Protecting RISC-V Processors against Physical Attacks

Mario Werner*, Robert Schilling*†, Thomas Unterluggauer*, and Stefan Mangard*
*Graz University of Technology
Email: {firstname.lastname}@iaik.tugraz.at
†Know-Center GmbH

***Abstract*—RISC-V is an emerging instruction-set architecture suitable for a wide variety of applications, which ranges from simple microcontrollers to high-performance CPUs. As an increasing number of commercial vendors now plans to adopt the architecture in their products, its security aspects are becoming a significant concern. For microcontroller implementations of RISC-V, one of the main security risks are attackers with direct physical access to the microchip. These physical attackers can perform highly powerful attacks that span from memory probing to power analysis up to fault injection and analysis.**

In this paper, we give an overview of the capabilities of attackers with direct physical device access, common threat models and attack vectors, and possible countermeasures. Besides, we discuss in more detail current approaches to secure RISC-V processors against fault injection attacks on the microchip itself. First, we show how to protect the control-flow against fault attacks by using an encrypted instruction stream and decrypting it on-the-fly in a newly added pipeline stage between the processor's fetch and decode unit. Second, we show how to protect conditional branches against fault injection by adding redundancy to the comparison operation and entangling the comparison result with the encrypted instruction stream. Finally, we discuss an approach to protect all pointers and memory accesses from tampering.

***Index Terms*—RISC-V, physical attacks, fault injection, countermeasures**

I. Introduction

Software is an essential building block in today's computing devices. To ensure their proper functioning, developers hence need to take care of implementing software correctly, *i.e.*, in the absence of both functional and security flaws. Hereby, software including software countermeasures is commonly designed by assuming the underlying processor executes correctly as well.

However, in embedded applications such as for the growing market of Internet of Things (IoT), this assumption is too strong as these applications often allow attackers to gain direct physical access to the computing hardware. Here, the attacker modifies the internal state of a processor, e.g., by inducing voltage or clock glitches [2], [4] or shooting with a laser on the chip [26]. This physical access facilitates a wide range of attacks that permit to skip instructions, change instruction opcodes or register values, redirecting a memory access, and modifying the control-flow, and thus renders the assumption of correct hardware invalid. While these attacks require physical access, attacks like exploiting the Rowhammer effect [11], [16] can even be induced via Javascript [11] or entirely remotely over a network interface [18], [30]. Eventually, even correctly implemented software without security vulnerabilities may deliver wrong results. Further, any software-based security mechanisms cannot be trusted anymore since they always assume the correct execution of the software.

Consequently, embedded computing platforms must incorporate security mechanisms to cope with this threat and protect the system against fault attacks. As RISC-V is an emerging open Instruction-Set Architecture (ISA) particularly in the field of embedded devices, RISC-V based platforms are increasingly susceptible to physical fault attacks and hence a highly attractive platform for designing and evaluating physical fault attacks and countermeasures.

Contribution: In this paper, we present an overview of techniques to protect RISC-V based embedded computing devices against fault attacks. First, we present Sponge-based Control-Flow Protection (SCFP), which is a mechanism to protect a processor's instruction stream against tampering by applying on-the-fly authenticated decryption. Doing so enforces Control-Flow Integrity (CFI) meaning that the software execution has to adhere to its Control-Flow Graph (CFG). Second, assuming CFI is enforced, we present a concept to protect conditional branches which ensure that, even under fault attacks, only genuine control-flow transfers within the protected CFG are possible. The concept utilizes a combination of AN-encoded data values, specially developed encoded comparison algorithms and linked branch operations to obtain this protection. Third, we present a new approach for securing memory accesses against tampering by transforming address errors into data errors. In this approach, we encode all data pointers and perform their pointer arithmetic in a protected domain where faults are detectable. These encoded pointers are used in linked memory accesses, where the address information is linked with the encoded payload data. Subsequently, accessing data with an incorrect address destroys the redundancy of the payload data.

Outline: The remainder of this paper is structured as follows. Section II introduces the attack vectors and discusses common threat models. The techniques we present to secure RISC-V processors against fault attacks are presented in Section III. Finally, Section IV briefly discusses related work on physical attacks beyond RISC-V and faults.

