Protecting Indirect Branches against Fault Attacks using ARM Pointer Authentication

Pascal Nasahl
Graz University of Technology
pascal.nasahl@iaik.tugraz.at

Robert Schilling
Graz University of Technology
robert.schilling@iaik.tugraz.at

Stefan Mangard
Graz University of Technology
Lamarr Security Research
stefan.mangard@iaik.tugraz.at

***Abstract*—With the growing number of embedded devices deployed in safety- and privacy-sensitive applications, such as in the automotive area or in the IoT, the hardening of these systems against attacks is getting essential. As these devices are physically accessible by an adversary, fault attacks are frequently used to hijack the control-flow of the executed program and bypass security defenses such as secure boot, gain arbitrary code execution, or retrieve sensitive information. To protect the control-flow from this threat, control-flow integrity (CFI) aims to be an effective and generic countermeasure.**

Although CFI aims to mitigate fault induced control-flow hijacking attacks, state-of-the-art CFI schemes do not protect addresses, allowing an attacker to still hijack the control-flow of indirect branches. To counteract this threat and detect unwanted bit flips, data encoding schemes are frequently used to add redundancy to these addresses. However, software-based data encoding schemes yield large runtime overheads, making them hard to deploy on a larger scale. To reduce this performance overhead, related work typically introduces custom CPU changes, which are not feasible for off-the-shelf systems, leaving a broad range of devices unprotected. Hence, software-based address redundancy schemes for commodity devices are needed to thwart fault attacks on indirect branches.

In this paper, we utilize the ARM pointer authentication feature of recent ARM architectures to efficiently protect the target addresses of indirect calls. In addition to the address protection, we further enhance the state update function of existing CFI schemes to protect the link between indirect control-flow transfers. To demonstrate how these defense mechanisms improve the protection of state-of-the-art CFI countermeasures, we integrate our address encoding and linking strategy into a previously introduced CFI scheme. We further extend a LLVM-based toolchain to automatically thwart fault attacks on indirect branches without user interaction. Our analysis shows an negligible overhead of less than 2.34% on average for protecting target addresses of indirect branches and the link between indirect branches for SPEC2017.

***Index Terms*—fault attacks, control-flow integrity, indirect branches, addresses, ARM pointer authentication**

I. Introduction

Fault attacks pose a severe threat to the security of the growing number of embedded devices. This attack methodology is used to hijack the control-flow of the target device to bypass secure boot on embedded controllers [17], [22], [62], to escalate privileges on Linux [59], [60], or to extract firmware and to gain arbitrary code execution on electronic control units (ECUs) in automotive applications [40], [44], [45], [66].

Protecting the integrity of the control-flow is a well-established technique to thwart fault-based control-flow hijack attacks. Here, control-flow integrity (CFI) schemes restrict the control-flow of the program during the execution to a narrow subset of execution paths. Typically, the set of valid execution paths through the program is statically determined at compile time using the control-flow graph (CFG) extracted from the code. Any control-flow violation, *i.e.*, control-flow deviations from this predefined path, are detected by CFI and hinder the attacker from exploiting the injected fault.

However, state-of-the-art CFI schemes aiming to protect the program execution against fault attacks [2], [24], [29], [46], [50], [52], [63] do not explicitly protect code-pointers. A fault to such an address can influence the execution of indirect branches allowing an adversary to bypass CFI and threaten the security of the protected program. Although CFI limits the set of reachable control-flow targets, the statically extracted CFG is only an over-approximation of the actual executed control-flow. Hence, an attacker, especially for indirect branches, could redirect the intended control-flow within the bounds of the approximated control-flow graph.

Hardening a CFI scheme against attacks on unprotected addresses requires adding redundancy to these addresses using data encoding schemes. Such schemes transform addresses to a different representation, allowing the system to detect a certain number of bit flips induced by faults. While arithmetic codes, such as ANB-codes [20], [51] or residue codes [37], [38] allow simple arithmetic operations on the encoded data, software-based schemes induce large runtime overheads. To reduce these large performance overheads, related work [39], [54] suggests intrusive hardware changes directly in the CPU. However, as custom hardware changes are unrealistic for commercial off-the-shelf systems, a broad range of devices remain unprotected, requiring efficient, software-based countermeasures.

Contribution

In this paper, we are addressing the issue of insufficient address protection in CFI schemes aiming to thwart control-flow hijacks using fault attacks on commodity devices from ARM. To efficiently protect addresses from targeted faults, which allow attackers to redirect the control-flow, we introduce a software-based hardware-assisted address redundancy scheme capable of detecting such faults. More concretely, we

are utilizing the pointer authentication extension of recent ARM architectures to cryptographically sign code-pointers used by indirect branches. We employ this feature to encode addresses allowing us to detect bit flips injected by a fault rather than protecting against classical software attacks. To protect addresses at program execution but also when stored in the binary, we encode code-pointers at compile time and verify the integrity at runtime. We showcase how the verification of these addresses using dedicated indirect branch instructions detects bit flips injected by a fault.

Our analysis of state-based CFI schemes further reveals that indirect control-flow transfers are insufficiently protected, allowing an attacker to hijack the control-flow. We propose an enhanced state update mechanism creating a link for such control-flow transfers to mitigate this attack vector. To automatically protect indirect branches from targeted fault attacks, we integrate the address encoding scheme and the hardened state update function into the LLVM-based toolchain of FIPAC [52], a previously introduced CFI scheme for ARM devices. To evaluate the performance overhead, we compiled a subset of the SPEC2017 benchmark suite with our custom toolchain. Our performance measurement for protecting indirect branches against fault attacks shows a negligible overhead of less than 2.34% on average for protecting indirect branches against fault attacks.

In summary, our contributions are as follows:

- We utilize the ARM pointer authentication feature to protect all addresses used by indirect branches from fault attacks.
- We propose a new state update function for signature-based CFI schemes protecting the link between indirect control-flow transfers.
- We integrate our control-flow protection mechanisms into the LLVM-based toolchain of FIPAC.
- We verify the functional correctness and the performance overhead of our scheme using the SPEC2017 benchmark and discuss security guarantees.

Paper Outline

In Section II, we discuss fault attacks and describe common countermeasures, such as control-flow integrity and data redundancy. Section III highlights foundations of state-based CFI schemes and discusses weaknesses of these schemes. To hinder an attacker from exploiting these weaknesses, we formulate design requirements in Section IV. Finally, in Section V, we describe the integration and implementation of our proposed features into a recently introduced CFI scheme and evaluate security guarantees and the performance overhead in Section VI. Section VII summarizes related work and Section VIII concludes this paper.

II. Background

This section first introduces fault attacks and fault inducing techniques. Then, we discuss the concept of control-flow integrity and differentiate between schemes targeting to protect against a software or a fault attacker. Finally, we highlight data redundancy schemes and the ARM pointer authentication, which can be used to enforce control-flow integrity.

A. Fault Attacks

In a fault attack, the adversary tampers with the physical parameters of the device's environment to trigger a fault in the memory, the instruction pipeline, or in other components of the processor. Long-established methods to physically induce faults are voltage [11] and clock glitches [55], laser [61] and electromagnetic pulses [41], and even heat [25]. New fault injection methods, such as Plundervolt [43], CLKscrew [57], VoltJockey [47], [48], or the Rowhammer effect [27], induce faults purely in software, which significantly increases the potential attack surface of fault attacks.

