SecWalk: Protecting Page Table Walks Against Fault Attacks

Robert Schilling*, Pascal Nasahl*, Stefan Weiglhofer*, Stefan Mangard*†
*Graz University of Technology
firstname.lastname@iaik.tugraz.at
†Lamarr Security Research

***Abstract*—The correct execution of a memory load and store is essential for the flawless execution of a program. However, as soon as devices are deployed in hostile environments, fault attacks can manipulate memory operations and subsequently alter the execution of a program. While memory accesses for simple processors with direct memory access can efficiently be protected against fault attacks, larger processors with virtual addressing lack this protection. However, the number of systems with larger application-class processors is growing, leaving many applications unprotected. It requires new countermeasures to efficiently protect memory accesses of application-class processors with virtual memory against fault attacks.**

In this work, we present SecWalk, a design to efficiently protect all memory accesses of a program in the virtual and physical memory domain against fault attacks. We enhance residual-based pointer protection with a hardware-based secure page table walk inside the memory management unit. The page table walk securely translates a protected virtual address to a protected physical address by exploiting the redundancy properties of encoded addresses and a linking mechanism in the memory management unit. Furthermore, we extend the protection domain for virtual addresses to the TLB to also protect fast translations.

To evaluate the overhead of our design, we integrate SecWalk to an FPGA-based hardware implementation of an open-source RISC-V processor. The hardware evaluation shows that SecWalk extends the area of the design by 10 %. The software evaluation on a set of microbenchmarks shows an average code and runtime overhead of 11.05 %. To show the applicability on real-life applications, we port the microkernel seL4 to SecWalk, which yields a code overhead of 13.1 % and a runtime overhead of 11.6 %. This evaluation shows the overhead is small considering that SecWalk automatically protects all memory accesses of arbitrary applications against fault attacks.

***Index Terms*—fault attacks, countermeasure, page table walk, virtual memory, risc-v**

I. Introduction

With the rise of the Internet-of-Things (IoT), powerful industrial computers, and the revolution of the automotive industry, the complexity of embedded devices is continually growing. While in the early days of the IoT, small and simple microcontrollers contented the computing power requirements, the increasing complexity also raises the computing workload. Consequently, the trend of using larger application-class processors running commodity operating systems (OSs) is emerging. However, those devices are attacked using fault attacks to redirect memory accesses to a different location to bypass privilege checks, signature verifications, or password checks. The recent publication of software-induced fault attacks increases the attack surface even more, making many devices vulnerable to fault attacks.

There are different methodologies for inducing faults, such as glitching the power or clock signal [1], [29], inducing electromagnetic radiation [41], or by shooting with a laser to the chip surface [57]. While those attack methodologies require physical access, the development of software-induced fault attacks relaxes this constraint. The Rowhammer effect [25] allows an attacker to manipulate bits in memory by frequently accessing its neighboring memory cells. This fault methodology can even be triggered via Javascript [21] or remotely via the network interface [34], [61]. With the recent publication of more advanced software-induced fault attacks, such as Plundervolt [42], VoltJockey [47]–[49], or CLKscrew [60], fault attacks also become a threat for larger systems.

Inducing a fault to the system does not necessarily compromise it or bypasses any security mechanisms. While past exploitation techniques for faults mainly focused on attacking cryptographic algorithms [3], [7], [15], more recent exploits use fault attacks to bypass security mechanisms of general-purpose software [43], [44], [62], [63]. Typically, these attacks either modify the control-flow of a program by skipping instructions, altering the program counter, or attack the data resulting in powerful exploits such as bypassing secure boot [12], [14], [65] or bypassing DRM protection mechanisms [19], [59]. Fault attacks on the virtual memory system of an application-class processor are exploited to gain kernel privileges [69].

In order to thwart fault attacks, dedicated countermeasures have been developed. While there exist particular fault countermeasures for specific cryptographic schemes [9], [26], [35], more general protection schemes try to harden the execution of an arbitrary program against fault attacks. They do this mainly by enforcing the integrity of data and/or the control-flow [38], [39], [53]. To protect data during storage or transmission against faults, special encoding schemes [11], [37] add redundancy to the data. These schemes transform data into a different representation, *i.e.*, into the encoded domain, to detect up to a certain number of bitflips. Improved schemes [17], [37] even support arithmetic operations directly in the encoded domain, without the need for decoding. To enforce the correct execution of a program, control-flow integrity (CFI) schemes [13], [55], [67] ensure that all instructions are executed correctly and in the correct sequence.

Memory operations currently lack comprehensive protection against fault attacks. To efficiently protect all kinds memory accesses for application-class processors against fault attacks, it requires dedicated countermeasures. Unfortunately, current mitigation mechanisms for memory accesses, such as AN-B codes [54], employ a runtime overhead between 200 % and 13000 %, making their deployment very expensive and not practical. Furthermore, they do not support dynamic or shared memory, thus, limiting the practical application of this countermeasure. Although other approaches [56] reduce the performance overhead to a minimum, they can only protect direct memory requests for processors without a memory management unit (MMU). Therefore, they cannot protect applications running on larger systems with a traditional operating system using virtual memory management and shared memory for inter-process communication. New and efficient mechanisms are required to protect larger applications with arbitrary memory accesses from the virtual memory domain.

Contribution: In this work, we present SecWalk, an efficient countermeasure to protect virtual memory accesses against fault attacks. Our approach allows us to protect all memory accesses of a program against fault attacks, even for large application-class processors. This work closes an open gap and protects the page table walk against fault attacks by linking the redundancy properties of virtual addresses to physical addresses. We encode pointers and addresses in the virtual memory domain using a multi-residue code, which redundancy is stored in the upper bits of the pointer. The proposed secure page table walk uses the redundancy of a pointer to securely translate the virtual address to an encoded and protected physical address. We exploit the arithmetic properties of encoded pointers to retrieve the correct page table entries during the address translation. The translated encoded physical address is then used to perform the actual secure memory access. This mechanism allows us to support all common memory allocations, such as dynamic or shared memory.

To evaluate the design, we integrate SecWalk into a hardware implementation of an open-source RISC-V processor. The evaluation of our prototype implementation shows that the hardware overhead increases the size of the processor design by less than 0.5 % in terms of flip-flops and 10 % in terms of lookup tables. To evaluate the software overhead, we develop a custom LLVM based toolchain to automatically instrument programs with SecWalk. On a set of microbenchmarks, SecWalk yields an average code overhead of 11.05 % and an average runtime overhead of 7.17 %. To showcase the applicability to larger programs, we integrate SecWalk to the commodity microkernel seL4, and automatically protect all memory accesses of the kernel an user threads (virtual and phyiscal accesses) against fault attacks. Instrumenting all pointer arithmetic and every memory accesses increases the code size by 13.1 % and the runtime overhead by 11.6 %.

Summarized, our contributions are:

- We propose SecWalk, a generic method to protect the page table walk against fault attacks. Combined with encoded pointers and pointer arithmetic, we protect all memory accesses of a program against fault attacks.
- We integrate SecWalk to the open-source RISC-V processor CVA6 and evaluate its overhead based on an FPGA implementation.
- To automatically protect arbitrary programs with SecWalk without user interaction, we develop a custom LLVM-based toolchain.
- To evaluate the software overhead and to show the practicability, we evaluate SecWalk on a set of microbenchmarks we port the microkernel seL4. We automatically replace all pointers, addresses, and memory accesses with their protected counterparts using our toolchain.

