FIPAC: Thwarting Fault- and Software-Induced Control-Flow Attacks with ARM Pointer Authentication

Robert Schilling[1], Pascal Nasahl[1], and Stefan Mangard[12]

[1] Graz University of Technology, Graz, Austria
firstname.lastname@iaik.tugraz.at
[2] Lamarr Security Research

Abstract. With improvements in computing technology, more and more applications in the Internet-of-Things, mobile devices, or automotive area embed powerful ARM processors. These systems can be attacked by redirecting the control-flow to bypass critical pieces of code such as privilege checks or signature verifications or to perform other fault attacks on applications or security mechanisms like secure boot. Control-flow hijacks can be performed using classical software vulnerabilities, physical fault attacks, or software-induced faults. To cope with this threat and to protect the control-flow, dedicated countermeasures are needed.
Control-flow integrity (CFI) aims to be a generic solution to counteract control-flow hijacks. However, software-based CFI typically either protects against software or fault attacks, but not against both. While hardware-assisted CFI can mitigate both, they require hardware changes, which are unrealistic for existing architectures. Thus, a wide range of systems remains unprotected and vulnerable to control-flow attacks.
This work presents FIPAC, a software-based CFI scheme protecting the execution at basic block granularity against software *and* fault attacks. FIPAC exploits ARM pointer authentication of ARMv8.6-A to implement a cryptographically signed control-flow graph. We cryptographically link the correct sequence of executed basic blocks to enforce CFI at this level. We use a custom LLVM-based toolchain to automatically instrument programs. The evaluation on SPEC2017 with different security policies shows a geometric mean code overhead between 51–91 % and a runtime overhead between 19–63 %. For embedded benchmarks, we measured geometric mean runtime overheads between 49–168 %. While these overheads are higher than for countermeasures against software attacks, FIPAC outperforms related work protecting the control-flow against faults. FIPAC is an efficient solution to protect software- *and* fault-based CFI attacks on basic block level on modern ARM devices.

1 Introduction

ARM-based systems are ubiquitous as billions of devices featuring such a processor are shipped, including mobile devices, the Internet-of-Things, or electronic control units. This growing trend continues, and ARM expects to embed up to

a trillion cores over the next two decades [33]. However, those devices are attacked using control-flow hijacks, posing a severe threat. These attacks hijack the control-flow to further bypass safety- and security-critical checks, such as privilege or password verifications, but they also bypass additional countermeasures implemented in software. Multiple attack methodologies covering different attacker models have been developed, which can induce control-flow hijacks.

Classical control-flow hijacks are performed in software, exploiting a memory vulnerability to modify code-pointers or return addresses. This strategy allows an adversary to perform powerful Turing-complete attacks, such as ROP [48] or JOP [11]. These techniques have successfully been used to attack many devices, from embedded devices to secure enclaves [21, 23, 29].

When considering faults, the attack surface of control-flow hijacks increases. Faults can manipulate the control-flow at a much finer granularity. Direct branches or calls are a target of fault-based control-flow attacks, allowing an attacker to arbitrarily jump. Consequently, faults on the control-flow are used to bypass security defenses. Recent work [62] describes a NaCl sandbox exploit, where Rowhammer was used to manipulate the branch target of an indirect branch. In [51], remote code execution was crafted by inducing bitflips via the network on the program's global offset table. There are several exploits where faults are used to bypass secure boot [16, 18, 54] or escalate Linux privileges [53, 56].

To counteract control-flow attacks, control-flow integrity (CFI) aims to protect the control-flow in different threat models. CFI addressing a software attacker [1,26,32,37] protect code-pointers enforcing coarse-grained CFI. They only protect indirect control-flow transfers, *i.e.*, indirect calls or returns, but they do not offer protection against faults. Fine-grained countermeasures [2, 17, 41] protect any control-flow transfer, *i.e.*, direct and indirect calls/branches or jumps and returns, to mitigate control-flow hijacks triggered by faults. Consequently, this comprehensive protection yields larger overheads when realized in software.

Although fine-grained CFI schemes are strong countermeasures against fault attacks, they do not consider software attackers in their threat model. While there exist countermeasures protecting against both threats, they presume intrusive hardware changes [12, 58]. Those schemes require to implement a custom processor, which is unrealistic for large-scale deployment, especially on closed architectures. This leaves many applications exposed to software- or fault-based control-flow attacks. Hence, there is a need for new countermeasures that protect programs against both threats but without hardware changes.

Contribution

We present FIPAC, a software-based CFI scheme protecting the execution at basic block granularity of ARM devices against software *and* fault attacks. FIPAC's threat model considers an attacker hijacking the control-flow on basic block level, independent of the attack methodology. We address this threat model and protect the control-flow by implementing a basic block level CFI scheme, using a keyed state update resistant to memory bugs. FIPAC cryptographically links the sequence of basic blocks at compile-time and verifies the executed sequence at runtime. We exploit ARM pointer authentication of ARMv8.6-A for efficient

linking and verification. We provide an LLVM-based toolchain to protect programs without user interaction. We validate the prototype using a simulator supporting ARMv8.6-A. To evaluate the runtime performance of FIPAC, we emulate the overheads of PA instructions and run SPEC2017 and other embedded benchmarks on existing hardware. Moreover, we provide a security evaluation and discuss different security policies. Summarized, our contributions are:

- We present an efficient basic block granular CFI protection scheme for ARM-based systems protecting the control-flow against fault *and* software attacks.
- We present a prototype implementation exploiting the ARM pointer authentication of the ARMv8.6-A.
- We provide a custom open-source[3] LLVM-based toolchain to automatically instrument and protect arbitrary programs.
- We perform a functional and performance evaluation based on SPEC2017 and other embedded benchmarks and discuss different security policies.

2 Background

This section introduces fault attacks control-flow attacks and discusses CFI.

2.1 Fault Attacks

In a fault attack, the attacker influences the device's operating conditions to manipulate an inner system state. Established fault attacks require physical access [5], but new methodologies, like Rowhammer [25], Plundervolt [39], or VoltJockey [44], allow an attacker to induce faults remotely in software, increasing the severity on commodity devices. Irrespective of the methodology, the fault model defines if the fault targets data or the control-flow. When targeting data, the induced fault is mainly used to break cryptographic primitives [7, 9, 44, 50]. Counteracting these attacks requires data redundancy schemes [6, 24] capable of detecting such faults. However, data protection schemes cannot prevent hijacks of the control-flow of a program using a fault. In this threat model [40,51,53,62], the adversary arbitrarily redirects the control-flow, e.g., to sensitive code blocks.

2.2 Control-Flow Attacks

In a control-flow attack, the adversary hijacks the program's control-flow to redirect it, by using software vulnerabilities or faults. In software-triggered control-flow attacks, the adversary exploits a memory bug and overwrites code- or data pointers, used for return addresses [48], jumps [11], or data-pointers [20].

Although faults can be used to attack the same control-flow (return addresses, code- or data pointers), faults increase the attack surface. While in a software attack the adversary is limited by the exploitability of the underlying memory bug, faults allow to hijack the control-flow arbitrarily. Faults can corrupt [30] or skip instructions [8], change the program counter [40,53,54], or modify addresses used by indirect or direct calls in registers, memory, or the code segment [38, 55]. These attacks target the control-flow within (intra) or over (inter) a basic

[3] Available at `https://github.com/Fipac/Fipac`

block, *i.e.*, consecutive instructions without control-flow. While intra basic block attacks allow the attacker to skip/manipulate individual instructions in a basic block, inter basic block attacks enable the attacker to redirect the control-flow to an arbitrary code position by corrupting addresses of branches/calls.

2.3 Control-Flow Integrity

To protect a program from intra/inter basic block control-flow attacks, enforcing control-flow integrity has shown to be an effective defense [1]. Existing software-based CFI schemes provide different enforcement granularities and either address a software *or* fault attacker but not both. Although there are schemes addressing both threats, they require hardware changes, which are not feasible for commodity systems.

Software CFI Schemes. Software CFI (SCFI) [1] protects the program from a software adversary performing control-flow hijacks. The coarse-grained CFI policy only protects indirect calls or returns. CPI [14] and CCFI [37] protect a broad range of forward- and backward-edges of the program by maintaining the integrity of code-pointers. PARTS [32] protects code-pointers by signing and verifying them using ARM pointer authentication before using them. If the verification fails, *i.e.*, the pointer authentication code (PAC) does not match the expected PAC, the application stops. PACStack [31] protects return addresses on the stack by utilizing PA to cryptographically link and verify them.