While faults in the past were mainly used to break cryptographic schemes [8]–[10], [18], [19], recent work [17], [22], [40], [44], [45], [58]–[60], [62], [66], [67] demonstrates that fault attacks can lever out security assumptions of the device under attack and bypass secure boot [17], [22], [62] or escalate privileges on Linux [59], [60]. Such attacks typically aim to induce a fault within a basic block or target to hijack the control-flow transfer between two basic blocks. While faults within a basic block either corrupt or skip individual instructions [43], [57], control-flow hijacking attacks redirect the control-flow to sensitive code parts [44], [67].

Due to the impact of fault attacks and the possibility of performing these attacks remotely or physically, faults pose a severe threat to many devices and require dedicated countermeasures.

B. Control-Flow Integrity

Control-flow integrity (CFI) is considered to be an effective countermeasure to thwart control-flow hijack attacks [1]. The enforcement of the control-flow integrity is implemented with various enforcement granularities and, therefore, also addresses different threat models.

1) Software CFI Schemes: Software CFI schemes (SCFI) aim to protect the control-flow of a program from a software attacker. These schemes consider an adversary exploiting memory vulnerabilities to overwrite code-pointers in memory and to perform attacks such as ROP [56] or JOP [13]. To address this threat model, SCFI schemes protect code-pointers to hinder an attacker from hijacking control-flow transfers. By maintaining the integrity of all code-pointers in the program, any tampering attempt on these pointers can be detected by SCFI schemes like CPI [28], CCFI [36], or PARTS [32].

2) Fault CFI Schemes: Protecting the integrity of code-pointers is insufficient when considering a fault attacker. SCFI schemes only protect indirect control-flow transfers, *i.e.*, indirect calls, which can be tampered by using a software vulnerability. However, FCFI additionally needs to protect direct control-flow transfers because a targeted fault on the code segment of the program can modify these transitions. Hence, FCFI schemes need to enforce the control-flow integrity at a much finer granularity than SCFI.

CFI for thwarting a fault attacker typically requires the scheme to extract the control-flow graph (CFG) at compile time. At runtime, the CFI enforcement policy can detect any deviation from this extracted CFG. Most software-based FCFI schemes [2], [24], [29], [46], [50], [52], [63] enforce the control-flow integrity at basic block granularity. Here, only transitions between basic blocks, *i.e.*, linear instruction sequences with a control-flow transfer at the end, which are in the set of allowed transitions determined by the CFG, are allowed. More precise enforcement strategies, *i.e.*, FCFI schemes [15], [64] operating on instruction granularity, typically require hardware-assistance to counteract high performance overheads.

C. Data Redundancy

In order to counteract fault attacks and protect data or addresses, redundancy is required. In general, there are two forms of redundancy available, temporal or spatial redundancy. Temporal redundancy aims to process or store data multiple times, thus providing protection against fault attacks. Concepts like instruction duplication [6], [42] are used to provide a generic protection against fault attacks.

In contrast to temporal redundancy, spatial redundancy adds additional bits to the data itself. Error detection codes [12], [23] are a well-known methodology to protect the data against fault attacks. While originally been developed to protect data during transmission and storage in harsh environments, *i.e.*, in space, those mechanisms also provide protection against fault attacks. A data encoding scheme transforms the payload data to a different representation, such that bit flips up to a certain number of faults are detectable, *i.e.*, the Hamming distance.

In the past, data encoding schemes have been developed to protect the data against fault attacks and ensure its integrity. For example, binary linear codes [23] can protect the data but also support simple logical operations. Compared to that, arithmetic codes, such as ANB-codes [20], [51] or residue codes [37], [38], support the protection during storage but also allow to perform simple arithmetic operations on the encoded data. However, pure software implementations of those encoding schemes are expensive. For example, ANB-codes have runtime overhead factors between 3 and 140. Similar to that, performing multiple module operations required for residue codes is also costly and thus not suitable for a pure software implementation.

To compensate for this performance penalty, it requires custom hardware support. For example, [39] and [54] modify the processor and add a dedicated ALU designed for multi-residue arithmetic. This allows the system to process encoded data with dedicated instructions, which have the same runtime overheads compared to plain arithmetic operations. While these approaches certainly reduce the performance penalty, it requires intrusive changes to the processor architecture. Thus, they are not realistic for commodity devices, especially for existing ARM devices.

D. ARM Pointer Authentication

ARM pointer authentication (PA) [49] was introduced with the ARMv8.3 architecture and is used to sign and verify the integrity of code- and data-pointers. ARM extended the instruction set with the dedicated sign instructions `pac*` to sign code- and data-pointers with key `A` or `B`, which can be set in privileged mode. Internally, these instructions use the tweakable block cipher QARMA [3] to calculate the MAC of the address, a modifier used as tweak, and the key. This MAC is then truncated to PAC_{size} bits and used as pointer authentication code (PAC). ARM stores this PAC in the upper, unused bits of a pointer to avoid any additional storage overhead. Although this method reduces the virtual address space, systems, such as Linux, anyway limit the address space size to 39- or 48-bit [35]. To verify the integrity of a signed pointer, the `aut*` instructions recompute the MAC of the address and compare it to the PAC stored in the upper bits. If the pointer integrity verification succeeds, the PAC is stripped from the pointer and can be used. In case of an integrity failure, the pointer is either invalidated (ARMv8.3) or an exception is triggered (ARMv8.6) [33]. The ARMv8.3 extension further includes instructions that verify the signed pointer before usage, e.g., the `braa` recomputes and compares the PAC and then branches to the destination.

The ARM PA feature is already used for pointer integrity in commercial products, beginning with the Apple iPhone XS [4]. Additionally, academic projects demonstrated that the PA extension can also be used to enforce software CFI [31], [32] and even fault CFI [52].

III. Problem Definition

In this section, we analyze state-of-the-art CFI schemes that protect from fault attackers and discuss exploitable design weaknesses.

A. A Motivating Example

To protect a program, such as the one in Listing 1, from control-flow hijacking attacks triggered by a fault, FCFI schemes aim to detect control-flow violations.

```
1  void B(); void C();
2  void D(); void E();
3
4  void A(string password)
5  {
6    void (*fun_ptr)(void) = NULL;          A1
7    ...
8    B();
9    ...                                    A2
10   if (password == "secret")
11     fun_ptr = &C;                        A3
12   else                                   A4
13     fun_ptr = &D
14   ...                                    A5
15   (*fun_ptr)();
16 }
```

Listing 1. Code snippet vulnerable to fault-based control-flow hijacking attacks.

Most of the FCFI schemes [1], [46], [52], [63]–[65] statically extract the control-flow graph (CFG) of the program

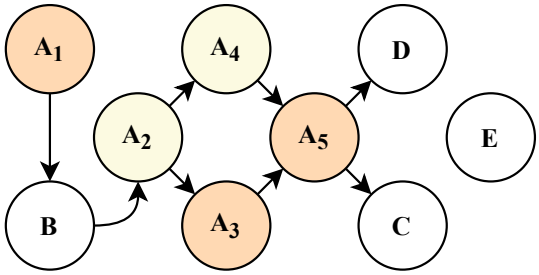

Fig. 1. Control-flow graph.