II. Attack Vectors and Common Threat Models

Physical attacks are a diverse topic with various facets. This section outlines the most common attack vectors which are encountered in the context of active physical attacks and discusses commonly used threat models.

A. Physical Attacks

Adversaries with physical access to a system can employ various techniques to tamper with the device and how it executes software. Most intuitively, all types of non-volatile external memory (e.g., on-board flash chips, SD-cards, HDDs, SSDs, ...) can easily be read or written by attackers. Subsequently, without special protection, manipulation of code and data in memory is possible. Furthermore, even when cryptography is used to enforce the memory content's confidentiality and integrity, replay attacks, like the NAND mirroring attack that targeted the iPhone [28], are often still possible. Similarly, volatile memory (e.g., DDR-RAM) can also be attacked. Cold boot attacks [13], for example, permit to extract secret data from memory when no encryption is used. Finally, even on live systems, interposing the bus communication between processor and memory is possible and common practice for debug purposes.

However, physical attacks on general purpose systems are by far not limited to external memory. Micro-probing of the chip and side channel attacks can, for instance, acquire information about the processed data. Similarly, fault attacks on the processor can tamper with the correct execution of software [4]. Such fault attacks intentionally violate presumed invariants of a device's operating conditions with the goal to introduce faults into the performed calculations. Commonly used invariants for the injection of faults are, for example, the supply voltage [3], the maximum frequency [2], [4], the allowed temperature range [27], and the tolerated amount of injected photoelectric [26], [29] or electromagnetic energy [21]. Surprisingly, even software-induced fault injection [11], [16], [18], [30] has already been demonstrated.

Especially in the context of general purpose processors, fault attacks have been shown to be very powerful. Adversaries can use precisely timed voltage and clock glitches to skip and repeat instructions deterministically [15], [17]. The Xbox 360, for example, has been compromised by such an attack [1]. Similarly, a targeted single bit flip in the `sudo` binary of a contemporary Ubuntu installation, which can be caused by, e.g., a Rowhammer attack or any other targeted fault injection method, is known [10] to be sufficient to gain root privileges.

B. Threat Models

Considering these types of attack vectors, the distinction between on- and off-chip attacks is often used to denote the security boundary in threat models. In particular, the following three model types are used in the context of physical attacks and countermeasures for general purpose processors.

a) Fully secure chip, malicious off-chip environment: This model is mostly used in system security papers that deal with physical attacks. Here, the actual processor is considered fully functional and not subject to faults or tampering. The off-chip components and busses, on the other hand, including storage memory (e.g., HDDs) and RAM, are assumed to be under full control of the attacker. In other words, model *(a)* completely disregards physical attacks on the chip and entirely focuses on memory-related attacks with perfect control. Interestingly, the resulting capability to perform arbitrary memory reads and writes is also pretty similar to software attacks that exploit memory vulnerabilities. Prominent examples for countermeasures that are typically deployed to thwart attacks in threat model *(a)* are memory encryption schemes as found in Intel SGX [12] and AMD SME [14]. Similarly, disk encryption schemes typically operate under this threat model.

b) Restricted on-chip attacks only, no off-chip memory: Threat model *(b)* can be considered to be the counterpart to threat model *(a)* given that it omits physical attacks on external memory and mostly focuses on side channel and fault attacks on the chip instead. The motivation for such a model is that, in many embedded applications, self-contained System-on-Chips (SoCs) without any external memory are deployed. These SoCs range from simple microcontrollers up to powerful application processors with Linux support. Subsequently, perfectly controlled tampering is considered hard and only more restricted attack models (e.g., random faults due to glitches, low number of controlled bit flips due to laser, limited number of probed wires, ...) are assumed. Software techniques that target reliability [22], as well as software countermeasures against special types of fault attacks [5], typically, operate in threat model *(b)*.

c) Restricted on-chip attacks, malicious off-chip environment: Finally, the strongest model, threat model *(c)*, combines both previously discussed attack capabilities. Here, restricted on-chip fault attacks, as well as arbitrary off-chip tampering, are permitted. By themselves, only very few techniques protect in this threat model. Instead, often a combination of countermeasures, which counteract attacks in model *(a)* or *(b)*, are used in practice. Note, however, that simply combining countermeasures does not necessarily provide the desired effect of securing an embedded system, especially when also software attacks have to be considered.