Fault CFI Schemes. Fault CFI schemes (FCFI) consider an attacker performing fault attacks, thus, operating on a finer granularity. FCFI schemes capable of *detecting* intra basic block control-flow hijacks, e.g., instruction skips, employ a global CFI state, which is updated with the execution of each instruction. Maintaining and checking a state at this granularity is expensive, so these schemes require hardware changes [12, 49, 58, 59]. As this is not possible for commodity devices, software-based FCFI schemes provide a trade-off between security and performance by protecting all control-flow transitions between basic blocks, hence, providing inter basic block CFI. In CFCSS [41] and SWIFT [46], each basic block is assigned a signature to update a global CFI state.

Pseudocode 1 CFI state update function.

```
1: function UPDATE(S, Sig_BB)
2:     r_1 ← Sig_BB
3:     S ← S ⊕ r_1
```

Pseudocode 1 shows an XOR-based state update function, like in CFCSS [41], where a global CFI state S is XORed with the basic block signature Sig_{BB}. At certain program locations, checks are included, comparing the CFI state to the expected value to detect control-flow deviations. This approach of CFCSS yields a runtime overhead between 107–426 % [17]. ACFC [57] reduces the performance

penalty down to 47 % by decreasing the checking precision and thereby reducing the security guarantees. Other approaches [19,27] annotate the source code with counter increment and verification macros to detect control-flow deviations.

3 Threat Model and Attack Scenario

This section presents the threat model, shows how it bypasses existing SCFI and FCFI schemes, and then states the required properties for secure SCFI schemes.

3.1 Threat Model

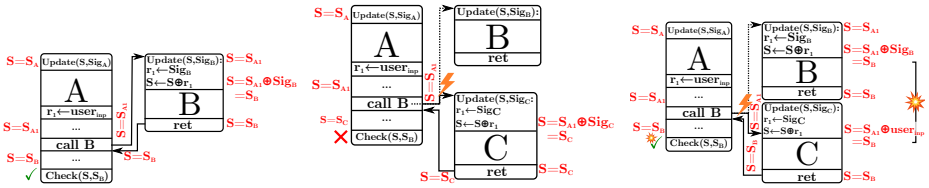
FIPAC considers an attacker performing software and fault attacks. This attacker aims to hijack direct or indirect control-flow transfers, *i.e.*, the threat model of FIPAC covers all transfers between basic blocks of the program, *i.e.*, direct, indirect, and conditional branches, direct and indirect calls, and arbitrary jumps. We consider attacks on the control-flow independently of the methodology, *i.e.*, we cover physical or software-induced fault attacks or software attacks. We expect the CFI protection to detect control-flow deviations to avoid further exploitation. The detection rather than its prevention aligns with threat models of related FCFI schemes. The attacker has binary access and can read all instructions and data. This threat model includes software attackers using this information to exploit a memory bug to conduct a control-flow hijack, e.g., manipulating code-pointers. We assume ARM pointer authentication to be cryptographically secure and that its keys are isolated from user applications.

We only consider control-flow hijacks on the CFG's edges, so we exclude attacks within a basic block, e.g., instruction skips. However, our assumed threat model aligns with several real-world exploits [10,40,62] hijacking the control-flow at these edges. Nevertheless, as security-critical code can still require stronger protection, we discuss the usage of FIPAC at instruction granularity in Section 7. DOP or faults on the data or the computation are not in the scope, including data used during a conditional branch or data used in cryptographic algorithms. To protect them, it requires orthogonal defenses, e.g., data encoding or instruction replication. For a full fault protection, a combination of both the protection of data and processing and a control-flow protection like FIPAC is required.

3.2 Attack Scenario

Bypassing SCFI. Most software CFI schemes [1,14,31,32,52,63] do not consider a faults in their threat model. As the programs' code section is immutable, SCFI schemes only protect indirect control-flow transfers but not direct calls and other branches. Hence, a targeted fault to the code segment of a program or directly within the execution, e.g., a fault on the program counter or the immediate value of a direct call, cannot be detected by SCFI.

Bypassing FCFI. The threat models of software-based FCFI schemes do not consider classical software attackers. Contrary to SCFI [1], where memory is considered to be vulnerable, typical FCFI schemes do not include this in their threat model. An attacker exploiting a memory bug can tamper the CFI state,



(a) Valid control-flow. **(b) Detectable attack.** **(c) Successful attack.**

Fig. 1: Attacker scenario to bypass FCFI.

which is maintained in software. As the state update function is known, an attacker-controlled CFI state can be crafted. Even a naïve combination of SCFI and FCFI, secure in their threat model, can be bypassed (Section 6.2) with a combined fault and software attack. To highlight the conceptional weaknesses of FCFI schemes, we demonstrate an attack bypassing FCFI with its state update function, shown in Pseudocode 1 and similarly used in many software-based FCFI schemes [41]. They compute their CFI states in software and load them into a register at some point. The goal is to exploit this instruction sequence of the state update to manipulate the CFI state to an attacker-defined value, *i.e.*, bypassing CFI.

Fig. 1 shows the attacker scenario for a control-flow hijack. Without an attacker, Fig. 1a shows a valid control-flow transfer, where `A` calls `B`. When entering `B`, the state update function updates the global state S to the beginning state S_B by XORing Sig_B to S. After returning from `B`, a CFI check verifies that S equals the pre-computed state S_B. In Fig. 1b, we consider an attacker redirecting the control-flow of the call from `B` to `C`. At the beginning of `C`, the state update XORs the current state S with the signature `C`. As this state $S = S_C$ deviates from the pre-computed state S_B, the control-flow hijack can be detected in the final check. Fig. 1c shows a successful attack on the control-flow, bypassing FCFI. The attacker controls register r_1, e.g., it is used to store user input, or it is modified due to a memory bug or fault. The adversary again redirects the control-flow from `B` to `C` but omits the signature load to r_1. Since r_1 is controlled by the attacker, who knows all states and signatures, the final state of `C` can be forged to match the end state of `B`. Eventually, the final CFI check in `A` cannot detect the control-flow hijack. Note, the control-flow redirect in Fig. 1b or 1c can either be performed with a software attack or by inducing faults.

3.3 CFI against Software and Fault Attacks

To protect the system against software *and* fault attacks and to enable large-scale usage, CFI schemes need to fulfill the following requirements:

1. The defense needs to enforce the CFI at a fine granularity, *i.e.*, at least on basic block level, to protect from a fault attacker.
2. The proper selection of the CFI state update function is essential, as it directly influences the security of the CFI scheme. Choosing a weak state update function, e.g., an XOR, allows an attacker to bypass the protection.

Furthermore, the state update function must be accumulating, meaning that the next CFI state depends on the value of the previous CFI state.

3. The protection should not require hardware changes and can be implemented in software to make the protection deployable for a wide range of devices.
4. To support legacy codebases and to enable easy deployment, the protection must be applied automatically, *i.e.*, during compilation, and must not require source code changes.

Previous fine-grained CFI protection with keyed update functions require expensive hardware changes and are not suitable for commodity devices. In [12, 58], the program is encrypted at compile-time, and the instructions are decrypted at runtime using control-flow dependent information. However, both schemes require intrusive hardware changes in the processor and are therefore inapplicable for large-scale deployment. Hence, there is a need for efficient CFI schemes considering software and fault attacks, which do not require hardware changes.

4 FIPAC

This section presents FIPAC, an efficient software-based CFI solution for ARM-based devices, fulfilling the abovementioned requirements. We first show the state-based CFI concept based on the work of Wilken and Shen [60,61] and then discuss how indirect calls are protected. Finally, we discuss the selection of the state update function and the check placement in the program.

4.1 Signature-Based Control-Flow Integrity

FIPAC is a state-based CFI protection scheme, where every basic block in the program corresponds to a well-defined CFI state. This state is maintained globally through the program execution. The CFI state is checked to match the expected state at certain program locations, indicating that no control-flow error occurred. To consider the history of the execution-flow, the next CFI state is linked with the previous one, allowing FIPAC to enforce the CFG.