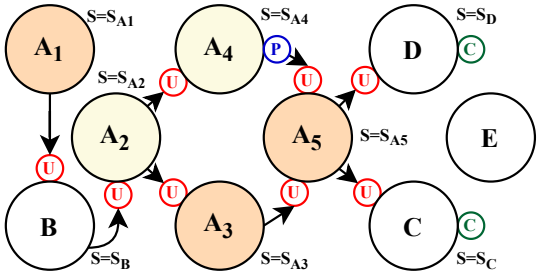

Fig. 2. Control-flow graph with state updates, patches, and checkes.

at compile time. When considering a FCFI scheme enforcing control-flow integrity at basic block level, the extracted CFG is similar to the one illustrated in Fig. 1.

To protect the program from fault-based control-flow hijacking attacks, these schemes restrict the control-flow to only valid edges of the CFG. In Fig. 1, basic block B can only be reached from basic block A_1 and basic block C and D from basic block A_5. As basic block E is never executed in the program, a control-flow redirection to this basic block violates the CFG.

To enforce the CFG at runtime, signature-based FCFI schemes internally maintain a global state S to accumulate the execution history of the program. This state is typically a counter [46] or a cryptographic chain [52]. Fig. 2 depicts the basic principle of updating (U), patching (P), and checking (C) this global CFI state S, which are explained as follows.

Updating: On each control-flow transfer, the current state S is updated when entering the next basic block.

$$S_N = f_U(S_C, ID_{BB}) = S_C \oplus ID_{BB} \tag{1}$$

Equation (1) shows the state update mechanism consisting of the current state S_C, the next state S_N, a unique basic block identifier ID_N, and the accumulating state update function $f_U()$. In the example in Fig. 2, on the control-flow transfer from A_1 to B, the state is updated from $S = S_{A_1}$ to $S = S_B = S_{A_1} \oplus ID_B$.

Patching: In most programs, the execution diverges to multiple execution flows and then merges again later. As this would create different states for each individual execution flow, signature-based CFI schemes require to patch the state to a common state on control-flow merges.

$$S_N = f_P(S_C, Patch_{BB}) = S_C \oplus Patch_{BB} \tag{2}$$

In Equation (2), this patching functionality is depicted. Similar to the state update function, the current state S_C is updated with the patch value $Patch_{BB}$. This patching mechanism assures that in basic block A_5 the state $S = S_{A_5}$ is generated by both branches. Here, the control-flow path through A_3 generates the state $S = S_{A_5} = S_{A_3} \oplus ID_{A_5}$ and the path through A_4 the same state $S = S_{A_5} = S_{A_4} \oplus Patch_{A_4} \oplus ID_{A_5}$.

Checking: To detect control-flow violations, regular checks of the global state S are required.

$$err = S_C \neq S_{expected_{BB}} \tag{3}$$

These checks, as shown in Equation (3), compare the actual state S_C with the precomputed state $S_{expected_{BB}}$ and trigger an exception on a mismatch.

B. Security Analysis

The security of signature-based FCFI schemes is based on the enforcement policy of the scheme, the precision of the extracted CFG, and the capability of the signature to detect state mismatches reliably. The enforcement policy is responsible of restricting the control-flow to the nodes of the CFG and is determined by the number and placement of the state updates and checks. While a larger number of updates and checks clearly increases the enforcement precision of the scheme, it also increases the runtime overhead of the scheme. Hence, software-based CFI schemes [46], [52], [63] typically update and check the state at basic block granularity to detect inter basic block control-flow hijacks. To also prevent intra basic block control-flow hijacks, e.g., instruction skipping, hardware-assisted schemes [15], [64] even update the state at instruction granularity.

The precision of the CFG is the fundamental cornerstone for the security of CFI schemes. However, this CFG typically is statically extracted at compile time and only offers a limited accuracy. As determining a precise set of valid targets for indirect branches is hard, the CFG used for CFI enforcement is possibly over-approximated and contains multiple target edges [16].

1) Indirect Branches: This over-approximation mainly affects indirect branches, where, in comparison to direct branches, the target address is not known at compile time. The statically extracted CFG is only an approximation of the actually executed control-flow and includes multiple targets for indirect branches. The indirect call in Line 15 in Listing 1 highlights the problematic of multiple, valid targets. Although CFI limits valid transitions from A_5 to C or D, an attacker still could redirect the control-flow within the set of valid targets using two possible fault targets:

IB1 Faulting Addresses: A targeted fault on the address used in an indirect branch allows the adversary to hijack the actual control-flow and redirect it to another target. The target address in `fun_ptr` could be tampered to point to D instead of C. As both basic blocks are in the list of

valid targets, a genuine state is generated in both branches and the state verification mechanism cannot detect the control-flow hijack.

IB2 Faulting the Branch: Even if the address is not modified by a fault, a targeted fault directly on the branch could redirect the control-flow. Here, the target address in `fun_ptr` used by the indirect branch points to C but a fault on this branch redirects the control-flow to D. Again, as both targets are possible paths through the CFG, the control-flow redirection remains undetected by the CFI scheme.

C. Threat Model

Similar to threat models of related CFI schemes [2], [24], [29], [46], [52], [63] protecting the control-flow of the program against fault attacks, we are considering an attacker having physical access to the system. This attacker is capable of inserting a fault using, e.g., a clock or a voltage glitch, and aims to hijack the control-flow of the program to redirect it to other sensitive code parts. We assume that the system already features a state-based CFI scheme, such as, for example, FIPAC [52], thwarting fault attacks on the control-flow. In addition to this threat model, we are considering an attacker aiming to bypass the protection of indirect branches using the fault targets **IB1** or **IB2**. These targets can be attacked by inducing a fault during program runtime or directly into the code stored in the instruction memory. Here, we consider faults flipping bits in addresses used by indirect branches stored in registers, in the immediate field of instructions, or directly in the code segment.

IV. Design

To address the threat model in Section III-C and counteract the identified weaknesses **IB1** and **IB2**, we show in this section how to thwart fault attacks on indirect branches on recent ARM architectures supporting pointer authentication.

1) Address Protection: Using a targeted fault **IB1** on an address used in an indirect branch allows an adversary to hijack the control-flow of the program. To counteract such attacks, protecting and verifying the integrity of these addresses throughout the program execution is required. However, as indicated in Section II-C, software-based redundancy schemes induce large performance overheads [51] and hardware-assisted schemes minimizing these performance overheads require intrusive hardware changes [54], making it difficult to deploy these schemes on off-the-shelf devices.

To efficiently and securely protect addresses used for indirect branches without hardware changes, we utilize the pointer authentication feature of recent commodity devices from ARM. This feature introduces, as described in Section II-D, dedicated instructions capable of cryptographically signing and verifying pointers.

Fig. 3 depicts a 64-bit pointer signed with the `pac*` instruction. This instruction uses an optional modifier and a preconfigured key K to calculate the MAC of the 64-bit address. In the latest ARMv8.6-A architecture, the result

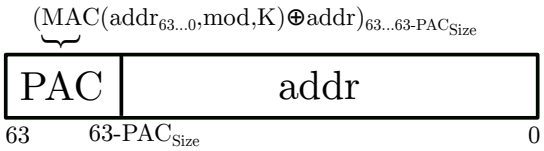

Fig. 3. Pointer signed with ARM pointer authentication.

is then XORed with the original address and stored in the upper, unused bits of the pointer. To verify the integrity of the signed pointer before usage, ARM provides dedicated `aut*` instructions and combined instructions, such as `blr*` or `br*`.