III. Fault Protection

Protecting the execution of software in a hostile environment requires different protection mechanisms to be in place. In this section, we briefly introduce our work which is tailored to counteract fault attacks and protect the correct execution of software. More detailed explanations and evaluation results can be found in the respective publications.

A. Protecting Instructions and the Control-Flow [34]

To be able to securely execute a program in the context of fault attacks, it is a necessity that the integrity of the instructions is maintained at all time, *i.e.*, from memory to execution. This does not only mean that each instruction has to be genuine (*i.e.*, no bit flips in the encodings), but also that their sequence has to be unaltered (*i.e.*, no skipped or repeated instructions). Every manipulation of the code should stop the execution of further instructions as soon as possible. Exactly this properties can be achieved with Sponge-based Control-Flow Protection (SCFP).

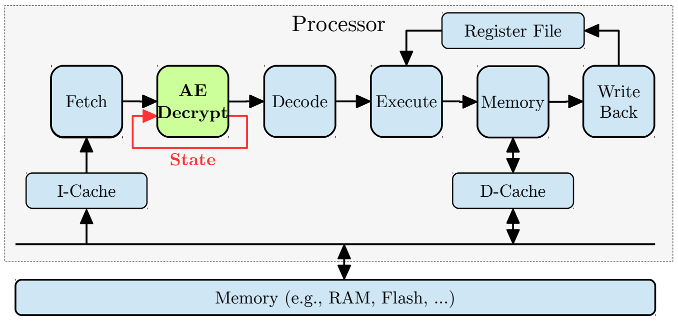

Figure 1. Extended processor pipeline with a stateful decryption stage.

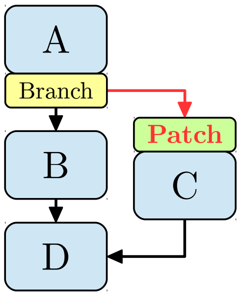

Figure 2. Example of patching the CFG of an if-then-else construct as part of the conditional branch instruction [34].

Approach: On a high level, the idea behind SCFP is to encrypt all code for a processor using a sponge-based authenticated-encryption primitive in a program-aware way. Namely, the sequence in which the instructions of a program are encrypted is determined by the sequence in which they eventually are executed. When such a program is executed on an SCFP-enabled processor, this special encryption approach permits to delay the actual decryption to the latest possible point. As shown in Figure 1, decryption is ideally performed with instruction-granularity as part of the processor pipeline just-in-time for execution. As the result, the majority of the system components, including the full memory hierarchy, the processor caches, the chip internal busses, and even the fetch unit that loads the instruction stream into the processor core only operates on encrypted instructions.

In case a wrong instruction is fed into the decryption unit, either due to bit flips in the encoding or because of software-based control-flow tampering, the respective instruction is decrypted incorrectly and yields a pseudo-random result. Additionally, due to the stateful cipher, also all subsequently decrypted instructions result in bogus values. In this situation, without knowledge of the cipher state and key, controlling the processor is next to impossible which effectively thwarts any further attacks. Still, eventually recovering from this pseudo-random execution state is also supported given that SCFP permits to perform context switches via interrupts. Therefore, as soon as the decoder detects an invalid instruction, the invalid instruction handler of the operating system can be used to handle the problem as desired.

Interestingly, a cryptographically strong Authentic Encrypted Execution (AEE) instantiation of the SCFP approach alone is already sufficient to protect instructions and their execution sequence against faults in the strongest threat model *(c)*. However, even weaker instantiations (e.g., AEE-Light), that are more likely to be used in practice given that they are cheaper to implement and faster, still achieve protection in threat model *(b)* and hinder software attacks considerably.