Programs do not have a linear control-flow but contain control-flow transfers, such as conditional branches, loops, or calls. Depending on which program path is executed, the CFI state for a certain basic block differs since it has more than one predecessor. When the control-flow merges, *i.e.*, for conditional branches, two different paths of CFI states merge and would turn into a state collision. To avoid that, we adopt generalized path signature analysis from Wilken and Shen and insert justifying signatures for correction. Fig. 2 shows a conditional branch, where the control-flow merges in basic block `E` and a loop, which control-flow merges in `A`. At the end of `D`, there is a state patch with $\mathrm{D_p}$, ensuring the CFI state at the beginning of `E` is the same, whether coming from `C` or `D`. Furthermore, `E` jumps back to `A`, forming a loop. Thus, a patch $\mathrm{E_p}$ is inserted at the end of `E`, correcting the CFI state to $S_{A_{Begin}}$. At the end in basic block `F`, a check compares the actual state with the expected value $\mathrm{F_C}$.

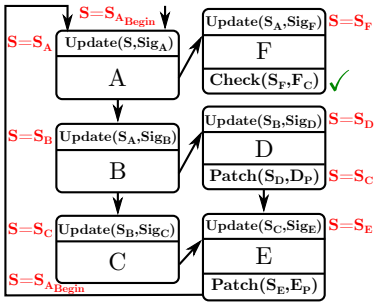


Fig. 2: Justifying signature for control-flow merges.

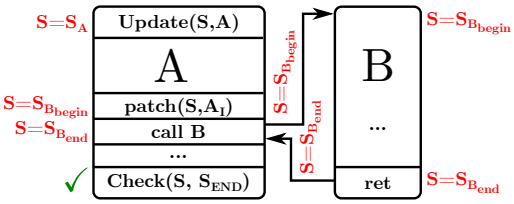


Fig. 3: CFI state patch for direct calls.

Direct Calls. In Fig. 3, function `A` directly calls `B`. To support calling `B` from multiple call sites, the beginning state of `B` always needs to be the same. Thus, we apply a justifying signature at the call site before the direct call, transforming the call site's CFI state to the beginning state of function `B`. When returning, the CFI state continues with the end state of the called function, here $S_{B_{end}}$.

Indirect Calls. Indirect calls require special handling of signatures, not covered by the work of Wilken and Shen. Determining the exact function that is being called during the indirect call is not always possible at compile-time. The best that FIPAC can do is to determine a possibly over-approximated set of potential call targets and enforce that the indirect call can only call one of them. Fig. 4 shows the patching for indirect calls and the interaction with direct calls.

To provide the CFI for indirect calls, FIPAC determines an intermediate CFI state S_I for every set of indirectly called functions. This can also lead to merging sets if the same function is called indirectly from different call sites. When performing an indirect call, the call site `A`, in ①, first patches its state S_A to an intermediate state $S_{I_{begin}}$, the same for all possible call targets of this indirect call. In ②, the indirect call is performed. At the beginning of the indirectly called function `B`, we transform the state, in ③, from the intermediate state $S_{I_{begin}}$ to the beginning state of $S_{B_{begin}}$. Furthermore, in ④, we set up the patch value used for the function return. We jump over the direct call entry in ⑤ and continue the execution of `B` until the return patch in ⑥. This patch transforms the end state $S_{B_{End}}$ of `B` to the common intermediate return state $S_{I_{End}}$ followed by a return. The caller `A` uses the pre-call signature S_A, which was saved, for a state update in ⑦, to transform the intermediate return state to a unique state for `A`. Note, the call site could simply continue with the execution using the state $S_{I_{end}}$. However, this would introduce undetectable control-flow vulnerabilities between different indirect call sites of the same function. Therefore, the patch with S_A is necessary to avoid different call sites continuing with the same signature and ensure that the function was actually called. The call site continues with the execution using the state $S_{A_{Iend}} = S_{I_{end}} \oplus S_A$, different for every call site.

Since any function must be callable with direct or indirect calls, the handling of indirect and direct calls interacts. On the right of Fig. 4, we show how `C` calls

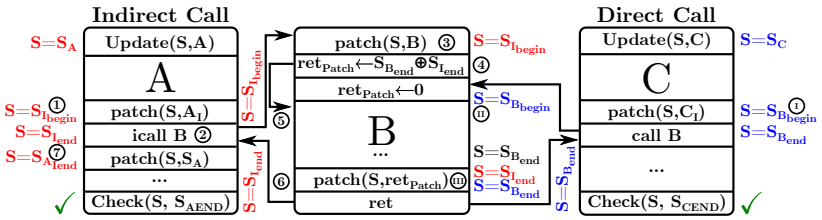


Fig. 4: CFI state patch for indirect calls.

B directly. In Ⓘ, a justifying signature is applied to transform C's CFI state to the beginning state $S_{B_{begin}}$ of B. The direct call does not jump to the beginning of B. Instead, it jumps to a dedicated entry point setting up the return patch ret_{Patch} to be zero (ⒾⒾ), and continues with the execution of B. At the end of the function in ⅢⅢ, the return patch ret_{Patch} is applied. Since the patch value is zero, this statement does not affect the state, which remains $S_{B_{end}}$. After the return, the call site then continues with the execution using the state $S_{B_{end}}$.

4.2 State Updates with Pointer Authentication

As discussed in Section 3.3, the state must not be computable by the attacker and must depend on all previous CFI states. FIPAC uses a chained cryptographic message authentication code (MAC) for the state update function to solve this problem. Thereby, we bind the security of FIPAC to a secret cryptographic key, which is unknown to the attacker and isolated by the OS. To efficiently implement such a cryptographic function, we exploit ARM pointer authentication (PA), introduced in ARMv8.3-A and updated in ARMv8.6-A [34]. It is designed to cryptographically sign pointers with a pointer authentication code (PAC) and verify their integrity before using it [45]. The PAC is computed as the MAC over the pointer and a modifier using QARMA [3]. Although pointers are 64-bit values, the size of the virtual address space limits the actual size of the pointer values. In AArch64 Linux, the virtual address space is typically configured for 39 or 48-bit [36], leaving the upper bits unused. ARM PA uses them to store the PAC value in the unused upper bit, thus having no storage overhead.

To use pointer authentication, ARMv8.6-A was extended for computing and verifying a PAC. The instructions `PACI*` and `PACD*` use the destination register as input, the source register as a modifier, and XOR the PAC in the upper bits of the destination register. The PAC can be verified by using the `AUTI*` and `AUTD*` instructions. On a successful verification, the PAC is removed from the address, and the pointer can be used. If the verification fails, `AUT*` instructions trap (this is different from ARMv8.3-A, which only sets an error bit).

This work uses the PA mechanism of ARMv8.6-A to implement the state update function rather than sign pointers. This extension fulfills the requirements needed for the state update. It uses a keyed mechanism and brings in the accumulating functionality required to link subsequent states.

4.3 Placement of Checks

Although the CFI check placement is essential for the security of the CFI scheme, there is no general solution for the correct placement. However, at minimum, there needs to be one check at the end of the program. For programs that do not return, *i.e.*, server programs, at least one CFI check in the main event loop is needed. This strategy, however, has the longest detection latency and the worst detection probability. To reduce the detection latency and improve the detection probability of CFI errors, more CFI checks are required. However, the granularity is a trade-off between overheads and security. The more checks inserted, the more overhead but also better detection probability and lower latency. At worst, a check is placed at the end of every basic block, yielding the best security but worst runtime and code performance. In between, there exist arbitrary policies with different trade-offs. For example, a generic policy places a CFI check at the end of each function. Even fully custom strategies of placing checks are possible. With the help of dynamic runtime profiling, a compiler can place the checks more efficiently. e.g., a policy can place a check after every 100^{th} basic block.

5 Implementation

FIPAC computes a rolling CFI state throughout the program's execution implemented in software on top of ARMv8.6-A without hardware changes. FIPAC exploits the PA instruction set extension to implement the cryptographic state update function. The `PACI*` and `PACD*` instructions cryptographically compute a MAC over a pointer and a modifier register and store the result in the upper bits of the pointer. In ARMv8.6-A, these instructions do not simply replace the upper bits of the pointer with the computed MAC but instead XOR them to the existing upper bits. Pseudocode 2 shows the simplified behavior of the `PACIA` instruction ignoring that the configuration bit 55 is excluded from the PAC.