To protect all addresses used by indirect branches, we utilize ARM's pointer authentication to sign and verify these addresses. More concretely, we exploit the PA feature to efficiently store address redundancy information, *i.e.*, the pointer authentication code, next to the actual address inside the pointer using hardware support. At compile time, we replace the unprotected addresses with their signed equivalent, *i.e.*, the address with the corresponding pointer authentication code (PAC). Furthermore, we replace all indirect branch instructions, *i.e.*, *branch register* or *branch and link register*, with their PA equivalent, which automatically verify the PAC before usage. Although signing code-pointers at runtime with the `pac*` instructions would be possible, an attacker still could induce a bit flip into the unprotected address before the PA instruction protects the address, or directly into the code segment of the program.

The approach of replacing addresses with their PA protected version and verifying them before usage yields, compared to software-based address protection schemes, several advantages. By embedding the redundancy information directly into the corresponding pointer, the design avoids the usage of additional registers and additional register pressure. Furthermore, as ARM's PA utilizes features of the underlying architecture, the generation and verification of the address redundancy information can be realized using a single instruction, avoiding large performance penalties. Most important, compared to related address protection schemes [54], our design does not require custom hardware changes and can be deployed on off-the-shelf hardware from ARM.

2) Linking the Branch: Independently whether the address is protected or not, inducing a bit flip when a code-pointer is used by the indirect branch instruction allows the adversary to hijack the control-flow of the program. Even though ARM PA provides dedicated branch instructions, e.g., `blr*` or `br*`, these instructions first verify the integrity of the address by recomputing and comparing the PAC, removeing the PAC from the pointer, and then use this unprotected address for the jump. Hence, a targeted bit flip still enables the attacker to redirect the control-flow to another valid edge in the control-flow graph, which cannot be detected by CFI.

To mitigate this attack vector **IB2**, we propose to enhance state-update functions of existing state-based CFI schemes. By merging the target address into the global CFI state at the

caller side and by removing this address at the callee, we are creating a link between the indirect control-flow transfer. More concretely, we are inducing the target address into the state using a XOR, and we remove this added address by XORing the current address at the callee into the state.

V. Implementation

This section first introduces FIPAC, the state-based CFI scheme we enhance with our indirect branch protection mechanisms. Then, we elaborate on how we integrate our pointer authentication-based address protection scheme and the hardened state update function linking indirect control-flow transfers into FIPAC.

A. FIPAC

FIPAC [52] is a CFI scheme protecting the control-flow against a software and fault attacker. Similar to other software-based FCFI schemes, FIPAC enforces control-flow integrity at basic block granularity using a statically extracted CFG. This CFI scheme exploits hardware features of the ARMv8.6-A architecture to efficiently derive a cryptographic state at each basic block entry. In FIPAC, the state update function uses the pointer authentication (c.f. Section II-D) instructions to efficiently create a MAC chain with the execution history.

$$S = \texttt{pacia}(S_P, PC, K_A) = MAC_{K_A}(S_P, PC) \oplus S_P \quad (4)$$

Equation (4) shows the update function, which cryptographically links the previous basic block with the current basic block, using the `pacia` instruction. The `pacia` instruction generates a MAC using the previous state S_P, the current program counter, and a key K_A and XORs the result to the previous state. This global CFI state S is then stored into a register exclusively reserved for FIPAC. To verify the integrity of the executed basic blocks, FIPAC uses the `autiza` instruction as checking mechanism. On an integrity verification failure, *i.e.*, a control-flow violation within the CFG at basic block granularity, this instruction triggers an exception. FIPAC automatically protects programs by providing a custom LLVM-based toolchain.

B. Address Protection

Faulting a code-pointer in the program allows the adversary to redirect the control-flow for indirect branches within the bounds of the CFG. To thwart this attack scenario **IB1**, we extend FIPAC to provide protection for all addresses used by indirect branches by exploiting hardware features of the underlying ARM architecture. More concretely, we replace these addresses at compile time with their encoded version embedding the PAC in the upper bits. In addition, we substitute all branch instructions with their PA equivalent.

To realize the address protection scheme, we introduce a custom `ModulePass` in the LLVM [30] middle-end of the FIPAC toolchain performing a data-flow analysis. We scan each function for indirect calls and track the corresponding virtual register in the IR containing the address information. By exploiting the `def-use` property of the static single assignment (SSA) form of LLVM, *i.e.*, each used virtual register is defined at exactly one position, we find the initial store instruction copying the address into a virtual register. If this address is a global value of the type function, we use the LLVM metadata functionality to tag the store with `defPAC` and the indirect branch instruction with `usePACBranch`. Inside the `IRTranslator` and the `AArch64InstructionSelector` pass in the LLVM backend, we utilize this metadata to replace the indirect branch instructions marked with `usePACBranch` with their PA equivalent, *i.e.*, `braaz` or `blraaz`. These instructions, which use a zero modifier, verify the integrity of the address by recomputing and comparing the PAC stored in the pointer before invoking the address. If the verification fails, an error is triggered. Furthermore, we replace all `adr` instructions marked with the `defPAC` tag with four consecutive `mov` instructions to load a 64-bit immediate value into a register. While the first `mov` instruction in Line 4 in Listing 2 clears the register and stores the first 16-bits of the address into the register, the other 16-bit values of the address are shifted using three `movk` instructions. As the address information is only available in the linker, we add a custom relocation target to the address stored in the register. This relocation target then is resolved in the linker, where we replace the actual address with the protected version, *i.e.*, we compute the PAC of the address and store it in the upper bits of the pointer.

```
1  <main>:
2    ...
3    ; adr x8, #function //defPAC
4    mov  x8, #function_PAC[15:0]
5    movk x8, #function_PAC[31:16], lsl #16
6    movk x8, #function_PAC[47:32], lsl #32
7    movk x8, #function_PAC[63:48], lsl #48
8    ...
9    ; br  x8 //usePACBranch
10   braaz x8
11   ...
```

Listing 2. Replacement of unprotected addresses and branch instructions with their PA protected version.

Furthermore, we utilize the SSA of the IR in our custom `ModulePass` to find instructions loading an global addresses of the function type into a virtual register, which is not used by an indirect branch instruction in the current function. In the backend, we identify these instructions tagged with `usePAC` and insert an `autiza` instruction afterwards, as highlighted in Line 11 in Listing 3. This instructions uses a zero modifier to verify the PAC and stores the unprotected address back to the target register. As performing a data-flow analysis over function boundaries is challenging, e.g., functions in external libraries or functions in other C files, which cannot be accessed by the `ModulePass`, we use this mechanism to pass unprotected addresses as function arguments.