Details: Besides the addition of the sponge-based authenticated-decryption module, implementing SCFP also requires software changes. Due to the stateful cipher, without software support, reuse of code in loops and via function calls would be impossible. We fix this problem in SCFP by injecting *patch values* into the state of the cryptographic primitive at specific points during the execution of a program. Each patch value whitelists exactly one specific control-flow transfer in the program's CFG by deliberately introducing a cipher state collision at the merge point in the graph. Subsequently, these patch values should be considered as part of the program code. Similar to regular instructions, they are placed by the compiler and their final value is computed during encryption of the binary.

Figure 2 illustrates the concept using a simple if-then-else construct. Two valid execution paths exist in the shown CFG, namely, both the Basic Block (BB) sequence A-B-D and the sequence A-C-D are valid in this example. To enable the potential execution of both paths at runtime, at the beginning of block D, exactly the same decryption unit state has to be present independent if BB B or BB C has been executed before block D. To ensure this invariant, a patch value has to be applied as part of the conditional branch instruction that jumps to basic block C. In other words, in this example, the patch value compensates the difference in cipher state between executing BB B and BB C.

Results: As a proof-of-concept, we implemented a custom software toolchain that fully automatically inserts the patches and encrypts the binary. Furthermore, we implemented SCFP, in an AEE-Light configuration, into a RISC-V processor based on the open-source RI5CY core [32]. The resulting processor supports the RV32IM ISA and most parts of the privilege architecture version 1.9. Our results demonstrate the practicality of SCFP with a size overhead of 19.8 % and performance overhead of 9.1 % on average.

B. Protection of Conditional Branches [24]

Ensuring solely that the execution of a program adheres to its CFG using CFI protection schemes is, although a good start, still insufficient in the context of fault attacks. In particular, these schemes do not ensure that control-flow transfers within the graph are consistent with the data that is processed. For instance, the decision on which successor of a conditional branch gets executed is still unprotected and can be faulted with a single bit flip. However, many security critical operations, e.g., the verification of a computed cryptographic signature, rely on the correct execution of conditional branches. Subsequently, these operation require protection to withstand physical attacks.

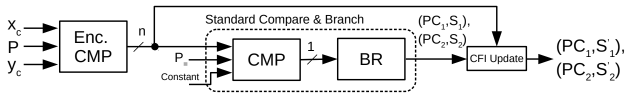
Standard Compare & Branch
x_c
P
y_c
Enc. CMP
n
$P_=$
Constant
CMP
1
BR
(PC_1,S_1), (PC_2,S_2)
CFI Update
(PC_1,S'_1), (PC_2,S'_2)

Figure 3. Protected conditional branch with state update and n-bit redundantly encoded comparison.

Independent of the actual implementation in an ISA, a generic conditional branch is an operation that consists of two micro-ops. First, a comparison operation is performed which compares its input operands, under a comparison predicate, to generate a 1-bit condition result. Second, the actual conditional branch is performed by updating the program counter based on the 1-bit condition result.

We identified three attack vectors, which all can be used to manipulate a conditional branch: (1) Modifying the input data of the comparison. (2) Attacking the comparison operation or the 1-bit result. (3) Attacking the direction of the branch operation.

Approach: To design an inherently secure conditional branch, all three attack vectors need to be mitigated. To prevent (1), we use AN-codes [22] and protect the data used for a conditional branch. While this encoding scheme protects the data during the storage in memory and registers, AN-codes also support protected arithmetic operations without the need of decoding and exposing data to an attacker. In particular, this encoding scheme natively supports the addition and subtraction operation without any changes. This property is exploited to prevent attack vector (2), namely the comparison. Instead of using an ordinary comparison operation of the processor's instruction set, which only yields a 1-bit condition result, we develop new comparison algorithms for all comparison predicates based on the arithmetic properties of AN-codes. These comparison algorithms instead return a syndrome containing only two values with a sufficiently large Hamming distance such that fault attacks are detectable. The last missing piece is the prevention of attack vector (3), namely, the protection of the branch operation itself. After performing the computation of the conditional branch, the comparison result already determines the branch target ahead of executing the conditional branch and therefore is a branch-target dependent value. At the branch target, the condition result can be used to explicitly alter the CFI state and to bind the computation of the comparison with the correct execution of the conditional branch.

