, extract sensible information, or gain privileged access to the device. In this work, a recently invented protection mechanism against fault attacks on processors, called Sponge-Based Control-Flow Protection, is implemented for the open-source `Ri5CY` CPU-core developed at ETH Zürich. It utilizes a cryptographic sponge, in combination with control-flow protection mechanisms originally developed for soft-error detection, to achieve control-flow integrity.

As an additional part of this work, the RISC-V privilege-ISA 1.9.1 and an MMU were implemented for the `Ri5CY` core. This allows the core to run seL4, an operating system with a formal proof of security, which was ported to the RISC-V architecture as part of this work. This helps to mitigate software attacks, which are not preventable by Sponge-Based Control-Flow Protection, such as attacks on the integrity and confidentiality of data.

Finally, cryptographic applications are often subject to side-channel attacks, which aim to extract secrets from the device by means of e.g. timing- or power-consumption measurements. While software countermeasures such as introducing random delay cycles or randomizing the sequence of processed data are possible, they are often insufficient. Several Keccak/SHA-3 peripherals, which are protected against side-channel attacks of different attack-orders by applying Domain Oriented Masking, were implemented.

Putting together all the pieces, they can be considered the basic building blocks of a security module. The security enhanced core, together with the SHA-3 peripherals,

were integrated and taped-out in an ASIC and the resulting area, runtime and code-size overhead were analyzed.

Contents

List of Figures

List of Tables

List of Acronyms

AES . . . . . .Advanced Encryption Standard
ALU . . . . . .Arithmetic Logical Unit
APB . . . . . .Advanced Peripheral Bus
ASIC . . . . . .Application-Specific Integrated Circuit
ASID . . . . . .Address Space IDentifier

CFI . . . . . .Control-Flow Integrity
CSM . . . . . .Continuous Signature Monitoring
CSR . . . . . .Control Status Register

DES . . . . . .Data Encryption Standard
DOM . . . . . .Domain Oriented Masking
DRAM . . . . .Dynamic Random Access Memory

EM-pulses . . .Electro Magnetic pulses
EX-stage . . . .Execute pipeline stage

GPSA . . . . . .Generalized Path Signature Analysis

IC . . . . . .Integrated Circuit
ID-stage . . . . .Instruction-Decode pipeline stage
IF-stage . . . . .Instruction-Fetch pipeline stage

IIS . . . . . .Integrated Systems Laboratory

IP . . . . . .Intellectual Property (core)

ISA . . . . . .Instruction Set Architecture

LRU . . . . . .Least Recently Used

LSU . . . . . .Load Store Unit

MMU . . . . . .Memory Management Unit

MXR . . . . . .Make eXecutable Readable

NIST . . . . . .National Institute of Standards and Technology

PC . . . . . .Program Counter

PD . . . . . .Page Directory (First level Page Table)

PLRU . . . . . .Pseudo Least Recently Used

PMP . . . . . .Physical Memory Protection

PPN . . . . . .Physical Page Number

PRNG . . . . . .Pseudo Random Number Generator

PSA . . . . . .Path Signature Analysis

PT . . . . . .Page Table

PTE . . . . . .Page Table Entry

PTW . . . . . .Page Table Walk (sometimes also Page Table Walker)

PUM . . . . . .Protect User Memory

RISC . . . . . .Reduced Instruction Set Computer

RNG . . . . . .Random Number Generator

RSA . . . . . .Rivest Shamir Adleman (a public-key cryptosystem)

SCA . . . . . .Side Channel Analysis

SCFP . . . . . .Sponge-Based Control-Flow Protection

SCM . . . . . .Standard Cell based Memory

SHA . . . . . .Secure Hash Algorithm

SRAM . . . . . Static Random Access Memory

TLB . . . . . Translation Lookaside Buffer

TRNG . . . . . True Random Number Generator

VCD . . . . . Value Change Dump (File Format)

VPN . . . . . Virtual Page Number

WB-stage . . . . Write-Back pipeline stage

Chapter 1

Introduction

1.1. Problem Statement

Embedded systems, smart cards and IoT devices are often exposed to physical attacks [54]. Some of the attacks are passive and non-invasive, like measurement of the power consumption or the electric emanation of a device. It was shown, that it is possible to recover secret keys and sensible data by means of such side-channel analysis (SCA) attacks [47, 41, 38].

Fault attacks, on the other hand, are semi-invasive or invasive physical attacks. An attacker usually uses her control over the voltage, clock-signals, reset signal, external memories, temperature and other external conditions to alter the device's behavior or make it malfunction [34, 42, 35]. E.g. in the case of smart cards the voltage, clock and reset signals are externally available and can thus easily be manipulated by an attacker. An adversary is also able to mount attacks that sound more exotic, such as depackaging the chip and utilizing lasers, EM-pulses or directly probe individual on-chip signals to achieve control over them. Although more expensive to perform, in certain application domains, like pay-TV systems, such attacks are considered economically feasible [65].

In the case of cryptographic functions, the introduced fault often causes the leakage of information that can be used to recover sensible information, such as device keys. Such fault attacks have first been applied on RSA [20], and were also applied to other prominent cryptographic algorithms such as AES [53], SHA-512 [60], or SHA-3 [11] to name prominent examples. Consequently a range of countermeasures have been proposed to secure cryptographic algorithms against attacks [18, 66, 32, 46].

However, while countermeasures against attacks on cryptographic algorithms received a lot of attention, the pressing issue of how to protect other parts of the system, such as the processor, against fault attacks, received a lot less research. Attacks on the control-flow

aim to introduce faults that aim to skip security checks, (partially) nullify registers, or execute privileged instructions without permission. Additionally, attacks on instruction and data memory pose similar threats.

The XBOX-360 hack is a good example of how to use glitches to circumvent security checks [1]. By applying a glitch on the reset line at the right point in time, it was possible to bypass a signature verification of a second-stage bootloader. This way a custom bootloader could be used that was able to run custom kernels. What most likely happened in this case, was that the reset glitch caused some of the processor's registers to be cleared. Thus, when applied at the right time, the *memcmp* function responsible for the signature check would appear to have returned zero and the check would pass.

Glitches can also be combined with other fault techniques. Korak *et al.* [43] showed that faults due to clock glitches can be made more effective by additionally varying the supply voltage at the time the glitch is introduced. By doing so it is even possible to change the voltage to target different parts of the circuit, because the change in propagation delay for logic-gates depends on the exact type of the cell. E.g. logic gates with multiple transistors in series are affected stronger by variations of supply voltage [33].

Another very simple, but obviously very powerful attack, is to tap the bus signals of an external memory. By doing so, basically arbitrary instructions and data can be inserted, skipped or modified, if no encryption is in place. By using the rowhammer attack developed by Kim *et al.* [37], bit flips in memory can be produced for a huge amount of DRAM chips in use today. This attack blurs the line between software attacks and physical fault attacks, and is especially concerning since the attack can be conducted remotely via Javascript execution [27].

The most powerful attacks are able to flip individual on-chip signals precisely by means of optical faults. Selmke *et al.* [57] showed that it is even possible to precisely control individual bits in SRAM cells.

1.2. Related Work

Several publications discuss the common solutions of adding redundant circuitry to make fault-attacks harder [46, 35, 34]. They also discuss the possibility of different approaches to prevent tampering, such as passive and active shields. However these tampering countermeasures do not provide full protection as demonstrated by Kömmerling and Kuhn [42].

Lalande *et al.* [44] presented a software-only countermeasure against fault attacks which is able to detect faults that jump over 2 consecutive C-code statements, but faults on the level of individual instructions cannot be detected reliably.

Hardware approaches to detect control-flow errors were first proposed in the context of soft-error detection. Building on the principles of Path Signature Analysis (PSA), Wilken

and Shen [71] developed Continuous Signature Monitoring (CSM). In their work, they rely on "signature"[1] checks that are inserted into the code. The signature is calculated at compile-time. A dedicated hardware monitor then computes the intermediate signature for every encountered instruction. When the signature-check instruction is encountered, the signature from the hardware monitor, which contains the information about which code has been executed, is compared against the reference signature computed at compile time. If they don't match, an error is raised. In order to allow different paths through the program, the signatures need to be "justified" before a merging point. Since detection of an error is only possible at the time of signature checks, the scheme was extended to also provide "horizontal signatures", and was termed Continuous Signature Monitoring. Horizontal signatures allow for continuous checking by simply associating a short reference signature of a few bits with every instruction. Again, when the horizontal signature bits calculated by the hardware doesn't match the reference signature, an error is raised.

Werner *et al.* [70] have analyzed the Generalized-PSA as well as the CSM scheme for their suitability in the setting of fault attacks. Their findings show, that CSM is very well suited as protection mechanism against fault attacks, when the attacker has only limited access to a certain number t of on-chip signals. Their analysis was performed for faults on the instruction stream. An underlying assumption is, that instruction memory is on-chip as an attacker would otherwise not be limited in how many bits of an instruction can be flipped.

All schemes that rely on explicit checks against some precomputed value, implicitly rely on the check itself to not be manipulated. An attacker only needs to, directly or indirectly, control just the 1-bit check result in order to thwart these protection mechanisms. Additionally when off-chip memory is considered, CSM can't provide any protection and additional measures, such as encrypting the off-chip memory, must be taken. In the case of cryptographic algorithms this was tackled *e.g.* by Yen *et al.* [72], but control-flow integrity measures still rely on explicit comparisons.

1.3. Fault-Attack Secure Processor Design

The major parts of this work can be summarized as follows and are further motivated below:

- Port of the seL4 operating system and necessary core modifications necessary to run it
- Implementation of a recently developed, yet to be published, control-flow integrity scheme called Sponge-Based Control-Flow Protection (SCFP), which provides protection against fault-attacks on the control-flow of a program.
- Side-Channel resilient Keccak peripherals.

[1] The term signature is used to describe the calculated checksum, not a cryptographic signature

It was desired to implement all of the components in the open-source Pulpino platform, respectively its `Ri5CY` core [62, 63], which serves as the basis for all modifications on the processor-core site. The `Ri5CY` core is a RISC-V core particularly suited for low power applications, featuring a four stage pipeline and several ISA extensions.

seL4. Looking at the security issues faced by embedded system designers in a top-down approach, it is first desirable to secure the software against remote attacks, as the software tends to be the easiest exploitable component. Obviously, building secure software is a very application specific task, also because what "secure" means is highly application dependent. However, in many scenarios it is desirable to use a platform that allows for isolation between tasks to achieve a separation of concerns, hence an operating system is desired. The seL4 micro-kernel, which has been formally proved to provide confidentiality and integrity of data is a good candidate for the software foundation of a secure platform [40]. In contrast to traditional operating system kernels, in which (potential) security vulnerabilities are commonly found, the formal proof conducted for seL4 provides a high assurance that its implementation is sound. This means that the kernel is free of e.g. classical memory handling errors (use-after-free, double-free, buffer-overflow, etc.), which account for the majority of found security vulnerabilities in traditional kernels such as the Linux kernel. Since no RISC-V port of the kernel existed that implemented sufficient functionality prior to this project, it needed to be written as part of this work. The initial work on a RISC-V port, done by Hesham Almatary [5, 6], for an older release of seL4 was extended for this purpose.

Core Modifications. The seL4 operating system relies on the concept of virtual memory and page-based memory management to provide isolation between processes and of resources. Since Pulpino, respectively its `Ri5CY` core didn't provide either an MMU nor any concept of a privilege mode which is required for seL4 to work, they also needed to be implemented. The RISC-V consortium is currently working on a privileged ISA specification [67]. Although the specification is still in flux, and during the course of this work three different versions (1.7, 1.9 and 1.9.1) were available, it made sense to stick to it as closely as possible, since coming up with a meaningful privilege ISA is a non-trivial task, which is demonstrated by the many iterations required by the RISC-V consortium. This has the additional benefit of not requiring any custom compiler modifications and the RISC-V GCC toolchain can be used.

Keccak. The implemented MMU together with a concept of privilege modes and privileged instructions, allow to run seL4 on the `Ri5CY` core. However, although now the provided platform can be used to write secure software, it is still vulnerable to physical attacks as stated above. Many secure systems rely on dedicated hardware implementations of cryptographic accelerators. These accelerators are designed to make physical attacks on the sensible cryptographic operation sufficiently difficult for the attacker, such

that she will search for another way to reach her goal. In the course of this work, the SCA-resilient Keccak peripherals from Gross *et al.* [26] were integrated into the system. They utilize the Domain Oriented Masking scheme [25] to mitigate SCA attacks.

SCFP. A secure operating system and cryptographic accelerators are not enough to secure the system against physical attacks. As mentioned, the control-flow of a program can be manipulated by a variety of attacks, from glitches over voltage variations to on-chip bit-flips. To mitigate attacks on the control-flow of the processor, a yet to be published scheme was implemented [69]. It uses a cryptographic sponge to encrypt/decrypt code and enters a secure state of random execution upon manipulation of the control-flow, instead of requiring explicit checksum or signature checks as it is done by state-of-the-art countermeasures. The approach ensures the integrity of the instruction stream from the instruction memory all the way to the instruction-decode stage. It can also protect against faults that don't influence the instructions directly, such as manipulation of branch decision signals. This scheme, called Sponge-Based Control-Flow Protection (SCFP), was implemented in the `Ri5CY` core.

As a final step that is missing to create a truly "secure element", data-integrity needs to be considered. This was not possible within the time-frame of this project and would need to be implemented in follow-up work. However, data integrity cannot be achieved without control-flow integrity, hence this work lays the necessary foundation.

In Chapter 2 the idea behind Sponge-Based Control-Flow Protection is explained and a brief description of Keccak is given. In Chapter 3 the implementations of MMU, privileged ISA, SHA-3 peripherals and SCFP are discussed. Chapter 4 presents the results of the ASIC implementation, e.g. in terms of runtime, memory, area overhead for the implemented SCFP, and Chapter 5 concludes this work.

Chapter 2

Theory / Background

2.1. Sponge-Based Control-Flow Protection

While countermeasures against corruption of the control-flow have been researched extensively [71, 23, 36], only very few publications have actually taken into account the possibility of introducing faults on purpose [10, 70]. Most countermeasures have hence been developed with a soft-error fault model in mind, in which faults occur randomly. This is obviously not the case with fault attacks, which can be as precise as flipping individual on-chip SRAM cells [57]. Building upon the principles of Continuous Signature Monitoring (CSM), a new control-flow protection scheme, called Sponge-Based Control-Flow Protection (SCFP) has been developed by Werner *et al.* [69].

Compared to state-of-the art countermeasures, it has a number of advantages:

- No explicit signature checks.
  A check with the signature results in a 1-bit value. Unless special measures are taken, an attacker just needs to gain control over this single 1-bit signal to thwart the whole scheme.
- Works in case of external memory.
  Inline Reference Monitoring (IRM) schemes rely on the correctness of the signature values [3]. It is hence highly inadvisable to store the instructions or signatures off-chip, where an attacker has full control over every instruction and signature.
- Signatures are no longer needed.
  The memory overhead can be drastically reduced without the needed signatures. Furthermore they also don't need to be fetched from memory any more, which results in smaller runtime overhead.

- Branch decisions can be protected.
  The CSM scheme neglects the problem of the branch decision being unprotected. Just like results of signature comparisons a branch decision is a 1-bit signal. An attacker that gets a hold on that signal can control conditional branches to her liking, without even caring about any signatures checks, since both the branch-taken as well as the opposite case are valid, this is basically undetectable.

The remainder of this section addresses how each of these advantages is achieved and what the trade-offs are.

2.1.1. Encryption/Decryption of a Program

In SCFP the whole program is encrypted at compile-time and decrypted at runtime using a cryptographic sponge function, which is a relatively new cryptographic primitive, developed by Bertoni *et al.* [15] and is used in many cryptographic applications today, most prominently in the winner of the SHA-3 competition KECCAK. Depending on the actual application of the device, the sponge-state might be big such as to provide actual encryption of instructions. E.g. by using a KECCAK-f[200] with 32-bit instructions, one would obtain a 168 bit capacity and thus a security of 84-bit [69]. This case is for example desirable if the instructions are stored in external memory, that is potentially under control of the attacker.

On the other hand, in the case of on-chip memory, a smaller sponge-state might be sufficient, e.g. a KECCAK-f[100] which would mean 68-bit capacity and thus 34-bit security. In such a case the encryption acts more as a scrambling than an encryption of cryptographic strength. However the probability of introducing a fault that would lead to a collision in the sponge state that would go undetected is low. Since not only any two collisions would do, the principle of a birthday attack doesn't apply. Given that the rate is under full control of the attacker (e.g. external memory), but the capacity isn't, the chance of producing a specific collision is bounded by 2^{-c} for a capacity of c bits.

Random Execution as Secure State Decryption of a modified ciphertext, or of ciphertexts given in the wrong order, results in the output of a random instruction which the attacker has no control over. Furthermore, the sponge capacity will be random as well. Unless the attacker is able to precisely control the whole sponge state, the error will further propagate and cause random instructions. Aside from possibly aborting the program due to the execution of the random instruction, the attacker will have no control over code execution. While random execution seems insecure on first thought, the probability for an attacker to gain any information from it are considerably low.

Even in modern very dense instruction sets, such as the RISC-V ISA, there is sufficient room for invalid instructions. Additionally, the system itself further constrains the amount of valid instructions, because some instructions are only executable in a certain

context/privilege-level, others would access non-existent memory and so on. In most cases (over 99.9% in our system as will be shown later) a random instruction thus leads to the triggering of an exception within a few cycles. Using exceptions is more robust than explicit signature checks, as the attacker needs control over different exception causes, if she wants to avoid trapping into exception routines continuously.

It should be noted, that even if the exception mechanisms cannot be relied on, the harm an attacker can cause can be easily limited by further hardware countermeasures. E.g. it is easy to imagine a gatekeeper mechanism in hardware, in which a certain sequence of instructions needs to be executed before a certain memory region becomes accessible for a limited amount of cycles. The configuration of an MMU would provide another well suited gate-keeping sequence.

By relying on the low probability of actually revealing sensitive information (which additionally must be detectable by the attacker as such), SCFP can get rid of the potentially vulnerable signature comparisons. Indeed, the signatures are not required at all now. This results in reduced code execution runtime and memory overhead.

Adjusting the Sponge State Similar to how signatures need to be adjusted ("justified") in CSM, the sponge-state needs to be "patched", such that the decryption is possible despite having different paths through the program. Since how this patching needs to be done depends on the used construction it is discussed in Section 2.1.2 .

2.1.2. The Sponge

The used sponge is a variation of the one used in APE [7]. The encryption in APE is performed as $C_i = f(P_i \oplus C_{i-1}||S_{C_i})$, where S_C is the sponge capacity. Likewise the decryption is done as $P_i = f^{-1}(C_i \oplus P_{i-1}||S_{C_i})$. A graphical depiction is given in Figure 2.1 . An obvious advantage of this, when compared to *e.g.* SpongeWrap [13] is, that the attacker can't influence the plaintext directly by manipulating the ciphertext. In constructions similar to SpongeWrap, a change in the ciphertext would lead to the same change in the plaintext ($\Delta C = \Delta P$), as can be seen in Figure 2.2 . This is not ideal, since an attacker can manipulate at least one instruction very precisely before the next one will be random. APE doesn't heave this weakness.

The used variation differs in that the input to the next transformation is not directly dependent on the previous plaintext during decryption, which can be seen in Figure 2.2 . This makes the whole construction act like a block cipher, instead of a stream cipher. In the APE-like construction only the state's capacity needs to be patched, since the state's rate doesn't have an influence on the following transformations. This allows to reduce the runtime as well as the memory overhead due to fetching the stored patches.

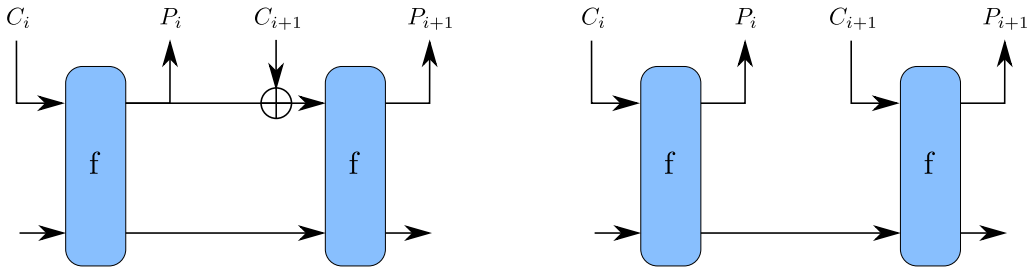

Figure 2.1.: Original (left) and modified (right) APE mode of operation with f as *de*cryption function

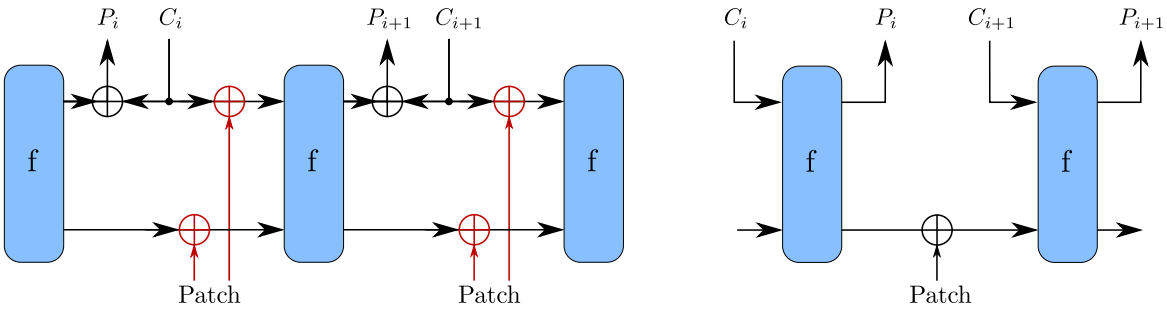

Figure 2.2.: Decryption and patching of the sponge-state in SpongeWrap (left) in comparison to the APE-like mode (right).

Patching the sponge state As briefly mentioned in Section 2.1.1 , the decryption requires to "patch" the sponge-state, similar to how signatures in CSM need to be adjusted. In contrast to CSM, the patching of the sponge-state when using the APE-like construction described above, must be done, whenever the control-flow diverges. This is a simple consequence of how encryption must be performed in a backwards fashion in APE in order to allow the decryption to be done forward. This means, that the control-flow graph (CFG) is traversed from the end to the beginning, and consequently a collision happens whenever the CFG diverges (hence the encryption paths converge).