Outline: The remainder of this paper is structured as follows. Section II provides background information on fault attacks, countermeasures, and discusses the page-based virtual memory. Section III-A discusses the threat model and existing fault attacks to virtual memory. Section IV presents SecWalk, an efficient mechanism to protect virtual memory accesses against fault attacks.

Section V describes the prototype implementation of SecWalk based on a RISC-V processor and discusses the toolchain, and in Section VI, we evaluate the performance of the implementation. Section VII discusses related work and shows how SecWalk is superior. Finally, Section VIII concludes this paper.

II. Background

This section first introduces fault attacks and then shows how existing redundancy mechanisms can protect against those attacks. Finally, we introduce the concept of page-based virtual memory, which SecWalk protects.

A. Fault Attacks

During a fault attack, the attacker influences the device's operating conditions with the goal of manipulating an inner state of the system. Such a fault can be induced in different ways, e.g., by manipulating the power supply [2], [6], [10], the clock signal [1], [46], the temperature [24], or by shooting with a laser or electromagnetic impulse onto the chip surface [5], [41], [57], [58]. While these fault methodologies require physical access to the device, more recent attacks have shown that this constraint can be relaxed. For example, new methodologies, such as the Rowhammer effect [25], can manipulate bits in memory solely in software by frequently accessing neighboring memory cells. This behavior can even be exploited remotely via Javascript [21] or over the network interface [34], [61]. More recently, software-induced fault attacks have also been used to induce faults directly to the CPU, e.g., with methods such as Plundervolt [42], VoltJockey [47]–[49], or CLKscrew [60].

B. Error Detection Codes

To counteract fault attacks and to protect data against unwanted manipulations, error detection codes (EDCs) [11], [45] are a long-established and well-studied research field. Their principle is to encode the data and add additional redundancy bits such that unwanted manipulations can be detected. While initially been developed to protect data during storage or transmission in harsh environments, new EDCs have been developed, which support the computation directly on encoded data. Such an encoding scheme has two advantages: First, it omits the necessity of decoding the data, which significantly improves the runtime performance when operating with encoded data. More importantly, it provides end-to-end protection of data throughout computation since the system never uses plain unencoded data. One example of such improved EDCs are binary linear codes [22], which natively support the computation of bitwise operations in the encoded domain. When dealing with arithmetic operations, so-called arithmetic codes are used, which natively support arithmetic operations on the encoded data.

1) AN-B Codes: One example of arithmetic codes are so-called AN codes [11], [18]. They are defined by multiplying a data value n with the encoding constant A, and thereby forming the codeword: $n_c = n \cdot A$. Note, the subscript c denotes the encoded value. Using this multiplication, only multiples of the encoding constant A are valid codewords; everything in between correlates to an invalid value. To verify the correctness of a codeword, a modulo operation with the encoding constant is performed, which is expected to return zero. To decode the codeword and to retrieve the original data, an integer division with the encoding constant A is performed. By multiplying the data with the encoding constant, the redundancy information is bound to the codeword and cannot be separated, thus the name *non-separable* encoding scheme. The redundancy properties of the AN code are defined by the encoding constant A. However, the proper selection of this constant is a challenging task and is currently only possible via exhaustive search [40]. Furthermore, also other parameters such as the maximum data size of the system influence the selection. Unfortunately, AN codes limit the value range for the payload data to be less than the encoding constant, to maintain proper error detection. However, this reduces the number of use cases for this encoding scheme for real-world applications. Furthermore, AN codes can only protect the arithmetic operations, but they do not protect the memory access of the data.

To extend the degree of protection, Forin and Schiffel et al. [18], [54] extend simple AN odes to AN-B codes. They add a unique signature B_n to every encoded data word n_c, yielding the encoding rule $n_c = A \cdot n + B_n$, where B_n is less than the encoding constant A. Since B_n is less than the encoding constant, an integer division can still be used to decode the data. However, when applying a modulo operation with the encoding constant to check for validity of the data, this must return B_n rather than zero. Since B_n is unique for every variable, it allows the code to detect wrong memory accesses. Schiffel et al. automated this process and developed a compiler toolchain to keep track of all assigned signatures and to insert the correct check operations. However, using this encoding scheme in practice is challenging. First, AN-B codes have a significant overhead of around 90 % on average on top of ordinary AN codes. Second, the signature B_n must be less than the encoding constant A, limiting the number of variables that can be protected. Furthermore, every location in memory requires a different signature, which is not practical.

2) Residue Codes: Residue codes [36] form a second class of arithmetic codes. In this encoding scheme, a codeword x_c is defined by concatenating the data with its residue $x_c = (x, r_x = M|x)$. Here, x denotes the payload data and r_x the redundancy part, the residue. The residue is computed as the remainder with respect to a modulus M, which defines the redundancy properties. Due to this concatenation, the tuple of data and residue are separable and therefore also called *separable* encoding scheme. This property allows the system to access the payload data without expensive decoding operations easily. Although the modulus M defines the robustness of this code, a simple bitflip on the data and on the modulus can create a new valid codeword. Thus, the Hamming distance between two simple residue encoded codewords is only 2.

To improve the robustness of residue codes, and to yield a higher Hamming distance, the number of residues can be increased, forming a multi-residue code [51], [52]. The modulus M is now defined by $M = \{m_0, \ldots, m_n\}$, where m_i is the actual modulus for one residue and n is the number of residues. Similar to AN codes, finding a good set of moduli is a challenging task and is currently only possible via exhaustive search but in a more efficient way [40]. Since (multi)-residue codes are arithmetic codes, they also natively support certain arithmetic operations. Here, the arithmetic operations are performed both on the payload data and on the residue. Equation (1) shows the arithmetic addition operation performed on multi-residue encoded data. First, the addition is performed on the plain payload data. Second, the addition is performed on every residue independently, followed by a modular reduction with the corresponding moduli m_i.

$$z_c = x_c + y_c = (x + y, \forall i : m_i | (r_{i,x} + r_{i,y})) \quad (1)$$

Like the addition operation, (multi)-residue codes also have native support for subtraction and multiplication operations.

C. Protection of Memory Accesses

Memory accesses are one of the most frequently used operations in a program after computation; thus, they require dedicated protection mechanisms against fault attacks. One protection mechanism tailored for this purpose are AN-B codes, as discussed before. However, its large runtime overheads between a factor of 2 and 130 make this scheme impractical for broader deployment.

A different approach to protect memory accesses is presented in [56]. They encode all addresses and pointers in a program using a multi-residue code and store the redundancy information in the upper bits of the pointer. The processor is extended to support pointer arithmetic directly on the encoded pointer with an extended instruction set. The memory access itself uses the encoded pointer and performs an xor-based link and unlink operation during the memory access to protect against wrong accesses. While this approach has reasonable overhead and can easily be applied to larger codebases via a custom toolchain, it only supports direct memory accesses without an MMU. This restriction limits the application of the countermeasure only to small processors without virtual memory management.

D. Page-based Virtual Memory

Page-based virtual memory [31] is a well-known and widely used architecture to decouple the physical memory layout from the application and OS. A memory management unit (MMU) decouples the virtual address space from the constraint physical address space. The memory of a program is fragmented into smaller, fixed-size pages. The operating system creates a mapping between pages in the virtual address space and the pages in the physical address space. These mappings, *i.e.*, the page table entries (PTEs), are stored in the page tables located in the page directory in the main memory. When running the program, the MMU uses the page tables to translate a virtual address to a physical address, called the *page table walk*. The physical address is eventually used for the actual memory access. As this translation is expensive, modern processors have a small cache in the MMU for storing the most recent translations, *i.e.*, the translation look-aside buffer (TLB), to have faster access to the physical address.