Figure 3 summarizes the overall concept. The image shows how the encoded comparison operation interacts with the actual conditional branch operation. Furthermore, the link between comparison and CFI state update is shown which ensures that errors during the branch operation can be detected via the CFI scheme.

Details: Ordinary comparisons use an initial subtraction and then evaluate the sign of the difference to retrieve the comparison result. This is even applies when the input data is already encoded with, e.g., an AN-code. Since such an operation removes all redundancy from the data encoding, we cannot use that for the comparison. However, we still use an initial subtraction. Since AN-codes are closed under subtraction, the difference is again a valid AN-code, either positive or negative. By casting this difference to an unsigned value, we intentionally destroy the code property for negative numbers due to the two's complement representation (we assume 32-bit numbers in this work). This cast is followed by a modulo operation using the encoding constant A of the AN-code, which maps the casted difference either to 0 or $2^{32}\%A$. This mapping is a syndrome with a sufficiently large Hamming distance. To avoid a zero condition value, which is possibly easier to fault, we add a constant C at the beginning of the comparison algorithm. A *less-than* comparison following this approach is described in Algorithm 1. The same principle applies to all other comparison predicates by either swapping the operands of the first subtraction, swapping the true and false constant of the comparison output, or by assembling the result from two encoded comparison operations.

Algorithm 1: AN-encoded < comparison.

```
Data: $x_c, y_c \in$ AN-code, $0 < C < A$.
Result: $cond \in \{C_1, C_2\}$.
begin
    diff $\longleftarrow$ (unsigned) $x_c - y_c + C$
    cond $\longleftarrow$ diff % A
end
```

Results: We implemented this protection scheme for conditional branches into the same RISC-V core that already supports SCFP by adding a new instruction named `bpdeq`. This instruction implements a conditional branch that tests for equality and uses the first operand for a CFI state update at the branch target. In other words, `bpdeq` provides the whole functionality in Figure 3 except for the encoded comparison itself. To further ease the application of this countermeasure, we extended the compiler to automatically identify conditional branches in annotated functions. The operands of the conditional branch, as well as all dependent operations, get automatically encoded to the protected AN-code domain. Furthermore, the branch conditions are computed with our encoded comparison algorithms and `bpdeq` instructions get inserted for the actual conditional branches. The evaluation shows, the overhead of the countermeasure is on par with state-of-the-art duplication approaches, where the conditional branch is replicated many times. However, by using AN-encoded data, our protection scheme not only protects the conditional branch alone, but also adds data protection.

C. Pointer Protection and Protected Memory Accesses [25]

Memory accesses are beside computation the most frequently used operations in today's programs. However, these operations are typically not protected although they often deal with critical data. An attacker capable of inducing faults can redirect a memory access and then gets access to possibly sensitive data. Two attack vectors need to be considered when designing such a protection mechanism. First, modifying the pointer allows an

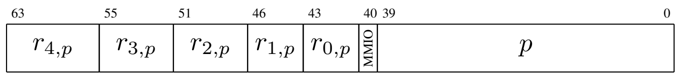

Figure 4. Encoded pointer representation. The actual 40-bit pointer value p, the MMIO tag bit, and 23 bits of redundancy r_p comprise an encoded 64-bit pointer.

attacker to redirect the memory access. Second, an attacker can modify the memory access directly, *i.e.*, by faulting the bus access. Current countermeasures, which try to protect against these attacks are too expensive [22] and are therefore not used. We overcome this problem and develop a new light-weight and secure memory architecture based on RISC-V to protect all memory accesses of a system.

Approach: In order to trust a pointer, the pointer requires redundancy. For this purpose, there are two main requirements. First, it must be easy to retrieve the original pointer value to support fast access to the actual memory location. Second, it is favorable to have an encoding scheme supporting arithmetic operations in order to extend the scope of protection to also include pointer arithmetic. When using such an encoding scheme, faults on a pointer up to a certain number of bit flips are detectable. Furthermore, also its pointer arithmetic is protected.