In essence, during encryption the sponge-state of all paths that converge on a certain basic block are calculated. Either one of the calculated sponge-states, or a completely different one is used for further encryption of the program. To get to a common sponge-state, a collision needs to be produced. Doing so by means of trying to produce a collision after the transformation, is obviously not feasible, but this is not required. By calculating the difference between the sponge states, a patch value can be simply exclusive-ored onto the current sponge state, which allows to proceed with the encryption, as was already hinted in Figure 2.2 .

The full procedure from encryption to decryption is now explained for the simple case of an if-else statement. Figure 2.3 acts as an illustration.

The encryption is performed from the last to the first instruction. Hence it starts by

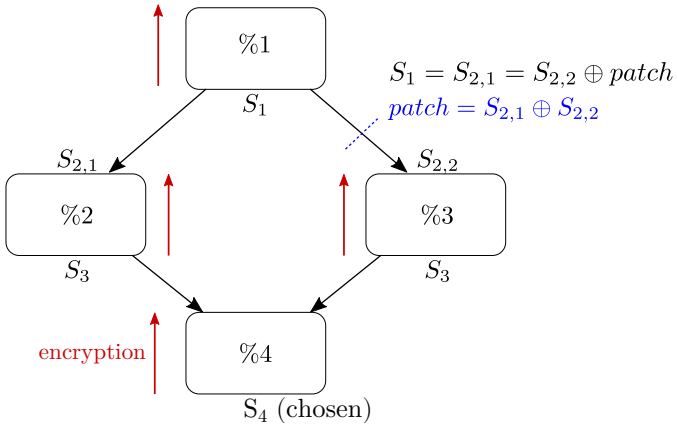

Figure 2.3.: Explanation of sponge-state patching for a simple if-then-else control-flow graph.

encrypting block %4. The encryption can proceed to encrypt blocks %2 and %3 in parallel. Note that the last encrypted instruction in %2, resp. %3, is dependent on the sequence of instructions in block %4, since the sponge capacity links them together. When starting to encrypt block %1, this can either be done with the sponge-state obtained from the encryption of %4→%3 or from the encryption of %4→%2. However, only one encrypted block %1 is desired, hence both encryption paths must be forced to the same state. This is done by means of the patch value as shown in Figure 2.3 . Either both edges (%1-%2, %1-%3) are patched such that the resulting state is the same, or one of the states is chosen as the "real" one and the other state is patched to the "real" state. Now that the state is the same, the encryption of %1 can continue.

During decryption, the patch value needs to be applied after the last instruction of block %1, to get to the sponge state at the beginning of block %2, resp. %3. For more complex control-statements a little more thought is required as to how this can be done practically, but the principle of patching at the edges of the CFG whenever there is a conflict in the sponge-state is exactly the same. How this is implemented is discussed in detail in Chapter 3 .

2.1.3. Choice of the permutation

The usage of the APE-like mode mandates, that the permutation must either be invertible or a block cipher must be used, effectively creating a keyed sponge [16]. For this prototype implementation the PRINCE block-cipher is used as transformation function.

PRINCE is a lightweight cipher especially designed for area constrained applications [21]. It is in principle possible to use an arbitrary block-cipher which can be used in the implemented APE-mode.

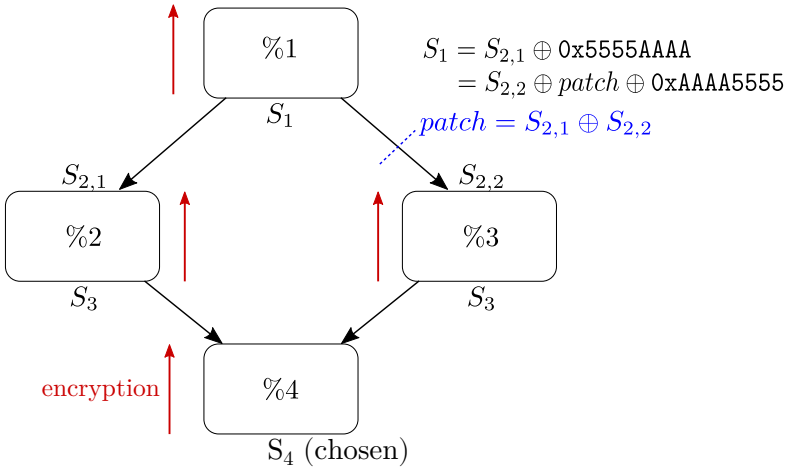

Figure 2.4.: Protecting the branch decision by linking the control-flow to the processed data.

Since we already had a working and tested version of the PRINCE cipher, it was an obvious choice. For future work, other interesting candidates for a lightweight block cipher might be PRESENT [19, 55], KLEIN [24], MCRYPTON [45] and HIGHT [29] to name just a few.

2.1.4. Protecting the branch decision

The branch decision of a conditional jump can be seen as a 1-bit value. Either the branch is taken, or not. In case an attacker gets hold of this value, either by direct manipulation with *e.g.* lasers or EM-pules, or by managing to skip the instruction in the EX stage without affecting any signals before the decoder, she can fully control which branch is chosen for execution. Although this is not as bad as arbitrary code execution, it still allows to skip vital security checks.

The solution in the SCFP scheme is to link the actual branch decision with the processed data that led to it. This can be achieved as follows. First, the decision of whether a branch gets taken or not is precomputed in software. The result of this computation is then stored in a register. The crucial part is, that, by design, there are only two possible values for this register, both of which are known at the time of encryption.

For instance, assume that the possible outputs of the software comparison are `0xAAAA5555` and `0x5555AAAA`. When the result of the software comparison is, to take the branch, it outputs `0xAAAA5555` otherwise `0x5555AAAA`. During encryption the sponge-state is adjusted on both CFG edges. This is depicted in in Figure 2.4 . At runtime, when the code is decrypted, the state is patched with the value in the register, regardless of the branch decision. When the branch is taken, an additional patch is needed, just as described previously.

In case an attacker now manipulates the branch decision, the sponge-state would still get patched with the register value that indicates the branch decision. Hence the decryption on the next instruction would fail, resulting in the start of random code execution.

Note that in the context of external memory, it is important that the value of the register (*e.g.* `0xAAAA5555` in the previous example) is unknown to the attacker. Otherwise it would be easy to craft the patch value such that it incorporates the register's value, which would then reduce the attack complexity to the control of the 1-bit branch decision signal again. To mitigate this, the patch value itself would need to be encrypted.

2.2. Protection of Cryptographic Accelerators Against Side-Channel Attacks

In this section, first the main properties of the Domain Oriented Masking (DOM) scheme, developed by Gross *et al.* [25], which is used to protect the KECCAK peripherals against side-channel-analysis (SCA) attacks, are shown. Afterwards a brief description of KECCAK is given. It is then explained how the KECCAK permutation can be protected against SCA-attacks by applying DOM to it.

2.2.1. Brief Description of the Masking Scheme

Domain Oriented Masking (DOM) is a masking scheme [59] that can be applied to protect cryptographic algorithms against SCA attacks [25]. It is secure in the d-probing model [31], which means, that tapping less than d signals would not be enough to extract any information about the processed data.

In contrast to threshold implementations, which rely on at least three component functions, as shown by Nikova *et al.* [51, 52], DOM only requires two shares, also called share-domains, for a first-order SCA resistant implementation, but is still secure in the presence of glitches, which have traditionally been a major source of leakage [48]. An additional benefit is, that in contrast to threshold implementations, which require a complete redesign, with DOM all the linear parts of a design just need to be replicated, and only the non-linear parts of the circuit require special attention and re-design. Furthermore, it is possible to extend the DOM-scheme to higher order protection in a generic manner, meaning that once the algorithm has been adapted for a first-order DOM implementation, it is possible to scale it to an arbitrary-order protection with no adjustments in the control-path and, dependent on how generically the design was described, with none to minimal adjustments to the data-path.

The DOM scheme requires an increased amount of randomness compared to a classical threshold implementation. Compared to the unprotected variant it also requires additional storage elements, but so does the classical threshold implementation. The fact that

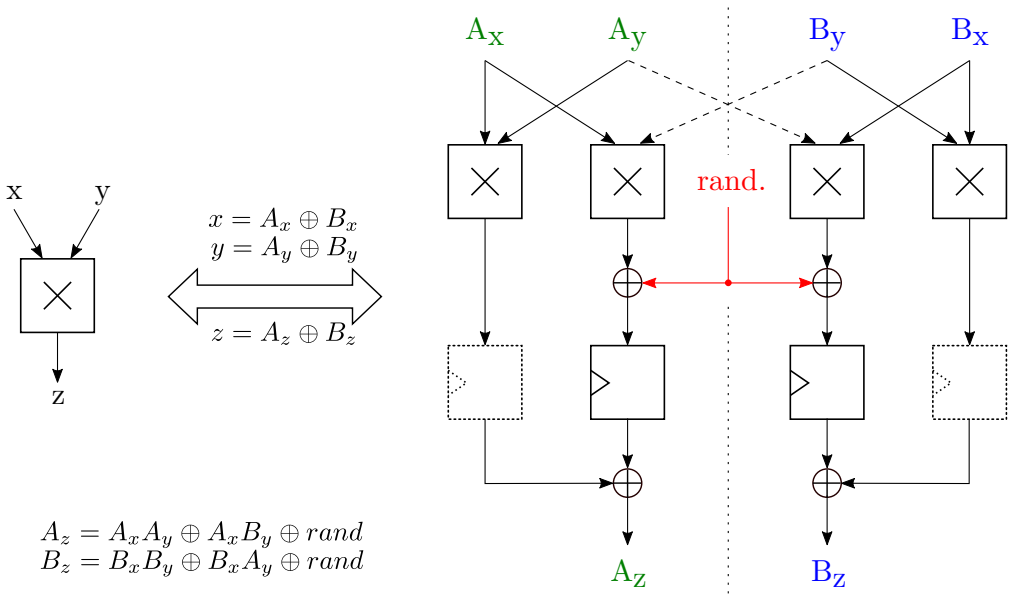

Figure 2.5.: A 1-bit multiplier (aka AND gate) in a 1^{st}-order DOM protected design.

classical threshold implementations also need flip-flops to not leak information, is often neglected, since it is easy to overlook, that the shared functions cannot be arbitrarily deeply nested without the possibility of glitches that leak information about all shared functions, which would allow to reconstruct the processed data. DOM requires $\frac{N \cdot (N-1)}{2}$ bits of randomness and $N \cdot (N-1)$ flip-flops for each 1-bit multiplier that must be protected. This is illustrated in Figure 2.5 for two share domains A and B. The dotted flip-flop in the inner-domain path is not mandatory for a functionally correct operation, but is often beneficial since it allows to use the DOM-multiplier as a pipeline stage.

Neglecting the problem of randomness generation, it was shown that this scheme performs very good in terms of required area. A second-order DOM-protected implementation for AES takes about the same area as a first-order threshold implementation with component functions [25].

2.2.2. Brief Description of Keccak

The very basics of Keccak, which are needed to understand the actual implementation are given in here. For a more detailed description of Keccak and sponges, the reader is referred to the work of Bertoni *et al.* [12, 16, 13].

The members of the Keccak family of sponge functions, differ in their state size, which is given by $25 \cdot 2^l$ where $l \in \{0, 1, 2, 3, 4, 5, 6\}$. These permutations are denoted as Keccak-f[25], Keccak-f[50], up to Keccak-f[1600]. The Keccak-f[1600] member was standardized in the FIPS-202 (SHA-3) standard [2].

The sponge-state, is logically divided into a rate r and a capacity c. The rate is considered public. The capacity is considered secret and must never be part of the output. A KECCAK permutation can be described as KECCAK[r,c], which then describes how much of the permutation is considered as public and how much is considered secret. E.g. the KECCAK[1088,512], instantiation is the sponge function used in SHA3-512 .

The KECCAK permutation consists of multiple steps:

- θ - Linear diffusion step
- ρ - Linear step that aims to provide inter-slice dispersion
- π - Linear step that transposes bits inside a slice and provides long-term diffusion
- χ - Non-linear step containing the S-Box lookup
- ι - Linear step, addition of a round constant

These steps are applied in the listed order every round. One permutations consists of multiple rounds. The exact number is given by $12 + 2 \cdot l$ for l being the number also used in the definition of the state size above. E.g. for KECCAK-f[1600] 24 rounds are performed per permutation.

2.2.3. Applying DOM to KECCAK

In order to apply DOM to KECCAK, only the χ step needs special attention. Since all other steps are linear, they can simply be replicated. This means, that e.g. for a first-order DOM-protected KECCAK peripheral, the whole sponge-state is instantiated twice. Also the linear θ, ρ and π steps are replicated for every domain. The ι step only needs to be applied to one of the domains, since it is an addition of the round constant.

The χ step is more tricky, because of the additional flip-flops in every DOM-AND gate which effectively makes every DOM-AND a sequential element as shown in Figure 2.5 . Basically there are two possibilities to insert flip-flops. Either they are only inserted on the cross-domain path (the path that combines values from different domains), or they are inserted in both the inner- and the cross-domain path. Only the flip-flops in the cross-domain paths are mandatory, since they are needed to prevent propagation of glitches which would lead to a power consumption that depends on a combination of the share domains. Inserting the additional flip-flops in the inner-domain path has a small area overhead, but by doing so, they can be used as a pipeline stage, so that a result is available at every cycle, after a latency of one cycle to fill the pipeline. On the downside, this has an area overhead of N inner-domain flip-flops for every DOM-protected 1-bit multiplier, where N is the number of shares. So the total number of flip-flops for a 1-bit multiplier increases to N^2.

The additional inner-domain flip-flops are preferable in applications which only have a small number of multipliers, but need to achieve a higher throughput. In KECCAK the χ

step can be done in a slice-based manner. This helps to restrict the polynomial increase in area due to the required N^2 flip-flops per 1-bit multiplier, since χ is not performed on the whole state at once, effectively reducing the amount of multipliers. This approach was chosen for the instantiated KECCAK peripherals as will be described in Section 3.4

Chapter 3

Design Implementation

In this chapter, the implementations of the privileged ISA 1.9.1, the Sponge-Based Control-Flow Protection (SCFP), the Memory Management Unit (MMU) as well as the Keccak peripherals are described in dedicated sections. It starts by discussing the changes to the core, which were needed for the privileged ISA. Then the implementation of the MMU and the SCFP unit are discussed. Finally the implementation of the DOM-protected Keccak peripherals are discussed.

3.1. Remus - Privileged ISA and `Ri5CY` Core Modifications

The `Ri5CY` core developed at the IIS at ETH Zürich served as a basis for all modifications [62].

In order to allow the seL4 port to run, the RISC-V privileged ISA draft 1.9.1, simply referred to as privileged ISA from now on, was partially implemented. This section describes the parts of the privileged ISA that were implemented. The MMU is described separately in Section 3.2 . The SCFP-unit is not part of the privileged ISA, but parts of this section describe modifications required in the core to make the SCFP-unit work and thus refer to it at several points. See Section 3.3 for a description of the SCFP-unit.

Coarse summary of changes to the core in comparison to the start of the project:

- Modified trap-vector layout
- Support for privilege modes (Machine/Supervisor/User)
  - New privilege instructions
  - New Control-Status-Registers (CSRs)

- New load/store/fetch exceptions
- New interrupt interface
- Additionally required stall cases
- Various Controller changes for correct trap-handling and context-switches
- Memory Management Unit (described in Section 3.2)
- Sponge-Based Control-Flow Protection unit in separate pipeline stage (described in Section 3.3)

The resulting core was given the name Remus .

In the remainder of this section, the implementation of the privileged ISA and the necessary modifications in the `Ri5CY` core are described. The privileged ISA specification itself is only described briefly. Please refer to the RISC-V manuals for details [68, 67].

3.1.1. Remus - Overview

An overview of the Remus core is given in Figure 3.1 . When the SCFP unit is unused the whole CFI pipeline stage is bypassed, and the core effectively only has 4 pipeline stages again. The forward-muxes from the LSU to the EX-stage have been omitted for clarity, but are still present like in the original `Ri5CY` core.

The MMU is not a dedicated pipeline stage, since the TLB lookups are done combinatorially. It acts like an external component, which uses the memory protocol of the LSU and fetch unit when an error must be signaled or a hardware page-table walk must be performed.

The `Ri5CY` core already commits certain instructions, including all the newly added instructions for the privileged ISA, in the instruction decode stage (ID-stage). Most register-register operations are committed in the execute stage (EX-stage). Only load and store instructions make it all the way to the write-back stage (WB-stage).

The `Ri5CY` core has been left mostly unmodified as far as instruction extensions are concerned. Only the hardware loop instructions (encoding space "custom-3") and multiply accumulate instructions (encoding space "custom-2") were replaced with the SCFP instructions (see Section 3.3.1). Please refer to the RISC-V User Manual [68] for the description of the available custom encoding spaces. The vector-, bit-manipulation, load-post and store-post extensions are still present.

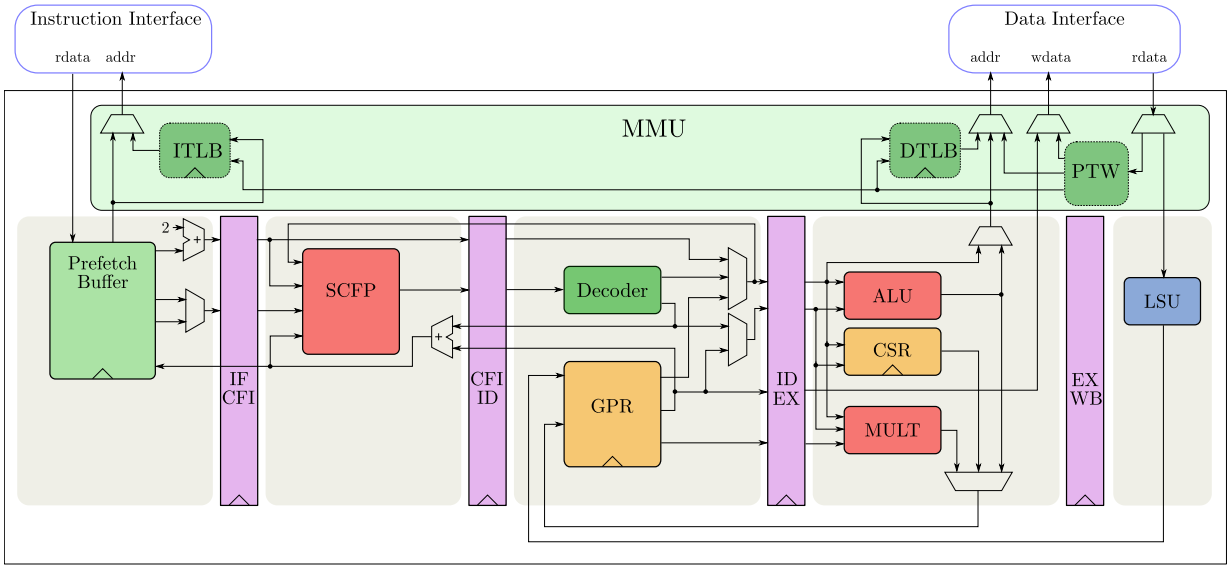

Figure 3.1.: Remus pipeline overview.

3.1.2. Additional CSRs and Instructions

The supported privilege modes are Machine, Supervisor and User (with User-Trap support). The Control-Status-Registers (CSRs) shown in Table 3.1 have been implemented, mostly following the privileged ISA 1.9.1 [67]. The Remus core can be configured to support only a subset of the privilege modes. Depending on the configuration, some CSRs are not available, or tied to constant values. See the privileged ISA for details [67].

Accessing an unimplemented CSR now raises an illegal instruction exception as mandated by the privileged ISA. The functionality of Debug/Trace CSRs, are not yet described by the specification and haven't been implemented. Their addresses are currently occupied by `Ri5CY` 's custom performance counter CSRs.

The following new instructions have been implemented to conform to the privileged ISA 1.9.1.

- `MRET`
- `SFENCE.VM` (Only valid when Supervisor support is configured)
  see Section 3.2.5 for a description of the inner workings of this instruction.
- `SRET` (Only valid when Supervisor support is configured)
- `URET` (Only valid when User-Trap support is configured)

address	M-Mode CSRs	address	S-Mode CSRs	address	User-Traps CSRs
0x300	MSTATUS	0x100	SSTATUS	0x000	USTATUS
0x301	MISA	0x102	SEDELEG	0x004	UIE
0x302	MEDELEG	0x103	SIDELEG	0x005	UTVEC
0x303	MIDELEG	0x104	SIE	0x040	USCRATCH
0x304	MIE	0x105	STVEC	0x041	UEPC
0x305	MTVEC	0x102	SSCRATCH	0x044	UCAUSE
0x340	MSCRATCH	0x102	SEPC	0x043	UBADADDR
0x341	MEPC	0x103	SCAUSE		
0x342	MCAUSE	0x104	SBADADDR		
0x343	MBADADDR	0x105	SIP		
0x344	MIP	0x180	SPTBR		
0xF11	MVENDORID				
0xF12	MARCHID				
0xF13	MIMPID				
0xF14	MHARTID				

Table 3.1.: Implemented CSRs.

3.1.3. Traps

Trap Delegation

The trap-delegation mechanism has been implemented, so traps can be handled in supervisor mode directly, without first having to go through machine mode code.

The MEDELEG CSR can be used to delegate exceptions to lesser privilege modes. Each bit in MEDELEG corresponds to an exception cause, with the bit-number equal to the cause number (see Table 3.2). Some bits are unused, because not all exception causes are meaningful in Remus . The "Environment call from H-mode", "Store address misaligned", "Load address misaligned" and "Instruction address misaligned" exceptions are not used in Remus , and can't be delegated. They are tied to constant zero.

The MIDELEG CSR can be used to delegate interrupts to lesser privilege modes. Each bit in MIDELEG corresponds to an interrupt cause, with the bit-number equal to the cause number (see Table 3.3). Only the external interrupts are supported, since software interrupts are not used in any of the adapted software, and timer interrupts are handled like ordinary external interrupts. Consequently all bits besides the external interrupt bits are tied to constant zero.

As described in detail in Section 3.1.6 , only the "Machine external interrupt" can be used in Patronus , because the event-unit doesn't support multiple privilege levels.