Key Management. By utilizing `PACIA`, FIPAC uses the `APIAKey`, which is managed in the kernel (`EL1`) and not accessible from user mode (`EL0`) [45]. To provide CFI protection with FIPAC for the kernel, the key management can be delegated to a higher privilege level, e.g., `EL2`. As PA instructions do not differentiate privilege levels, these instructions can be used in `EL0` and `EL1`. To prevent cross-EL attacks [4], FIPAC protected user and kernel tasks can either use different keys for each privilege level (e.g., `APIAKey` for `EL0` and `APIBKey` for `EL1`), or the key manager in `EL2` could swap the keys on mode transitions. As the key needs to be known at compile-time, the prototype implementation of FIPAC statically configures the `APIAKey` in a kernel module in `EL1`. We discuss the dynamic configuration of the PA keys in Section 7.

Interrupts. FIPAC supports interrupts and OS interactions without any change. When an interrupt diverts the control-flow to the kernel, it saves all registers of the user application, including the current CFI state. The CFI state is restored after resuming from the interrupt, allowing the program to continue.

Pseudocode 2 Simplified behavior of `PACIA` in the 15-bit configuration.

```
1: function PACIA(Xd, Xm)
2:     PAC[63:0] ← ComputePAC(Xd[47:0], Xm, K)
3:     Xd[63:48] ← Xd[63:48] ⊕ PAC[63:48]
4:     Xd[47:0]  ← Xd[47:0]
```

```
adr   x2,  #4
pacia x28, x2
```

Listing 1.1: State update with `PACIA`.

```
mov  x2, #patch
eor  x28, x28, x2
```

Listing 1.2: CFI state patch.

```
mov    x2, #const
eor    x2, x28, x2
autiza x2
```

Listing 1.3: CFI check with `AUTIZA`.

5.1 CFI Primitives

We first discuss the CFI primitives and then show how they protect different control-flow instructions.

CFI State and Updates. Instead of signing a pointer with `PACIA`, we use it to compute the CFI state. The upper bits of a `PACIA` computation (the size depends on the virtual memory configuration, but we use a 15-bit configuration), the PAC bits, denote our CFI state. To accumulate the CFI state, the `PACIA` instruction is always executed on the same "pointer", in our case, the CFI state stored in `Xd`. The `PACIA, Xd, Xn` instruction computes a PAC of register `Xd` with `Xn` as a modifier and XORs it to the upper bits of `Xd`. For each basic block, a unique identifier, *i.e.*, the program counter (PC), is used as the modifier `Xm` for this instruction. By subsequently XORing the new CFI state to the previous one, we create a dependency link between succeeding basic blocks. We store the global CFI state in the exclusively reserved general-purpose register `x28`.

Listing 1.1 shows the CFI state update, placed at the beginning of each basic block. `ADR, x2, #4` first loads a unique constant for the basic block to a temporary register `x2`, in this case, the program counter. We use this constant to compute a new PAC, which gets XORed to the previous CFI state in `x28`.

State Patches. To inject a justifying signature needed for control-flow merges, we use the instruction sequence from Listing 1.2. We load an immediate constant to a temporary register in `x2`, which gets XORed to the CFI state in `x28`, thus correcting it to a target state. The computation of this immediate constant happens during the post-processing stage, as discussed in Section 5.3.

State Checks. A check compares the current CFI state with the expected state at this program location and executes an error handler on a mismatch. Such instruction sequences typically involve conditional branches, which slows down the program execution, as they impact the instruction pipeline. We also exploit the PA instructions for efficiently performing the necessary CFI checks. Similar to generating a PAC, ARM also provides `AUTI*` and `AUTD*` instructions to verify the integrity of PACs. In ARMv8.6-A, these instructions even trap on an invalid PAC verification. Since we use `PACIA` to compute a PAC, it is tempting to directly

use `AUTIZA` for verification. However, the CFI state in `x28` is not a valid PAC value in the classical sense. Instead, it is an accumulated XOR-sum of many valid PAC values combined do not form a valid PAC anymore. Thus, we cannot directly use the `AUTIZA` instruction to verify the CFI state.

At every location in the program, we know the expected CFI state at compile-time. Thus, we can compute a differential constant, which is XORed the CFI state, transforming it to a valid PAC. By applying this constant to the CFI state, we receive a valid PAC that can be verified with `AUTIZA`. This constant is determined in the post-processing tool and explained in Section 5.3. In Listing 1.3, we show the corresponding assembly sequence. We first insert an instruction sequence that patches the current CFI state to a valid PAC value using a constant for this program location. Then, we use the `AUTIZA` instruction to verify the integrity of this PAC value. On a control-flow deviation, applying the constant to the incorrect CFI state in `x28` generates an invalid PAC, which the `AUTIZA` instruction detects. If the check fails, `AUTIZA` traps and stops the program.

CFI checks can be placed arbitrarily within the program. FIPAC supports three strategies: one check at the end of a program, a check at the end of every function, or a check at the end of every basic block. The check strategy directly impacts performance and security, which is discussed in Section 6.

5.2 Protection of Control-Flow Instructions

We now discuss how the CFI primitives are used to protect different control-flow instructions. At the beginning of each basic block, we insert a PA-based CFI state update sequence. This instruction sequence uniquely updates the CFI state for the current basic block based on the previous state value.

Protection of Direct Branches, Jumps, and Conditional Branches. These control-flow instructions create control-flow merges, where state collisions occur. At control-flow merges, our compiler instruments those instructions and inserts the state patches for justifying signatures. Note, the final patch values are determined during the post-processing, as discussed in the next section. To identify the locations of patches, we compute the inverted maximum spanning tree over the edges of the CFG, defining the patch locations.

Direct Calls. Direct calls are instrumented with state patches at call site, transforming the state to the beginning state of the called function. When returning from a directly called function, the caller's CFI state continues with the callee's end state. Note, functions are instrumented to only have single return nodes.

Indirect Calls and Returns. At the call site, indirect calls are instrumented to stack the current CFI state and patch the state for the intermediate state for this set of indirect calls. When returning, the pre-call state saved on the stack is retrieved and XORed to the CFI state to provide a link over the indirect call.

Indirect calls require more complicated instrumentation besides the call site. As discussed in Section 4.1, the function header of an indirectly called function needs to set up the patch value used during the function's return. However,

```
1    mov x1 , #I_PATCH     ; Indirect call entry point
2    eor x28 , x28 , x1    ; Patch to beginning state of function
3    mov x1 , #RET_PATCH   ; Load return patch
4    b #8
5    mov x1 , #0           ; Direct Call entry point
6    ...                   ; sets up zero patch
7    eor x28 , x28 , x1    ; Apply return patch
8    ret
```

Listing 1.4: Function entry points for indirect and direct calls.

a function generally does not know how it was called and must support being called directly and indirectly. We solve this problem by adding a second function entry point, one for direct calls and the second one for indirect calls.

We add a custom function entry for indirect calls in the compiler, shown in Listing 1.4. This entry patches the intermediate state of the indirect call to the beginning state of the called function (Line 1-3). We then load the CFI update patch (Line 4), used during the function's return, and jump, in Line 5, over the direct call entry point. When the function is called directly, it jumps to the direct call function entry in Line 6, setting up a zero-patch for the return. During the function return, Line 8 uses the previously set up return patch. For direct calls, where the return patch is zero, this statement has no effect, but for indirect calls, it patches the end state to the intermediate return state. The compiler is unaware that the inserted instructions have control-flow and implement a second function entry. Thus, direct calls also use the second entry point, which is exclusively for indirect calls. We correct this during the post-processing, where all direct calls get rewritten to the second entry.

5.3 Toolchain

Our prototype toolchain uses a combination of both approaches, shown in Fig. 5. We use a custom compiler ② based on the LLVM compiler framework [28], to insert all necessary state update and patch instructions using two backend passes during the compilation of a program ①. We extend the AArch64 backend and reserve the general-purpose register `x28`, which is exclusively used to store the CFI state, disable tail calls, and ensure that functions have only a single return point. The compiler emits an instrumented ELF binary ③, but the concrete state patches and check values are set to zero. In a second step, we use a post-processing tool (PP) ④, which has access to the compiled and linked binary to compute all expected states and insert the patch updates.