Inside a function, we scan for indirect branch instructions

```
1  <main>:
2    ...
3    ; adr x8, #function //defPAC
4    mov  x8, #function_PAC[15:0]
5    movk x8, #function_PAC[31:16], lsl #16
6    movk x8, #function_PAC[47:32], lsl #32
7    movk x8, #function_PAC[63:48], lsl #48
8    str x8, [sp, #16]
9    ...
10   ldr x0, [sp, #16] ; //usePAC
11   autiza x0
12   b #function
13 <function>:
14   paciza x0
15   str x0, [sp, #16] ; //defPAC
16   ...
17   ldr x8, [sp, #16]
18   ; br  x8 //usePACBranch
19   braaz x8
```

Listing 3. Passing protected addresses to functions.

```
1  <main>:
2    ...
3    state_update(S)
4    eor S, S, x8
5    br  x8
6    ...
7  <function>:
8    adr x27, #function
9    eor S, S, x27
10   state_patch(S)
11   ...
```

Listing 4. Target address insertion into the global CFI state S.

and use the SSA to check, if the target address is passed as a function argument. In this case, we tag the branch instruction with `usePACBranch` to translate it later to an protected branch instruction and tag the store instruction with `defPAC`. In the backend, we extend all store instructions tagged with `defPAC` with a `paciza`, as shown in Line 14 in Listing 3. The `paciza` instructions calculates the PAC of the address using key K_A and the zero modifier and stores this PAC into the upper bits of the pointer.

C. Linking the Branch

To protect indirect branches from threat **IB2**, we induce the target address of the indirect branch into the global CFI state at the caller and remove this address at the callee, which allows us to detect control-flow hijacking attempts.

Listing 4 highlights the basic principle of this protection mechanism. After the actual state update **state_update(S)** of the CFI scheme, we merge the target address of the indirect branch, which is stored in a register, into the state S using an XOR. At the entry of the called function, we first remove the induced address by XORing the state with the address of the current function. Eventually, the CFI scheme patches the state using **state_patch(S)**.

In order to integrate this mechanism into the LLVM toolchain of FIPAC, we created a custom `MachineFunctionPass` in the LLVM backend scanning for indirect branch instructions. Between the already existing CFI state update function and the indirect call instruction, as depicted in Line 4 in Listing 4, we insert a bitwise exclusive OR (`eor`) instruction inducing the target address stored in register `x8` into the global state register. To remove this address from the state, our LLVM pass extends the function header of the callee with two instructions, as illustrated in Line 8 and 9 in Listing 4. We utilize the `adr` instruction, which allows to form a PC-relative address using a (negative) offset, to determine the starting address of the current function. Then, we again use an `eor` to add this address to the current state S. As this function could be invoked from different callers and state-based CFI schemes require to have a single unique CFI state at a certain position, removing the address induced into the state at the callee is necessary.

D. Combination

When combining both approaches, the caller side XORs the protected address containing the PAC in the upper bits of the pointer to the global CFI state S. At the callee, the unprotected address of the invoked function is determined using the `adr` instruction. To compute the encoded pointer, we insert an additional `paciza` between the `adr` instruction and the XOR correction the state. On a valid indirect control-flow transfer, this instruction computes the same PAC as on the caller side.

E. Key Management

The precomputation of the encoded addresses at compile time requires the toolchain to have access to the key for the PAC generation. As the system at runtime needs to have the same key for verifying the encoded addresses using the `autiza`, `blraaz`, and `braaz` instructions, we use a pre-shared key K_A. This PA key is configured by a custom Linux kernel module running in the kernel mode. We discuss security implications and alternative key sharing approaches in Section VI-D.

F. Compatibility with other CFI schemes

The protection of addresses used by indirect branches can be integrated into other CFI schemes related to FIPAC, if they are deployed on hardware supporting ARM pointer authentication. State-based CFI schemes, such as CFCSS [46], ACFC [63], or SWIFT [50], update and verify a global state on each basic block. As our approach thwarting **IB1** encodes addresses used by indirect branches and does not influence this state generation and verification mechanism of the underlying CFI scheme, our scheme is fully compatible with these schemes.

To protect from attack vector **IB2**, the state update mechanism of these CFI schemes could be extended to induce the target address into the state at the caller side and remove this

address at the callee. By using an XOR, this approach works for schemes using a counter [24] or signatures [65].

VI. Evaluation

In this section, we first evaluate the performance impact and the code size overhead of hardening indirect branches against fault attacks. Then, we demonstrate the functional correctness of our proposed changes on an emulator supporting the required pointer authentication instructions. Finally, we analyze security guarantees of our enhanced CFI scheme thwarting fault attacks on indirect branches.

A. Performance Evaluation

To evaluate the performance overhead introduced by the additional protection of indirect branches, we compile a set of benchmarks and execute them on an ARM development board. However, currently, there is no open development board available supporting the ARM pointer authentication instructions introduced in ARMv8.3-A. Although Apple offers this feature in their mobile processors [26], iOS restricts the usage of PA by custom software [4]. While pointer authentication currently is not broadly available, with the announcement of ARM supporting PA in the upcoming ARMv9-A architecture [34], we expect more devices featuring this extension in the near future. Due to the lack of openly available hardware, we emulate the PA instructions on the Raspberry Pi 4 Model B [21] consisting of a 64-bit ARMv8-A SoC. To emulate these instruction, we reuse the cycle accurate emulation approach introduced by PARTS [32] and used by related work [31], [52]. This PA-analogue consists of four consecutive XORs simulating the cycles needed for a PA instruction and one memory access.

a) SPEC2017: To quantify the performance overhead introduced by the protection of indirect branches, we execute the SPECspeed2017 [14] benchmark suite on the Raspberry Pi. More concretely, we compile all C-based benchmarks without OpenMP support of the SPECspeed2017 Integer test suite with our customized toolchain using the cycle accurate emulation approach for different protection configurations.

In Fig. 4, we depict the performance overhead induced by the protection of indirect branches on top of the baseline, *i.e.*, the SPEC2017 benchmark compiled with the FIPAC toolchain. On average, encoding addresses and verifying them at each indirect branch using the dedicated `blraaz` and `braaz` instructions yields a performance overhead of 1.50%. The protection of the link between indirect control-flow transfers induces a runtime overhead of 0.83% on average. For the combination of both protection mechanism, we measured an average performance overhead of 2.34%.

B. Code Size Overhead Evaluation

Protecting indirect branches from threats **IB1** and **IB2** requires the insertion of additional instructions into the program. More concretely, for the address protection of **IB1**, we replace `adr` instructions containing an address used by an indirect branch with four consecutive `mov` instructions containing the

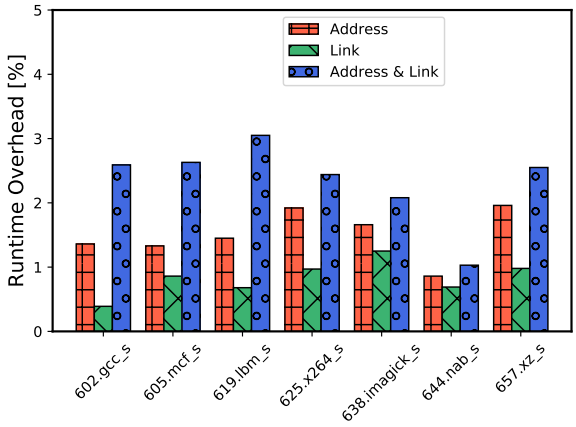

Fig. 4. SPECspeed2017 performance overhead for different protection configurations.

PAC and the address. Furthermore, to pass protected addresses as function arguments, we insert additional `autiza` and `paciza` instructions to the function headers and entries. To address threat **IB2**, we induce the target address of the indirect branch into the global CFI state S at the caller side and then correct the CFI state at the callee using additional XORs.