While the encoding scheme protects the pointer value, the actual memory access is still unprotected. To protect the access, the goal is to bind the redundant address information to the data when storing a register value to the memory. Instead of directly writing a register value D_{Reg} into memory at a specific address p (*i.e.*, $\texttt{mem}\left[p\right] = D_{Reg}$), a little more work has to be performed in our scheme. Namely, as shown in Equation 1, the linking function l_p has to be evaluated in order to determine the value that is actually written to the memory at address p.

$$\texttt{mem}\left[p\right] = l\left(p, D_{Reg}\right) = l_p\left(D_{Reg}\right) \qquad (1)$$

Details: To efficiently protect pointers, we use multi-residue codes [19] providing us with several advantages. The protection level of the code can be scaled by changing the number of residues. Furthermore, residue codes are separable codes, which give direct access to the plain pointer value, and support arithmetic directly in the encoded domain. To avoid any overhead regarding storage, we store the redundancy information of the code directly in the pointer by reducing the address space and make space for the residues. Concretely, we reduce the address to 40-bits, allowing us to store up to 24-bits of redundancy information. In Figure 4, the final pointer layout comprising the redundancy bits and one tag bit is shown.

The second part of the protection scheme is linking the redundant address information p with the data D_{Reg} in order to detect wrong bus accesses. This linking function l_p can be created in different forms, but at least requires two properties to be fulfilled. First, it needs to be a permutation and therefore being a bijective mapping having an inverse function l_p^{-1}. This inverse is used when performing the unlink operation during a memory read. The second requirement is to ensure that addressing faults are detectable. Here, data encoded under one address should yield a modified value when being decoded under a different address, *i.e.*, $\forall p, p', D_{Reg} : p \neq p' \rightarrow l_{p'}^{-1}\left(l_p\left(D_{Reg}\right)\right) \neq D_{Reg}$. Note, the modified value should yield an invalid code word in terms of the used data encoding scheme.

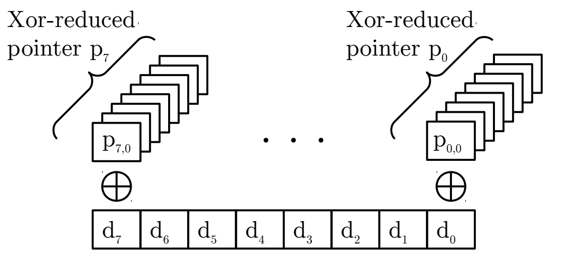

Figure 5. Byte-wise data linking of a 64-bit word. Each byte gets xored with its respective xor-reduced encoded address.

In our work, we explore a lightweight approach and show that a simple xor operation is enough and fulfills the required properties. Here, when storing data in memory, it is linked by xor-ing the address to the data, *i.e.*, $\texttt{mem}\left[p\right] = p_r \oplus D_{Reg}$. When reading data back, the inverse unlinking operation $D_{Reg} = p \oplus \texttt{mem}\left[p\right]$ is performed. While this approach applies to any data granularity, it has two disadvantages. First, by using the address directly to perform the linking operation, close bytes likely have the same linking pad. Second, real-life applications often use misaligned data accesses with arbitrary size, which often arises with memory functions like, e.g., *memcpy*.

To overcome these limitations, we compute a dedicated linking pad for every byte in a transfer. Each byte address is still a multi-residue encoded value to provide the necessary diffusion during the linking. However, since the data and its address have different bit sizes, an xor-based *compression* function is used to compute the final linking pad. In particular, each 64-bit address is reduced to an 8-bit value by xor-ing each sub-byte, *i.e.*, $p' = \bigoplus_{i=0}^{7} p_i$. In Figure 5, we show this linking approach for a 64-bit data word.

Results: The concept of multi-residue encoded pointers and linked memory accesses is integrated into a 64-bit RISC-V prototype implementation based on the open-source *RI5CY* core [32], which is part of the PULP project [31]. To efficiently deal with encoded pointers, we extended the instruction set and added new instructions to add, subtract, encode, and decode pointers. Furthermore, we added new memory operations to load and store data with the proposed linking approach.