Exc. Nr.	Exc. Description
0	Unused (Instruction address misaligned)
1	Instruction access fault
2	Illegal instruction
3	Breakpoint
4	Unused (Load address misaligned)
5	Load access fault
6	Unused (Store address misaligned)
7	Store access fault
8	Environment call from U-mode
9	Environment call from S-mode
10	Unused (Environment call from H-mode)
11	Environment call from M-mode
≥ 12	Unused

Table 3.2.: Exception causes.

IRQ Nr.	IRQ Description
0-3	Unused ((U|S|H|M)-mode software interrupt)
4-7	Unused ((U|S|H|M)-mode timer interrupt)
8	User external interrupt
9	Supervisor external interrupt
10	Hypervisor external interrupt
11	Machine external interrupt
≥ 12	Unused

Table 3.3.: Interrupt causes.

Additional Exceptions

In addition to the exceptions previously implemented in `Ri5CY` , three additional exceptions are needed for the correct operation of the MMU. First the possibility to signal errors on the data interface, e.g. load- and store-errors, must be provided. Secondly the possibility to signal errors on the instruction interface, e.g. fetch-errors, is needed.

While a load/store-error immediately results in a load/store-exception, this is not the case for fetch-errors. Because a control-transfer in the ID-stage or EX-stage might have changed the control-flow and the fetch request which caused the error would actually never have occurred. As a consequence, when a fetch-error is raised, it is stored as pending and the fetch-unit stops fetching instructions. When the pipeline runs empty without changing the control-flow, the stored fetch-error triggers an exception. In case the control-flow changes before the pipeline ran empty, meaning the PC to fetch is changed by the controller, the fetch-error is cleared and will not trigger an exception.

The privileged ISA 1.9.1 specifies 'cause' values for fetch-/load-/store-exceptions respectively, so they have been adopted as shown in Table 3.2 . The trap-vector is the same for all three exceptions and the `(M|S|U)CAUSE` CSR value has to be investigated inside the trap handler to determine the exception cause.

Trap Vector Table

SCFP needs a different trap-vector layout, namely it requires eight bytes per vector. Four bytes are needed for the initial patch value and another four bytes for the unconditional control-transfer to the actual trap handler. Additionally, a new handler for load-, store- and fetch- exceptions has been introduced.

The adjusted trap-vector table is depicted in Figure 3.2 .

3.1.4. Controller Modifications

Traps are usually handled within a higher privilege level. This also means that the entry into a trap must already be performed with the access rights of the higher privilege level.

In the previous implementation of the controller this could not be achieved, since the PC was changed immediately on trap-entry or exit. To solve this, an intermediate state is needed. First the signal that a trap is entered or exited is given, such that the context is prepared. In the next cycle, the PC of the fetch-unit is changed to the trap-handler's entry address, respectively the trap-return target address. This way the fetch is already performed within the correct context, e.g. the privilege level is correct and virtual address translation is configured accordingly.

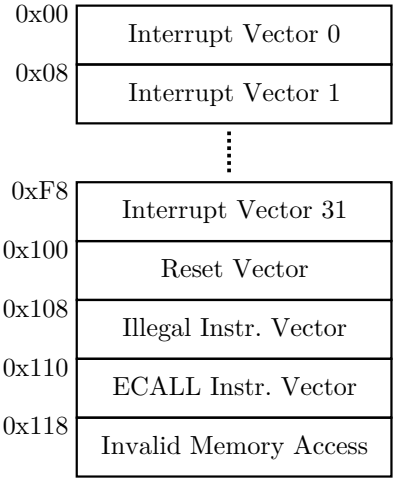

Figure 3.2.: Layout of the trap vector table.

Additional Stall Cases

SCFP-patch stall. This stall is needed, because it cannot be assumed, that the patch value that follows some CFI-instructions is always available to the CFI unit when the instruction is seen in the Decode-stage. In practice this stall is almost never observed. In Patronus , it can only happen when the instruction-fetch and load-store unit access the same memory bank simultaneously, and the fetched instruction would be the patch value.

CSR stalls. Since it is now possible to write to CSRs which have an effect on the current state of the core in a way, that the following instruction will behave differently, an additional stall case is needed when writing to these CSRs. As an example, consider a write to the `MSTATUS` CSR, immediately followed by an `MRET` instruction. E.g. when writing to the `MPP` bits in `MSTATUS` , it is important that this happens before executing the `MRET` instruction, which applies its effects in the Instruction-Decode stage already.

New Load/Store stalls. It is possible that a load or store in the EX-stage or WB-stage might fail, e.g. due to accessing an invalid physical address, or because of a page-fault in case the MMU is active. In such a case it is important that no instructions which alter the core's state (can't be replayed) are committed before the load/store instruction. Concretely, the additional instructions that can cause a load/store stall are `SFENCE.VM`, `MRET`/`SRET`/`URET` and `ECALL`. Additionally, illegal-instruction exceptions as well as interrupts and instruction access faults must be delayed until it is sure, that no load/store fault is raised.

E.g. if an `ECALL` is executed while a load/store is ongoing, the system would trap to the ecall trap-handler. When a load/store fault is raised after this is done, the system state is not what would logically make sense in an in-order pipeline, since for example the current privilege level isn't the one that actually caused the load/store fault. This can in general not be worked around in software, and consequently a load/store must cause a stall, until it is sure that it won't raise an error.

Context-Switches

On a context-switch, e.g. due to interrupts or exceptions, the program counter of the last instruction that has not started to be executed, meaning that it has not yet applied any of its effects, must be stored. For example on a load/store-exception the PC of the load/store instruction that causes the exception needs to be saved. Hence the PC is now passed through the whole pipeline together with the instruction to handle such cases.

Interrupts are treated as exceptions in the ID-stage, SCFP-stage or IF-stage in this regard, depending on where the first not yet executed instruction is in the pipeline. This means, that on an interrupt, either the PC of the ID-stage, the SCFP-stage, or the IF-stage gets saved.

Note that instructions in the EX-stage or WB-stage can in general not be replayed, since they might already be in the process of applying its effects. More concretely, think of load/stores again, for which a bus access gets granted and once done so, the transfer must be completed. So loads/stores are instructions that apply their effects over multiple cycles and must be completed once they started to do so. Aborting such an instruction would (in the current implementation) leave the system in an inconsistent state of having an instruction applied partially. As a consequence, interrupts must be delayed until any instruction that currently applies its effects (and doesn't have a safe way to be aborted) has been committed.

Further note, that the result of an instruction might be an exception. Since exceptions cannot simply be switched off, or stored on a context-switch to be handled at a later point, they must have higher priority than interrupts once triggered. Otherwise, the exception gets triggered within the interrupt routine, which is already in a different context (the kernel usually). Effectively this would allow the triggering of exceptions within another context than the one that raised it, which is obviously undesired for sanity reasons.

The controller, which is responsible to stall the core, trigger context-switches and change the control-flow, has been adapted to take this into account.

3.1.5. Deactivating RVC when CFI active

When the SCFP-unit is active, the compressed decoder, which is located in the instruction fetch stage is disabled. Otherwise the compressed decoder would decode ciphertext when

the two least-significant bits are not '11' by chance. In case the RVC extension is desired the compressed decoder would need to be located after the SCFP unit, either still within the SCFP-stage, or already in the ID-stage. While there is no conceptional limitation that would prohibit this, there is no support for RVC instructions within the SCFP post-processing tools that would allow to test such a modification. Hence RVC instructions are not supported with an active SCFP-unit, but can still be used when it is inactive.

3.1.6. Interrupt Line Modifications

The external-interrupt interface has been adapted to work with a new event-unit, which can signal an interrupt request on one of 4-different interrupt lines paired with a 5-bit number to describe the interrupt. Each interrupt line would correspond to one privilege level and hence hints which is the least-privileged level that might handle the interrupt. This would allow to configure the event-unit, to signal that certain interrupts may be handled at lower privilege levels.

All four external interrupts, also those allowed to be handled by lesser privilege levels according to the event unit, are by default handled in Machine-Mode. The MIDELEG CSR can be used to delegate them directly to Supervisor-Mode if desired.

Since the Event-Unit wasn't finished in time for the tape-out, a version that signals interrupts on a single line paired with a 5-bit number was used in the end. The interrupt request line is wired to the Machine-Mode external interrupt line of the core. Enabling of interrupts and delegation can thus be done with the `MIE` bit in the `MSTATUS` CSR, and the corresponding `MEIE` field (bit number 11) in the `MIDELEG` register. Please note that only all, or no external interrupts can be delegated in hardware.

3.1.7. Deviations from specification

Context-switch. The context-switch itself works slightly different then described in the privileged ISA 1.9.1.

On trap entry from privilege mode x to mode y:

```
yPIE  = yIE
yIE   = 0
yPP   = x
privlvl = y
```

On trap return from mode y to mode x:

```
privlvl = yPP
yPP   = UMODE
yIE   = yPIE
yPIE  = 1
```

Since there is no User-Traps support in PATRONUS , and either all or no interrupts are delegated, this is actually functionally equivalent to what is specified in the privileged ISA 1.9.1. A program adhering to the privileged ISA should work without any modifications in this regard.

Performance Counters. The `MCYCLE`, `MINSTRET`, `MHPMCOUNTERx`, `M(U|S|H)COUNTERENx` CSRs were not implemented, as the same functionality is already implemented in the hardware performance counters. They are hardwired to zero.

Interrupt CSRs. The `(M|S)IE` and `(M|S)IP` CSRs are hardwired to zero, as the Event-Unit is responsible for enabling and signaling individual interrupts.

3.2. Memory Management Unit

The implementation of the MMU at a block-diagram level is shown in Figure 3.3 .

What follows is a short description of the sequence how the TLB hit and miss cases are handled. The load-store-unit (LSU) is taken as an example, but the sequence for the prefetch interface is the same.

The LSU sets off a request with a certain virtual address (vaddr). The tuple of `(ASID, vaddr)` forms the tag which is compared against all the valid tags in the TLB.

Hit. On a hit, the translation stored in the TLB that contains the physical page number and the corresponding access-rights is selected. The access-rights are checked against the current type and privilege of the access. If successful, the MMU forwards the request with the translated memory address. If a permission error occurs, the access is not forwarded and an access-error is signaled to the LSU.

Miss. On a miss, the Page-Table Walker becomes active and loads the page table entries from memory. If an error occurs it is forwarded to the LSU. If the walk succeeds, the new entry is stored inside the TLB. The entry to replace is determined by the current state of the Pseudo-LRU tree. After insertion of the new entry, the TLB signals a hit again, the PTW becomes inactive and the sequence for a hit as described above is exercised.

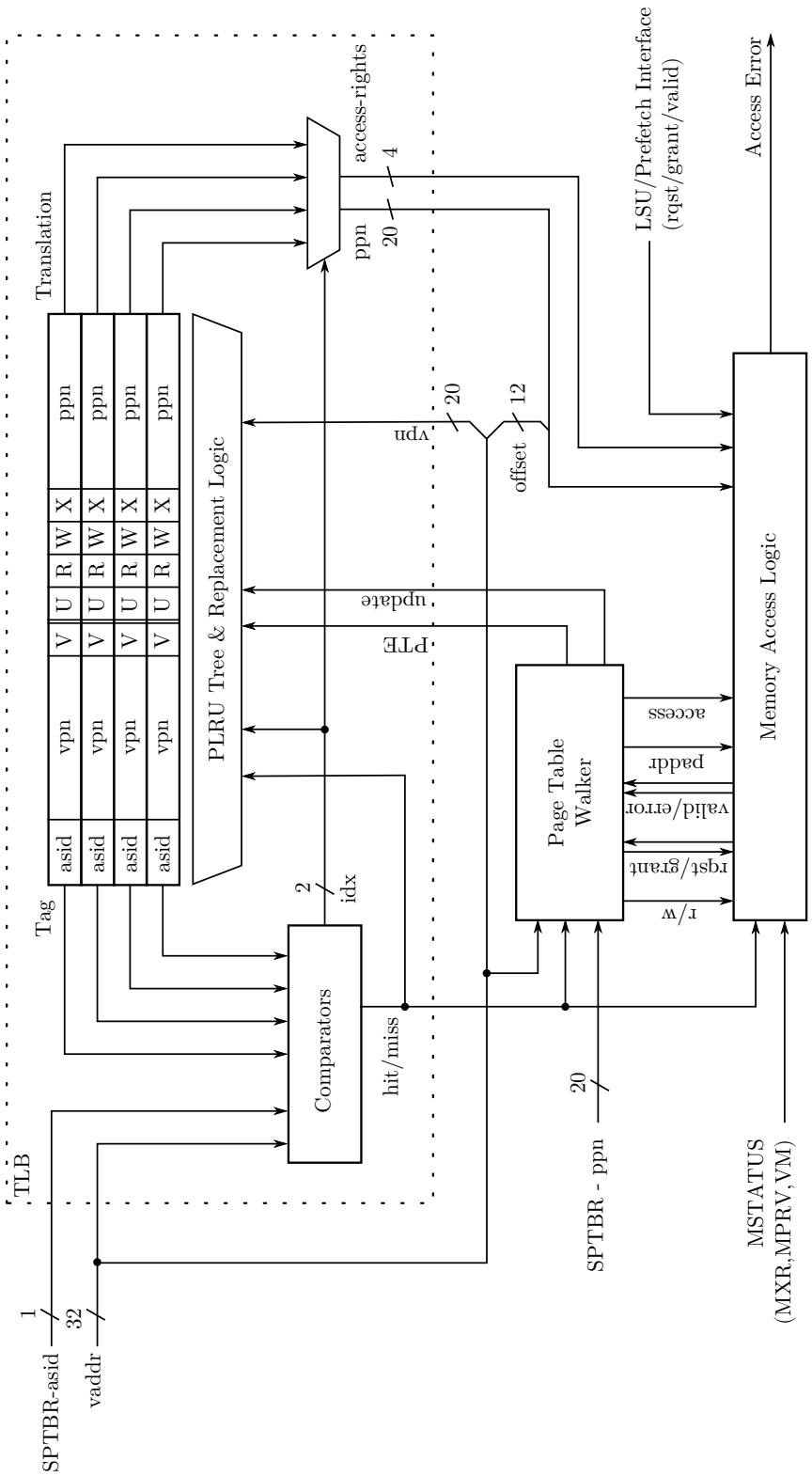

Figure 3.3.: MMU block diagram. Only one of the two TLBs is shown for increased clarity.

3.2.1. Overview

The implemented Memory Management Unit (MMU) is modeled after the sv32-MMU as described by the privileged ISA 1.9.1 [68]. The primary purpose of the MMU was to run the RISC-V port of the seL4 operating system. Not all functionality specified in the privileged ISA has been implemented, since some of the described features are optional, and others are not required for the correct operation of the seL4 microkernel. Only features that are actually supported in the RISC-V port of seL4 are also implemented in hardware. This makes the MMU less feature-rich, but increases the assurance that the MMU works as specified, since the vital parts could be tested more extensively. Deviations from the specification are documented in Section 3.2.7 .

The MMU translates 32-bit virtual addresses into 32-bit physical addresses. It does so combinatorially, meaning that the MMU could be added to the core without requiring an additional pipeline stage. It can thus be seen as an extension of the load-store unit (LSU) respectively the instruction-fetch-unit.

The following list provides an overview of the MMU instantiated in Remus .

- 32-bit virtual address to 32-bit physical address translation via 2-level page tables
- 4-entry data TLB
- 4-entry instruction TLB
- 4kB pages and 4MB "Megapages"
- Hardware Page Table Walker
- Pseudo Least Recently Used (PLRU) page replacement
- Support for the Protect User Memory (PUM) flag
- Support for the Make eXecutable Readable (MXR) flag
- Support for the Execute-Only pages
- 1-bit ASID value (to ease porting of seL4)
- Combinatorial address translation without extra pipeline stage

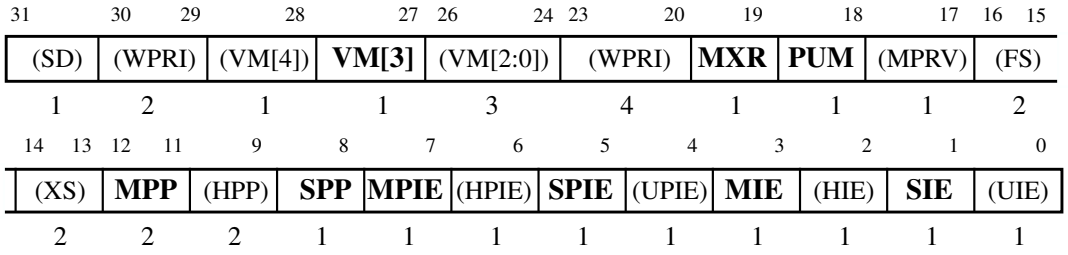

Figure 3.4.: `MSTATUS` - Machine mode status register (`SSTATUS` is a restricted view on this CSR)

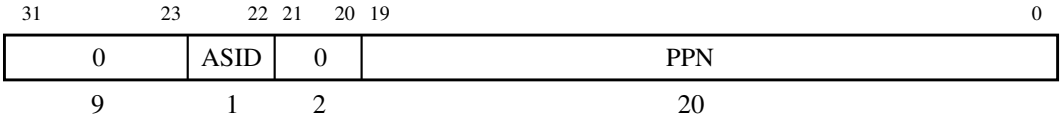

Figure 3.5.: `SPTBR` - Supervisor Page Table Base Register

3.2.2. Parameterization

The MMU provides a few very basic possibilities for configuration at synthesis time via parameters.

The size of the data TLB as well as the instruction TLB can be parameterized individually and can be chosen to be any power of two. Note, that since the TLB is implemented as a fully associative cache with flip-flops as storage elements, the area requirement for larger powers of two might become substantial and other solutions, such as implementing the MMU in a dedicated pipeline stage and using fast SRAM blocks for storage, should be evaluated. The Pseudo Least Recently Used (PLRU) page replacement algorithm is written generically to work with more TLB entries without any modifications.

The number of ASID bits is also parameterizable. Since the RISC-V port of seL4 requires only a 1-bit ASID, this is what has been chosen as the default and is also instantiated in Patronus . Using more ASID bits will add to the time required for the TLB lookup, since more bits need to be checked before a TLB-hit can be signaled. Similarly, since every TLB entry has an ASID, the TLB size inevitably increases.

3.2.3. Configuration via CSRs

The `MSTATUS` and `SPTBR` CSRs contain multiple bits relevant for MMU configuration as described in the privileged ISA 1.9.1 and shown in Figure 3.4 and Figure 3.5 . In `MSTATUS` the fields which are actually supported are shown in bold, while all fields in brackets are implemented as constant-zero values.

A short listing and description of the CSRs' contents is given below. Refer to the privileged ISA draft 1.9.1 for more details.

MSTATUS:

- VM determines the virtual memory mode. If VM[3] is '1' the MMU is active in S-mode and U-mode.
- Protect User Memory (PUM)
  When the PUM bit in the MSTATUS CSR is set, all accesses to a page marked accessible in U-mode will trigger an exception when translation is performed with supervisor privilege level. This is effectively a kernel-debugging and, to some extend, a security feature. For instance, in case the kernel would start executing user code with kernel privilege level, a fetch-error exception will be raised if the PUM bit is set.
- Make eXecutable Readable (MXR)
  By default, loads and stores from/to execute-only pages (i.e. pages that can only be accessed via the instruction interface) cause a load/store exception respectively. When the MXR flag is set, loads from execute-only pages are allowed, but stores are still forbidden. In case the flag gets reset although there are still execute-only pages in the data-TLB, the next load from the execute-only page will trigger a load-exception again.

SPTBR:

- Physical Page Number (PPN) are the upper 20-bits of the physical address of the process' page directory.
- Address Space IDentifier (ASID)
  The current ASID gets stored alongside a valid translation whenever a TLB entry is updated. It is only a 1-bit value and was implemented to ease the porting of seL4, not to provide improved performance.

Note: the MPRV bit in MSTATUS has not been implemented. See Section 3.2.7 for details.

3.2.4. Interfaces to Core and Memory.

The MMU has been designed to use the memory protocol of the Ri5CY core [62] and replicated in Figure 3.6 for the sake of completeness. When inactive, all signals just pass through the MMU. No translation or access-right checks are performed and the hardware page-table walker (HW-PTW) is inactive. The load-store unit (LSU) and instruction-fetch unit effectively have direct access as if the MMU is non-existent.

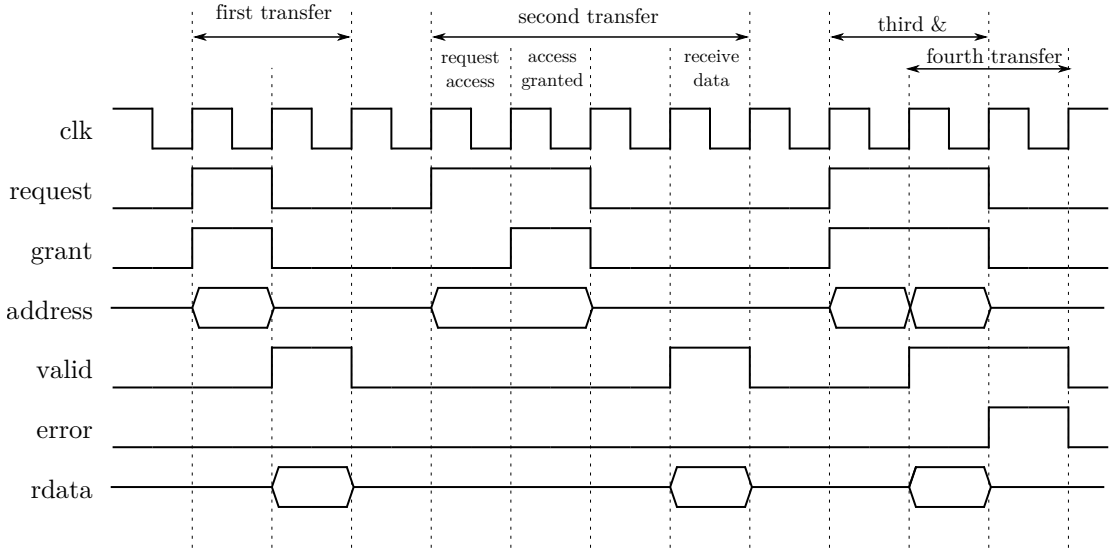

Figure 3.6.: Four memory accesses of the `Ri5CY` core: A single transfer with immediate bus response, a single transfer with delayed grant and delayed valid responses and a back-to-back transfer in which the second access causes an error.