III. Threat Model and Attack Scenario

This section first presents the threat model and then shows how existing attacks in this threat model hijack virtual memory accesses. Finally, we discuss the required properties for protected memory accesses in the virtual memory domain.

A. Threat Model

In this work, we consider a powerful attacker capable of inducing faults with the goal of redirecting a virtual memory access. We consider attacks on the memory access independently of the used methodology, *i.e.*, we cover physical or software-induced fault attacks. The attacker aims to hijack the memory access by attacking the register file where a pointer is stored, pointer arithmetic, the memory access itself, or by manipulating the translation between the virtual and physical address. This includes faults to the MMU, the TLB, or to the page table entries stored in memory. Furthermore, we assume that the payload data of the application in memory is protected with a data encoding scheme.

Note, fault attacks on other parts of the processor, e.g., the instruction pipeline, the instruction pointer, or on the actual computation on other data, are not in the scope of this work. It requires orthogonal countermeasures, e.g., hardware-enforced control-flow integrity tailored to fault attacks [13], [55], [67], which ensures the authentic and genuine execution of the instruction stream and its control-flow graph. The computation can either be protected with instruction replication or by using a data encoding scheme that supports encoded arithmetic operations. For a complete protection against fault attacks, a combination of the protection of memory accesses such as SecWalk, the control-flow, and the computation is required. We now show how faults are used to hijack memory accesses in the virtual memory domain.

B. Faults on Virtual Memory

When dealing with larger application-class processors with virtual memory and MMUs, no efficient protection mechanism exists, leaving virtual memory accesses vulnerable to fault attacks. Especially, the page table walk, which translates a virtual to a physical address, is prone to fault attacks, which eventually leads to wrong memory accesses. Fig. 1 illustrates the unprotected page table walk leading to a wrong address translation due to a fault. The virtual address VA is translated to a physical address PA during the page table walk. A precise fault in this page table translation can redirect the page table walk to return a different physical address PA$_F$, thus redirecting the subsequent memory access to a different location. This attack vector exists even if virtual or physical addresses include redundancy mechanisms such as data encoding. There is no efficient way of protecting the page table walk, and thus, memory accesses from the virtual domain against fault attacks. Similar to that, also the MMU internal optimization buffer, *i.e.*, the translation look-aside buffer (TLB), suffers from the same attack vector. A fault can redirect the TLB to return a different and wrong page table entry, and thus, redirecting a memory access to a wrong location.

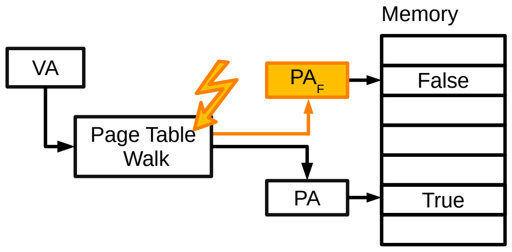

Fig. 1: Attack vector: A faulted page table translation leads to a wrong memory access.

Such an attack is presented in [64], where they use electromagnetic fault injection to induce faults to the MMU of a System-on-Chip. In their experiments, they are able to fault the virtual to physical mapping, therefore, redirecting the memory access to a different location.

[69] describes a kernel privilege escalation, where the Rowhammer effect is used to manipulate the PTE stored in memory. By inducing faults to the PTE, the attacker is able to redirect the virtual to physical mapping of an attacker-controlled page. Eventually, this results in having read and write access to the attacker process's own page tables, yielding access to all physical memory allowing the attacker to escalate privileges.

C. Requirements for Protected Virtual Memory Accesses

To protect memory accesses in the virtual memory domain against fault attacks with an easy application and to mitigate attacks as discussed above, a protection scheme needs to fulfill the following requirements.

1) Pointers and addresses require an efficient protection mechanism against fault attacks, which also covers pointer arithmetic.
2) A link between the accessed data and the protected memory address is required to ensure the correct memory element was accessed.
3) In order to protect the virtual memory domain, the translation between virtual and physical addresses, including the TLB, must propagate the address redundancy.
4) To support arbitrary applications, the protection mechanism of virtual memory must support shared memory. Therefore, any linking between payload data and addresses must only operate on physical addresses.

5) To support legacy codebases and to enable the easy deployment, the memory protection must be applied automatically, *i.e.*, during compilation, and must not require source code modifications.

Previous protection mechanisms for memory accesses are either not efficient or do not support the protection of virtual and shared memory [54], [56]. Hence, there is a need for new and efficient protection schemes, protecting *all* memory accesses against fault attacks.

IV. Design

This section presents SecWalk, an efficient protection scheme against fault attacks for all memory accesses in the virtual and physical memory domain, fulfilling the key requirements discussed above. We first introduce the design of protected pointers and then discuss the protected page table walk and TLB protection needed for virtual and shared memory.

A. Protected Pointers and Memory Accesses

Residual codes are an efficient method to protect arithmetic operations against fault attacks. These codes can also be used to protect pointers and their respective pointer arithmetic. Similar to [56], we embed the redundancy of the residue code in the upper bits of the pointer by reducing its address space. Our design uses the moduli set $M = \{5, 7, 17, 31, 127\}$ to protect pointers and addresses, which yields a Hamming distance of $D = 5$, capable of detecting up to four bitflips. Fig. 2 shows a virtual memory address, where the upper bits denote the residue redundancy and the lower 39-bits the original pointer value. This separation – a residue-code is a separable code – supports the direct access to the payload data without a dedicated decode operation, which is crucial for a fast memory lookup on the unencoded address space. The address space of the pointer is reduced to 39-bits allowing the pointer to store up to 25-bits for redundancy purpose. The smaller address space aligns with existing systems such as Linux [20] for RISC-V, which only uses 39-bits in its virtual address space. To efficiently operate on encoded pointers, we add new instructions to encode, decode, add, and subtract encoded pointers.

To protect the actual memory access, we establish a link between the encoded address and the actual data in the memory access. The linking operation scrambles the actual data when being written to memory and unscrambles it when reading it back using its encoded address information. We use a simple xor-based link on byte granularity, where each encoded byte address scrambles the corresponding byte in the data. Only when reading from the correct memory location, the unscramble operation succeeds, and the correct data is loaded into the register of the processor. As the payload data uses a data encoding scheme, the unlink operation of a wrong memory access destroys the payload's redundancy properties. Thus, the wrong access is detectable in software.

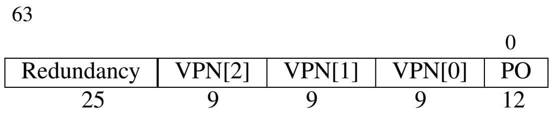

Fig. 2: Encoded virtual address in Sv39. The upper 25-bits denote the redundancy information of the multi-residue code.

B. Secure Page Table Walk

The page table walk is the main operation to translate a virtual address to a physical address, which is eventually used for the memory access. In a protected program, all addresses, virtual and physical ones, are protected using the residual-based encoding scheme as described before. We now present the secure page table walk that translates a protected virtual address to a protected physical address and establishes a protected link in between. The design focuses on the RISC-V Sv39 virtual memory system [66], but the protection mechanism itself is generic and can also be applied to other virtual memory architectures.

In Sv39, a 39-bit virtual address is grouped to a 27-bit virtual page number (VPN) and a 12-bit page offset (PO). During the three-step deep page table walk (the page table walk may abort early for larger pages), the VPN is translated to a 44-bit physical page number (PPN). The page offset remains untranslated. The final physical address is computed by concatenating the retrieved PPN with the page offset, forming a 64-bit address for the memory access. In Fig. 2, we show the layout of a virtual address. Note, the upper bits of the address are used to store the redundancy information of the multi-residue code of the virtual address.