The average code size overhead for thwarting **IB1** is 0.16% and 1.30% for **IB2** on average for the SPEC2017 benchmark, as illustrated in Tab. I. As the FIPAC toolchain automatically creates two function entry points, one for indirect and one for direct calls, we automatically extend the indirect function header entry with the `adr` and `eor` instruction. Hence, the code size overhead is larger for protecting the link between indirect control-flow transfers than for the address encoding using the PA feature. When combining the address and the indirect control-flow transfer protection, we measured an average code size overhead of 1.92%. Note that these overheads are on top of the underlying CFI scheme, *i.e.*, FIPAC.

C. Functional Evaluation

To verify the functional correctness of our proposed CFI enhancements for FIPAC, we executed the protected SPEC2017 benchmark on a recent Linux kernel on the QEMU [7] em-

TABLE I
Code size overhead for SPECspeed2017.

Testcase	Address	Link	Address & Link
602.gcc_s	0.13	1.76	2.70
605.mcf_s	0.23	1.34	1.55
619.lbm_s	0.27	0.56	1.03
625.x264_s	0.10	1.37	1.95
638.imagick_s	0.20	1.06	1.59
644.nab_s	0.11	1.17	1.81
657.xz_s	0.11	1.87	2.79
Average	**0.16**	**1.30**	**1.92**

ulator. However, as FIPAC requires the pointer authentication features of the ARMv8.6-A architecture and the latest QEMU 6.0 version only offers PA provided in ARMv8.3-A, we enhanced QEMU to support the `EnhancedPAC2` and `FPAC` features [52]. The successful execution of SPEC2017 compiled with our extended LLVM compiler hardening indirect branches validates the functional correctness of our proposed CFI policy refinement.

D. Security Evaluation

As highlighted in Section III, a fault attacker can hijack the control-flow of a program even if a CFI scheme is in place. By inducing a targeted fault into an indirect branch, an attacker can redirect the control-flow within the bounds of the CFG. In this section, we analyze how the protection of indirect branches improves security guarantees for attacker models **IB1** and **IB2**.

1) Address Protection: In a CFI scheme without explicit indirect branch protection, a single fault into an address used by an indirect branch allows the adversary to hijack the control-flow. When protecting these addresses using the ARM pointer authentication feature, an attacker now needs to induce two faults, one in the address and one in the pointer authentication code. As this PAC, in comparison to other data redundancy schemes, is calculated using a keyed MAC, an attacker, without having access to this key, cannot predict a valid PAC for a target address. Hence, even when having the capability of inducing two precise faults, the likelihood of generating a valid address and PAC pair with a fault is low.

For **IB1**, a bit flip in a protected address is detected by the next verification instruction, e.g., `blraaz`. These instructions recompute the PAC using the given address and compare it with the PAC stored in the upper bits of the pointer. If either the address or the PAC is erroneous, the comparison fails and, in the ARMv8.3-A architecture, an error bit is set in the pointer. When using this corrupted pointer, e.g., in the `blraaz` instruction, an error is triggered. As a single error bit can easily be flipped by a fault, we recommend using the ARMv8.6-A architecture, where an authentication error automatically triggers an exception directly in the authentication mechanism.

2) Linking the Branch: To address attacker model **IB2**, we merge the target address of the indirect branch at the caller side into the global CFI state and remove this address at the callee. Now, to bypass this protection mechanism, an adversary needs to induce an additional, precise fault on the XOR, which removes the address from the global state. Combining this concept with the address protection, *i.e.*, XORing the PA protected address into the state and remove this address by recomputing the PAC using the `paciza` instruction, further improves security guarantees.

3) Key Management: Protecting addresses by replacing them at compile time with their protected PA equivalent requires the toolchain to have access to the used key as the key needs to be identical for the verification at runtime. However, in our threat model considering a physical attacker, we argue that a static key still provides strong protection against faults. An attacker without knowing the key needs to induce a targeted fault into the address and the PAC with the goal of crafting a valid PAC. This requires to flip up to PAC_{size} bits in the PAC. When extending our threat model with an adversary being capable of extracting the PA keys from the highest privilege level, this attacker could precompute a valid PAC. However, the adversary still needs to have the capabilities of inducing a precise fault into the address and also into the PAC. Additionally, the underlying CFI scheme already restricts the control-flow to only valid edges of the CFG. To support dynamic keys, a kernel module configuring ephemeral keys for the PA feature and recomputing and replacing all PACs in the program using binary rewriting could be integrated into the OS.

4) Function Arguments: In our current prototype implementation, we do not conduct an exhaustive data-flow analysis over function boundaries. Instead, before passing a protected address as an argument to a function, we verify the integrity of the pointer and remove the PAC using the `autiza` instruction. This approach allows us to pass code-pointers as function arguments to external functions, e.g., provided by a dynamically linked library. Inside of a function, we analyze the function arguments and protect all addresses used by indirect branch instructions using the `paciza` instruction. Hence, addresses passed as function arguments in a register or on the stack are, for a short moment, unprotected, allowing an adversary to induce a bit flip. However, in a future version of our prototype, this attack vector can be avoided by using a statically linked library and performing a data-flow analysis on the compiled binary and replace unprotected addresses using binary rewriting. Note that this approach also would reduce the performance impact of mitigating attacker model **IB1**, as no additional `autiza` and `paciza` instructions are required.

5) Conditional Branches: Previous research [5], [58] has shown that inducing a fault into a conditional branch allows an attacker to hijack the control-flow and, for example, bypass secure boot. To thwart fault attacks on conditional branches, protection for the operands, the comparison, and the branch itself is required [53]. Although addressing these attack vectors using the ARM pointer authentication feature is possible, there are some major limitations with this approach. While the PA feature could be used to protect operands of the conditional branches by embedding the PAC into the upper bits of the value, performing arithmetic operations or comparisons directly on the protected value are not possible. Hence, before each operation, the PAC needs to be removed, allowing an attacker to still induce a bit flip. In addition, storing the PAC in the upper bits of the value reduces the data size from 64-bits to $64 - PAC_{size}$ bits. While this is not a limitation for pointers, as the addresses in pointers do not occupy the whole 64-bit space, reducing the data size for values is not practical for different scenarios. Hence, CFI schemes, independently if they are extended with our indirect branch protection, need to be complimented with addition countermeasures addressing attacks on conditional branches.

VII. Related Work

This section summarizes related work and compares them to our address protection and branch linking approach.

A. Code-Pointer Integrity

Code-pointer integrity (CPI) [28] is a SCFI scheme protecting sensitive code-pointers in a program from an adversary exploiting a memory vulnerability by storing metadata to these pointers in an isolated memory region. As CPI considers a software attacker in its threat model, CPI protects accesses to these safe memory regions. However, as a targeted fault still can tamper the metadata stored in the protected memory, CPI cannot hinder an attacker hijacking the control-flow using faults.

Similar to CPI, CCFI [36] protects code-pointers in the program by storing a MAC next to the pointer. By utilizing features of the hardware, *i.e.*, AES-NI, the `macptr` and `checkptr` instructions compute and verify the MAC for each sensitive pointer. Although this approach protects code-pointers stored in memory from fault attacks, these pointers are stored unprotected in registers, allowing a fault to tamper addresses and redirect the control-flow.