Description of operation. When the MMU is set active by writing to the `VM` field in the `MSTATUS` CSR (see Figure 3.4), it assumes control over the communication with memory when the current privilege level is lower than M-mode.

The 'request' signals from the core are no longer directly connected to the bus. The request is only forwarded to the bus when a valid translation is found, and the access conforms to the specified access-rights for the page, In case no valid entry could be found the HW-PTW is started, see Section 3.2.6 for a description.

In case of the instruction-interface, the 'grant', 'valid' and 'rdata' signals are always forwarded to the core. The MMU is simply responsible for making sure that the requested virtual address has a valid translation and that the access-type doesn't violate the page's access-rights.

In case of the data-interface, the 'grant', 'valid' and 'rdata' signals are only forwarded when the HW-PTW is inactive. The MMU is thus also responsible for arbitrating the access to the data bus between the LSU and the HW-PTW. The HW-PTW always has priority, but outstanding transfers must be completed before it can take over.

Access-right errors. In case a valid translation is found in the TLB, but the access-type violates the specified access-rights (e.g. a write to a read-only page), the MMU doesn't forward the request to the bus. Since the instruction-fetch and load-store-unit expect errors to always be signaled together with a 'valid' signal, which must be preceded with a 'grant', the MMU first signals the core that the request would've been granted. In

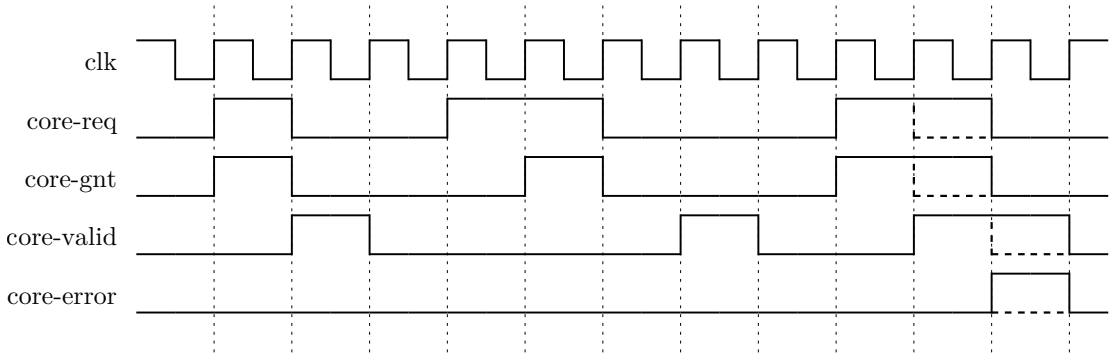

Figure 3.7.: Typical memory accesses with the last one causing an MMU access-right error. The actual bus signal is shown in dashed lines for the erroneous access.

the following cycle it then asserts the core's 'valid' signal together with the 'error' signal to complete the protocol with the core.

Figure 3.7 shows a few typical memory accesses with the last access causing an error. The error cause could be either a bus error, due to a non-existent physical address, or an MMU translation error, due to an access-right violation, in this case. Both cases look identical from the viewpoint of the core. In case of an access-right violation, the last request will not be visible on the actual data bus, but only to the MMU, which adheres to the protocol to signal an error. The actual signals on the data bus are shown in dashed lines for the last access. For the other accesses it is identical with the core's view.

Errors signaled by the MMU. Errors that can lead to exceptions are triggered in the following cases.

- The access doesn't have the appropriate access-rights according to the access-right bits of a valid translation.
- An invalid PTE is encountered during a page-table walk, hence no valid translation exists.
- The bus signals an error condition. This can happen already during the page-table walk, as well as when performing the translated access.

A data-access-error always leads to an exception. The type of exception depends on the type of access that triggered it. In case of a load-access, a load-exception is triggered. In case of a store-access, a store-exception is triggered. These are the exceptions standardized by the RISC-V privileged ISA. See Section 3.1.3 for details on how these exceptions are handled.

An instruction error inside the MMU is equivalent to a fetch-error of the instruction fetch unit. The fetch-error gets only handled when the pipeline ran empty, since it is

possible that an instruction in the decode or execute stage actually changed the PC and the fetch-error would've never been triggered. Likewise, interrupts and load/store errors have higher priority than fetch-errors.

3.2.5. Data and Instruction TLB

Choice of the TLB sizes. The size of the data- and instruction-TLB make up for most of the area of the MMU [1].

Usually systems with MMU tend to be a lot bigger and have external memory with long access times. This makes larger TLBs mandatory, since misses are very costly. E.g. even the small ARM Cortex-A5 MPCore has a unified main TLB with 128 entries [8].

However, in our case we only have internal memory and the latency for a hardware page-table walk takes only a few cycles (see Section 3.2.6). Additionally what is an appropriate TLB size is also heavily dependent on the workload, and the general-purpose benchmarks might not be representative for the intended use cases. Since the point of the MMU is to be able to run seL4, it was thus conceived reasonable to use the seL4-testsuite as benchmark.

The hit-rate for different sizes of data- and instruction TLBs was evaluated on the Virtual Platform (Section 3.2.9) for different sizes of TLBs. For a instruction TLB size of 16 entries a 99.6% hit-rate is achieved. Almost no replacements happened in this case, only the flushes during context-switches cause the hit-rate to be smaller than one. The hit-rate with 4 entries it is still as high as 98.6% and drops slightly to 98.1% for two entries. It was decided, that basically quadrupling the area of the MMU isn't worth this small increase in hit-rate, especially since memory accesses are not very expensive in the first place. Additionally no significant difference in hit-rate could be observed for an instantiation that uses the LRU instead of a Pseudo-LRU replacement algorithm.

Hence a 4-entry TLB with a Pseudo-LRU replacement algorithm has been chosen. The evaluation for the data-TLB shows similar results, and the same configuration (4-entries, Pseudo-LRU) has been chosen. As an additional confirmation that this choice is sane, the RISC-V Rocket Core also has 4-entries per TLB in its smallest configuration ("Tiny-Config") [64].

Implementation of the TLB. The data and instruction TLB each have 4 entries. They act as fully associative caches with a Pseudo Least Recently Used (PLRU) page replacement algorithm. A single data- and instruction-TLB entry is shown in Figure 3.8 and Figure 3.9 respectively.

[1] Synthesis results for MMU with 4-entry data/instruction-TLB: 3.8kGE MMU, including 1.5kGE instruction-TLB and 1.6kGE data-TLB.

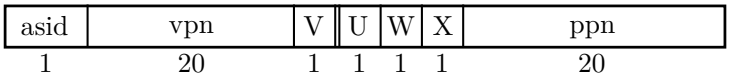

Figure 3.8.: Data-TLB entry

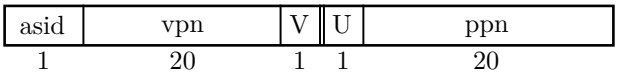

Figure 3.9.: Instruction-TLB entry

When a translation could not be found, a Pseudo Least Recently Used (PLRU) replacement algorithm is used to determine which entry in a TLB should be replaced next. Every access to the data- or instruction-TLB updates the respective PLRU tree, whose state determines the next candidate for replacement.

Flushing the TLBs

The TLBs can be flushed by means of the `SFENCE.VM` instruction, with finer-grained control given by the current ASID bits of the `SPTBR`.

Two different kinds of flushes are possible

- Full flush - Effectively flushing the whole TLB
- ASID flush - All entries that correspond to the ASID stored in the `SPTBR` at the time of executing the flush instruction are flushed. Note that the 1-bit ASID in Remus has been implemented to ease the porting of seL4 from the ARM version that has a special handling of ASIDs. It explicitly is not intended to increase performance. During normal execution the ASID should stay at either zero or one. Switching the ASID between processes is unlikely to have a significant performance gain.

In the current implementation The TLB ignores the virtual address that can optionally be specified with the `SFENCE.VM` instructions. Hence a TLB-flush acts on all mappings of the current ASID in this case. See Section 3.2.7 for details.

The actual flush is done by simply invalidating the 'valid' bits of the affected TLB entries.

It is safe to call `SFENCE.VM` directly after changing the ASID bits in the `SPTBR` CSR, due to the CSR-write-stall (see Section 3.1.4).

The recommended sequence for flushing the TLB in Patronus is shown in Listing 3.1 . Note that the shown sequence will invalidate all entries of the *current* ASID. Invalidating all entries of a process with another ASID requires changing the ASID bits in the `SPTBR` first. This is a limitation of the current specification of the privileged ISA. Note that if the flush is done while the MMU is active, the translation for the page from which the code is executed needs to be re-fetched by means of a hardware page-table walk.

As a consequence, when flushing all entries for a user-process with a certain ASID, the TLB will contain a translation for a kernel page with a user ASID. Since the 'user'-bit in the TLB entry is not set, this is not a security issue, but the user-process would crash when trying to access the virtual memory address that the kernel used. In operating systems in which some of the virtual memory is reserved for the kernel (e.g. everything >3GB is reserved for the kernel), such as in seL4, this is not a problem.

The only way to circumvent this limitation if accessing the full address space from user space should be allowed, is, to do the TLB flush, and the context-switch back to user, in M-mode software and provide an SBI call for it.

```
static inline void flush() {
  asm volatile (
    "SFENCE.VM_x0;"
    "NOP;"
    "NOP;"
    "NOP;"
    "NOP;"
    );
}
```

Listing 3.1: Recommended procedure for a TLB flush

Important: Due to a bug discovered only after tape-out, it is important to not change the fetch address immediately after an `SFENCE.VM` instruction, which could happen in rare cases when the prefetch buffer already contains a jump when `SFENCE.VM` is executed. The obvious workaround is to always insert `NOP`s, or other instructions not influencing the PC, after the `SFENCE.VM` instruction. This problem was fixed after tape-out. See Section 3.2.8 for details.

3.2.6. Hardware Page Table Walker

In the following paragraphs, the process of walking through the page-table entries (PTE) to find the correct translation is termed "data page-table walk" (data-PTW) when the walk is triggered by a data-TLB miss. Likewise, an "instruction page-table walk" (instruction-PTW) is used when the walk is triggered by a miss in the instruction-TLB.

Description of the Page Table Walk.

Figure 3.10 shows the memory accesses, by means of the handshake signals, for a successful page-table walk for a 4kB page translation, and for a page-fault.

The HW-PTW is started when an TLB miss is signaled. In case of an instruction-TLB miss, any ongoing transfer on the data-memory side needs to be completed before the

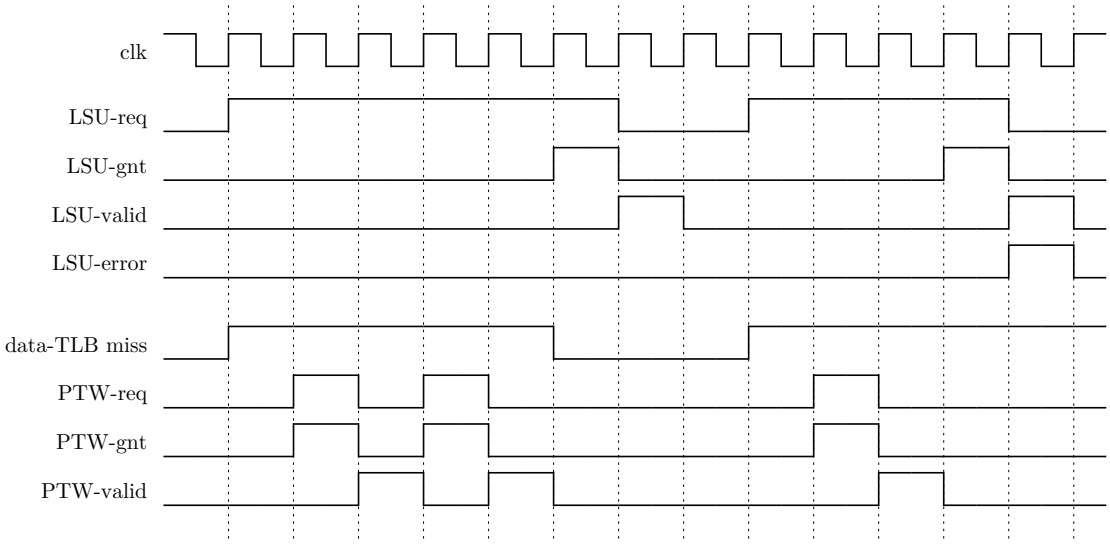

Figure 3.10.: Memory accesses of a successful page-table walk for a 4kB page translation and a failed page-table walk.

HW-PTW can take over. In case a data-TLB and an instruction-TLB miss happen concurrently, the instruction-TLB miss has priority, hence an instruction-PTW is done first. After successful completion the data-PTW will be started.

The HW-PTW performs a sequence of data-memory accesses in order to find the physical address corresponding to the virtual address that caused the TLB miss. The physical page number (PPN) of the first-level page-table (aka page directory), is taken from the `SPTBR` CSR (see Figure 3.5). The HW-PTW then repeatedly loads the PTE, checks the access-rights bits, and calculates the address of the PTE in the next level page-table until a leaf PTE is reached. If an error is encountered, the HW-PTW is stopped and an error is propagated to the LSU or instruction-fetch-unit respectively. The translation is not stored in case of an error during the page table walk. An error during translation occurs when:

- The bus signals an error (e.g. because the physical address is non-existent).
- The valid bit in the PTE is not set.
- The 'writeable' flag is set without the 'readable' flag, since write-only pages are illegal according to the RISC-V privileged ISA 1.9.1 .

The exact process of the lookup is described in more detail in the privileged ISA [67] and not replicated here.

Timing. As is already depicted in Figure 3.10 , starting the PTW is done when a TLB-miss is signaled at the positive clock edge, hence it takes 1 cycle for the PTW to take

over. In case there is no congestion on the memory side, each page-table lookup takes two cycles, back-to-back memory transfers are not supported. Hence the total lookup takes 3 cycles for a 4MB page and 5 cycles for a 4kB page translation. On the 4^{th} respectively the 6^{th} cycle after the request, the new translation is present in the TLB, and the grant is signaled to the LSU or fetch-unit.

Additional Checks on leaf-PTEs. The leaf-PTEs have additional checks that allow to save one bit per entry in each of the TLBs.

Any valid entry in the data-TLB must translate to a readable page, since no write-only pages are allowed according to the RISC-V privileged ISA 1.9.1. By checking that the page is readable, before the translation is stored in the data-TLB, it is possible to omit the 'readable'-bit of a TLB entry. Due to the implemented `MXR` flag a separate execute-only bit in the TLB entry is needed, that specifies whether the page was execute-only when its translation was stored.

Any valid entry in the instruction-TLB must translate to an executable page. Hence the 'executable' flag in the instruction-TLB can be omitted, when it is checked before the translation is stored in the instruction-TLB.

Concurrent accesses to data and instruction memory. While the data-memory interface is used by the HW-PTW, the instruction-interface can still be used by the fetch-unit. Translation of instruction-fetches are possible concurrently with loads/stores on the data-interface. Hence the MMU would work well with an out-of-order pipeline in this regard. Since the core has an in-order pipeline, only the instruction-prefetch-buffer is filled while a data-PTW is in progress.

3.2.7. Deviations from the privileged ISA draft 1.9.1

The sv32 MMU described by the RISC-V privileged ISA 1.9.1 differs slightly in a few points from the implemented MMU.

Physical address size. The privileged ISA specifies a physical address of 34 bits. The virtual address is 32-bit, hence each individual program can only address 32-bit. The 34-bit physical address is intended for 32-bit systems that provide more than 4GiB physical address space, up to 16GiB to be precise. Since PATRONUS itself has a mere 640kiB of SRAM, and all peripheral memory addresses can easily be mapped into a 4GiB address space, it consequently also has a 32-bit address bus. The two most significant bits have hence been dropped in the design already. Note that extending the current version to support 34-bit is as simple as expanding the page directory base address inside the `SPTBR` CSR from 20 to 22-bit and adjusting a few hard-coded bit-widths inside the MMU to use

the full 12-bit of PPN[1] in the PTE instead of just 10-bit. Also note, that the RISC-V seL4 version only supports a 32-bit physical address space at the moment.

Flush dependent on virtual addresses. Additionally to the described techniques to flush the TLBs, the privileged ISA also describes the optional implementation of flushing individual TLB entries by means of a virtual address. When the ASID-bits of the `SPTBR` CSR are non-zero, the `SFENCE.VM` instruction may specify a register which contains the virtual address of the page to be flushed. Since this mechanism is optional and not implemented in the RISC-V port of seL4, this has not been implemented in hardware. Adding this functionality is trivial and the needed steps are already documented in the RTL code of the TLBs.

MPRV flag. This flag is located in the `MSTATUS` register and is described by the privileged ISA as allowing load- and store-operations to be protected by physical memory protection (PMP) mechanisms as if the current privilege level is set to `MPP` . Additionally it also allows to do translations as if the current privilege level is set to `MPP` . In applications it is intended to be only used in small parts of machine-mode code for e.g. address translation in software. Patronus doesn't have any physical memory protection, making this flag effectively useless for this use-case. The other use-case of using the MMU translation mechanism from within M-mode software is not used in seL4 and is an inversion of concerns. M-mode is not required to handle the virtual memory for lesser privilege modes, hence there is no use-case for this in our target applications. Consequently `MPRV` is unused and the system behaves like MPRV is fixed to zero.

Accessed and Dirty bits. Since we don't support swapping, Accessed and Dirty bits are ignored by the hardware at the moment. The rationale behind this is simplicity and a small increase in performance, since no memory accesses are wasted to write the Accessed and/or Dirty bits when first writing or reading from a page. It is still possible to implement all functionality needed for swapping in software, by means of writing invalid page entries and setting Accessed and Dirty bits in the trap handler, as it is done in the ARM version of Linux, for example. Hence not implementing the Accessed/Dirty write-back in hardware simply lowers performance in case the system actually starts to swap pages often, which should be avoided in embedded systems in the first place.

Global bits. The TLB doesn't check whether the Global flag of a TLB entry is set or not when flushing an entry with a non-zero ASID. Since Global flags are a potential source of errors the RISC-V seL4 port doesn't use them currently. Consequently the hardware also doesn't support it.

3.2.8. Known Problems

The listed problems were fixed immediately after the tape-out of Patronus , but the Remus core in Patronus is affected.

It is possible that a change in the program counter while performing a page-table walk to get the translation for an instruction page, causes a wrong translation to be stored. The fetch address was considered immutable after the request has been set off by the fetch stage. This is not the case when e.g. a control-transfer instruction is still present in the prefetch-buffer, as it can propagate through the pipeline while the PTW is still in progress. There are three scenarios where this can cause problems.

- When flushing the TLB, since a PTW is performed directly afterwards in case the flush is done in S-mode.
- When the program runs into another page sequentially (not jumping into the page), this triggers an instruction-PTW and there was a control-transfer directly at the previous page border.
- An interrupt is triggered during an instruction-PTW

This first case can be prevented by either doing TLB flushes only in M-mode, or inserting four NOPs after the flush, so that the prefetch-buffer won't contain control-transfer instructions. The others are rare cases, but can't really be prevented in a reasonable manner.

3.2.9. Virtual Platform Implementation

A functional model of the MMU was first implemented on the virtual platform.

The version of the virtual platform is slightly different in functionality from the RTL implementation. Namely it doesn't support the Protect User Memory (`PUM`) and Make eXecutable Readable (`MXR`) flags. Furthermore, the full 10 ASID bits are theoretically available in the virtual component as opposed to the 1-bit ASID implemented in the RTL code. Cropping the ASID to 1-bit is done outside the MMU, by means of not allowing to write the all ASID bits to the `SPTBR` CSR. A flush on specific virtual addresses is also possible in contrast to the RTL implementation, which can only flush all entries corresponding to a specific ASID (see Section 3.2.5).

The virtual component provides several options for customization by means of constants in the class implementation.

- It is possible to change the replacement algorithm. Least Recently Used (LRU) and Pseudo-LRU were implemented, in order to compare their effectiveness.
- It is furthermore possible to adjust the TLB associativity and number of entries.

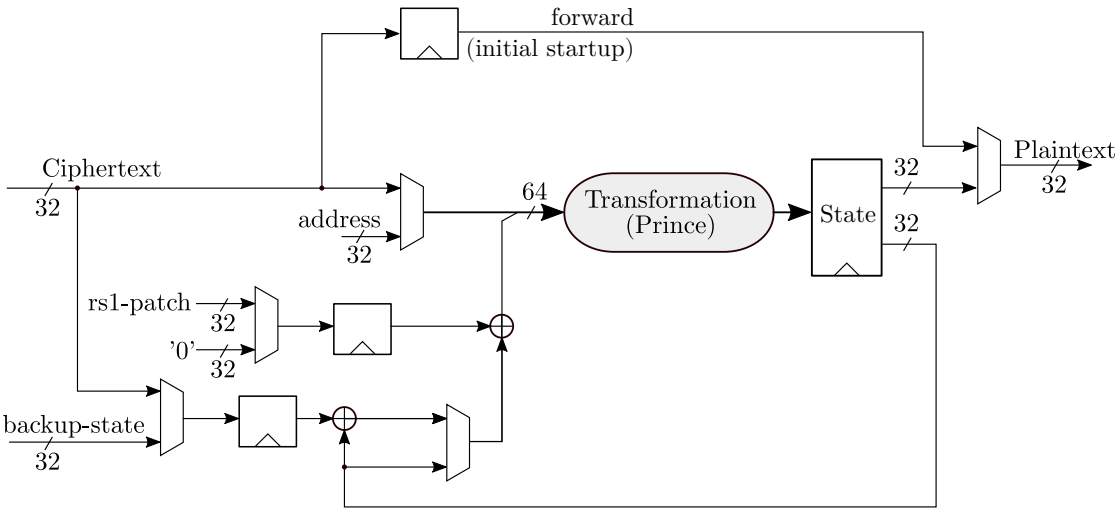

Figure 3.11.: The datapath of the SCFP pipeline stage.

- The interpretation of the virtual page number can be changed to allow more than two levels of page-tables. The number of offset bits directly used for physical address translation and number of page-table offset bits are adjustable. This feature is not fully fleshed out, as only the sv32 version is needed, but serves as a starting point in case e.g. a sv39 MMU should be implemented.

3.3. Sponge-Based Control-Flow Protection

An overview of the implementation is given in Figure 3.11 .