To achieve a secure page table walk, we need to establish a link between the protected virtual address and the translated physical address. Only when performing the correct page table walk, this link can be verified, and the page table walk is genuine. The verification is done by checking the integrity of the encoded PPN in the page table entry after applying the respective unlink operation.

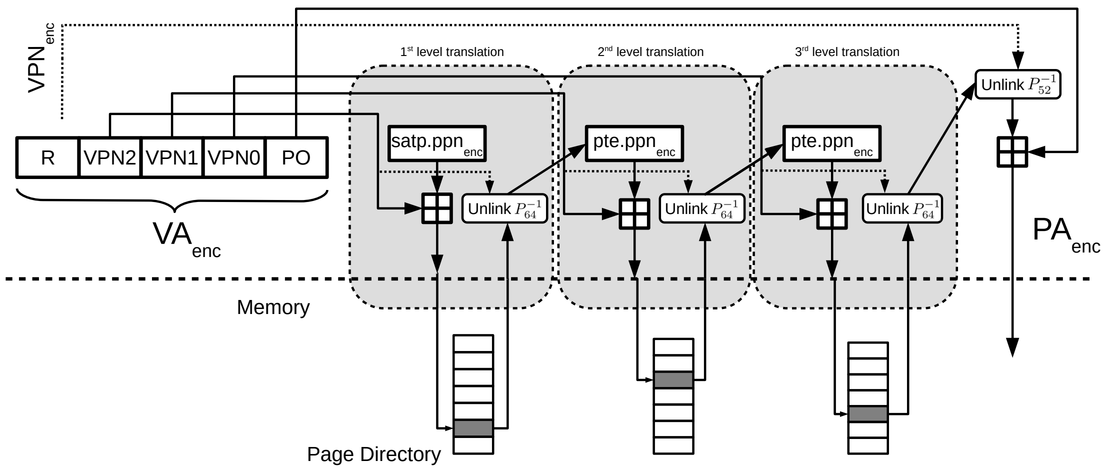

Fig. 3: Secure page table walk with linked page table entries.

Otherwise, the translation yields an invalid PPN in terms of the encoding scheme. Due to the redundancy properties of the encoding scheme, the invalid PPN can be detected. However, the link, which is based on the virtual address, must not influence the actual physical address nor the data stored in the memory. This property is needed to support shared memory, where different virtual addresses map to the same physical address and data.

To design a protected link between the virtual and physical address of the page table walk and to make faults detectable, we add redundancy to a page table entry. We encode the PPN within the PTE using the same multi-residue code as used for pointers. We extend the size of the PPN by 8-bits to a total size of 52-bit, to include the redundancy information of the multi-residue code. Together with the 12-bit page offset, this forms a 64-bit physical address. Since the PPN is aligned to the page size of 4 KiB or larger, the lower 12-bits of the physical address pointed by the PPN are always zero. Eventually, PPN $\times\ 2^{12}$ forms a valid codeword in terms of the multi-residue code, which can be verified. In Fig. 4, we show the modified PTE, including the redundancy of the encoded PPN that we use to verify the correct translation. By including 25-bits of redundancy in the physical page number, we also reduce the physical address space to 39-bits.

The page table walk subsequently reads new page table entries, based on the VPN of the virtual address, from memory, to determine the final physical address. In SecWalk, the PTEs are linked with the corresponding part of the VPN. Before using the PTE, it needs to be unlinked, followed by the verification of the residual

$\overbrace{\hphantom{\text{Redundancy PPN[2] PPN[1] PPN[0]}}}^{\text{PPN}_{enc}}$

63 0

Res	Redundancy	PPN[2]	PPN[1]	PPN[0]	RSW	Status Bits
2	25	9	9	9	2	8

Fig. 4: Sv39 page table entry with the extended encoded PPN to store the redundancy information.

integrity of the encoded PPN. If this check succeeds, the correct PTE was loaded from memory, and no wrong lookup was performed. If the check fails, it corresponds to an invalid memory read of the PTE or a manipulation of the PTE in memory. These steps, *i.e.*, the page table walk, are repeated until the final PTE is successfully loaded and the last encoded PPN is obtained. The last PPN itself is linked a second time with the fully encoded VPN, thus providing an end-to-end link between the encoded VPN and PPN. Finally, the physical address is computed by taking the encoded PPN and performing an encoded addition with the encoded page offset.

We achieve the link by applying a special linking function P_x to the PTEs during the page setup. During the page table walk, we apply the respective inverse unlink operation P_x^{-1}. Note that x denotes the bitwidth on which the linking is applied, which is 64-bit for linking the PTE. The final 52-bit encoded PPN in the last-level PTE is linked twice with a linking function where $x = 52$. This last step is needed to also incorporate the residual redundancy of VPN$_{enc}$ to the page table walk. As soon as the page table walk is faulted and a wrong PTE is loaded, the unlinking step destroys the data such that the redundancy verification fails, which eventually causes a trap in the processor. Here, a fault during the address translation is transformed to a data error on the PTE, which is detectable due to its redundancy properties.

In RISC-V, the page table walk starts with a base register storing the initial physical page number. Similar to PPNs within a PTE, we also encode the initial PPN to the multi-residue domain stored within the control and status (CSR) register `satp_enc`.ppn_{enc}. Note, we require a new CSR for this purpose to fit in the extended encoded PPN. The protected translation of the encoded virtual address VA$_{enc}$ to the encoded physical address PA$_{enc}$ works as follows. The suffix $_{enc}$ denotes multi-residue encoded data, $\boxplus$ the encoded addition, and $\boxminus$ an encoded subtraction. For Sv39, PAGE_SIZE is 2^{12}, and PTE_SIZE is 8.

1) Let a be `satp_enc`.ppn_{enc} $\times$ PAGE_SIZE and i = 2.
2) Let VPN$_{enc}$ = VA$_{enc}$ $\boxminus$ Enc(PO), where PO is the 12-bit page offset of the virtual address. Verify the lower 12-bit of VPN$_{enc}$ to be zero.
3) Let the linked PTE$_l$ be the value of the linked PTE at address a $\boxplus$ VA.$vpn[i]$ $\times$ PTE_SIZE.
4) Perform the unlink step: PTE = P_{64}^{-1}(PTE$_l$, VA.$vpn[i]$).
5) If PTE.$r = 1$ or PTE.$x = 1$, we have a leaf PTE. Go to step 7.
6) The PTE is a pointer to the next level of the page table. Check the integrity the residue integrity of PTE.ppn_{enc} $\times$ PAGE_SIZE. Fail if not valid. Let a be PTE.ppn_{enc} $\times$ PAGE_SIZE and $i = i - 1$. If $i < 0$, fail out. Continue at step 3.
7) A leaf PTE was found. Perform the second unlink operation of the PPN by PTE.ppn_{enc} = P_{52}^{-1}(PTE.ppn $\times$ PAGE_SIZE, VPN$_{enc}$) and check the residual integrity of PTE.ppn_{enc}. Fail if not valid.
8) The page table translation finished. The translated encoded physical address is given as PA$_{enc}$ = PTE.ppn_{enc} $\boxplus$ PO$_{enc}$.

Note, original physical memory access and physical memory protection (PMP) checks of RISC-V during the address translation are still in place. The page table walk returns an encoded physical address, which then is used for the linked memory access. In Fig. 3, we visualize the page table walk, using the steps as described before.