The toolchain supports instrumented or non-instrumented libraries, but only instrumented libraries have CFI. Instrumented libraries must be linked statically, such that the PP tool can replace the patch and check values in the binary. The toolchain also supports inline assembly and external assembler files. However, the programmer's responsibility is to insert the necessary state update and patch sequences into the assembly code. If the assembly code is not instrumented, the code is still fully functional but does not have CFI protection. The toolchain currently supports the instrumentation of programs written in C. However, extending the support to other languages supported by LLVM, e.g., C++ or Rust, only requires more engineering work but no changes to the design of FIPAC.

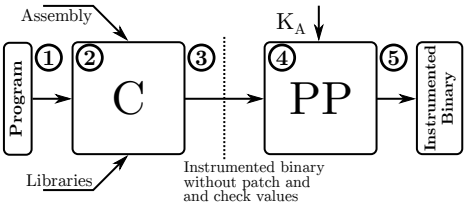


Fig. 5: Custom toolchain to build protected binaries.

Post-Processing Tool. The post-processing tool performs the call rewriting, the CFI state computation, the insertion of the patch values, and the computation of the CFI check values. It has access to the PA key and consumes the instrumented binary with zeroed patches and checks. The tool rewrites all direct calls to use the second function entry point (the first one is used for indirect calls). Next, it computes the CFI state for every location in the program. Every function is assigned a random start signature, which is propagated through all PAC-based state updates of the function. At a control-flow merge, the state values of both branches are known such that the tool can compute the justifying signature as the XOR-difference between both states and replaces the patch values `#patch`. The post-processing tool knows the CFI state at every location in the program; thus, it can also compute the XOR-differences to form a valid PAC. For `AUTIZA`-based check sequences, it replaces `#const` with the corresponding XOR-difference.

6 Evaluation

This section discusses the security guarantees of FIPAC and analyzes its overheads for different checking policies.

6.1 Security Evaluation

FIPAC considers a software and a fault attacker aiming to hijack control-flow transfers between basic blocks. To protect these control-flow transfers, FIPAC performs a state update of the global CFI state S at the beginning of every basic block allowing FIPAC to detect inter basic block manipulations.

Software Attacker Protection. A software attacker is able to hijack the control-flow by modifying indirect calls or returns by exploiting a memory bug. FIPAC mitigates these hijacks, *i.e.*, ROP or JOP, by ensuring that the executed control-flow follows the statically derived CFG. When entering a basic block, FIPAC derives a new state considering the execution history and a unique basic block identifier. On a control-flow hijack, the attacker redirects the control-flow to a basic block that is not in the set of valid targets. Hence, the state update derives a faulty state, which is detectable by the following check. If the attacker omits the update, e.g., by redirecting the control-flow to the middle of the basic block, the check before the return instruction detects the wrong state, mitigating ROP attacks. Suppose the attacker omits the state update, e.g., by redirecting the control-flow to the middle of the basic block. In that case, the check before the return detects the wrong state, mitigating ROP attacks.

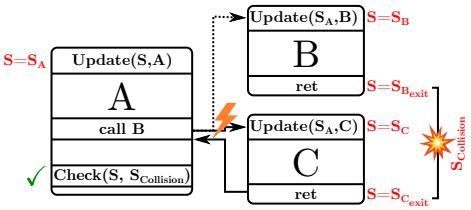


Fig. 6: Control-flow hijack from `B` to `C`. Due to a state collision, the control-flow hijack is not detected.

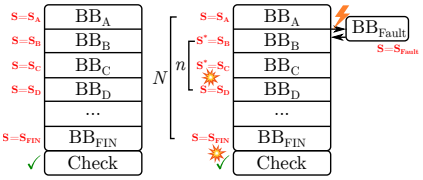


Fig. 7: A coarse-grained check policy. After n updates, a collision rectifies the faulty state.

Compared to other CFI schemes, which only consider a fault attacker, FIPAC uses a keyed state update to prevent a software or combined attacker from forging a CFI state. Equation 1 depicts the state update function ignoring the excluded bit 55 for simplification purposes. This function consists of the secret key K_A, the current state S, and a unique identifier Sig_{BB} for the basic block. The secret key K_A, inaccessible by the adversary, is initialized at boot time and ensures that the attacker cannot forge a specific state.

$$S = \text{Update}(S, Sig_{BB}, K_A) = S \oplus \text{MAC}_{K_A}\left(Sig_{BB}\right)_{PAC_{Size}} \tag{1}$$

Fault Attacker Protection. While mitigating software-triggered control-flow attacks only requires protecting a subset of control-flow transfers, *i.e.*, returns and indirect calls, thwarting a fault attacker necessitates the protection of all control-flow transfers. Hence, in addition to SCFI schemes, FIPAC also updates the CFI state for direct calls and branches, detecting any faults on addresses stored in the memory, registers, or during the execution.

Detection of a Control-Flow Violation. FIPAC does not prevent a control-flow hijack; instead, it detects an attack after the control-flow violated the CFG at the next check. This is the best that software-based CFI can do, as they cannot verify branches or calls ahead of executing them. If an attacker skips the check at the end of the basic block/function, the hijack is not detected in the first place. However, depending on the checking policy, a new check occurs at the end of the next basic block or function. Since the CFI state is invalid at this point, it requires the attacker to skip all subsequent checks such that the control-flow attack is not detectable. Control-flow attacks, which redirect the execution to the program's end, are not detectable, as there is no check anymore.

CFI State Collision Probability. Due to the truncated MAC, state collisions are possible with a probability of $P_{Coll}=\frac{1}{2^{PAC\_SIZE}}$, which can lead to a bypass. Fig. 6 illustrates a control-flow hijack, redirecting the call from `B` to `C`. When returning to the caller `A`, the state mismatch $S_{C_{exit}} \neq S_{B_{exit}}$ should be detected by the check of FIPAC. However, with probability P_{Coll}, a state collision $S_{C_{exit}}=S_{B_{exit}}=S_{Coll}$ occurs, and the control-flow attack remains undetected.

Checking Policy. To reliably detect state collisions, the sufficient placement of checks, *i.e.*, the checking policy, is crucial for the security of FIPAC. However, properly placing CFI checks is a challenging problem with no general solution. Fig. 7 shows the problem of a too coarse-grained checking policy. Left, a valid control-flow from basic block BB_A to BB_{FIN} is shown. Right, the attacker manages to redirect the control-flow to BB_{Fault} and therefore alter all subsequent states to S^*. However, with a probability of P_{Coll}, a state collision occurs after each state update. In this example, after n updates, a collision occurs, and S^* becomes S_D. Thus, the state S is valid again, and the control-flow hijack cannot be detected in further CFI checks. To give a quantitative measure on the security of the check placement, we analyze the probability of undetectable state collisions between subsequent checks. $\text{MP}_{Coll_N} = 1 - \left(1 - \frac{1}{2^{PAC\_SIZE}}\right)^N$ denotes the minimum probability that a state collision occurs in one of N state updates. After 50,000 state updates, the state collision probability reaches 78 %, and after 250,000 updates almost 100 % for a 15-bit PAC.

Selecting the checking policy is a trade-off between security and performance. Although a precise policy, *i.e.*, a check at each basic block, maximizes the detection probability of a control-flow hijack, the performance overhead also increases. While a loose checking policy, e.g., a check at the program's end, might be sufficient for small programs, programs with a high number of executed basic blocks might be vulnerable. Between these two policies, arbitrary checking strategies can be selected; for example, a check at the end of each function. A more advanced check strategy can incorporate additional information, e.g., runtime profiling. This allows the compiler to better decide where checks are needed to enforce a lower bound of the minimum detection probability of CFI errors.

A check at the end of a function is a good trade-off between runtime overhead and security. For example, SPEC2017 consists of 28391 functions. 12583 of these functions, or 44 %, contain only a single basic block with a check at the end. Thus, calling such a function is equivalent to performing a CFI state check at the call site. For example, calling this function within a loop containing no explicit checks implicitly performs a state validation at each loop iteration.

We analyzed the number of basic blocks per function for SPEC2017. The number of functions with a small number of basic blocks is much larger than functions comprising a large number of basic blocks. Almost 75 % of all functions consist of less than 13 basic blocks, which is in favor of our checking policy, since smaller functions perform a CFI check earlier than large ones. Thus, the detection probability of a state mismatch is higher. To summarize, we expect that a CFI check at the end of each function is a good trade-off for a static policy.