In order to efficiently use this countermeasure without manually instrumenting all code on assembly level, we integrate it to the LLVM compiler project. The compiler identifies all pointers and transforms them and their respective pointer arithmetic to the encoded multi-residue domain. Finally, all memory accesses are replaced by their protected counterpart. On average, the overhead on our prototype platform regarding code size is about 10 % and the runtime overhead is less than 7 %.

IV. Discussion

In this paper, we presented parts of our ongoing work to counteract the emerging threat of physical attacks on general purpose processors. However, of course there is also other notable research in the context of physical attacks. ATRIUM [35], for example, that was also presented with a RISC-V prototype, is a runtime remote attestation mechanism that permits to detect errors in instructions. Another interesting approach that, similar to SCFP, provides confidentiality of code and CFI in the presence of faults is SOFIA [6], [7].

Also in the field of side channels attacks, or more generically passive physical attacks, a lot of interesting research is performed. For example, cryptographic accelerators [23] and memory encryption techniques [33] based on re-keying [20] are a promising approach to counteract side channels on an algorithmic level. On the logic level, modern masking approaches [9], which traditionally secure cryptographic implementations, can also protect the datapath of a RISC-V ALU [8].

In summary, physical attacks and countermeasures are getting more important. Free and open-source ISAs like RISC-V, support this development and provide academia and industry with powerful architectures that facilitate the needed research.

Acknowledgment

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402) and by the Austrian Research Promotion Agency (FFG) via the competence center Know-Center (grant number 844595), which is funded in the context of COMET - Competence Centers for Excellent Technologies by BMVIT, BMWFW, and Styria.

References

[1] "The xbox 360 reset glitch hack," https://free60project.github.io/wiki/Reset_Glitch_Hack.html, accessed: 2018-11-26.

[2] R. J. Anderson and M. G. Kuhn, "Low cost attacks on tamper resistant devices," in *Security Protocols Workshop – SPW*, 1997.

[3] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J. Seifert, "Fault attacks on RSA with CRT: concrete results and practical countermeasures," in *Cryptographic Hardware and Embedded Systems – CHES*, 2002.

[4] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The sorcerer's apprentice guide to fault attacks," *Proceedings of the IEEE*, vol. 94, 2006.

[5] T. Barry, D. Couroussé, and B. Robisson, "Compilation of a countermeasure against instruction-skip fault attacks," in *Workshop on Cryptography and Security in Computing Systems – CS2@HiPEAC*, 2016.

[6] R. de Clercq, J. Götzfried, D. Übler, P. Maene, and I. Verbauwhede, "SOFIA: software and control flow integrity architecture," *Computers & Security*, vol. 68, 2017.

[7] R. de Clercq, R. D. Keulenaer, B. Coppens, B. Yang, P. Maene, K. D. Bosschere, B. Preneel, B. D. Sutter, and I. Verbauwhede, "SOFIA: software and control flow integrity architecture," in *Design, Automation & Test in Europe – DATE*, 2016.

[8] H. Groß, M. Jelinek, S. Mangard, T. Unterluggauer, and M. Werner, "Concealing secrets in embedded processors designs," in *Smart Card Research and Advanced Applications – CARDIS*, 2016.

[9] H. Groß, S. Mangard, and T. Korak, "An efficient side-channel protected AES implementation with arbitrary protection order," in *Topics in Cryptology – CT-RSA*, 2017.

[10] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another flip in the wall of rowhammer defenses," in *IEEE Symposium on Security and Privacy – S&P*, 2018.

[11] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in javascript," in *Detection of Intrusions and Malware & Vulnerability Assessment – DIMVA*, 2016.

[12] S. Gueron, "A memory encryption engine suitable for general purpose processors," *IACR Cryptology ePrint Archive*, vol. 2016, 2016. [Online]. Available: http://eprint.iacr.org/2016/204

[13] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: Cold boot attacks on encryption keys," in *USENIX Security Symposium*, 2008.

[14] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption," http://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, accessed: 2017-04-17.