Linking Function: The general idea of the secure page table walk uses a linking function $P_x(y,k)$ to link the PTE with its corresponding parts of VPN. This link is performed on the whole PTE and in the last step only on the encoded PPN, thus requiring two different block sizes (52- and 64-bit). In the linking function x denotes the block size, y the data being linked, and k the linking key.

To make the link secure but also practical, the un/linking function needs to fulfill three requirements.

1) The linking function $P_x(y,k)$ needs to be a bijective mapping, implying that there exists an inverse unlinking function $P_x^{-1}(y,k)$ such that $y = P_x^{-1}\left(P_x\left(y,k\right),k\right)$. During the page directory setup, the PTE gets linked using its corresponding part of the VPN as the linking key k. When performing the page table walk, the respective unlink operation is applied to retrieve the correct PTE data again.
2) The unlinking function is used to detect wrong page table walks by verifying the redundancy of the unlinked PTE. Thus, the unlinking function must not yield a correct codeword in terms of the data encoding scheme if wrong data is accessed.
3) Third, the linking function needs to provide diffusion over the whole data word, e.g., over the 64-bit PTE when $x = 64$. The diffusion is needed to mix all bits of the PTE, *i.e.*, the status bits and the PPN. Thus, an arbitrary fault on the PTE, even only on a status bit, also affects the redundancy bits of the encoded PPN. Therefore, a simple byte-granular xor-based linking function, such as the one used in [56], is not sufficient as there is no intra-word diffusion.

Many functions fulfill these requirements, but we aim for a small and efficient design in this work. We use a two-round reduced version of the PRINCE block cipher [8] to perform a 64-bit link, meeting the requirements discussed above. A round-reduced version of PRINCE is sufficient as the linking function only requires a diffusion and no cryptographic strength. The second linking function uses a two-round reduced version of PRINCE as well, but with a reduced block size to 52-bit. The encryption operation of the cipher performs the linking operation, and the decryption operation performs the unlinking step, respectively.

C. TLB Design

The translation between virtual and physical addresses is a complex multi-step process, including multiple memory accesses under the hood. Modern processors have a dedicated cache for storing the most recent translations to speed up this operation, *i.e.*, the translation look-aside buffer (TLB). This buffer stores the most recent translations between virtual and physical addresses to avoid a costly MMU translation. The TLB is indexed using the VPN of the virtual address and returns the corresponding PTE if available. We apply the same 64-bit linking mechanism to secure this translation as used in the page table walk. The PTE in the TLB is linked using the encoded VPN_{enc} of the virtual address. When retrieving a PTE entry from the TLB, the PTE is unlinked, and the redundancy of the included PPN is verified. Only when using the correct encoded VPN_{enc} for unlinking, the redundancy properties of the encoded PPN are preserved and the lookup is valid. Otherwise, if the wrong or faulted VPN is used for unlinking, the redundancy properties of the encoded PPN are destroyed. In this case, the MMU traps and stops the application.

D. Page Directory Setup

When setting up virtual memory, it is necessary to create the corresponding mappings between virtual and physical addresses, *i.e.*, the page directory. This configuration is a manual task and is typically performed in software when the OS initializes a new process, or a process asks for more memory. As discussed in Section IV-B, the different levels of the page table are linked using parts of the VPN as the linking key with a final link of the whole encoded VPN at the end. It is the page directory setup's responsibility to create these links.

There are different approaches possible for establishing these links. While creating this link can purely be done in software, we aim for a hardware-centric approach since the linking functionality is needed anyways for the TLB. We add a new instruction `vpnlink1 rd, rs1, rs2`, which creates the 64-bit link between the layers of the page tables. Furthermore, we add a second instruction `vpnlink2 rd, rs1, rs2`, which creates the final 52-bit link for the last PPN.

E. Shared Memory Support

SecWalk natively supports shared memory. By not having a hard link between the virtual address and the data in memory, a process can map the same physical page to multiple virtual addresses. Similarly, multiple processes can map the same physical page in their address space to allow inter-process communication. Due to the design of the page table walk, shared memory does not require to share any information between multiple mappings, as it is required for other protection schemes in of related work.

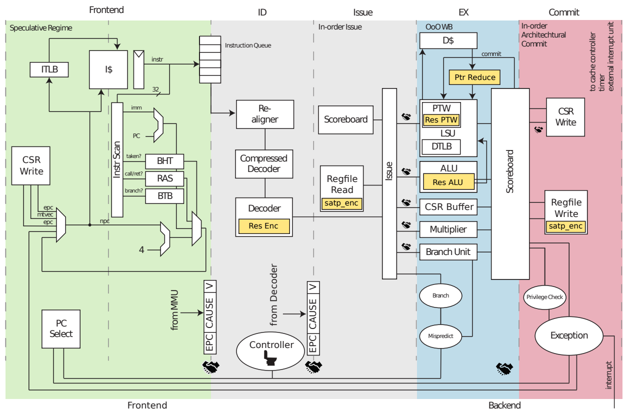

Fig. 5: CVA6 hardware architecture with SecWalk. The yellow parts indicate changes in the design.

V. Implementation

In this section, we first describe the hardware architecture of SecWalk and then discuss its custom toolchain to automatically instrument and protect arbitrary programs.

A. Hardware Implementation

We integrate SecWalk into the open-source RISC-V processor CVA6 [68], formerly known as Ariane. CVA6 is a 64-bit, application-class, 6-stage, single issue, in-order RISC-V CPU written in SystemVerilog capable of running operating systems. In Fig. 5, we show the modified hardware architecture of the processing system (the yellow parts indicate changes or additions). To support new instruction to deal with encoded pointers, e.g., add, subtract, encode, or decode, we extend the decoder and add a dedicated residue ALU. Furthermore, we add a CSR `satp_enc` to store the multi-residue encoded base address of the page directory needed for the page table walker. To support the linking operations needed for the page table setup, we add two new instructions `vpnlink1` and `vpnlink2`, to the decoder, which perform the 64-bit and 52-bit linking operation based on a round-reduced implementation of the PRINCE cipher.

In Fig 6, we show the modified load-and-store unit (LSU) of the system. The LSU adds a new XOR-unit to the load- and store-unit, which is responsible for performing the linked memory address using the compressed encoded address coming from the ptr-reduce module. Furthermore, the MMU adds the residue-based page table walker, which transforms the encoded virtual address to the encoded physical address used for memory access in the load- or store-unit. The MMU has dedicated access to the memory to retrieve the page table entries needed for the address translation.

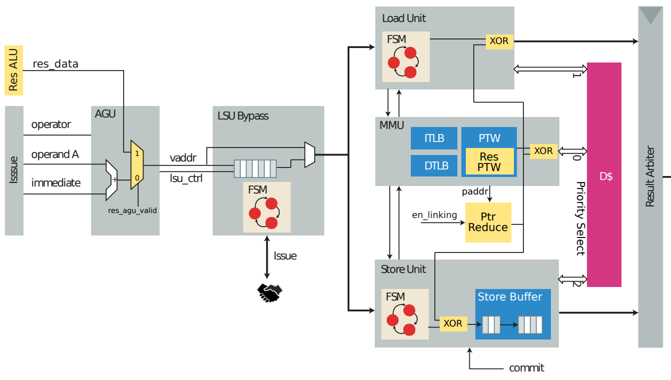

Fig. 6: Hardware architecture the load-store-unit of CVA6.