3.3.1. Integration into RISC-V ISA

In order to patch the sponge-state to a known value, to make runtime decryption without immense code duplication possible, additional CFI instructions are needed. As elaborated in Chapter 2, the patching of the state must be done at the edges of the control-flow graph in the used APE construction, and is desirable to do in the Spongewrap construction in order to not leak control-flow information unnecessarily. Therefore, the control-transfer instructions of the RV32I ISA would now need to be modified, such that they expect an additional patch value after the instruction. This patch value is not encrypted itself (due to the obvious chicken-egg problem), but can be considered as Associated Data, whose manipulation results in an invalid state, consequently leading to random execution. In the APE-like mode of operation, the patch needs to be absorbed into the sponge's capacity (the rate is solely given by the ciphertext). The absorption of Associated Data into the sponge's capacity was shown to be secure [56, 49].

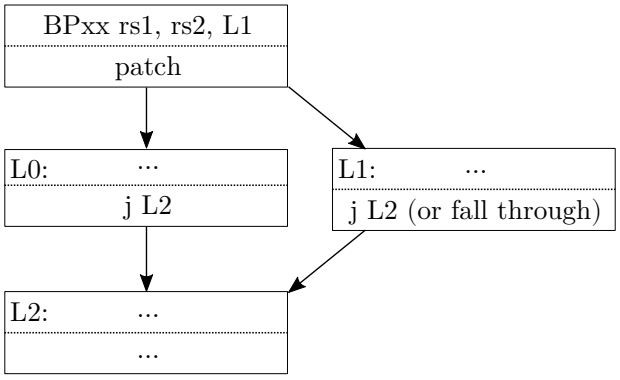

Figure 3.12.: Concept of conditional jumps (branches) in the SCFP implementation.

Since the implementation is experimental, with the purpose of evaluating the efficacy of the SCFP-concept, it is desirable to minimize the impact on the core and not give new meanings to already existing instructions. The custom-2 and custom-3 encoding spaces have been chosen in order to implement instructions that could not be fitted into RV32I as brownfield extensions. In the following, the needed ISA extensions and their behavior is described.

Conditional Jumps

Conditional jumps in the RISC-V ISA are done by means of the branch instructions `BEQ`, `BNE`, `BGE`, `BLT` and their unsigned complements `BGEU`, `BLTU`. There is no additional compare instruction and status register like in e.g. the x86 or ARM instruction sets [30, 9]. The backwards merge point is directly at the branch instruction (see Figure 3.12). Since a branch can only diverge into exactly two paths only one patch value is needed. In principle it would be possible to have a patch value for both paths, but this would just be unnecessary overhead if one of the paths can be specified as the default.

In addition to the control-flow in Figure 3.12 , the sequence of permutations is illustrated in Figure 3.13 .

In theory it is possible to place the patch value at either the target address or adjacent to the branch instruction at the next address. Placing it at the target has a small performance benefit when the THEN branch is the one that is likely to get taken, since the patch value doesn't need to be skipped. An argument against patches at the jump target address, is, that it complicates the control-flow when e.g. the target jump address is reachable by another path. This is the case *e.g.* for the entry point of a loop, where a *jump* into the loop would be required, in order to not execute the patch value.

In (optimized) loops, where the branch is the last instruction in the loop, it is often better to have the patch value directly next to the branch instruction. Most of the time the branch condition is true and the branch gets taken (also see the amount of taken

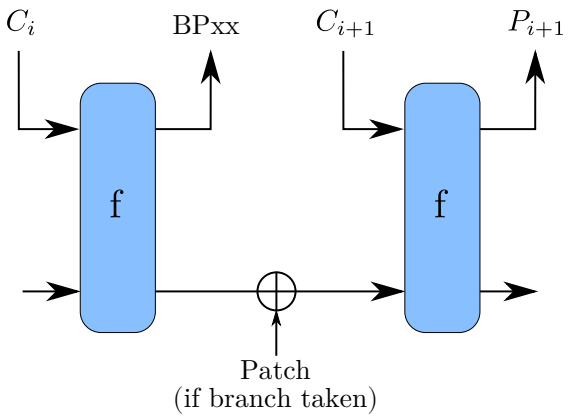

Figure 3.13.: Sequence of permutations for a conditional jump (branch) in the SCFP implementation.

31 25	24 20	19 15	14 12	11 7	6 2	1 0	
imm[12|10:5]	rs2	rs1	0 0 0	imm[4:1|11]	1 0 1 1 0	1 1	BPEQ
imm[12|10:5]	rs2	rs1	0 0 1	imm[4:1|11]	1 0 1 1 0	1 1	BPNE
imm[12|10:5]	rs2	rs1	1 0 0	imm[4:1|11]	1 0 1 1 0	1 1	BPLT
imm[12|10:5]	rs2	rs1	1 0 1	imm[4:1|11]	1 0 1 1 0	1 1	BPGE
imm[12|10:5]	rs2	rs1	1 1 0	imm[4:1|11]	1 0 1 1 0	1 1	BPLTU
imm[12|10:5]	rs2	rs1	1 1 1	imm[4:1|11]	1 0 1 1 0	1 1	BPGEU
imm[12|10:5]	rs2	rs1	0 1 0	imm[4:1|11]	1 0 1 1 0	1 1	BPDEQ
			condition		custom-2		

Figure 3.14.: Encoding of conditional jumps in the SCFP implementation.

branches in the detailed performance evaluation in Chapter A). In that case, a patch after the branch is desirable as the patch value has already been fetched and is available in a previous pipeline stage when the branch instruction is committed.

In order to not change the semantic of the base ISA, new instructions corresponding to the ones in the base ISA need to be added. However, there is not enough space inside a 25-bit RISC-V encoding space to specify both variants (patch at target, patch follows instruction) of the instruction. Spending another encoding space just for the less common case that a patch at the target would be the more sensible option from a performance point of view, would be excessive, especially since no toolchain exists that supports this at the moment. Hence the patch is always placed directly after the branch instruction.

The encoding of the instructions is shown in Figure 3.14 . Conditional SCFP-jumps use the custom-2 encoding space and mirror the encoding space of usual conditional jumps.

In contrast to (some versions of) the openRISC ISA and the MIPS architecture, RISC-V doesn't have delay slots. However for SCFP this delay slot could be used meaningfully,

since it can contain the patch value. Hence when SCFP is implemented for other ISAs, the relative performance impact for conditional jumps is going to be less than that for RISC-V.

BPDEQ The `BPDEQ` instruction doesn't have a pendant in the BRANCH encoding space. It is used to link the branch decision to the data being processed, as was described in Chapter 2.

Because the branch instruction actually takes two operands, both possible results of the software comparison are stored into registers. To build on the example from Chapter 2, the software comparison outputs `0xAAAA5555` to both registers in case the branch should be taken. When the result of the software comparison is, to not take the branch, it outputs `0x5555AAAA` to the first and `0xAAAA5555` to the second register. The `BPDEQ` instruction then operates on those two registers. The sponge-state is patched with the value in the first register, regardless of the branch decision.

The instruction's effects are depicted below in algorithmic form. SPC is the sponge capacity. It is patched with the first register operand $rs1$. Then the branch is either taken, in which case another patch with the appended patch value is needed to adjust the state, or not.

$SPC \leftarrow SPC \oplus rs1$
if $rs1 = rs2$ **then**
    $SPC \leftarrow SPC \oplus patch$
    $PC \leftarrow PC + offset$
else
    $PC \leftarrow PC + 4$
end

The patching of the state when taking the branch is done concurrently with the patching of $rs1$.

Due to the overhead imposed by the software comparison, a trade-off between security and performance might be required for some applications. Especially in the not security-critical parts of the software, it might be required to not use this security feature due to performance reasons. As a consequence, both the ordinary protected branch instructions and this new BPDEQ instruction are included in the ISA. This will also allow to measure performance and efficacy of the BPDEQ instruction in comparison with the other SCFP branch instructions.

Direct Function Calls

Function calls are either implemented as direct jumps (`JAL`) or indirect jumps in the RISC-V ISA. The return from a function is done by an indirect jump (`JALR`) with the

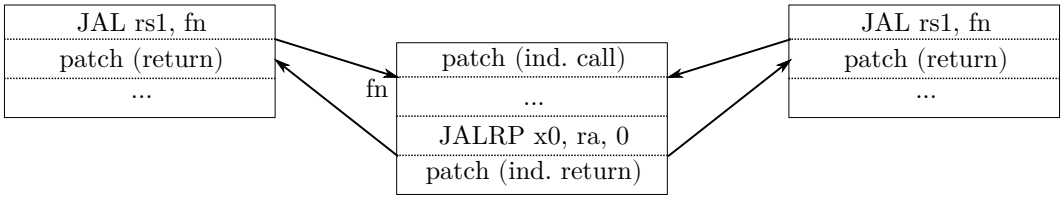

Figure 3.15.: Concept of direct function calls in the SCFP implementation.

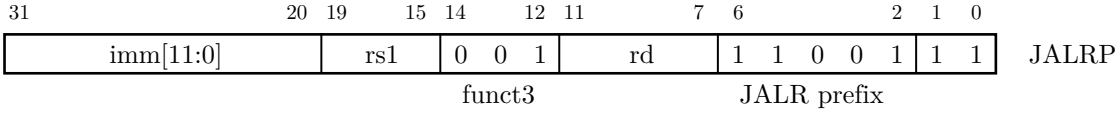

Figure 3.16.: Encoding of `JALRP` in the SCFP implementation.

return address as the target and the zero register as storage place for this jumps return address, effectively discarding it. The control flow of a function call in the SCFP implementation is depicted in Figure 3.15 .

In the simple case of a direct function call, it can be seen that no patching is required when entering the function, since there is no merge point in backwards direction. Consequently a standard `JAL` instruction is used. However, when returning from the function, it is obvious that patching of the sponge state is needed, since it can potentially return to multiple different locations. Since the list of possible return targets can be huge and have variable length, it is easier to locate the patch that needs to be applied at the return target location instead of appending it to the return instruction. The corresponding instruction is `JALRP` which can be implemented as a brownfield extension within the JALR encoding space. The encoding of the `JALRP` instruction is shown in Figure 3.16 .

When a `JALRP` instruction is encountered, the instruction at the jump target address is interpreted as a patch value that gets absorbed into the sponge capacity to adjust the state to the value obtained during encryption. However, this would have a substantial problem. An attacker that knows multiple possible return points of a function could try to manipulate the control flow by returning to one of the other possible return points instead of the one that actually called the function.

This scenario might be especially bothersome when using external memory, where an attacker can manipulate the fetch address easily. In order to make this attack harder, an additional permutation with the jump target address is done, as is depicted in Figure 3.17 . Since this address is an internal 32-bit value, it is assumed that it is hard for the attacker to actually control it well enough to change it to the address of another possible return point.

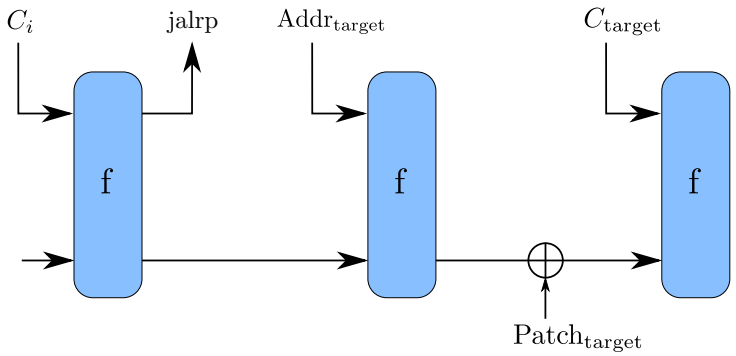

Figure 3.17.: Sequence of permutations for a return from a directly called function in the SCFP implementation.

Indirect Function Calls

In addition to direct function calls, indirect function calls need to be considered as well. Indirect function calls are done via `JALR` instructions in RISC-V, just like a function return. Unlike direct function calls however, the called function is in general not known, and identification of all the possibly callable functions is hard.

One way to handle these cases is, to patch the state capacity to a specific value on the indirect function call. This requires an additional patch value for such indirect calls, which are placed directly after the jump instruction. Additionally a patch value at the function entry is required in order to adjust the state to the one obtained during encryption. Just like it was the case with the return from a directly called function, an attacker could easily use indirect functions as gadgets this way, since all the functions that are indirectly callable would have the same intermediate state. Like before this can be solved, by performing an additional permutation with the jump target address.

There's also a difference between a return from a directly called function for indirectly called function. It follows the same principle as indirect function calls. The state gets patched to a fixed value before it is permuted with the return target address and patched again with the patch at the target address.

The full sequence for a indirect function call is depicted in Figure 3.18 and in Figure 3.19 .

There are cases when a direct function call could also be performed by a `JALR` instruction, e.g. because the callee is located in a memory location far away from the caller. So in essence the `JALRP` instruction must serve two purposes. A mechanism is needed to tell apart direct and indirect function calls. This can be achieved without introducing an extra instruction by simply utilizing an unused bit in the `JALR` instruction. The LSB of immediate value of the `JALR` instruction is also the LSB of the target address in order to avoid an additional instruction format [68]. Instructions cannot be located on a 1-byte

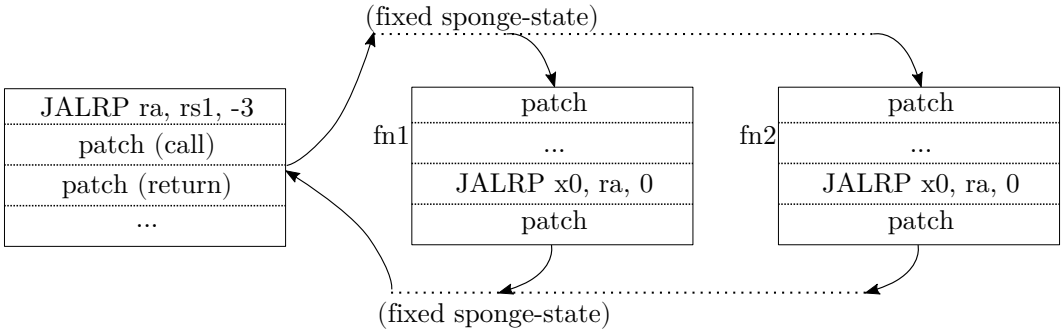

Figure 3.18.: Concept of indirect function calls in the SCFP implementation.

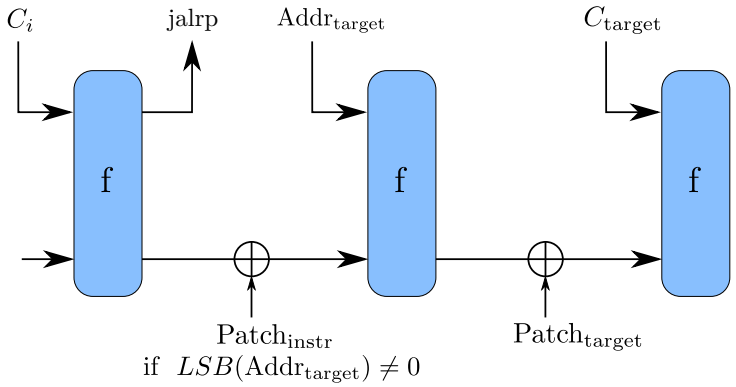

Figure 3.19.: Sequence of permutations for indirect calls/returns in the SCFP implementation.

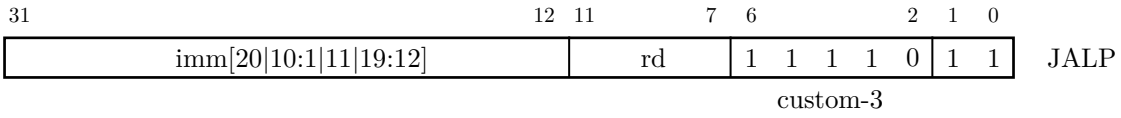

Figure 3.20.: Encoding of JALP in the SCFP implementation.

boundary, so this bit is unused in practice and is envisioned by the RISC-V designers to be a place to give implementation specific hints for e.g. the branch predictor. In the context of SCFP, it is used to tell apart direct and indirect function calls and thus we can use a single `JALRP` instruction to handle both direct and indirect function calls.

A nice property of doing it like this is, that in case it turns out to be possible to identify all the indirectly callable functions in a program, the indirect function calls can be handled just like the returns from directly called functions. It turns out, that no adjustments in hardware would be needed for this, and is simply a matter of how smart the toolchain is.

Unconditional Jump

An additional Jump-And-Link-Protected (`JALP`) instruction is added in order to make the CFI instructions mirror the control-transfer instructions from the base ISA completely. Since direct function calls don't require patching when calling, they also don't require a CFI instruction for the call itself. So in contrast to what the name implies it is not a drop-in replacement for all `JAL` instructions used previously. It is used when unconditional jumps need patching. While a branch instruction with an always-true condition like BEQ x0,x0,offset is semantically the same, the range of the `JALP` instruction is bigger. Hence it is used for unconditional jumps to far-away targets that might require patching. The patch value is located after the `JALP` instruction, just like protected branch instructions.

This instruction must be implemented in a dedicated encoding space. The custom-3 encoding space has been chosen for it. The encoding is shown in Figure 3.20

Additional CSRs

The SCFP implementation allows to have different keys for different privilege levels. This means, that it is possible to have user programs encrypted with a different key than the underlying kernel. Similarly the kernel and firmware don't need to share a key. Table 3.4 gives an overview of the additionally required CSRs.

In a 64-bit RISC-V implementation the CSRs would have a length of 64-bit, in which case the `*KEYOH` and `*KEY1H` CSRs are not required. The `USPONGE` register is only available when U-mode traps are supported. Likewise the key CSRs are only implemented when the respective privilege mode is implemented.

CSR	address	reset value	access	description
MSPONGE	0x350	0	RW	Interrupted task's sponge-state (M-mode trap)
MCFICTRL	0x351	0	RW	SCFP configuration
MKEY0	0x352	0	RW	Bits 31..0 of M-mode key-part k_0
MKEY1	0x353	0	RW	Bits 31..0 of M-mode key-part k_1
MKEY0H	0x362	0	RW	Bits 63..32 of M-mode key-part k_0
MKEY1H	0x363	0	RW	Bits 63..32 of M-mode key-part k_1
SSPONGE	0x150	0	RW	Interrupted task's sponge-state (S-mode trap)
SKEY0	0x152	0	RW	Bits 31..0 of key-part k_0
SKEY1	0x153	0	RW	Bits 31..0 of key-part k_1
SKEY0H	0x162	0	RW	Bits 63..32 of key-part k_0
SKEY1H	0x163	0	RW	Bits 63..32 of key-part k_1
USPONGE	0x050	0	RW	Interrupted task's sponge-state (U-mode trap)
UKEY0	0x052	0	RW	Bits 31..0 of key-part k_0
UKEY1	0x053	0	RW	Bits 31..0 of key-part k_1
UKEY0H	0x062	0	RW	Bits 63..32 of key-part k_0
UKEY1H	0x063	0	RW	Bits 63..32 of key-part k_1

Table 3.4.: Additional CSRs for SCFP.

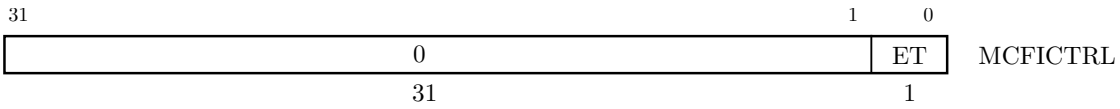

Figure 3.21.: SCFP control register

Configuration. The content of the MCFICTRL is only accessible from within M-mode. The CSR is shown in Figure 3.21 .

The Encrypt Traps (ET) field in the CFICTRL register controls whether the SCFP unit remains active when a trap is entered, as described in Section 3.3.2 .

ET = 0 ... Drop to plain execution on trap-entry. (default)
ET = 1 ... SCFP-unit not switched off at trap-entry.

Keys. The CSR names correspond to the two parts of the key used in the PRINCE block cipher, called k_0 and k_1 with the full key $k = k_0||k_1$. Hence the full key is given as k = KEY0H||KEY0||KEY1H||KEY1 for 32-bit implementations (as in the case of REMUS) and k = KEY0||KEY1 in case of a (hypothetical) 64-bit implementation. The values of the key CSRs are used directly without further buffering as soon as the corresponding privilege mode becomes active. Hence the implementation doesn't allow to easily change keys from within the running privilege mode and consequently M-mode keys can only be changed when the SCFP-unit is inactive.

Sponge Backup. The (M|S|U)SPONGE CSRs are used to save the sponge-state, which corresponds to its capacity in the employed APE-like mode of operation, on trap-entry. On execution of a (M|S|U)RET instruction, the corresponding (M|S|U)SPONGE CSR is applied as a patch to the current sponge state (which should, by design, be zero on trap return) in order to restore the state and continue with the execution of the interrupted task. By saving the SPONGE and KEY CSRs to (adequately protected) memory, it is possible to return to different U-mode programs and also to different S-mode programs, although the later would likely be used less frequently. See Section 3.3.2 for details on the procedure.

3.3.2. Traps

Trap Entry. A trap entry can be seen as a restart of the program and is illustrated in Figure 3.22 . The sponge-state is saved into a CSR and set to zero. The further procedure for trap-entry is similar to the one for the return from a directly called function. It can be seen as a special case with a patch value of zero for the patch value that would follow a JALRP instruction. First a permutation with the trap-entry address is performed, before being patched with the trap-entry patch value, which is needed to get to the initial state needed for trap decryption.

The sponge-state prior to the permutation with the trap-entry address is saved in the (M|S|U)SPONGE CSR, dependent on which privilege level the trap handler executes in. If the trap is not delegated by means of the delegation CSRs [67], but by software, it should be ensured that the higher privilege mode writes the xSPONGE CSR of the lesser privilege mode "x" just like it would write *e.g.* the corresponding xCAUSE or xEPC CSRs.

Because the initial state required for decryption is predetermined by the backwards-encryption of the APE-like mode, an initial patch value on trap entry is required. In the case of Remus , a trap-vector table, which contains the jumps to the actual trap-handlers, is used. Since each vector originally is only 4-byte wide, the patch value doesn't have space and the width of one table entry needed to be extended to 8-byte. The first 4-byte are the patch value, the second 4-byte the (encrypted) jump. This changes the layout of the trap-vector table as shown in Figure 3.2 .

Trap Return. The sponge-state is zero at the point of return by design (if no fault occurred) and can thus be patched with the previously saved sponge-state. This is simply achieved by starting the APE-like encryption with a zero state. A possible advantage is, that any manipulation inside the trap handler will cause random execution, even when the trap-handler is left by chance due to a (M|S|U)RET instruction.