PARTS [32] utilizes the PA feature of recent ARM architectures to sign and verify all code- and data-pointers in the program. Although this approach is similar to our work, PARTS only considers a software adversary in their threat model. Hence, PARTS only signs these pointers at runtime, allowing a fault attacker to either induce a fault directly in the address before the pointer is signed using a `pac*` instruction or directly into the code segment of the binary. Additionally, PARTS is vulnerable to attack vector **IB2**, as after the verification of the PAC, the unprotected address is used for the jump. In comparison to PARTS, our approach protects addresses used by indirect branches from faults throughout the program's whole execution life cycle and the binary in the instruction memory by replacing unprotected addresses with their protected version during compile time. By merging the target address of indirect branches directly into the global CFI state and removing this address at the callee, we further prevent an attacker from exploiting **IB2**.

VIII. Conclusion

In this work, we showcased how the missing address protection in state-of-the-art fault CFI schemes allow an attacker to hijack the execution of indirect branches by inducing targeted faults. While data encoding schemes could mitigate these attacks, software-based schemes yield large runtime overheads and hardware-based systems require intrusive hardware changes.

To efficiently protect indirect branches on commodity devices from ARM, we propose a hardware-assisted data redundancy scheme. We utilize the pointer authentication feature of recent ARM architectures to embed a MAC into all addresses used by indirect branches. By replacing unprotected addresses at compile time with their protected equivalent, we thwart fault attacks on these addresses stored in registers, in the immediate field of instructions, or directly in the binary. We further replace all indirect branch instructions with their PA equivalent, automatically verifying the integrity of the MAC and triggering an exception on mismatch. Additionally, we enhance the state update mechanism of signature-based CFI schemes to protect the link between indirect control-flow transfers. By merging the target address of the indirect branch into the global CFI state at the caller side and correcting the state at the callee with the current address, we assure that the executed control-flow follows the intended control-flow. To demonstrate how these defense mechanisms improve the protection of state-of-the-art CFI schemes, we integrate our address encoding and linking strategy into FIPAC, a previously introduced CFI scheme. The integration of these countermeasures into the LLVM-based toolchain of FIPAC allows to automatically protect indirect branches of programs without user interaction. To evaluate the hardware-assisted address redundancy scheme and the protection of indirect control-flow transfers, we compiled a subset of the SPEC2017 benchmark with our custom LLVM-based toolchain and executed the protected binary on a Raspberry Pi. Our evaluation shows an negligible runtime overhead of less than 2.34% on average.

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] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13:4:1–4:40, 2009.

[2] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-FLAT: Control-Flow Attestation for Embedded Systems Software. In *Conference on Computer and Communications Security – CCS*, pages 743–754, 2016.

[3] Roberto Avanzi. The QARMA Block Cipher Family. Almost MDS Matrices Over Rings With Zero Divisors, Nearly Symmetric Even-Mansour Constructions With Non-Involutory Central Rounds, and Search Heuristics for Low-Latency S-Boxes. *IACR Trans. Symmetric Cryptol.*, 2017:4–44, 2017.

[4] Brandon Azad. Examining pointer authentication on the iphone xs. https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html, 2019. [accessed 2020-09-22].

[5] Guillaume Barbu, Guillaume Duc, and Philippe Hoogvorst. Java Card Operand Stack: Fault Attacks, Combined Attacks and Countermeasures. In *Smart Card Research and Advanced Applications – CARDIS*, volume 7079 of *LNCS*, pages 297–313, 2011.

[6] Thierno Barry, Damien Couroussé, and Bruno Robisson. Compilation of a Countermeasure Against Instruction-Skip Fault Attacks. In *Cryptography and Security in Computing Systems – CS2@HiPEAC*, pages 1–6, 2016.

[7] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference – USENIX ATC*, pages 41–46, 2005.

[8] Eli Biham and Adi Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In *Advances in Cryptology – CRYPTO*, volume 1294 of *LNCS*, pages 513–525, 1997.

[9] Johannes Blömer and Jean-Pierre Seifert. Fault Based Cryptanalysis of the Advanced Encryption Standard (AES). In *Financial Cryptography – FC*, volume 2742 of *LNCS*, pages 162–181, 2003.

[10] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). In *Advances in Cryptology – EUROCRYPT*, volume 1233 of *LNCS*, pages 37–51, 1997.

[11] Claudio Bozzato, Riccardo Focardi, and Francesco Palmarini. Shaping the Glitch: Optimizing Voltage Fault Injection Attacks. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019:199–224, 2019.

[12] David T. Brown. Error Detecting and Correcting Binary Codes for Arithmetic Operations. *IRE Trans. Electron. Comput.*, 9:333–337, 1960.

[13] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Conference on Computer and Communications Security – CCS*, pages 559–572, 2010.

[14] Standard Performance Evaluation Corporation. Spec cpu 2017. https://www.spec.org/cpu2017, 2019. [accessed 2020-09-22].

[15] Ruan de Clercq, Johannes Götzfried, David Übler, Pieter Maene, and Ingrid Verbauwhede. SOFIA: Software and control flow integrity architecture. *Comput. Secur.*, 68:16–35, 2017.

[16] Ruan de Clercq and Ingrid Verbauwhede. A survey of Hardware-based Control Flow Integrity (CFI). *CoRR*, abs/1706.07257, 2017.

[17] Jan Van den Herrewegen, David F. Oswald, Flavio D. Garcia, and Qais Temeiza. Fill your Boots: Enhanced Embedded Bootloader Exploits via Fault Injection and Binary Analysis. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021:56–81, 2021.

[18] Christoph Dobraunig, Maria Eichlseder, Hannes Groß, Stefan Mangard, Florian Mendel, and Robert Primas. Statistical Ineffective Fault Attacks on Masked AES with Fault Countermeasures. In *Advances in Cryptology – ASIACRYPT*, volume 11273 of *LNCS*, pages 315–342, 2018.

[19] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Stefan Mangard, Florian Mendel, and Robert Primas. SIFA: Exploiting Ineffective Fault Inductions on Symmetric Cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018:547–572, 2018.

[20] Philippe Forin. Vital coded microprocessor principles and application for various transit systems. *IFAC Proceedings Volumes*, 23(2):79–84, 1990.

[21] Raspberry Pi Foundation. Raspberry pi 4. https://www.raspberrypi.org/products/raspberry-pi-4-model-b/, 2021. [accessed 2021-01-28].

[22] Free60.org. Reset Glitch Hack. http://free60.org/wiki/Reset_Glitch_Hack. [accessed 2020-09-19].

[23] Richard W Hamming. Error detecting and error correcting codes. *Bell Labs Technical Journal*, 29(2):147–160, 1950.

[24] Karine Heydemann, Jean-François Lalande, and Pascal Berthomé. Formally verified software countermeasures for control-flow integrity of smart card C code. *Comput. Secur.*, 85:202–224, 2019.

[25] Michael Hutter and Jörn-Marc Schmidt. The Temperature Side Channel and Heating Fault Attacks. In *Smart Card Research and Advanced Applications – CARDIS*, volume 8419 of *LNCS*, pages 219–235, 2013.

[26] Apple Inc. Apple soc security. https://support.apple.com/guide/security/apple-soc-security-sec87716a080/web. [accessed 2021-04-26].

[27] Yoongu Kim, Ross Daly, Jeremie S. Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *International Symposium on Computer Architecture – ISCA*, pages 361–372, 2014.

[28] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer Integrity. In *Operating Systems Design and Implementation – OSDI*, pages 147–163, 2014.