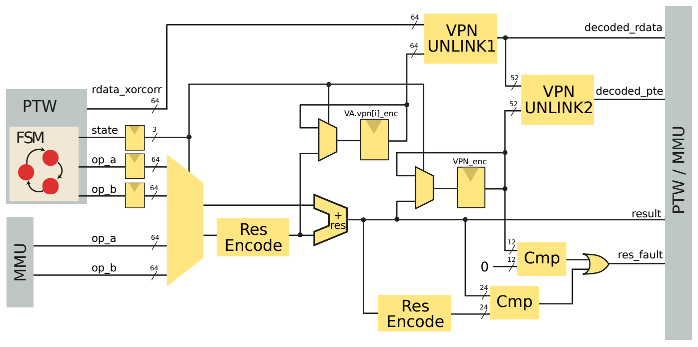

Fig. 7: Residue page table walker exploiting the redundancy properties of residue codes.

As shown in Fig. 6, we extend the MMU with a dedicated residue page table walker (ResPTW), detailed in Fig 7. The residue PTW performs the additional operations needed by the existing PTW and MMU to enable secure virtual memory accesses. The block diagram in Fig. 7 shows an overview of the implementation of the ResPTW, where it receives its data from the PTW or MMU and provides the results to the same two units. The original PTW, together with the residue PTW, performs the multi-level address translation according to the design of SecWalk. When an intermediate PTE is read, the 64-bit `vpnunlink1` operation decodes the whole PTE using the corresponding part of the VPN as the linking key. If a leaf PTE is read, the residue PTE performs the final 52-bit `vpnunlink2` operation to unlink the encoded page number using VPN$_{enc}$ as the linking key. Both `vpnunlink` operations are based on a round-reduced version of the PRINCE block cipher with different block sizes. The final address is computed by adding the encoded page offset to the final PPN from the leaf PTE. Note, a residue addition is performed rather than a simple concatenation to yield an encoded physical address, which can be used to access the memory.

If the TLB already contains the requested translation, the PTW is not needed, and the MMU only requests the computation of PA$_{enc}$. The `vpnunlink` operations of the residue PTW are used to decode the accessed entry from the TLB. Note, all residual operations, *i.e.*, an addition, contain an integrated check with respect to the redundancy bits. As soon as an invalid codeword is detected, the MMU traps, leading to aborting the program execution.

Although the prototype of SecWalk is based on the CVA6 processor, the protection mechanism is generic. Thus, SecWalk is compatible with other 64-bit RISC-V designs such as Rocket [**?**], (Sonic)Boom [**?**], [**?**], or and many others. The only hard requirement is being able to modify the core, i.e., having access to the source code. Furthermore, the protected page table walk itself is generic; thus, it is also applicable to other architectures. For example, the ARM AArch64 architecture supports a similar 39-bit addressing scheme, where SecWalk can be added if core changes are possible.

B. Toolchain Implementation

To automatically compile arbitrary software for SecWalk, we develop a custom toolchain based on the LLVM compiler [30]. We extend the RISC-V backend of the compiler to automatically encode all pointers to the multi-residue domain and to emit residue operations for pointer arithmetic. Furthermore, we replace all memory accesses with linked memory accesses, which use the residue encoded pointer for addressing. Note, the toolchain currently does not support the automatic instrumentation of inline assembly code. If a program uses inline assembly, it requires the developer to manually modify the assembly code to use protected pointers and memory accesses.

To run a protected program, it requires support from the operating system. When starting a new application, the operating system takes care to set up the memory mappings of the process. This part of the program requires a modification to take the linked page table entries into account. It needs to incorporate `vpnlink1` and `vpnlink2` to set up the link such that the hardware page table walker can unlink them when needed. This task is a manual process and is not covered by the LLVM-based toolchain.

VI. Evaluation

In this section, we first provide an evaluation showcasing the overheads of SecWalk in terms of hardware, code size, and runtime. We then discuss the security properties and how it protects against the defined threat model.

A. Hardware Evaluation

To measure the hardware overhead, we synthesize the design for a Xilinx Kintex-7 series FPGA. Our evaluation shows, the prototype implementation of SecWalk increases the area of the design by less than 0.5 % in terms of flip-flops and 10 % in terms of lookup tables. In Tab. I, we further split the utilization of the overheads between the handling of protected pointers and the changes related to virtual address translation in the MMU. Note, the hardware changes of SecWalk do not affect the critical path of CVA6, and the synthesis still reaches the original target frequency of 50 MHz.

B. Performance Evaluation

To evaluate the performance of SecWalk, we measure the code and runtime overhead of set of microbenchmarks and then extend the evaluation to a microkernel. We use the custom LLVM-based toolchain to automatically instrument the programs and to transform all pointer arithmetic and memory instructions to the protected domain. The startup code configures the MMU and maps the virtual and physical pages accordingly. In Fig. 8, we summarize the runtime and code overhead for the microbenchmark suite. SecWalk adds an average runtime overhead of 7.17 % and an average code size overhead of 11.05 %.

To showcase the applicability of our design for a larger application, we port the formally verified microkernel seL4 [16], [27], [28] to SecWalk, which is used in many security-critical applications. seL4 already supports RISC-V, but still requires minor adoptions for our design. First, we shift the operating system's address space to fit into the modified address layout of encoded pointers with its reduced address space. The instrumentation of the assembly code of seL4 requires manual modification, but these changes are minimal. The most crucial change in software is setting up the page tables using the custom linking instructions. These instructions

TABLE I: Hardware utilization of SecWalk.

Hardware Overhead	LUTs [%]	Flip-flops [%]
Protected Pointers	6.4	0.12
Residue PTW	3.6	0.32
Sum	10.0	0.44

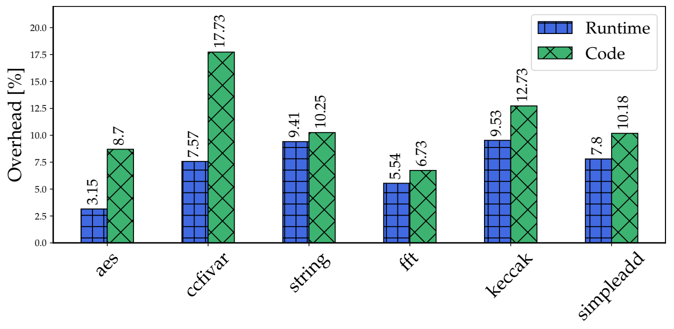

Fig. 8: Performance evaluation of SecWalk using microbenchmarks.

are used to create the link between the virtual and physical addresses in the page directory, which are unlinked during the page table walk. When compiling seL4 with the extensions of SecWalk, we see an increase of 13.1 % in code size, which is solely coming from using protected pointers and pointer arithmetic. Conceptually, the only actual software overhead for the protection of virtual memory is the setup of page tables using the new linking instructions, which is negligible. When running the protected seL4 kernel on the prototype, the runtime in terms of cycles increases by 11.6 %. Both overhead numbers are reasonable considering that all pointers, all pointer arithmetic, and every memory access of the system is protected against fault attacks.

C. Security Evaluation

The protection of addresses and pointers in the virtual memory domain using the multi-residue code with the described set of moduli yields codewords with a Hamming distance of $D = 5$ bits. Thus, this encoding scheme is able to detect up to four bitflips on pointers and addresses and its supported pointer arithmetic. Suppose the compiler detects an operation that is not supported by multi-residue codes, *i.e.*, a bitwise operation. In that case, it decodes the encoded pointer, performs the unsupported operation on the plain data, and then re-encodes the data back to the multi-residue domain. While the prototype implementation currently leaves the pointer unprotected for a short moment, other forms of redundancy, e.g., spatial redundancy, can be used to protect the pointer during such an operation. For example, instruction replication [4], [23] can be used to protect pointer arithmetic through unsupported operations by the multi-residue code. Note, such unsupported operations only occur very rarely, as pointer arithmetic tends to use simple operations such as additions and subtractions, which can operate in the protected domain.