On a trap-return by means of an (M|S|U)RET instruction the (M|S|U)SPONGE CSR gets patched to the current state (which should be zero by design): $SPC \leftarrow SPC \oplus$ xSPONGE. The KEY CSRs are not handled specially as the active privilege level is used to select which key gets used.

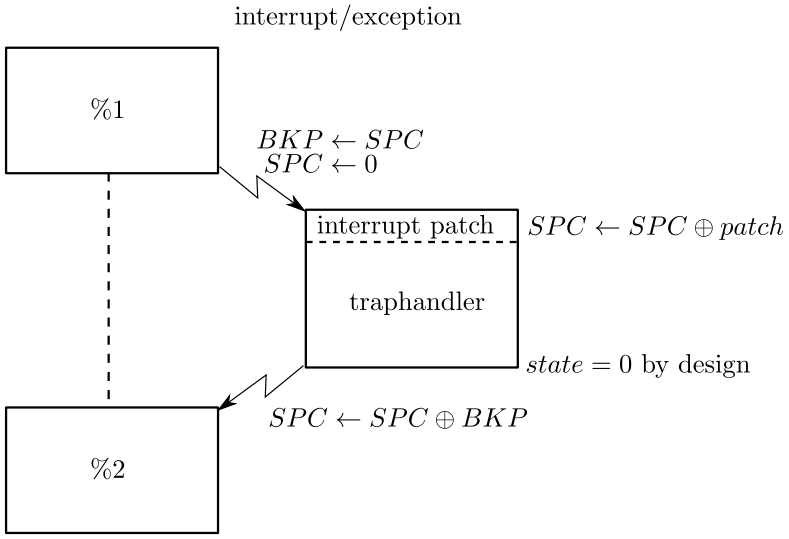

Figure 3.22.: Concept of handling traps in the SCFP implementation.

Unencrypted Traps. When the `ET` bit in the `CFICTRL` CSR is set, the SCFP unit enters a special state in which no encryption is performed. The SCFP pipeline stage is effectively inactive and the core behaves like a 4-stage instead of a 5-stage pipeline again. This mechanism has been implemented since at the time of its first implementation, traps could not be properly encrypted by the used toolchain. It has been kept, since it has been conceived as potentially useful for the purpose of evaluation of the SCFP scheme.

The SCFP-unit becomes active again when a `(M|S|U)RET` instruction is executed. It is not possible to have nested unencrypted traps in the current implementation, as it is not trivially possible to determine whether the executed `(M|S|U)RET` instruction belongs to a nested trap (hence returning to unencrypted code) or not. A wake-up mechanism, which is able to work with unencrypted traps, would need to follow an approach similar to the `xIE` bits in the `MSTATUS` register, which is more complex to implement and was conceived to provide no benefit for the intended use-case of evaluation. Note that this doesn't apply to *encrypted* traps, which can nested be arbitrarily.

Saved sponge-state for ECALL and illegal instructions. `ECALL` instructions and illegal instructions both cause an exception. However in contrast to all other instructions that cause exceptions, they should not be replayed, because this would cause an infinite loop. Hence for these to kind of exceptions, the sponge-state *after* the instruction passed through the SCFP-unit is saved. In contrast, load/store exceptions save the sponge-state before the instruction passed through the SCFP-unit.

The alternative approach would be to save both, the state prior as well as that after the instruction. The software trap-handler could then decide which one should be used on trap-return by means of writing the correct one to `(M|S|U)SPONGE`. Since there is little

reason for an `ECALL` or illegal instruction to be replayed, and the alternative would just require more trap handling code, it was decided to go with the less flexible choice of handling them specially.

3.3.3. Start Procedure

Since it was desirable to still allow execution without active SCFP-unit, *e.g.* for testing, a dedicated start procedure is needed. The SCFP-unit becomes active on the first encountered JALRP which should signal an indirect function call by having the LSB of the target address set to one. The instruction is handled exactly like described previously in Section 3.3.1 : The following patch is applied, then the permutation with the target address is performed and finally the patch at the target is applied.

3.3.4. Toolchain limitations

The RISC-V GCC toolchain currently cannot be used to directly emit code for SCFP. Dedicated post-processing tools, which have been provided by Mario Werner from TU Graz, must be used on the compiled binary to instrument it.

The SCFP instructions are implemented as assembly macros, which expand to the equivalent control-transfer instruction with `NOP`s as placeholders for the patch values. This mostly limits the supported code to handwritten assembly at the moment. However a simple script that does text-replacement of the standard control-transfer instructions with the equivalent SCFP assembly macros on intermediate assembly files turned out to work very well. By instrumenting a few more complicated parts of a program by hand and using the text-replacement tool on the intermediate assembly output, it was possible to translate most of the used benchmarks. Only the Coremark and Dhrystone benchmarks couldn't be instrumented this way. The emitted jump-tables are the cause for this, since they don't work well with text-replacement due to the additional patch values required at the target of indirect jumps. The aggressive inlining used in the other benchmarks helps in this regard, since jump tables (resp. the case statements) are optimized away in all cases there.

In a first post-processing step, the control-transfer instructions are replaced by the actual SCFP instruction encodings. In a second step the instrumented assembly is linked to a binary to get the final address locations. The binary is then disassembled again, the code is encrypted and the `NOP` placeholders are replaced with the actual patch values.

3.4. Keccak peripherals

The Keccak peripherals should serve the purpose of evaluating the DOM-scheme, with focus on its applicability on Keccak. Multiple variants of the Keccak peripherals with

varying orders of SCA-protection were instantiated. All of the instantiated peripherals use a Keccak-f[1600] permutation, since this is the one that many new constructions use, e.g. Keyak and Ketje, two of the last-round candidates in the CAESAR competition [17, 14].

- An unprotected Keccak-f[1600] instance serving as a baseline
- A first-order DOM-protected variant
- A first-order DOM-protected variant with an additional optimization to reduce the required randomness
- A second-order DOM-protected variant

The second-order DOM-protected Keccak peripheral is, as far as could be ensured by thorough search of related literature, the first higher-order SCA-protected Keccak variant taped-out in an actual ASIC.

There is no restriction in hardware as to how much of the state is interpreted as rate and how much as capacity. It is solely up to the software to decide how much of the state it writes or reads respectively, which allows for greater flexibility.

A high level view of the Keccak peripherals is shown in Figure 3.23 .

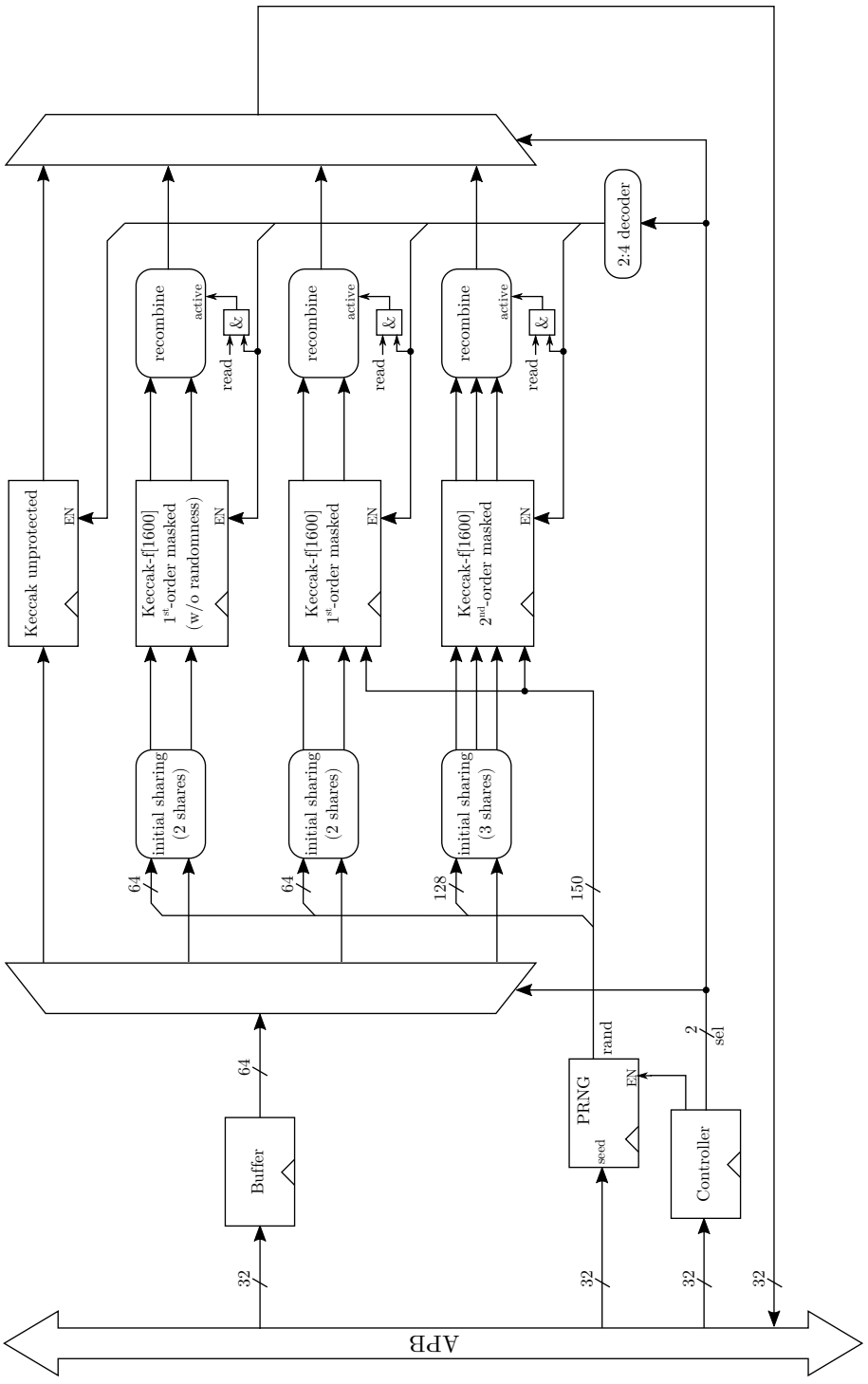

Figure 3.23.: Block diagram of the Keccak peripherals

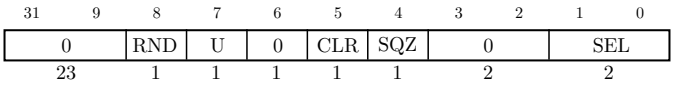

Figure 3.24.: Control Register

3.4.1. Memory Map / Configuration Registers

Table 3.5 shows a list of registers that are used to read/write configuration as well as data from/to the Keccak peripherals.

Figure 3.24 shows the control register for the Keccak peripherals. Only one peripheral can be active at any time. Which one is active is determined by the `SEL` bits in the control register:

`SEL = 0` $\rightarrow$ unprotected Keccak.
`SEL = 1` $\rightarrow 1^{st}$-order DOM-protected Keccak.
`SEL = 2` $\rightarrow 1^{st}$-order DOM-protected Keccak with randomness optimization.
`SEL = 3` $\rightarrow 2^{nd}$-order DOM-protected Keccak.

Writing to the `SQZ` bit triggers a permutation of the selected peripheral. The permutation will get triggered automatically when the full state has been written. `SQZ` allows to only write a part of the state (at 32-bit granularity). Since the rate is written first, this allows to effectively determine the sponge rate in software.

Writing '1' to `CLR` clears the sponge-state of the selected peripheral.

Writing to `U` has no effect in the current implementation, but preserves the value that is written to it.

Writing '1' to `RND` triggers an update of the LFSR. One LFSR update actually consists of 150 recursions, thus producing 150-bit of pseudo-randomness per update. The RND bit is automatically reset after the LFSR update.

The DATA register is used to read/write from/to the Keccak peripheral selected by means of the `SEL` bits in the `CTRL` register. Two counters are used to keep track of how much data has been read, respectively written. The counters are reset when a permutation is triggered, the full state has been read/written, or the `CTRL` register gets written. As soon as the full state was written a permutation is triggered automatically.

The PRNGx registers are used as the seed value for the PRNG.

The base address of the Keccak peripherals in PATRONUS is `0x1A102200`.

register	offset	reset value	access	description
DATA	0x00	0	RW	Data for selected Keccak peripheral
CTRL	0x1C	0	RW	Configuration register
PRNG0	0x20	1	RW	LFSR state bits 31..0 (for seed)
PRNG1	0x24	0	RW	LFSR state bits 63..32 (for seed)
PRNG2	0x28	0	RW	LFSR state bits 95..64 (for seed)
PRNG3	0x2C	0	RW	LFSR state bits 127..96 (for seed)
PRNG4	0x30	0	RW	LFSR state bits 159..128 (for seed)
PRNG5	0x34	0	RW	LFSR state bits 166..160 (for seed)

Table 3.5.: Register map for Keccak peripherals.

3.4.2. Instantiated variants

The area required for the χ step of the DOM peripherals grows exponentially with the number of slices processed in parallel in this step. It is desirable to keep the number of slices which are processed in parallel in the χ step to a minimum. The linear steps do not see such an exponential increase. When higher throughput is desired, the first thing to increase is the number of slices processed in the linear steps.

For the used IP, it is not possible to choose a different amount of processed slices for the χ step than in any linear step. The exception is, that all linear steps may be done at once, and the χ step being allowed to take several cycles. This configuration was used, as it allows for higher throughput and minimizes the time spent for the linear steps, which are not of interest for the intended evaluation, as it is expected that any information leakage should be extractable faster from the non-linear χ step. Figure 3.25 shows this configuration for a Keccak-f[1600] variant utilizing first-order masking (two shares).

The amount of slices processed in one cycle inside the χ step was determined by the randomness available, which was generated with a PRNG (see Section 3.4.3). The chosen PRNG is able to generate 150-bit per cycle. In the case of second-order protected Keccak variants using 150-bit of randomness allows to process two slices in the χ step in parallel, which increases throughput by x.x times compared to a 1-slice-in-parallel implementation (not accounting the readout of state, which is not required for the measurements). Likewise, the first-order protected variant achieves x.x times the throughput of a 1-slice-in-parallel implementation.

SCA Considerations (combining/read all state). Recombination of the shares in order to get the final result must only be done when the result gets read. Wiring the recombination logic directly to the state would be a fatal flaw and provide virtually no protection against side-channel attacks, apart from an increased noise level. This was taken into account by using operand-isolation, concretely by only recombining the state

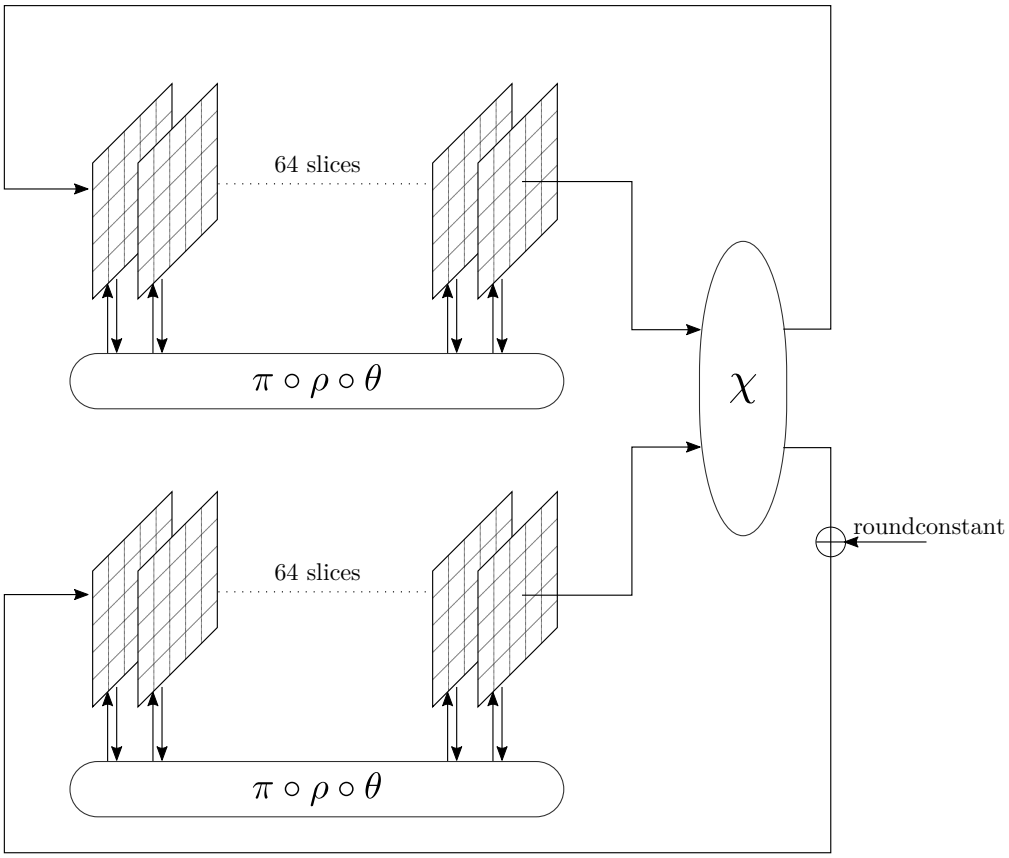

Figure 3.25.: Illustration of the instantiated Keccak-f[1600] configuration for two share domains

when the Keccak permutation is finished and a read from the `DATA` register is performed by the APB master. Figure 3.23 also illustrates this.

3.4.3. Random Number Generator (RNG)

The initial sharing requires 1-bit for every added share domain. Hence when absorbing a 64-bit word of Keccak-f[1600], 64-bit of randomness are required for the first-order protected Keccak variant (two share domains) and 128-bit for a second-order protected variant (three share domains). For the initial sharing it is also possible to buffer the required randomness, making this a secondary issue.

As described by Gross *et al.* [25], the randomness requirement of the DOM-scheme is $(N^2 - N)/2$ per protected 1-bit multiplier (aka AND-gate), with N being the number of share domains. The χ step is the only non-linear step in the design. The Keccak S-Box used in the χ mapping, is a 5-bit S-Box which requires five 1-bit multipliers. Each S-Box operates on a single *Row*. Since the Keccak instances operate in a slice-based manner. Since each slice consists of 5 rows, the χ step requires 25 1-bit multipliers. In summary, the first-order protected Keccak needs 25-bit randomness per slice, the second-order protected variant 75-bit per processed slice.

By using 150-bit of randomness per cycle it is possible to not require an additional buffer mechanism for the initial sharing, which requires 128-bit for the second-order variant. This of course doubles the number of slices that can be processed in parallel in the χ step to four and two slices for the first- and second-order SCA protected Keccak variants respectively.

However, this randomness requirement is huge. High-throughput True-RNGs only reach a throughput of typically below 10 Mbit/s [22, 61]. Hence a Pseudo-RNG (PRNG) must be used. For this purpose a LFSR has been used as a PRNG. While it is not recommended to use a single LFSR as PRNG in a final product, since an attacker might be able to recover the LFSR state through SCA, it is considered feasible for evaluation purposes. For final products a RNG as described by Sunar *et al.* [61] should be used.

For the production of (pseudo-) random numbers needed for the DOM protected χ step of the design, a 167-bit primitive LFSR has been used [4]. The LFSR-recursion is shown in (3.1). The LFSR-recursion has been unrolled to produce a 150-bit pseudo-random number every cycle. While it would be possible to use a LFSR with slightly smaller length, the chosen primitive LFSR is the first with a length greater than 150-bit which only requires a single tap, which decreases the required area, especially when unrolling it. A single tap also minimizes the noise introduced by the LFSR itself, which is desirable for SCA evaluation.

$$S_1 \leftarrow S_{161} \oplus S_{167} \tag{3.1}$$

3.5. seL4

3.5.1. Description

seL4 is a microkernel with a formal proof of security [40, 39], that has its roots in the L4 family of microkernels [28]. More precisely, it provides a proof that the C-implementation of the kernel adheres to its abstract specification. This specification entails a proof of integrity, meaning that no kernel data can be modified from userspace directly, and confidentiality, meaning that no kernel data is accessible from within userspace [58]. From a security point of view it is important to notice, that as with all proofs, they prove exactly what is specified, which is not necessarily (but at least very likely) what was intended.

Parts of the kernel are not covered by the proof. This entails all assembly code and code that manages the MMU and its TLBs [40]. Furthermore, at the time of this writing, only the ARM version of seL4 has been fully verified. The proof for this ARM version again is for one specific configuration of the iMX6 platform, not for all ARM architectures in general. In contrast the x86 and x86_64 versions of seL4 don't yet have a formal proof of correctness.

Proving all platforms and even the existing x86 and x86_64 ports is a time consuming endeavor. Consequently a proof of a seL4 RISC-V port is not within the scope of this thesis, whose main focus is on securing the system against physical attacks.

The seL4 microkernel manages resources with a capability based approach. The initial userspace thread receives full capabilities of all (non-kernel) memory and devices. The capabilities themselves are stored inside the kernel space and modification of the capabilities needs to be done by the kernel. In a secure system, the initial thread will strictly control to which thread requires which capabilities.

3.5.2. Modifications

The RISC-V port has been kept conceptually as close to the ARM version of seL4 as the architecture permits. Because the implemented MMU relies on 4kB pages, the memory management parts are however usually closer to the x86 version than to the ARM version. Apart from memory management, most of the ARM version could be used with small adaptations.

Since the kernel API has stayed untouched, the reader is referred to the seL4 specification for any details [50]. The implementation merely adheres to the specification in this regard.

Modifications in architecture independent code. Currently the full register file is considered as User Context. However, this required to increase the maximum message length by one word from 32 to 33. This part actually affects the architecture independent parts of the kernel as well, and hence other architectures such as ARM and x86.

Further adaptations. While the microkernel is the heart of the seL4 operating system, it is not very useful on its own, due to the very nature of a microkernel. Hence not only the microkernel needed porting to the RISC-V architecture, but also most of the supporting userspace libraries. Again, some of the ports could be based on some work from Almatary [5], although the whole library ecosystem did change significantly after this very basic initial port.

The following libraries have been modified to support the RISC-V kernel port.

- seL4_libs
- seL4_util_libs
- seL4_tests
- seL4_tools
- libmuslc

This is basically the full set of libraries required to write a meaningful minimal program that can run on the seL4 operating system. Again, since no API modifications have been performed, the reader is referred to the documentation [50].