6.2 Security Comparison

Tab. 8 compares CFI schemes addressing software [1, 14, 31, 32, 37, 52, 63] or fault [19, 27, 41, 46, 57] adversaries with FIPAC. Software CFI schemes, like PARTS [32] or CPI [14], enforce CFI at a coarse granularity by protecting a wide range of forward- and backward edges on function level. Although these approaches mitigate software attacks (🐞) exploiting a memory vulnerability, they fail to protect against a fault attacker (ϟ). FCFI schemes enforce CFI at a

finer granularity to protect the control-flow from fault attacks, *i.e.*, on basic block or instruction level. In contrast to a software attacker exploiting memory bugs, a precise fault can tamper with direct and indirect control-flow transfers. While software-based FCFI schemes protect all control-flow transfers from faults (ϟ), they fail to protect against software adversaries (🐞). As the state update of these schemes are counters or predictable IDs, an adversary can use a memory bug to modify the state and prevent the detection of a control-flow hijack.

To protect against control-flow attacks from a fault and software attacker, it is tempting to naïvely combine existing schemes such as PARTS with FCFI, e.g., CFCSS. While these schemes are secure in their own threat model, a combined fault and software attack (💣) can bypass them. First, the adversary gains control over a register used for the FCFI state update. Then, it redirects the control-flow to a wrong function, e.g., with a fault. Finally, the tampered register is used for the state update, thus, can forge a valid CFI state.

To protect against fault and software attacks and to support a large-scale deployment, FIPAC fulfills the key requirements stated in Section 3.3. First, FIPAC comprehensively enforces CFI for transfers between basic blocks. Hence, our scheme operates on a much finer granularity than typical software CFI schemes. Second, FIPAC uses, in comparison to fault CFI schemes, a keyed state update function to mitigate attacks targeting to manipulate the global CFI state. FIPAC is implemented in software and is applied automatically during compilation.

6.3 Functional Evaluation

To evaluate the functional correctness of FIPAC, we compiled SPEC2017 [13] and Embench [42] with our LLVM-based toolchain. We executed these instrumented binaries on the QEMU 6.0 [43], which we modified to support PAC of ARMv8.6-A. In QEMU, we started the 5.4.58 Linux kernel and initialized the PA keys during the boot procedure before starting the benchmarks.

6.4 Performance Evaluation

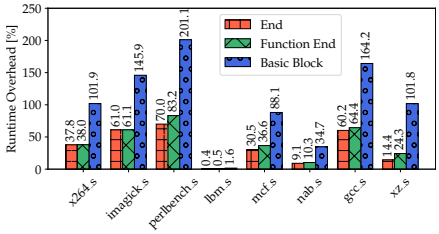
To the best of our knowledge, there is currently no publicly available device supporting ARMv8.6-A. To conduct our performance evaluation on hardware, we use the Raspberry Pi 4 Model B [15]. Since the ARM Cortex-A72 CPU is based on ARMv8-A without PAC, we emulate the runtime overhead of the PA

Fig. 8: Protection guarantees and vulnerabilities for SCFI and FCFI schemes compared to FIPAC.

	SCFI		FCFI		FIPAC
	Prot.	Vuln.	Prot.	Vuln.	
Return Addr.	🐞	💣	ϟ	🐞	✔
Indirect Calls	🐞	💣	ϟ	🐞	✔
Indirect Br.	🐞	💣	ϟ	🐞	✔
Direct Calls		ϟ	ϟ	💣	✔
Direct Br.		ϟ	ϟ	💣	✔

✔ Full 🐞 Software ϟ Fault 💣 Combined

Fig. 9: Runtime overhead for SPECspeed 2017.



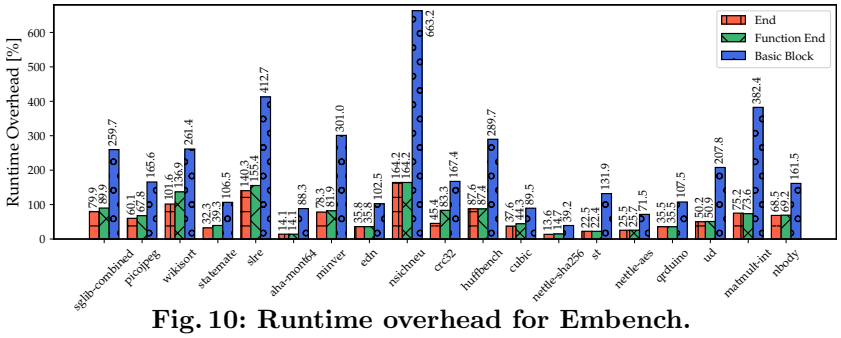


Fig. 10: Runtime overhead for Embench.

instructions in software by replacing them with their PA-analogue, *i.e.*, four consecutive XORs. PARTS [32] evaluated this sequence to model the timing of native PA instructions, which is also used in related work [31].

SPEC 2017. To measure the performance overhead of FIPAC, we compiled all C-based benchmarks with OpenMP support disabled of SPECspeed 2017 Integer. We enabled three different checking policies, from coarse-grained to fine-grained checks, to compare the performance penalty introduced by them. More concretely, we configured FIPAC to insert a CFI check at the end of the program, at the end of every function, or at each basic block. Verifying the CFI state at the end of every basic block has the largest geometric mean penalty in code size of 90.6 % as it requires 3 additional instructions per basic block. Interestingly, placing a CFI check at the end of every function only has a geometric mean overhead of 52.5 %, slightly higher than a single check at the program end with a geometric mean penalty of 50.6 %. Due to this small increase in code size but its stronger security guarantees, this policy is a good trade-off. Fig. 9 shows the runtime overhead of FIPAC compared to the baseline without protection. The coarse-grained checking policy with a single check at the program end introduces the smallest geometric mean runtime overhead of 18.8 %. The fine-grained checking policy with CFI checks at the end of every basic block has the largest geometric mean runtime penalty of 62.9 %. Interestingly, the intermediate policy with a check at the end of each function introduces a geometric mean runtime overhead of 22.1 %. This is only a small increase compared to a single check at the end, but it provides much better security. These runtime overheads are outperforming related work with overheads between 107–426 % [17].

Embench. To evaluate FIPAC on embedded workloads, we use Embench. The geometric mean code overheads are between 55–95 %, and the runtime overheads are between 49–168 % (Fig. 10), depending on the checking policy. This increased overhead is due to Embench's small codebase with a larger number of control-flow transfers compared to application-grade benchmarks like SPEC.

7 Discussion

This section discusses the hardware requirements of FIPAC, how it can be implemented on other architectures, and future improvements.

FIPAC Hardware Requirements. FIPAC requires pointer authentication from ARMv8.6-A. Although it is not yet widely available in existing processors, ARM has already announced the successor ARMv9-A [35]. Hence, we expect new designs, e.g., Apple's new processors, to feature ARMv8.6-A or even ARMv9-A.

FIPAC on ARMv8.3-A. FIPAC can also be implemented on ARMv8.3-A with the following adaptions. ARMv8.3-A PA instructions only compute a new PAC without accumulation, which must be done manually using an additional `eor` instruction per state update. This increases the overhead of an update to 3 instructions and requires one more register. `autiza` in ARMv8.3-A cannot be used as a check as it does not trap. However, ARMv8.3-A features `blraa`, a branch with link operation with pointer authentication, which traps if the jump-target contains an invalid PAC. This instruction can be misused to perform a CFI check. First, we transform the known CFI state to a valid PAC with the address of the next instruction. When executing this branch, it first verifies the target address and, if valid, jumps to the next instruction. If the PAC, and therefore also the CFI state, is invalid, the verification traps. Both solutions increase code size and runtime overheads compared to the prototype of FIPAC.

Similar to ARMv8.6-A, there is currently no open hardware available for ARMv8.3-A yet. Although Apple offers cores, such as the M1 and A14 [22], they restrict the usage of this feature. iOS applications are not allowed to load kernel modules; thus, FIPAC cannot configure the PA keys. FIPAC may run on the Apple M1 core with PA of ARMv8.3-A. However, we currently do not have access to such a device, and it requires future research to clarify if PA key access is possible in the EL1 kernel mode or if Apple restricts it.