[29] Jean-François Lalande, Karine Heydemann, and Pascal Berthomé. Software Countermeasures for Control Flow Integrity of Smart Card C Codes. In *European Symposium on Research in Computer Security – ESORICS*, volume 8713 of *LNCS*, pages 200–218, 2014.

[30] Chris Lattner and Vikram S. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Symposium on Code Generation and Optimization – CGO*, pages 75–88, 2004.

[31] Hans Liljestrand, Thomas Nyman, Lachlan J Gunn, Jan-Erik Ekberg, and N Asokan. Pacstack: an authenticated call stack. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[32] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *USENIX Security Symposium*, pages 177–194, 2019.

[33] Arm Limited. Arm architecture reference manual. https://developer.arm.com/documentation/ddi0487/latest/, 2020. [accessed 2021-01-26].

[34] ARM Limitedv. Arm's solution to the future needs of ai, security and specialized computing is v9. https://www.arm.com/company/news/2021/03/arms-answer-to-the-future-of-ai-armv9-architecture. [accessed 2021-04-26].

[35] Catalin Marinas. Memory layout on aarch64 linux. https://www.kernel.org/doc/html/latest/arm64/memory.html, 2021. [accessed 2021-01-26].

[36] Ali José Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: Cryptographically Enforced Control Flow Integrity. In *Conference on Computer and Communications Security – CCS*, pages 941–951, 2015.

[37] James L Massey. Survey of residue coding for arithmetic errors. *International Computation Center Bulletin*, 3(4):3–17, 1964.

[38] James L Massey and Oscar N García. Error-correcting codes in computer arithmetic. In *Advances in Information Systems Science*, pages 273–326. Springer, 1972.

[39] Marcel Medwed and Stefan Mangard. Arithmetic logic units with high error detection rates to counteract fault attacks. In *Design, Automation & Test in Europe – DATE*, pages 1644–1649, 2011.

[40] Alyssa Milburn, Niek Timmers, Nils Wiersma, Ramiro Pareja, and Santiago Cordoba. There will be glitches: Extracting and analyzing automotive firmware efficiently. *Black Hat USA*, 2018.

[41] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller. In *Fault Diagnosis and Tolerance in Cryptography – FDTC*, pages 77–88, 2013.

[42] Nicolas Moro, Karine Heydemann, Emmanuelle Encrenaz, and Bruno Robisson. Formal verification of a software countermeasure against instruction skip attacks. *J. Cryptogr. Eng.*, 4:145–156, 2014.

[43] Kit Murdock, David F. Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *IEEE Symposium on Security and Privacy – S&P*, pages 1466–1482, 2020.

[44] Pascal Nasahl and Niek Timmers. Attacking autosar using software and hardware attacks. In *escar USA*, 2019.

[45] Colin O'Flynn. BAM BAM!! On Reliability of EMFI for in-situ Automotive ECU Attacks. *IACR Cryptol. ePrint Arch.*, 2020:937, 2020.

[46] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Control-flow checking by software signatures. *IEEE Trans. Reliab.*, 51:111–122, 2002.

[47] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies. In *Conference on Computer and Communications Security – CCS*, pages 195–209, 2019.

[48] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. VoltJockey: Breaking SGX by Software-Controlled Voltage-Induced Hardware Faults. In *IEEE Asian Hardware Oriented Security and Trust Symposium – ASIANHOST*, pages 1–6, 2019.

[49] Inc. Qualcomm Technologies. Pointer authentication on armv8.3. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf, 2017. [accessed 2021-01-26].

[50] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software Implemented Fault Tolerance. In *Symposium on Code Generation and Optimization – CGO*, pages 243–254, 2005.

[51] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. ANB- and ANBDmem-Encoding: Detecting Hardware Errors in Software. In *Computer Safety, Reliability, and Security, 29th International Conference, SAFECOMP 2010, Vienna, Austria, September 14-17, 2010. Proceedings*, volume 6351 of *LNCS*, pages 169–182, 2010.

[52] Robert Schilling, Pascal Nasahl, and Stefan Mangard. FIPAC: Thwarting Fault- and Software-Induced Control-Flow Attacks with ARM Pointer Authentication. *CoRR*, abs/2104.14993, 2021.

[53] Robert Schilling, Mario Werner, and Stefan Mangard. Securing conditional branches in the presence of fault attacks. In *Design, Automation & Test in Europe – DATE*, pages 1586–1591, 2018.

[54] Robert Schilling, Mario Werner, Pascal Nasahl, and Stefan Mangard. Pointing in the Right Direction - Securing Memory Accesses in a Faulty World. In *Annual Computer Security Applications Conference – ACSAC*, pages 595–604, 2018.

[55] Bodo Selmke, Florian Hauschild, and Johannes Obermaier. Peak Clock: Fault Injection into PLL-Based Systems via Clock Manipulation. In *Proceedings of the 3rd ACM Workshop on Attacks and Solutions in Hardware Security Workshop, ASHES@CCS 2019, London, UK, November 15, 2019*, pages 85–94, 2019.

[56] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Conference on Computer and Communications Security – CCS*, pages 552–561, 2007.

[57] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In *USENIX Security Symposium*, pages 1057–1074, 2017.

[58] N Timmers and A Spruyt. Bypassing secure boot using fault injection. *Black Hat Europe*, 2016, 2016.

[59] Niek Timmers and Cristofaro Mune. Escalating Privileges in Linux Using Voltage Fault Injection. In *Fault Diagnosis and Tolerance in Cryptography – FDTC*, pages 1–8, 2017.

[60] Niek Timmers, Albert Spruyt, and Marc Witteman. Controlling PC on ARM Using Fault Injection. In *Fault Diagnosis and Tolerance in Cryptography – FDTC*, pages 25–35, 2016.

[61] Jasper G. J. van Woudenberg, Marc F. Witteman, and Federico Menarini. Practical Optical Fault Injection on Secure Microcontrollers. In *Fault Diagnosis and Tolerance in Cryptography – FDTC*, pages 91–99, 2011.

[62] Aurélien Vasselle, Hugues Thiebeauld, Quentin Maouhoub, Adèle Morisset, and Sébastien Ermeneux. Laser-Induced Fault Injection on Smartphone Bypassing the Secure Boot-Extended Version. *IEEE Trans. Computers*, 69:1449–1459, 2020.

[63] Rajesh Venkatasubramanian, John P. Hayes, and Brian T. Murray. Low-Cost On-Line Fault Detection Using Control Flow Assertions. In *International Symposium on On-Line Testing and Robust System Design – IOLTS*, pages 137–143, 2003.

[64] Mario Werner, Thomas Unterluggauer, David Schaffenrath, and Stefan Mangard. Sponge-Based Control-Flow Protection for IoT Devices. In *IEEE European Symposium on Security and Privacy – EURO S&P*, pages 214–226, 2018.

[65] Mario Werner, Erich Wenger, and Stefan Mangard. Protecting the Control Flow of Embedded Processors against Fault Attacks. In *Smart Card Research and Advanced Applications – CARDIS*, volume 9514 of *LNCS*, pages 161–176, 2015.

[66] Nils Wiersma and Ramiro Pareja. Safety != Security: On the Resilience of ASIL-D Certified Microcontrollers against Fault Injection Attacks. In *Fault Diagnosis and Tolerance in Cryptography – FDTC*, pages 9–16, 2017.

[67] Google Project 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 2021-04-27].