3.5.3. Limitations and further considerations

The RISC-V version of seL4 is functional, but lacks a few features in comparison with the mature ARM and x86 versions. For once, the IPC fastpath [28] is only partly implemented, which means, that this functionality cannot be used at all currently. Furthermore, not all tests are currently passing. Furthermore a lot more testing is definitely needed before the port can be considered really stable.

The RISC-V seL4 version assumes a sv32 MMU as described in the privileged ISA 1.9.1, although not every feature is required. It currently doesn't make use of:

- 34-bit physical address space (only 32-bit is supported)
- User memory protection from within supervisor mode (see `PUM` field in `MSTATUS`)
- The memory privilege functionality (see `MPRV` field in `MSTATUS`)
- Making execute-only pages temporarily readable (see `MXR` field in `MSTATUS`)
- Multiple ASIDs (only a 1-bit ASID is used)

- Fine grained TLB flushes (always the full TLB is flushed)
- Global TLB entries
- Accessed and Dirty bits (swapping is unsupported)

The port was done for an older release of seL4 and later updated to the 3.2 release. However, during the course of this thesis there were two more major releases of seL4. The impact on the current port has not been analyzed yet, and upgrading to the new release version might break the port seriously.

Chapter 4

Results

The implementations of the Remus core and Keccak peripherals, as described in Chapter 3, have been taped out in a UMC65 process. Due to the memory requirement of the seL4 operating system most of the chip area is occupied by dense RAM macros (`SHKA65_16384X8X4CM16`, respectively `SHKA65_8192X8X4CM16`). Synthesis was performed with Synopsys Design Compiler 2016.12, which was provided with UMC's low leakage low-, regular- and high-voltage threshold standard cell libraries (`UMK65LSCLLMVBBL_B`, `UMK65LSCLLMVBBR_B`, `UMK65LSCLLMVBBH_B`) to choose the gate mappings from. Further backend design was performed with Cadence Innovus 2016.10. A 10ns clock period and the worst-case delay corner (1.08V/125°C) were used as constraints throughout the backend design for the Remus core.

An overview of the chip's layout is given in Figure 4.1 . The major parts of the chip have been colored and annotated.

This chapter focuses on the Remus and Keccak core and neglects all other peripherals, the μDMA and the other cores (Eeny and ZeroRISC) implemented on the same chip, which were not directly part of this work.

4.1. Remus - Overview

4.1.1. Area

In the taped-out Patronus chip, the Remus core occupies an area of about 89.5 kGE, for a clock period constraint of 10ns with worst-case delay corner libraries (1.08V / 125°C).

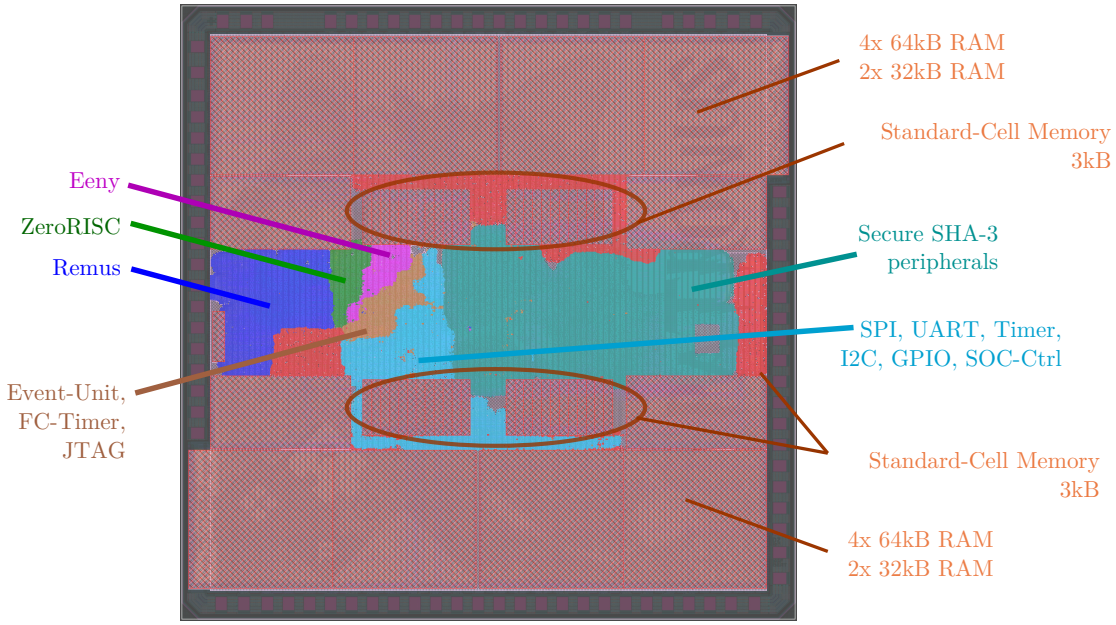

Figure 4.1.: Patronus layout. Major parts of the design are highlighted and annotated.

	IF-stage	SCFP-unit	ID-stage	EX-stage	LSU	MMU	CSRs	Total
Area (in kGE)	4.0	25.3	16.7	16.2	2.4	3.8	10.3	79.0
Area (in %)	5.1	32.0	21.2	20.6	3.0	4.8	13.0	100

Table 4.1.: Remus post-synthesis area for worst-case libraries (1.08V,125°C)

A more detailed analysis of the core is given in Table 4.1 . These values are obtained post-synthesis with the already mentioned constraints of 10ns clock period, worst-case delay corner.

As can be observed, the SCFP unit causes an area overhead of roughly 47%. The largest contributor to this area increase is the used Prince block cipher with 23kGE (91% of the increase). A more detailed discussion is given in Section 4.2.4 .

4.2. Sponge-Based Control-Flow Protection

4.2.1. Fault Detection

The idea behind Sponge-Based Control-Flow Protection (SCFP), is to use the property of the decryption, that a modified ciphertext will result in random plaintext. Hence it defines a secure state as execution of random plaintext. Since every ciphertext depends on the correct decryption of all previous ciphertexts, an error in the ciphertext stream, due

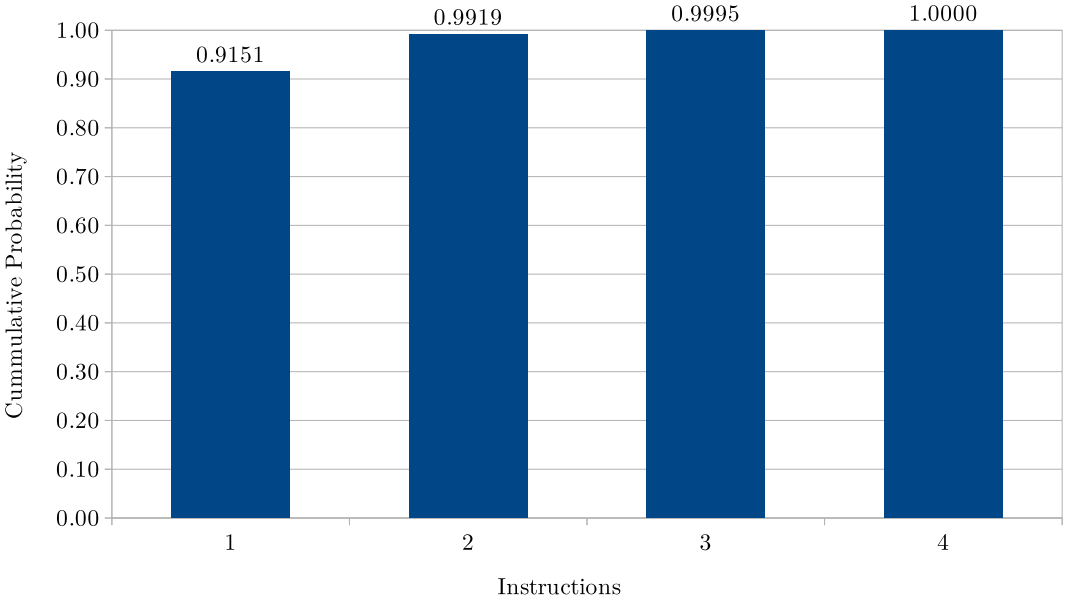

Figure 4.2.: Cumulative probability of triggering an exception within a certain amount of instructions for triggering 16483 random code executions due to invalid sponge states and decryption keys.

to faults, immediately, irreversibly, triggers execution of random plaintext. The program will need to be restarted in order to behave normally again. The same principle applies if the attacker manages to either corrupt the capacity or the key of a keyed permutation.

In any case it might still be desirable to detect a corruption. A nice property of SCFP is, that no explicit comparisons with reference signatures are needed, but this also implies that detection is not directly part of the scheme. However, as it turns out, the probability of triggering an exception when executing random instructions is high. In the concrete implementation in the Remus core the cumulative probability of triggering an exception within a certain number of executed instructions is given in Figure 4.2 . As can be seen it is highly likely, that the first random instruction will already trigger an exception. Detection probability then further rises with each executed instruction.

The measurements for these probabilities have been obtained, by changing the sponge capacity and the key in a random fashion inside a common trap handler. When returning to the actual program code, decryption will fail, respectively output random plaintext. How many instructions it takes until an exception gets triggered was measured by simply setting up the hardware performance counter that measures this metric. Care was taken, that no conditional branching is performed after setting up the performance counter, since that would cause invalid measurements.

The minimum value of the performance counter then was confirmed to correspond to

an exception at the first decrypted instruction. Assuming that the used PRINCE cipher doesn't produce output that can be correlated to its input, this approach should yield representative measurements for faults on ciphertext, sponge capacity and keys. Obviously this does only cover faults prior to the instruction decode stage, since faults after the SCFP stage don't result in random code execution and hence are in general not protected by SCFP and consequently also not detectable.

It should further be noted, that while this detection mechanism is for free, it cannot be relied on. The reasoning for this is the same as the reasoning why a checking mechanism might not be considered very reliable. A triggered exception boils down to a one bit value at some point. If the attacker can get a hold on this signal, detection is no longer possible. In the concrete case of the REMUS core however, it is at least possible that the random instruction causes different types of exceptions. The possible exceptions are load-,store-,fetch- and illegal instruction exceptions. Fully controlling multiple of these signals might be considered infeasible, hence the detection mechanism is arguably slightly more robust than a comparison which only yields a 1-bit value.

4.2.2. Runtime Overhead

The runtime overhead due to the SCFP unit is shown in Table 4.2 . The overhead ranges from 3% to 18%, dependent on the relative amount of control-transfer instructions executed for the respective benchmark. With an average of 10% overhead for our benchmark set, the runtime impact of SCFP is considered small when compared to implementations of signature-checking and software-only approaches.

The implementation of Werner *et al.* [70] has an overhead of 2% to about 71% when implementing continuous signature checking for a Cortex-M3. The software approach from Lalande *et al.* [44], increases execution time by 48% to 106% for an x86 target machine and 109% to 400% for a Cortex-M3 processor. However, it is important to note that a direct comparison to these implementations is not possible, because they implement their methods for x86 and ARM, but not for RISC-V. In particular, every processor obviously needs custom tailoring to minimize the overhead of the SCFP pipeline stage. Furthermore, these methods neglect the protection of the branch decision and the signature check.

The runtime increase is roughly a direct measure of the amount of branches (unconditional jumps are comparatively rare in all tests, see Chapter A), scaled by the average IPC for the respective test. The reason for this direct relation is, that every branch causes exactly one extra cycle of delay in practice, either directly by skipping the patch value, or indirectly when attributing the delay due to the additional SCFP pipeline stage to the branching instruction itself.

In Figure 4.3 the results are shown in a visually more comprehensive way. The short-running benchmarks are omitted for space reasons. Exact numbers of all benchmarks are available in Appendix A.

Benchmark	cycles SCFP inactive	cycles SCFP active	Percentage of CF-Instructions	Runtime overhead
aes_cbc	199199	217785	11.13%	9.33%
bubblesort	148946	169186	20.08%	13.59%
conv2d	28312	30020	6.35%	6.03%
Coremark	435468	505134	19.61%	16.00%
crc32	1741517	1873001	9.11%	7.55%
dhrystone	721701	848040	21.85%	17.51%
fdctfst	8583	8959	4.00%	4.38%
fft	243162	260396	8.39%	7.09%
fir	120138	130296	12.05%	8.46%
ipm	17338	19435	16.36%	12.09%
keccak	817094	871456	7.20%	6.65%
matrixAdd	11233	12316	12.63%	9.64%
matrixMul16_dotp	5931	6573	13.51%	10.82%
matrixMul16	800764	871770	11.27%	8.87%
matrixMul32	42320	43572	3.06%	2.96%
matrixMul8_dotp	5927	6574	13.53%	10.92%
matrixMul8	732808	804033	12.63%	9.72%
sha	228109	238388	4.80%	4.51%
stencil	1407	1589	14.39%	12.94%
sudokusolver	190657	225320	25.99%	18.18%
towerofhanoi	60137	63317	6.51%	5.29%
Average			12.12%	9.64%

Table 4.2.: The relative amount of executed control-flow instructions and resulting runtime overhead for the used set of benchmarks.

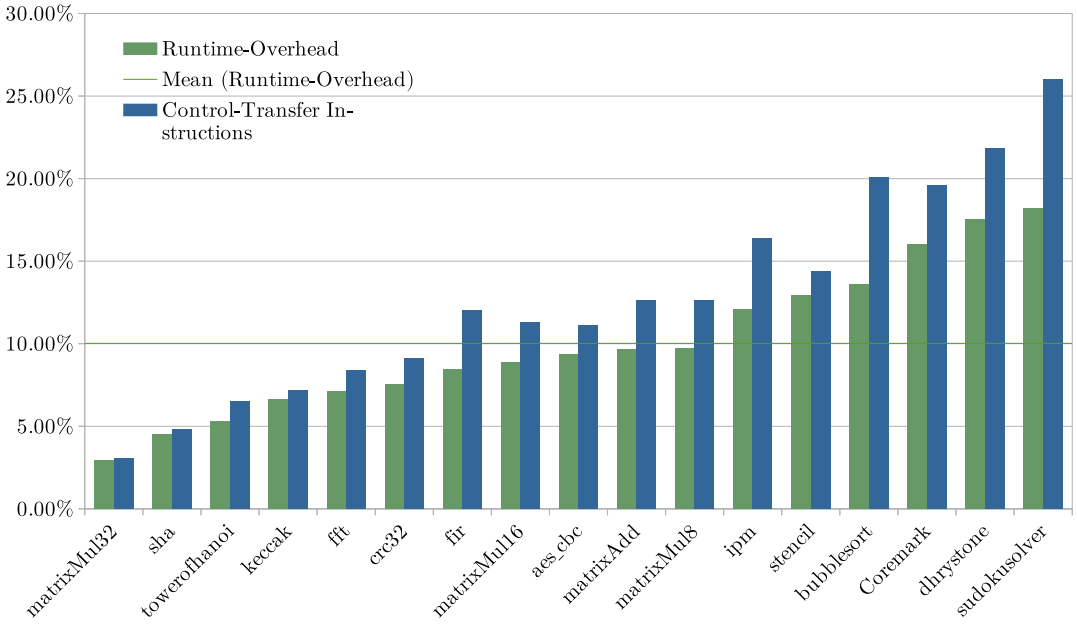

Figure 4.3.: Runtime overhead for the set of benchmarks.

	Score (Iterations/sec)		Score / MHz	
	SCFP inactive	SCFP active	SCFP inactive	SCFP active
Coremark	114	99	2.28	1.98
Dhrystone	69283	58961	0.79	0.67

Table 4.3.: Coremark and Dhrystone benchmark results for REMUS with and without active SCFP unit at a simulation clock frequency of 50MHz.

Due to the popularity of the Coremark and Dhrystone benchmarks, Table 4.3 shows the runtime scores for them with and without active SCFP unit. The results were obtained with a simulation clock frequency of 50MHz. The more usual DMIPS measure for Dhrystone is obtained by dividing the score by 1757, which is the number of iterations per second on a VAX 11/780 machine, which is nominally a 1 DMIPS machine.

4.2.3. Binary size overhead

Due to the additional patch values inserted after control-transfer instructions, the code section of the program necessarily increases. Table 4.4 shows the increase of the `.text` section. For the used benchmarks, the increase ranges from 5% to 20%, with an average of 13%.

Benchmark	Baseline (in bytes)	SCFP instrumented (in bytes)	Overhead
fir	3096	3256	5.17%
aes_cbc	7708	8204	6.43%
keccak	5952	6364	6.92%
fft	2516	2716	7.95%
matrixMul16	12412	13408	8.02%
matrixMul32	9876	10860	9.96%
sha	4928	5464	10.88%
matrixMul8	9348	10380	11.04%
ipm	6136	6832	11.34%
towerofhanoi	6588	7604	15.42%
dhrystone	8532	9884	15.85%
sudokusolver	8776	10172	15.91%
crc32	5676	6612	16.49%
coremark	19120	22324	16.76%
stencil	5436	6400	17.73%
matrixAdd	4704	5620	19.47%
bubblesort	4668	5596	19.88%
Average			12.66%

Table 4.4.: Increase of binary `.text` section due to SCFP.

	Area (in kGE)				
	30ns	20ns	13ns	10ns	6.7ns
IF-stage	3.3	3.6	3.4	4.0	5.5
SCFP-unit	10.9	10.9	13.4	25.3	26.0
Prince (12 rounds)	8.9	8.8	11.4	23.0	23.3
ID-stage	15.3	15.4	16.0	16.7	17.6
EX-stage	14.4	14.5	15.5	16.2	16.7
WB-stage	1.8	1.8	1.9	2.4	3.2
MMU	3.4	3.5	3.7	3.8	3.9
iTLB	1.4	1.5	1.5	1.5	1.6
dTLB	1.4	1.5	1.5	1.6	1.6
CSR	8.9	8.9	9.9	10.3	10.3
Remus Total	58.3	58.8	64.2	79.0	83.6
SCFP Overhead	23.1%	22.8%	26.6%	47.3%	45.4%

Table 4.5.: Remus post-synthesis area requirement for different clock constraints, using worst-case libraries (1.08V/125°C).

4.2.4. Area Requirement

The area requirement for the Remus core for different constraints is shown in Table 4.5 . As can be seen, the area requirement of the SCFP unit depends heavily on the used constraint. This is expected, since the Prince cipher is fully unrolled. Usage of a different cipher could potentially yield significant improvements here.

While the Prince cipher is close to, it is never the most critical path, which is always the path to memory in both typical-case and worst-case scenarios.

4.2.5. Power Analysis

The power analysis was performed for multiple benchmarks of the list given in Table 4.2 . Short-running benchmarks were preferred as post-layout simulation with exact timing annotations took a significant amount of time.

For the power analysis itself, the dumped VCD files from the post-layout simulations were used. For the simulation, the RC delays were extracted from the layout and used to back-annotate the simulated netlist. The simulation clock frequency was 50MHz. For power analysis the typical case view with 1.2V core supply voltage at 25°C was used. The VDDIO and VSSIO rails were not considered in this analysis.

The simulation and power analysis was performed for programs with and without active SCFP unit. The results can be seen in Table 4.6 . On average the power consumption

Benchmark	Remus Power (in mW) SCFP inact.	SCFP act.	Overhead	Patronus Power (in mW) SCFP inact.	SCFP act.	Overhead
aes_cbc	7.45	9.25	24.10%	13.59	15.36	13.02%
bubblesort	6.18	8.16	32.13%	13.21	15.05	13.93%
conv2d	7.40	8.97	21.22%	14.48	15.48	6.91%
fdctfst	7.56	9.35	23.68%	13.89	15.55	11.95%
fft	7.31	9.13	24.93%	13.93	15.42	10.70%
fir	6.98	9.20	31.78%	13.45	15.71	16.80%
keccak	7.71	9.57	24.22%	14.52	16.38	12.81%
matrixAdd	7.18	9.06	26.12%	13.54	15.50	14.48%
matrixMul16_dotp	7.56	9.23	22.10%	13.89	15.75	13.39%
matrixMul8_dotp	7.36	9.13	23.97%	13.94	15.68	12.48%
sha	7.88	9.80	24.31%	14.04	15.85	12.89%
stencil	6.71	8.53	27.02%	13.58	15.25	12.30%
Average	7.27	9.11	25.46%	13.84	15.58	12.60%

Table 4.6.: SCFP power overhead for selected set of benchmarks for Remus and Patronus with 50MHz simulation clock frequency.

overhead of SCFP was determined to be 13% for the whole chip and 25% when only the Remus core is considered.

4.3. Keccak peripherals

4.3.1. Area requirement

The area requirements of the instantiated Keccak peripherals are shown in Table 4.7 . The shown numbers are post-synthesis results for a constraint of 10ns clock period and worst-case delay corner (1.08V/125°C). In contrast to other parts of the circuit the Keccak peripherals are hardly influenced by more stringent clock constraints, and no significant difference in area could be observed for clock periods between 6.7ns and 30ns.

The biggest contributors to area are the linear steps (θ, ρ, π), since they are applied in a single cycle and hence operate on the whole 1600-bit state simultaneously. It can be seen that all the linear parts of the Keccak peripheral (θ, ρ, π, State) are increasing linear with the protection order, as is expected for a DOM protected peripheral, since they are basically just replicated for each share-domain [25, 26].

The non-linear χ step operates only on four and two slices respectively for the protected Keccak instances and hence doesn't contribute as much to the overall area. Due to the $\iota \circ \chi$ steps of the 2^{nd}-order protected peripheral acting only on two slices, in contrast to

	Area (in kGE)				
	θ	$\pi \circ \rho$	$\iota \circ \chi$	State	Total
Unprotected	6.4	2.4	5.6	18.0	32.8
1^{st}-order	13.1	4.2	4.1	36.3	58.0
1^{st}-order (less rand.)	13.0	4.0	3.6	36.1	57.1
2^{nd}-order	19.3	6.3	4.6	54.0	84.6
PRNG					1.9
Buffer&Control					21.6
Sum					255.9

Table 4.7.: Area requirement of the individual Keccak peripherals in kGE.

the four slices that the 1^{st}-order protected peripherals are operating on, the area required by $\iota \circ \chi$ is roughly equal for the different protection orders.

4.3.2. Throughput

In terms of throughput it was desired, that it would be as high as possible, in order to get useful measurements in a reasonable time. Sending a data word to the peripheral takes 3 cycles per 32-bit word. In the end, filling the whole 1600-bit sponge state thus takes 150 cycles. Likewise, reading the whole state also requires 150 cycles. As already noted, it is not required to write the whole state. Writing only part of the state and triggering a permutation by writing to the control register as described in Chapter 3 is possible as well.