FIPAC on Other Architectures. The design of FIPAC is generic and could also be implemented on other architectures. It is tempting to implement FIPAC on x86 with the AES-NI [47], supporting partial encryption with one instruction. However, we see limitations with this approach. First, AES-NI operates on a 128-bit state, also requiring to embed 128-bit patches. Second, one AES-NI operation only computes one round, just providing scrambling and no cryptographic strength. Third, it requires the encryption keys to be held in general-purpose registers. Thus, there is no key isolation between the user and the kernel. Hence, we do not envision FIPAC to be implemented with AES-NI.

Dynamic Key Handling. FIPAC uses a static PA key configured by the OS. However, ARM pointer authentication supports up to five keys for different domains. By using different keys, FIPAC could isolate the control-flow of the kernel and user programs. For better isolation between applications, FIPAC could embed the PA key in the binary, allowing applications to use different keys. Existing key exchange algorithms are then used to protect the embedded PA key. The OS has access to a private key for the key exchange, can read the PA key, and configure the system before starting the binary. To dynamically change the PA key, the post-processing can be integrated into the OS. Before starting the application, the OS chooses a random key and performs the post-processing

step, *i.e.*, the computation of the CFI states, patches, and check values. Thus, every invocation of the application is different in terms of FIPAC related patch and check values, which also hardens the attack surface.

Instruction Granular Protection. FIPAC does not protect the linear instruction sequence within a basic block as it only performs state updates at the beginning of a basic block. If a more fine granular protection is required, *i.e.*, intra basic block security, FIPAC supports the placement of state updates within security-critical basic blocks. For such pieces of code, the state update from Listing 1.1 is placed after every instruction to emulate instruction-granular CFI. Instruction granular CFI increases the overhead and adds two additional instructions per instruction to protect. Automatically identifying such critical pieces of code is a challenging task and not in the scope of this work. Instead, it requires the developer to manually place a check, e.g., via inline assembly.

Compatibilty. FIPAC uses the instruction address for the signature computation. When ASLR is enabled, it leads to randomized signatures not being compatible with the static computation. This problem can be solved by using static numbers to compute the signatures or by integrating dynamic key handling in the OS.

FIPAC is a software-based CFI protection scheme, and therefore, comes with certain degrees of flexibility compared to hardware-centric approaches. As FIPAC supports arbitrary checking policies on the same system, critical applications, e.g., running within a TEE or an enclave, can have a stronger checking policy than a non-critical application. FIPAC is backward compatible and supports non-instrumented applications.

8 Conclusion

We presented FIPAC, a fine-granular software-based CFI protection scheme for upcoming ARM-based hardware. FIPAC offers fine granular control-flow protection on basic block level for both fault and software attacks. The design exploits a cryptographically secure state update function, which cannot be recomputed without knowing a secret key. FIPAC utilizes ARM pointer authentication of ARMv8.6-A, to efficiently implement the keyed CFI state update and checking mechanism. We provide a toolchain to automatically instrument and protect applications. The evaluation of FIPAC with the SPEC2017 benchmark with different security policies shows a geometric mean runtime overhead between 19–63 % and is slightly larger for small embedded benchmarks. FIPAC is a software-based CFI protection, requires no hardware changes, and outperforms related work.

Acknowledgments

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. Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control-flow integrity. In: Conference on Computer and Communications Security – CCS'05 (2005), https://doi.org/10.1145/1102120.1102165
2. Alkhalifa, Z., Nair, V.S.S., Krishnamurthy, N., Abraham, J.A.: Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection. IEEE Trans. Parallel Distributed Syst. (1999), https://doi.org/10.1109/71.774911
3. Avanzi, R.: 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), https://doi.org/10.13154/tosc.v2017.i1.4-44
4. Azad, B.: Examining Pointer Authentication on the iPhone XS. https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html (2019), [accessed 2021-12-10]
5. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The Sorcerer's Apprentice Guide to Fault Attacks. Proc. IEEE (2006), https://doi.org/10.1109/JPROC.2005.862424
6. Bertoni, G., Breveglieri, L., Koren, I., Maistri, P., Piuri, V.: Error Analysis and Detection Procedures for a Hardware Implementation of the Advanced Encryption Standard. IEEE Trans. Computers (2003), https://doi.org/10.1109/TC.2003.1190590
7. Biham, E., Shamir, A.: Differential Fault Analysis of Secret Key Cryptosystems. In: Advances in Cryptology – CRYPTO'97 (1997), https://doi.org/10.1007/BFb0052259
8. Blömer, J., da Silva, R.G., Günther, P., Krämer, J., Seifert, J.: A Practical Second-Order Fault Attack against a Real-World Pairing Implementation. In: Fault Diagnosis and Tolerance in Cryptography – FDTC'14 (2014), https://doi.org/10.1109/FDTC.2014.22
9. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). In: Advances in Cryptology – EUROCRYPT'97 (1997), https://doi.org/10.1007/3-540-69053-0_4
10. Carré, S., Desjardins, M., Facon, A., Guilley, S.: Exhaustive single bit fault analysis. A use case against Mbedtls and OpenSSL's protection on ARM and Intel CPU. Microprocess. Microsystems (2019), https://doi.org/10.1016/j.micpro.2019.102860
11. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: Conference on Computer and Communications Security – CCS'10 (2010), https://doi.org/10.1145/1866307.1866370
12. de Clercq, R., Götzfried, J., Übler, D., Maene, P., Verbauwhede, I.: SOFIA: Software and control flow integrity architecture. Comput. Secur. (2017), https://doi.org/10.1016/j.cose.2017.03.013
13. Corporation, S.P.E.: SPEC CPU 2017. https://www.spec.org/cpu2017 (2019), [accessed 2021-12-10]
14. Evans, I., Fingeret, S., Gonzalez, J., Otgonbaatar, U., Tang, T., Shrobe, H.E., Sidiroglou-Douskos, S., Rinard, M., Okhravi, H.: Missing the Point(er): On the Effectiveness of Code Pointer Integrity. In: IEEE Symposium on Security and Privacy – S&P'15 (2015), https://doi.org/10.1109/SP.2015.53

15. Foundation, R.P.: Raspberry Pi 4 Model B. https://www.raspberrypi.org/products/raspberry-pi-4-model-b (2020), [accessed 2021-12-10]
16. Free60.org: Reset Glitch Hack. https://free60.org/Reset_Glitch_Hack/, [accessed 2021-12-10]
17. Goloubeva, O., Rebaudengo, M., Reorda, M.S., Violante, M.: Soft-Error Detection Using Control Flow Assertions. In: IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology – DFT'03 (2003), https://doi.org/10.1109/DFTVS.2003.1250158
18. Group, N.: There's A Hole In Your SoC: Glitching The MediaTek BootROM. https://research.nccgroup.com/2020/10/15/theres-a-hole-in-your-soc-glitching-the-mediatek-bootrom, [accessed 2021-12-10]
19. Heydemann, K., Lalande, J., Berthomé, P.: Formally verified software countermeasures for control-flow integrity of smart card C code. Comput. Secur. (2019), https://doi.org/10.1016/j.cose.2019.05.004
20. Hu, H., Shinde, S., Adrian, S., Chua, Z.L., Saxena, P., Liang, Z.: Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In: IEEE Symposium on Security and Privacy – S&P'16 (2016), https://doi.org/10.1109/SP.2016.62
21. Hund, R., Holz, T., Freiling, F.C.: Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In: USENIX Security Symposium – USENIX'09 (2009), http://www.usenix.org/events/sec09/tech/full_papers/hund.pdf
22. Inc., A.: Apple SoC security. https://support.apple.com/guide/security/apple-soc-security-sec87716a080/web, [accessed 2021-12-10]
23. Jaloyan, G., Markantonakis, K., Akram, R.N., Robin, D., Mayes, K., Naccache, D.: Return-Oriented Programming on RISC-V. In: Asia Conference on Computer and Communications Security – AsiaCCS'20 (2020), https://doi.org/10.1145/3320269.3384738
24. Joshi, N., Wu, K., Karri, R.: Concurrent Error Detection Schemes for Involution Ciphers. In: Cryptographic Hardware and Embedded Systems – CHES'04 (2004), https://doi.org/10.1007/978-3-540-28632-5_29
25. Kim, Y., Daly, R., Kim, J.S., Fallin, C., Lee, J., Lee, D., Wilkerson, C., Lai, K., Mutlu, O.: Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In: International Symposium on Computer Architecture – ISCA'14 (2014), https://doi.org/10.1109/ISCA.2014.6853210
26. Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., Song, D.: Code-Pointer Integrity. In: Operating Systems Design and Implementation – OSDI'14 (2014), https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov
27. Lalande, J., Heydemann, K., Berthomé, P.: Software Countermeasures for Control Flow Integrity of Smart Card C Codes. In: European Symposium on Research in Computer Security – ESORICS'14 (2014), https://doi.org/10.1007/978-3-319-11212-1_12
28. Lattner, C., Adve, V.S.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Symposium on Code Generation and Optimization – CGO'04 (2004), https://doi.org/10.1109/CGO.2004.1281665
29. Lee, J., Jang, J.S., Jang, Y., Kwak, N., Choi, Y., Choi, C., Kim, T., Peinado, M., Kang, B.B.: Hacking in Darkness: Return-oriented Programming against Secure Enclaves. In: USENIX Security Symposium –