[15] D. Karaklajic, J. Schmidt, and I. Verbauwhede, "Hardware designer's guide to fault attacks," *IEEE Trans. VLSI Syst.*, vol. 21, 2013.

[16] Y. Kim, R. Daly, J. Kim, C. Fallin, J. 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 *International Symposium on Computer Architecture – ISCA*, 2014.

[17] T. Korak and M. Hoefler, "On the effects of clock and power supply tampering on two microcontroller platforms," in *Fault Diagnosis and Tolerance in Cryptography – FDTC*, 2014.

[18] M. Lipp, M. T. Aga, M. Schwarz, D. Gruss, C. Maurice, L. Raab, and L. Lamster, "Nethammer: Inducing rowhammer faults through network requests," *CoRR*, vol. abs/1805.04956, 2018. [Online]. Available: http://arxiv.org/abs/1805.04956

[19] J. L. Massey and O. N. García, "Error-correcting codes in computer arithmetic," in *Advances in Information Systems Science*, 1972.

[20] M. Medwed, F. Standaert, J. Großschädl, and F. Regazzoni, "Fresh re-keying: Security against side-channel and fault attacks for low-cost devices," in *Progress in Cryptology – AFRICACRYPT*, 2010.

[21] D. Samyde, S. P. Skorobogatov, R. J. Anderson, and J. Quisquater, "On a new way to read data from memory," in *Security in Storage Workshop – SISW*, 2002.

[22] U. Schiffel, A. Schmitt, M. Süßkraut, and C. Fetzer, "ANB- and anbdmem-encoding: Detecting hardware errors in software," in *Computer Safety, Reliability and Security – SAFECOMP*, 2010.

[23] R. Schilling, T. Unterluggauer, S. Mangard, F. K. Gürkaynak, M. Muehlberghuber, and L. Benini, "High speed ASIC implementations of leakage-resilient cryptography," in *Design, Automation & Test in Europe – DATE*, 2018.

[24] R. Schilling, M. Werner, and S. Mangard, "Securing conditional branches in the presence of fault attacks," in *Design, Automation & Test in Europe – DATE*, 2018.

[25] R. Schilling, M. Werner, P. Nasahl, and S. Mangard, "Pointing in the right direction - securing memory accesses in a faulty world," *CoRR*, vol. abs/1809.08811, 2018. [Online]. Available: http://arxiv.org/abs/1809.08811

[26] 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 – CARDIS*, 2015.

[27] S. Skorobogatov, "Low temperature data remanence in static ram," University of Cambridge, Tech. Rep., 2002. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-536.pdf

[28] ——, "The bumpy road towards iphone 5c NAND mirroring," *CoRR*, vol. abs/1609.04327, 2016. [Online]. Available: http://arxiv.org/abs/1609.04327

[29] S. P. Skorobogatov and R. J. Anderson, "Optical fault induction attacks," in *Cryptographic Hardware and Embedded Systems – CHES*, 2002.

[30] A. Tatar, R. K. Konoth, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi, "Throwhammer: Rowhammer attacks over the network and defenses," in *USENIX Annual Technical Conference – USENIX*, 2018.

[31] P. Team, "Pulp - open hardware, the way it should be!" https://www.pulp-platform.org/, 2018, accessed: 2018-05-15.

[32] ——, "Ri5cy: Risc-v core," https://github.com/pulp-platform/riscv, 2018, accessed: 2018-11-22.

[33] T. Unterluggauer, M. Werner, and S. Mangard, "Meas: memory encryption and authentication secure against side-channel attacks," *Journal of cryptographic engineering*, 2018.

[34] M. Werner, T. Unterluggauer, D. Schaffenrath, and S. Mangard, "Sponge-based control-flow protection for iot devices," in *European Symposium on Security and Privacy – EURO S&P*, 2018.

[35] S. Zeitouni, G. Dessouky, O. Arias, D. Sullivan, A. Ibrahim, Y. Jin, and A. Sadeghi, "ATRIUM: runtime attestation resilient under memory attacks," in *Conference on Computer-Aided Design – ICCAD*, 2017.