For the SCA-protected KECCAK variants, the number of cycles for the actual permutation is limited by the slices that can be processed in parallel in the χ step. An overview is given in Table 4.8 .

For one round of the first-order protected variants, this means 1 cycle for the linear steps and 16 ($= 64/4$) cycles for the χ step with one additional cycle to fill the DOM-pipeline of the χ step itself. For 24 rounds this results in 432 ($= 18 \cdot 24$) cycles.

For one round of the second-order protected variant, the linear steps are also done in 1 cycle, but 32 ($= 64/2$) cycles are needed for the χ step again with one additional cycle for filling of the DOM-pipeline. This results in 816 ($= 34 \cdot 24$) cycles for 24 rounds.

The unprotected variant, used as comparison, does one round per cycle. Effectively requiring 24 cycles for a full permutation.

	Time for one permutation (in cycles)		
	$\theta \circ \pi \circ \rho$	$\iota \circ \chi$	Total (24 rounds)
Unprotected	1/round	with lin. steps	24
1^{st}-order	1/round	17/round	432
1^{st}-order (less rand.)	1/round	17/round	432
2^{nd}-order	1/round	33/round	816

Table 4.8.: Area requirement of the individual Keccak peripherals in kGE.

Chapter 5

Conclusion and Future Work

In this work, a secure processor design was implemented based on the `Ri5CY` core. It implements a soon to be published scheme of protecting the control-flow of programs against fault-attacks. It was shown that this scheme can be efficiently implemented with a minimal runtime overhead that ranges from 3% to 18% and a code size overhead of 5% to 20%, for the set of benchmarks adjusted for the scheme. It requires what can be considered a reasonable amount of area ($\tilde{2}$5kGE) which corresponds to an area overhead of roughly 45%, which is significantly better than approaches that require full redundancy. However using other sponge permutations and possibly round reduced versions of them should be considered for future work, since the permutation is the biggest contributor to the area requirement.

Furthermore a minimal version of the privilege ISA 1.9.1 was implemented. This also entails a Memory Management Unit (MMU), so the implemented core is now able to run programs and operating systems which rely on page-based virtual memory for isolation. The small version of the MMU requires less than 4kGE of area, doesn't require an additional pipeline stage, has separate instruction- and data-TLBs and provides a hardware page table walker. MMU and privilege ISA were designed to allow to run a port of the formally verified seL4 microkernel. This seL4-port for the RISC-V architecture was also written as part of this work and was based on previous porting efforts for older versions of seL4.

Additionally, multiple side-channel-analysis (SCA) resilient versions of Keccak peripherals were implemented, which utilize the recently published Domain-Oriented-Masking (DOM) scheme. The implemented peripherals serve as cryptographic accelerators with the main purpose of evaluating the efficacy of the DOM scheme against SCA-attacks in an ASIC implementation. One of the implemented DOM-protected SHA-3 peripherals is designed to provide 2nd-order protection against SCA, making it the first higher-order protected implementation of SHA-3 taped-out in an actual ASIC.

A variety of things can be considered in potential future work.

5.1. Sponge-Based Control-Flow Protection

It would be possible to not only protect the instruction stream until the decode stage, but beyond. This can be done by feeding back signals from later stages into the sponge construction. E.g. signals which indicate the type of instruction make sense in this regard. The pipeline would likely need modifications for this, as it is currently possible, that the decoding stage executes instructions like unconditional jumps without having to wait on the execute stage.

Because the PRINCE cipher, which was used as the sponge's permutation function, turns out to grow rapidly in area when even decent speed is required, alternative ciphers and permutations should be evaluated. Additionally, finding a permutation small enough to be merged into the instruction decode stage would also be interesting. This would again require changes in the current pipeline model, as the execution of jumps and other instructions which are currently committed in the decode stage should be moved into the execute stage.

The current tooling makes the whole process of writing software that can utilize the SCFP unit very tedious, as parts need to be implemented in hand-written assembly and automatic instrumentation of intermediate assembly can be unreliable, which can lead to decryption failures during runtime. Thus further work on the toolchain is needed to make this scheme practical for a broader range of applications.

5.2. Keccak / SHA-3 peripherals

While the SHA-3 peripherals are secure against side-channel analysis, they don't provide protection against fault-attacks. As only few fault-attack resilient implementations of KECCAK have yet been proposed, this would be a potentially interesting research topic.

Furthermore, a true or hybrid random number generator should be implemented instead of the currently used PRNG.

5.3. seL4

The current RISC-V port of seL4 must be considered as a minimal implementation that is inefficient in certain cases (e.g. very conservative flushing of TLBs) and provides less functionality than other ports (e.g. no "fastpath" syscalls). Furthermore a new version of seL4 was released after finishing the RISC-V port. Bringing the port up-to-date and

implementing missing functionality would be desirable. Also, the port is by no means formally verified, which is a project totally out-of-scope for this work.

5.4. MMU and Privileged Specification

Once the privilege specification's development slows down, a reevaluation of the current implementation will need to be performed. Especially the current interrupt and exception mechanisms and the performance counters, which are deviations from the specification would need to be adjusted.

It should be considered to move the MMU into its own pipeline stage. This not only simplifies the design, but would allow to implement versions with increased TLB sizes that don't severely impact the critical path.

Appendix A

Detailed Evaluation Results

All performance measurements for the SCFP-unit are given in Table A.1 . The benchmarks with a "-nocfi" suffix do not utilize the SCFP unit, while the ones with the "-cfi" suffix do. The amount of instructions differs very slightly between the two variants of a benchmark. This is due to the fact, that the instrumented CFI assembly can't cope with tail calls. Tail call have thus been substituted with a usual push-call-pop-return sequence.

Benchmark	Timer	Performance counter values									
	cycles/iteration	cycles	instr.-cnt	Ld-stall	Jr-stall	imiss	mem_load	mem_store	jumps	branch	Branch-taken
aes_cbc-cfi	42533	217785	171137	340	5	764	17824	8113	2923	16119	9639
aes_cbc-nocfi	38958	199199	171112	340	0	804	17819	8108	2923	16119	9639
bubblesort-cfi	168885	169186	100819	19300	1	1126	39632	20033	11	20236	10113
bubblesort-nocfi	148700	148946	100812	19800	0	496	39631	20031	11	20236	10113
conv2d-cfi	5765	30020	23922	0	5	232	9069	393	63	1454	1019
conv2d-nocfi	5459	28312	23897	0	0	262	9064	388	63	1454	1019
Coremark-cfi	523093	505134	335595	18799	4	2386	56531	14011	7155	58665	30180
Coremark-nocfi	450996	435468	335574	18805	0	1706	56527	14007	7155	58665	30180
crc32-cfi	1870755	1873001	1443649	0	1	5398	131331	269	9	131457	131326
crc32-nocfi	1739539	1741517	1443642	0	0	5278	131330	267	9	131457	131326
Dhrystone-cfi	848030	848040	508025	4000	0	9591	112001	70002	24006	87000	56999
Dhrystone-nocfi	721692	721701	508025	4000	0	6668	112001	70002	24006	87000	56999
fdctfst-cfi	1457	8959	7707	0	5	37	834	853	63	244	229
fdctfst-nocfi	1426	8583	7682	0	0	65	829	848	63	244	229
fft-cfi	48893	260396	206877	600	5	1494	33154	25513	703	16654	10889
fft-nocfi	45801	243162	206852	591	0	1212	33149	25508	703	16654	10889
fir-cfi	26005	130296	95262	0	5	602	19009	978	63	11409	9499
fir-nocfi	23975	120138	95237	0	0	886	19004	973	63	11409	9499
ipm-cfi	3634	19435	12262	1125	5	81	3004	2463	83	1919	1384
ipm-nocfi	3230	17338	12237	1107	0	115	2999	2458	83	1919	1384
keccak-cfi	173718	871456	733432	13418	0	3328	138809	102503	12158	40649	15259
keccak-nocfi	162882	817094	733432	13420	0	2162	138809	102503	12158	40649	15259
matrixAdd-cfi	12244	12316	8450	0	1	106	2050	1041	9	1057	1023
matrixAdd-nocfi	11173	11233	8443	0	0	101	2049	1039	9	1057	1023
matrixMul16_dotp-cfi	6025	6573	4559	0	1	46	1158	271	13	602	518
matrixMul16_dotp-nocfi	5404	5931	4552	0	0	53	1157	269	13	602	518
matrixMul16-cfi	418511	871770	628868	0	3	9576	136212	6183	21	70852	68667
matrixMul16-nocfi	384774	800764	628851	0	0	5460	13209	6179	21	70852	68667
matrixMul32-cfi	17647	43572	37392	0	5	225	14127	2615	19	1124	1067
matrixMul32-nocfi	17333	42320	37375	0	2	189	14124	2611	19	1124	1067
matrixMul8_dotp-cfi	6010	6574	4554	0	1	48	1157	270	13	602	518
matrixMul8_dotp-nocfi	5400	5927	4547	0	0	55	1156	268	13	602	518
matrixMul8-cfi	394404	804033	561302	2	7	6210	135201	4664	23	70853	68668
matrixMul8-nocfi	359521	732808	561285	2	4	6410	135198	4660	23	70853	68668
sha-cfi	47629	238388	204932	1300	10	779	21744	9473	648	9194	8464
sha-nocfi	45574	228109	204882	1300	0	620	21734	9463	648	9194	8464
stencil-cfi	1272	1589	1077	65	1	16	269	266	26	128	65
stencil-nocfi	1135	1407	1070	65	0	8	268	264	26	128	65
sudokusolver-cfi	224991	225320	137245	27420	2	421	36208	15761	2385	33278	7983
sudokusolver-nocfi	190386	190657	137238	27420	1	427	36207	15759	2385	33278	7983
towerofhanoi-cfi	62917	63317	47170	3248	2	216	20358	16851	925	2143	1011
towerofhanoi-nocfi	59786	60137	47163	3195	1	268	20357	16849	925	2143	1011

Table A.1.: Detailed performance metrics for the set of benchmarks used to evaluate the SCFP unit in the Remus core.

Bibliography

[1] Reset Glitch Hack - Free60. `http://free60.org/wiki/Reset_Glitch_Hack`.

[2] SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. Technical Report NIST FIPS 202, National Institute of Standards and Technology, July 2015.

[3] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.

[4] P. Alfke. Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators, July 1996.

[5] H. M. K. Almatary. seL4 microkernel RISC-V port. `https://github.com/heshamelmatary/seL4-riscv-mk`, 2015.

[6] H. M. K. Almatary. *Operating System Kernels on Multi-Core Architectures*. Msc by research, University of York, Jan. 2016.

[7] E. Andreeva, B. Bilgin, A. Bogdanov, A. Luykx, B. Mennink, N. Mouha, and K. Yasuda. APE: Authenticated permutation-based encryption for lightweight cryptography. In *International Workshop on Fast Software Encryption*, pages 168–186. Springer, 2014.

[8] ARM. Cortex-A5 MPCore Technical Reference Manual. Technical Report ID021016, ARM, May 2010.

[9] ARM Limited. ARM1156T2F-S^{TM} Technical Reference Manual. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0290g/Chdigebi.html`, 2007.

[10] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha. Hardware-Assisted Run-Time Monitoring for Secure Program Execution on Embedded Processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(12):1295–1308, Dec. 2006.

[11] N. Bagheri, N. Ghaedi, and S. K. Sanadhya. Differential Fault Analysis of SHA-3. In *Progress in Cryptology – INDOCRYPT 2015*, pages 253–269. Springer, Cham, Dec. 2015.

[12] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. On the Indifferentiability of the Sponge Construction. In *Advances in Cryptology – EUROCRYPT 2008*, pages 181–197. Springer, Berlin, Heidelberg, Apr. 2008.

[13] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications. In *Selected Areas in Cryptography*, pages 320–337. Springer, Berlin, Heidelberg, Aug. 2011.

[14] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, and R. V. Keer. CAESAR submission: Keyak v2. `http://keyak.noekeon.org/Keyakv2-doc2.2-diff.pdf`, Sept. 2016.

[15] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Cryptographic sponge functions. *Submission to NIST (Round 3)*, 2011.

[16] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. On the security of the keyed sponge construction. In *Symmetric Key Encryption Workshop (SKEW 2011)*, 2011.

[17] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer. CAESAR submission: Ketje v2. `http://ketje.noekeon.org/Ketjev2-doc2.0.pdf`, Sept. 2016.

[18] J. Blömer, M. Otto, and J.-P. Seifert. A New CRT-RSA Algorithm Secure Against Bellcore Attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 311–320, New York, NY, USA, 2003. ACM.

[19] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In *Cryptographic Hardware and Embedded Systems - CHES 2007*, pages 450–466. Springer, Berlin, Heidelberg, Sept. 2007.

[20] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In *Advances in Cryptology — EUROCRYPT '97*, pages 37–51. Springer, Berlin, Heidelberg, May 1997.

[21] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, and others. PRINCE–a low-latency block cipher for pervasive computing applications. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 208–225. Springer, 2012.

[22] M. Bucci, L. Germani, R. Luzzi, A. Trifiletti, and M. Varanonuovo. A high-speed oscillator-based truly random number source for cryptographic applications on a smart card IC. *IEEE transactions on computers*, 52(4):403–409, Apr. 2003.

[23] A. Francillon, D. Perito, and C. Castelluccia. Defending embedded systems against control flow attacks. In *Proceedings of the First ACM Workshop on Secure Execution of Untrusted Code*, pages 19–26. ACM, 2009.

[24] Z. Gong, S. Nikova, and Y. W. Law. KLEIN: A New Family of Lightweight Block Ciphers. In *RFID. Security and Privacy*, pages 1–18. Springer, Berlin, Heidelberg, June 2011.

[25] H. Gross, S. Mangard, and T. Korak. Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order. *IACR Cryptology ePrint Archive*, 2016:486, 2016.

[26] H. Gross, D. Schaffenrath, and S. Mangard. D-th Order Threshold Implementations of Keccak. unpublished.

[27] D. Gruss, C. Maurice, and S. Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 300–321. Springer, Cham, July 2016.

[28] G. Heiser and K. Elphinstone. L4 Microkernels: The Lessons from 20 Years of Research and Deployment. *ACM Trans. Comput. Syst.*, 34(1):1:1–1:29, Apr. 2016.

[29] D. Hong, J. Sung, S. Hong, J. Lim, S. Lee, B.-S. Koo, C. Lee, D. Chang, J. Lee, K. Jeong, H. Kim, J. Kim, and S. Chee. HIGHT: A New Block Cipher Suitable for Low-Resource Device. In *Cryptographic Hardware and Embedded Systems - CHES 2006*, pages 46–59. Springer, Berlin, Heidelberg, Oct. 2006.

[30] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual. `https://software.intel.com/en-us/articles/intel-sdm`, Sept. 2016.

[31] Y. Ishai, A. Sahai, and D. Wagner. Private circuits: Securing hardware against probing attacks. In *Annual International Cryptology Conference*, pages 463–481. Springer, 2003.

[32] M. Joye, P. Manet, and J.-B. Rigaud. Strengthening hardware AES implementations against fault attacks. *IET Information Security*, 1(3):106, 2007.

[33] H. Kaeslin. *Digital Integrated Circuit Design: From VLSI Architectures to CMOS Fabrication*. Cambridge University Press, 1st edition, Apr. 2008.

[34] B. S. Kaliski, Ç. K. Koç, and C. Paar, editors. *Cryptographic Hardware and Embedded Systems–CHES 2002: 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002: Revised Papers*. Number 2523 in Lecture notes in computer science. Springer-Verlag, Berlin ; New York, 2002.

[35] D. Karaklajic, J.-M. Schmidt, and I. Verbauwhede. Hardware Designer's Guide to Fault Attacks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(12):2295–2306, Dec. 2013.

[36] D. S. Khudia and S. Mahlke. Low Cost Control Flow Protection Using Abstract Control Signatures. In *Proceedings of the 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '13, pages 3–12, New York, NY, USA, 2013. ACM.

[37] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 361–372, Piscataway, NJ, USA, 2014. IEEE Press.

[38] I. Kizhvatov. *Physical Security of Cryptographic Algorithm Implementations*. PhD thesis, University of Luxembourg, 2011.

[39] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive Formal Verification of an OS Microkernel. *ACM Trans. Comput. Syst.*, 32(1):2:1–2:70, Feb. 2014.

[40] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[41] P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *Advances in Cryptology — CRYPTO' 99*, pages 388–397. Springer, Berlin, Heidelberg, Aug. 1999.

[42] O. Kömmerling and M. G. Kuhn. Design Principles for Tamper-Resistant Smartcard Processors. *USENIX Workshop on Smartcard Technology*, 1999.

[43] T. Korak and M. Hoefler. On the Effects of Clock and Power Supply Tampering on Two Microcontroller Platforms. pages 8–17. IEEE, Sept. 2014.

[44] J.-F. Lalande, K. Heydemann, and P. Berthomé. Software Countermeasures for Control Flow Integrity of Smart Card C Codes. In *Computer Security - ESORICS 2014*, pages 200–218. Springer, Cham, Sept. 2014.

[45] C. H. Lim and T. Korkishko. mCrypton – A Lightweight Block Cipher for Security of Low-Cost RFID Tags and Sensors. In *Information Security Applications*, pages 243–258. Springer, Berlin, Heidelberg, Aug. 2005.

[46] P. Maistri. Countermeasures against fault attacks: The good, the bad, and the ugly. In *On-Line Testing Symposium (IOLTS), 2011 IEEE 17th International*, pages 134–137. IEEE, 2011.

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

[48] S. Mangard, N. Pramstaller, and E. Oswald. Successfully Attacking Masked AES Hardware Implementations. In *Cryptographic Hardware and Embedded Systems – CHES 2005*, pages 157–171. Springer, Berlin, Heidelberg, Aug. 2005.

[49] B. Mennink, R. Reyhanitabar, and D. Vizár. Security of Full-State Keyed and Duplex Sponge: Applications to Authenticated Encryption. *IACR Cryptology ePrint Archive*, 2015:541, 2015.

[50] NICTA. seL4 Reference Manual Version 3.1.0. `https://sel4.systems/Info/Docs/seL4-manual-3.1.0.pdf`, June 2016.

[51] S. Nikova, C. Rechberger, and V. Rijmen. Threshold implementations against side-channel attacks and glitches. In *International Conference on Information and Communications Security*, pages 529–545. Springer, 2006.

[52] S. Nikova, V. Rijmen, and M. Schläffer. Secure hardware implementation of non-linear functions in the presence of glitches. *Journal of Cryptology*, 24(2):292–321, 2011.

[53] G. Piret and J.-J. Quisquater. A Differential Fault Attack Technique against SPN Structures, with Application to the AES and Khazad. In *Cryptographic Hardware and Embedded Systems - CHES 2003*, pages 77–88. Springer, Berlin, Heidelberg, Sept. 2003.

[54] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady. Security in Embedded Systems: Design Challenges. *ACM Trans. Embed. Comput. Syst.*, 3(3):461–491, Aug. 2004.

[55] C. Rolfes, A. Poschmann, G. Leander, and C. Paar. Ultra-Lightweight Implementations for Smart Devices – Security for 1000 Gate Equivalents. In *Smart Card Research and Advanced Applications*, pages 89–103. Springer, Berlin, Heidelberg, Sept. 2008.

[56] Y. Sasaki and K. Yasuda. How to Incorporate Associated Data in Sponge-Based Authenticated Encryption. In *Topics in Cryptology — CT-RSA 2015*, pages 353–370. Springer, Cham, Apr. 2015.

[57] B. Selmke, S. Brummer, J. Heyszl, and G. Sigl. Precise Laser Fault Injections into 90 nm and 45 nm SRAM-cells. In *Smart Card Research and Advanced Applications*, pages 193–205. Springer, Cham, Nov. 2015.

[58] T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein. seL4 Enforces Integrity. In *Interactive Theorem Proving*, pages 325–340. Springer, Berlin, Heidelberg, Aug. 2011.

[59] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[60] A. Shoufan. A fault attack on a hardware-based implementation of the secure hash algorithm SHA-512. In *Reconfigurable Computing and FPGAs (ReConFig), 2013 International Conference On*, pages 1–7. IEEE, 2013.

[61] B. Sunar, W. J. Martin, and D. R. Stinson. A Provably Secure True Random Number Generator with Built-In Tolerance to Active Attacks. *IEEE Transactions on Computers*, 56(1):109–119, Jan. 2007.

[62] A. Traber and M. Gautschi. PULPino: Datasheet. `http://www.pulp-platform.org/wp-content/uploads/2017/02/datasheet.pdf`, Nov. 2016.

[63] A. Traber, M. Gautschi, and P. D. Schiavone. RI5CY: User Manual. `http://www.pulp-platform.org/wp-content/uploads/2017/02/ri5cy_user_manual.pdf`, Jan. 2017.

[64] University of California Berkeley. Rocket Chip Generator. `https://github.com/ucb-bar/rocket-chip`, Mar. 2017.

[65] J. G. van Woudenberg, M. F. Witteman, and F. Menarini. Practical Optical Fault Injection on Secure Microcontrollers. pages 91–99. IEEE, Sept. 2011.

[66] D. Vigilant. RSA with CRT: A New Cost-Effective Solution to Thwart Fault Attacks. In *Cryptographic Hardware and Embedded Systems – CHES 2008*, pages 130–145. Springer, Berlin, Heidelberg, Aug. 2008.

[67] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanović. The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.9.1. Technical Report UCB/EECS-2016-161, EECS Department, University of California, Berkeley, Nov. 2016.

[68] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi. The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.1. Technical Report UCB/EECS-2016-118, EECS Department, University of California, Berkeley, May 2016.

[69] M. Werner, T. Unterluggauer, and S. Mangard. Sponge-Based Control-Flow Protection Against Fault Attacks. unpublished.

[70] M. Werner, E. Wenger, and S. Mangard. Protecting the Control Flow of Embedded Processors against Fault Attacks. In *Smart Card Research and Advanced Applications*, pages 161–176. Springer, Cham, Nov. 2015.

[71] K. Wilken and J. P. Shen. Continuous signature monitoring: Low-cost concurrent detection of processor control errors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(6):629–641, 1990.

[72] S.-M. Yen, S. Kim, S. Lim, and S. Moon. RSA Speedup with Residue Number System Immune against Hardware Fault Cryptanalysis. In *Information Security and Cryptology — ICISC 2001*, pages 397–413. Springer, Berlin, Heidelberg, Dec. 2001.