The page table entries contain a multi-residue encoded PPN with a Hamming distance of $D = 5$ bits. The secure page table walk incorporates multiple operations in the encoded multi-residue domain. Throughout the translation of the virtual address, all residue additions of the page table walk are followed by a check operation in hardware, as depicted in Fig. 7. Thus, faults cannot accumulate over multiple operations on the multi-residue code. Summarized, the page table walk adds a protection against four random bitflips.

VII. Related Work

Starting with the ARMv8.3-A instruction set, ARM developed a feature named ARM pointer authentication [33], [50]. This feature adds new instructions allowing the software to sign and verify a pointer cryptographically. The truncated MAC is thereby stored in the upper bits of the pointer, reducing its address space. Before accessing the memory, the pointer is authenticated, and the MAC is removed from the pointer. Then, a memory load or store operation can access the memory using the authenticated pointer. While ARM pointer authentication has similar design decisions, its scope of protection is different. They protect special pointers at runtime, *i.e.*, the stack pointer, to protect against classical software attackers [32]. However, they cannot protect pointer arithmetic, nor can they protect the memory access itself.

There are related works in the context of protecting memory accesses against fault attacks. AN-B codes [54] assign each variable a dedicated signature B at compile-time. When reading the data back from the memory, this signature is verified using the underlying data encoding scheme of AN-B codes. If this signature cannot be verified, it means the memory access was redirected and read from a different location. Due to the static assignment of these signatures at compile-time, AN-B codes can only protect static memory and no dynamic allocations. Furthermore, they do not support shared memory, thus providing only a limited scope of protection for their expensive costs.

The work in [56] adds redundancy to the pointer to perform linked memory accesses. To compensate for the overheads of encoded pointer arithmetic, they extend the processor with new instructions and develop a compiler using them. While their overheads are reasonably low, their protection mechanism only supports bare-metal applications of small embedded use cases. There is no support for virtual and shared memory; thus, it cannot protect memory accesses of application-class processors against faults.

SecWalk is superior to other protection mechanisms for memory accesses. While it has a low performance penalty, SecWalk outperforms related work in terms of supported features. SecWalk supports the protection of virtual memory accesses against fault attacks, including dynamic allocations and shared memory between different processes. In Tab. II, we summarize the comparison of SecWalk against AN-B codes and purely encoded pointers.

VIII. Conclusion

The correct execution of a load or store operation is essential for the security of the system. With the rise of more powerful embedded systems, operating systems with virtual memory are commonly deployed in the IoT. When fault attacks are considered, virtual memory accesses cannot be trusted as there are different attacks possible, which redirect the memory to a different location. Currently, there is no economic mechanism available that protects virtual memory accesses against fault attacks, including dynamic and shared memory.

In this work, we closed this gap and presented SecWalk, an efficient design to protect all memory accesses of a program in the virtual and physical domain against fault attacks. SecWalk protects all pointers and addresses in the virtual address space using a multi-residue code with no additional storage overhead. Furthermore, this encoding scheme supports encoded operations, thus also protecting the pointer arithmetic. We extend the domain of protection, and develop a secure page table walk, that propgates the redundancy from the virtual address to the physical address used for the memory access. The core idea of SecWalk is to add redundancy to page table entries, add a linking mechanism between virtual and physical addresses, and then verify the redundancy properties on the page table walk. The protection is comprehensive, covering the virtual address domain, the address translation within the MMU and TLB, and the actual memory access using the translated physical address. Furthermore, SecWalk

TABLE II: Feature comparison of SecWalk comapared to related work.

Protection Scheme	Protection of Virtual Memory	Protection of Shared Memory	Overhead
ARM Pointer Authentication	✗	✗	Low
AN-B Codes	✓	✗	High
Encoded Pointer	✗	✗	Low
SecWalk	✓	✓	Low

supports arbitrary applications, including dynamic and shared memory.

We implemented SecWalk on an open-source RISC-V processor and mapped the design to an FPGA to showcase the hardware overhead. We developed a custom LLVM-based toolchain to automatically instrument arbitrary programs without user interaction. To evaluate the performance of SecWalk, we compile and execute a set of microbenchmarks. Furthermore, we integrate SecWalk to the existing microkernel seL4 to show its applicability on real-life applications using dynamic and shared memory. Our evaluation shows, the hardware, and software overheads of SecWalk are reasonable, considering that it protects all memory accesses of a program against fault attacks.

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).

References

[1] 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.

[2] A. Barenghi, G. Bertoni, E. Parrinello, and G. Pelosi, "Low Voltage Fault Attacks on the RSA Cryptosystem," in *Fault Diagnosis and Tolerance in Cryptography – FDTC*, 2009.

[3] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures," *Proceedings of the IEEE*, vol. 100, 2012.

[4] T. Barry, D. Couroussé, and B. Robisson, "Compilation of a Countermeasure Against Instruction-Skip Fault Attacks," in *Cryptography and Security in Computing Systems – CS2@HiPEAC*, 2016.

[5] E. Biham and A. Shamir, "Differential Fault Analysis of Secret Key Cryptosystems," in *Advances in Cryptology – CRYPTO*, 1997.

[6] J. Blömer and J. Seifert, "Fault Based Cryptanalysis of the Advanced Encryption Standard (AES)," in *Financial Cryptography – FC*, 2003.

[7] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract)," in *Advances in Cryptology – EUROCRYPT*, 1997.

[8] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, P. Rombouts, S. S. Thomsen, and T. Yalçin, "PRINCE - A Low-Latency Block Cipher for Pervasive Computing Applications - Extended Abstract," in *Advances in Cryptology – ASIACRYPT*, 2012.

[9] K. Bousselam, G. D. Natale, M. Flottes, and B. Rouzeyre, "On Countermeasures Against Fault Attacks on the Advanced Encryption Standard," in *Fault Analysis in Cryptography*, 2012.

[10] C. Bozzato, R. Focardi, and F. Palmarini, "Shaping the Glitch: Optimizing Voltage Fault Injection Attacks," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019.

[11] D. T. Brown, "Error Detecting and Correcting Binary Codes for Arithmetic Operations," *IRE Trans. Electron. Comput.*, vol. 9, 1960.

[12] A. Cui and R. Housley, "BADFET: Defeating Modern Secure Boot Using Second-Order Pulsed Electromagnetic Fault Injection," in *Workshop on Offensive Technologies – WOOT*, 2017.

[13] 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.

[14] J. V. den Herrewegen, D. F. Oswald, F. D. Garcia, and Q. Temeiza, "Fill your Boots: Enhanced Embedded Bootloader Exploits via Fault Injection and Binary Analysis," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021.

[15] C. Dobraunig, M. Eichlseder, H. Groß, S. Mangard, F. Mendel, and R. Primas, "Statistical Ineffective Fault Attacks on Masked AES with Fault Countermeasures," in *Advances in Cryptology – ASIACRYPT*, 2018.

[16] D. Elkaduwe, G. Klein, and K. Elphinstone, "Verified Protection Model of the seL4 Microkernel," in *Verified Software: Theories, Tools, Experimentsy – VSTTE*, 2008.

[17] C. Fetzer, U. Schiffel, and M. Süßkraut, "AN-Encoding Compiler: Building Safety-Critical Systems with Commodity Hardware," in *Computer Safety, Reliability and Security – SAFECOMP*, 2009.