USENIX'17 (2017), https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-jaehyuk
30. Liao, H., Gebotys, C.H.: Methodology for EM Fault Injection: Charge-based Fault Model. In: Design, Automation & Test in Europe – DATE'19 (2019), https://doi.org/10.23919/DATE.2019.8715150
31. Liljestrand, H., Nyman, T., Gunn, L.J., Ekberg, J., Asokan, N.: PACStack: an Authenticated Call Stack. CoRR (2019), http://arxiv.org/abs/1905.10242
32. Liljestrand, H., Nyman, T., Wang, K., Perez, C.C., Ekberg, J., Asokan, N.: PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In: USENIX Security Symposium – USENIX'19 (2019), https://www.usenix.org/conference/usenixsecurity19/presentation/liljestrand
33. Limited, A.: Inside the numbers: 100 billion ARM-based chips. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/inside-the-numbers-100-billion-arm-based-chips-1345571105, [accessed 2021-12-10]
34. Limited, A.: Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile. https://documentation-service.arm.com/static/5fa3bd1eb209f547eebd4141 (2020), [accessed 2021-12-10]
35. Limitedv, A.: 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 2021-12-10]
36. Marinas, C.: Memory Layout on AArch64 Linux. https://www.kernel.org/doc/html/latest/arm64/memory.html (2020), [accessed 2021-12-10]
37. Mashtizadeh, A.J., Bittau, A., Boneh, D., Mazières, D.: CCFI: Cryptographically Enforced Control Flow Integrity. In: Conference on Computer and Communications Security – CCS'15 (2015), https://doi.org/10.1145/2810103.2813676
38. Moro, N., Dehbaoui, A., Heydemann, K., Robisson, B., Encrenaz, E.: Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller. In: Fault Diagnosis and Tolerance in Cryptography – FDTC'13 (2013), https://doi.org/10.1109/FDTC.2013.9
39. Murdock, K., Oswald, D.F., Garcia, F.D., Bulck, J.V., Gruss, D., Piessens, F.: Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In: IEEE Symposium on Security and Privacy – S&P'20 (2020), https://doi.org/10.1109/SP40000.2020.00057
40. Nasahl, P., Timmers, N.: Attacking AUTOSAR using Software and Hardware Attacks. In: escar USA (2019)
41. Oh, N., Shirvani, P.P., McCluskey, E.J.: Control-flow checking by software signatures. IEEE Trans. Reliab. (2002), https://doi.org/10.1109/24.994926
42. Patterson, D., Bennett, J., Palmer Dabbelt, C.G., Madhusudan, G.S., Mudge, T.: Embench™: A Modern Embedded Benchmark Suite. https://www.embench.org, [accessed 2021-12-10]
43. QEMU: QEMU the FAST! processor emulator. https://www.qemu.org (2020), [accessed 2021-12-10]
44. Qiu, P., Wang, D., Lyu, Y., Qu, G.: VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies. In: Conference on Computer and Communications Security – CCS'19 (2019), https://doi.org/10.1145/3319535.3354201
45. Qualcomm Technologies, I.: Pointer Authentication on ARMv8.3. https://www.qualcomm.com/media/documents/files/

whitepaper-pointer-authentication-on-armv8-3.pdf (2017), [accessed 2021-12-10]
46. Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I.: SWIFT: Software Implemented Fault Tolerance. In: Symposium on Code Generation and Optimization – CGO'05 (2005), https://doi.org/10.1109/CGO.2005.34
47. Rott, J.K.: Intel® Advanced Encryption Standard Instructions (AES-NI). https://software.intel.com/content/www/us/en/develop/articles/intel-advanced-encryption-standard-instructions-aes-ni.html (2012), [accessed 2021-12-10
48. Shacham, H.: 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'07 (2007), https://doi.org/10.1145/1315245.1315313
49. Sugihara, M.: A Dynamic Continuous Signature Monitoring Technique for Reliable Microprocessors. IEICE Trans. Electron. (2011), https://doi.org/10.1587/transele.E94.C.477
50. Tang, A., Sethumadhavan, S., Stolfo, S.J.: CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In: USENIX Security Symposium – USENIX'17 (2017), https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang
51. Tatar, A., Konoth, R.K., Athanasopoulos, E., Giuffrida, C., Bos, H., Razavi, K.: Throwhammer: Rowhammer Attacks over the Network and Defenses. In: USENIX Annual Technical Conference – USENIX ATC'18 (2018), https://www.usenix.org/conference/atc18/presentation/tatar
52. Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., Pike, G.: Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In: USENIX Security Symposium – USENIX'14 (2014), https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice
53. Timmers, N., Mune, C.: Escalating Privileges in Linux Using Voltage Fault Injection. In: Fault Diagnosis and Tolerance in Cryptography – FDTC'17 (2017), https://doi.org/10.1109/FDTC.2017.16
54. Timmers, N., Spruyt, A., Witteman, M.: Controlling PC on ARM Using Fault Injection. In: Fault Diagnosis and Tolerance in Cryptography – FDTC'16 (2016), https://doi.org/10.1109/FDTC.2016.18
55. Trouchkine, T., Bouffard, G., Clédière, J.: Fault Injection Characterization on Modern CPUs. In: Information Security Theory and Practice – WISTP'19 (2019), https://doi.org/10.1007/978-3-030-41702-4_8
56. van der Veen, V., Fratantonio, Y., Lindorfer, M., Gruss, D., Maurice, C., Vigna, G., Bos, H., Razavi, K., Giuffrida, C.: Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In: Conference on Computer and Communications Security – CCS'16 (2016), https://doi.org/10.1145/2976749.2978406
57. Venkatasubramanian, R., Hayes, J.P., Murray, B.T.: Low-Cost On-Line Fault Detection Using Control Flow Assertions. In: International Symposium on On-Line Testing and Robust System Design – IOLTS'03 (2003), https://doi.org/10.1109/OLT.2003.1214380
58. Werner, M., Unterluggauer, T., Schaffenrath, D., Mangard, S.: Sponge-Based Control-Flow Protection for IoT Devices. In: IEEE European Symposium on Security and Privacy – EURO S&P'18 (2018), https://doi.org/10.1109/EuroSP.2018.00023
59. Werner, M., Wenger, E., Mangard, S.: Protecting the Control Flow of Embedded Processors against Fault Attacks. In: Smart Card Research and Advanced Applications – CARDIS'15 (2015), https://doi.org/10.1007/978-3-319-31271-2_10

60. Wilken, K.D., Shen, J.P.: Continuous Signature Monitoring: Efficient Concurrent-Detection of Processor Control Errors. In: International Test Conference – ITC'88 (1988), https://doi.org/10.1109/TEST.1988.207880
61. Wilken, K.D., Shen, J.P.: Continuous signature monitoring: low-cost concurrent detection of processor control errors. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. (1990), https://doi.org/10.1109/43.55193
62. Zero, G.P.: Exploiting the DRAM rowhammer bug to gain kernel privileges. https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html (2015), [accessed 2021-12-10]
63. Zhang, M., Sekar, R.: Control Flow Integrity for COTS Binaries. In: USENIX Security Symposium – USENIX'13 (2013), https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/Zhang