[18] P. Forin, "Vital coded microprocessor principles and application for various transit systems," *IFAC Proceedings Volumes*, vol. 23, 1990.

[19] Free60.org, "Reset Glitch Hack," http://free60.org/wiki/Reset_Glitch_Hack, [accessed 2021-05-05].

[20] A. Ghiti, "Virtual memory layout on risc-v linux," https://www.kernel.org/doc/html/latest/riscv/vm-layout.html, 2021, [accessed 2021-02-26].

[21] 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.

[22] R. W. Hamming, "Error detecting and error correcting codes," *Bell Labs Technical Journal*, vol. 29, 1950.

[23] J. S. Hu, F. Li, V. Degalahal, M. T. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Compiler-Directed Instruction Duplication for Soft Error Detection," in *Design, Automation & Test in Europe – DATE*, 2005.

[24] M. Hutter and J. Schmidt, "The Temperature Side Channel and Heating Fault Attacks," in *Smart Card Research and Advanced Applications – CARDIS*, 2013.

[25] Y. Kim, R. Daly, J. S. 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.

[26] Á. Kiss, J. Krämer, P. Rauzy, and J. Seifert, "Algorithmic Countermeasures Against Fault Attacks and Power Analysis for RSA-CRT," in *Constructive Side-Channel Analysis and Secure Design – COSADE*, 2016.

[27] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: formal verification of an operating-system kernel," *Commun. ACM*, vol. 53, 2010.

[28] 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 *Workshop on System Software for Trusted Execution – SysTEX@SOSP*, 2009.

[29] 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.

[30] C. Lattner and V. S. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *International Symposium on Code Generation and Optimization – CGO*, 2004.

[31] S. H. Lavington, "The Manchester Mark I and Atlas: A Historical Perspective," *Commun. ACM*, vol. 21, 1978.

[32] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J. Ekberg, and N. Asokan, "PAC it up: Towards Pointer Integrity using ARM Pointer Authentication," in *USENIX Security Symposium*, 2019.

[33] A. Limited, "Arm architecture reference manual armv8, for armv8-a architecture profile," https://documentation-service.arm.com/static/5fa3bd1eb209f547eebd4141, 2020, [accessed 2020-09-22].

[34] M. Lipp, M. T. Aga, M. Schwarz, D. Gruss, C. Maurice, L. Raab, and L. Lamster, "Nethammer: Inducing Rowhammer Faults through Network Requests," *arXiv abs/1805.04956*, 2018.

[35] T. Malkin, F. Standaert, and M. Yung, "A Comparative Cost/Security Analysis of Fault Attack Countermeasures," in *Fault Diagnosis and Tolerance in Cryptography – FDTC*, 2006.

[36] J. L. Massey, "Survey of residue coding for arithmetic errors," *International Computation Center Bulletin*, vol. 3, 1964.

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

[38] M. Medwed and S. Mangard, "Arithmetic logic units with high error detection rates to counteract fault attacks," in *Design, Automation & Test in Europe – DATE*, 2011.

[39] M. Medwed and J. Schmidt, "A Generic Fault Countermeasure Providing Data and Program Flow Integrity," in *Fault Diagnosis and Tolerance in Cryptography – FDTC*, 2008.

[40] ——, "Coding Schemes for Arithmetic and Logic Operations - How Robust Are They?" in *Information Security Applications – WISA*, 2009.

[41] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz, "Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller," in *Fault Diagnosis and Tolerance in Cryptography – FDTC*, 2013.

[42] K. Murdock, D. Oswald, F. D. Garcia, J. V. Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based Fault Injection Attacks against Intel SGX," in *IEEE Symposium on Security and Privacy – S&P*, 2020.

[43] P. Nasahl and N. Timmers, "Attacking autosar using software and hardware attacks," in *escar USA*, 2019.

[44] C. O'Flynn, "BAM BAM!! On Reliability of EMFI for in-situ Automotive ECU Attacks," *ePrint 2020/937*, 2020.

[45] W. W. Peterson, *Error-correcting codes*. M.I.T. Press [u.a.], 1961.

[46] G. Piret and 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.

[47] P. Qiu, D. Wang, Y. Lyu, and G. Qu, "VoltJockey: Breaking SGX by Software-Controlled Voltage-Induced Hardware Faults," in *Asian Hardware Oriented Security and Trust Symposium – AsianHOST*, 2019.

[48] ——, "VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies," in *Conference on Computer and Communications Security – CCS*, 2019.

[49] ——, "VoltJockey: Abusing the Processor Voltage to Break Arm TrustZone," *GetMobile Mob. Comput. Commun.*, vol. 24, 2020.

[50] I. Qualcomm Technologies, "Pointer authentication on armv8.3," https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf, 2017, [accessed 2020-09-16].

[51] T. R. N. Rao, "Biresidue Error-Correcting Codes for Computer Arithmetic," *IEEE Trans. Computers*, vol. 19, 1970.

[52] T. R. N. Rao and O. N. Garcia, "Cyclic and multiresidue codes for arithmetic operations," *IEEE Trans. Inf. Theory*, vol. 17, 1971.

[53] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software Implemented Fault Tolerance," in *International Symposium on Code Generation and Optimization – CGO*, 2005.

[54] 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.

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

[56] R. Schilling, M. Werner, P. Nasahl, and S. Mangard, "Pointing in the Right Direction - Securing Memory Accesses in a Faulty World," in *Annual Computer Security Applications Conference – ACSAC*, 2018.

[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 – CARDIS*, 2015.

[58] S. P. Skorobogatov and R. J. Anderson, "Optical Fault Induction Attacks," in *Cryptographic Hardware and Embedded Systems – CHES*, 2002.

[59] M. Steil and F. Domke. The xbox 360 security system and its weaknesses. https://www.youtube.com/watch?v=uxjpmc8ZIxM. [accessed 2021-05-05].

[60] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management," in *USENIX Security Symposium*, 2017.

[61] 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*, 2018.

[62] N. Timmers and C. Mune, "Escalating Privileges in Linux Using Voltage Fault Injection," in *Fault Diagnosis and Tolerance in Cryptography – FDTC*, 2017.

[63] N. Timmers, A. Spruyt, and M. Witteman, "Controlling PC on ARM Using Fault Injection," in *Fault Diagnosis and Tolerance in Cryptography – FDTC*, 2016.

[64] T. Trouchkine, S. K. Bukasa, M. Escouteloup, R. Lashermes, and G. Bouffard, "Electromagnetic fault injection against a complex cpu, toward new micro-architectural fault models," *Journal of Cryptographic Engineering*, 2021.

[65] A. Vasselle, H. Thiebeauld, Q. Maouhoub, A. Morisset, and S. Ermeneux, "Laser-Induced Fault Injection on Smartphone Bypassing the Secure Boot-Extended Version," *IEEE Trans. Computers*, vol. 69, 2020.

[66] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanović, “The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.12-draft,” EECS Department, University of California, Berkeley, Tech. Rep., 2020.

[67] M. Werner, T. Unterluggauer, D. Schaffenrath, and S. Mangard, “Sponge-Based Control-Flow Protection for IoT Devices,” in *European Symposium on Security and Privacy – EuroS&P*, 2018.

[68] F. Zaruba and L. Benini, “The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-ready 1.7GHz 64bit RISC-V Core in 22nm FDSOI Technology,” *arXiv abs/1904.05442*, 2019.

[69] G. P. Zero, “Exploiting the dram rowhammer bug to gain kernel privileges,” https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html, 2015, [accessed 2020-10-22].