# SFP: Providing System Call Flow Protection against Software and Fault Attacks

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

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

Martin Unterguggenberger
martin.unterguggenberger@lamarr.at
Graz University of Technology
Lamarr Security Research
Graz, Austria

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

## Abstract

With the improvements in computing technologies, edge devices in the Internet-of-Things or the automotive area have become more complex. The enabler technology for these complex systems are powerful application core processors with operating system support, such as Linux, replacing simpler bare-metal systems. While the isolation of applications through the operating system increases the security, the interface to the kernel poses a new threat. Different attack vectors, including fault attacks and memory vulnerabilities, exploit the kernel interface to escalate privileges and take over the system.

In this work, we present SFP, a mechanism to protect the execution of system calls against software and fault attacks providing integrity to user-kernel transitions. SFP provides system call flow integrity by a two-step linking approach, which links the system call and its origin to the state of control-flow integrity. A second linking step within the kernel ensures that the right system call is executed in the kernel. Combining both linking steps ensures that only the correct system call is executed at the right location in the program and cannot be skipped. Furthermore, SFP provides dynamic CFI instrumentation and a new CFI checking policy at the edge of the kernel to verify the control-flow state of user programs before entering the kernel. We integrated SFP into FIPAC, a CFI protection scheme exploiting ARM pointer authentication. Our prototype is based on a custom LLVM-based toolchain with an instrumented runtime library combined with a custom Linux kernel to protect system calls. The evaluation of micro- and macrobenchmarks based on SPEC 2017 show an average runtime overhead of 1.9 % and 20.6 %, which is only an increase of 1.8 % over plain control-flow protection. This small impact on the performance shows the efficiency of SFP for protecting all system calls and providing integrity for the user-kernel transitions.



*HASP '22, October 1, 2022, Chicago, IL, USA*

ACM ISBN 978-1-4503-9871-8/22/10.
https://doi.org/10.1145/3569562.3569565

## CCS Concepts

• **Security and privacy → Tamper-proof and tamper-resistant designs**; **Side-channel analysis and countermeasures**.

## Keywords

System Call Flow Protection, Control-Flow Integrity, Fault Attacks.

**ACM Reference Format:**
Robert Schilling, Pascal Nasahl, Martin Unterguggenberger, and Stefan Mangard. 2022. SFP: Providing System Call Flow Protection against Software and Fault Attacks. In *Hardware and Architectural Support for Security and Privacy (HASP '22), October 1, 2022, Chicago, IL, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3569562.3569565

## 1 Introduction

Devices in the Internet-of-Things, automotive area, or industrial computers are getting more complex and powerful. While in the past, those systems used deeply embedded processing units with bare-metal applications, they now are based on powerful application-grade processors with the support for operating systems [40]. Off-the-shelf operating systems, e.g., Linux, build the foundation for complex software [10]. They isolate different programs, manage privileges, or restrict access to particular memory regions. User programs can only access kernel features via a small but well-defined interface, the system call (syscall) interface. For this reason, this interface to the kernel is a prominent target for attackers to escalate privileges and gain access to the system [48].

One way of manipulating the system call interface is control-flow hijacking, which can be conducted with different methodologies. Classical control-flow attacks performed in software exploit a memory vulnerability to modify a code-pointer or return address on the stack to redirect the execution of the program. When fault attacks are considered in the threat model, the attack surface in the kernel interface increases even more. While faults can manipulate the control-flow on a much finer granularity, e.g., they can manipulate direct branches, they can also manipulate the system call being executed. A control flow hijack can skip or change which system call gets executed, possibly with a critical security impact. Furthermore, precise faults can directly manipulate which system call gets executed by manipulating the system call register containing the system call number.

One way to counteract control-flow attacks is a generic mechanism called control-flow integrity (CFI) [1]. CFI exists at different granularities, depending on which threat model is considered. In a classical software setting, only indirect branches are protected since those are the only ones an attacker can manipulate. Faults pose a more severe threat, thus requiring even more robust protection. Fine-grained instrumentation [2, 22, 36] protects the control-flow of a program on basic-block or even instruction-level [13, 53]. As a result, these countermeasures protect direct or indirect branches or even the whole instruction sequence. Instruction-granular protection requires intrusive hardware changes to deal with the performance penalty, which is unsuitable for commodity devices.

CFI can be enforced in different security domains. While traditionally, CFI was only used to protect user-space applications, different CFI protection schemes can also protect the kernel [16, 21]. However, currently, there are no CFI protection schemes available providing protection between different security domains, *i.e.*, the transitions between the user-space program and the kernel. Thus, the large attack surface, the transitions between user programs to the kernel remain unprotected. Hence, there is a need for new countermeasures that protect the software interface to the kernel and provides system call flow integrity for commodity devices.

## Contribution

In this work, we solve the problem of the unprotected system call interface and provide system call flow protection on top of CFI, protecting the interface to the kernel against both software and fault attacks. SFP cryptographically links the system call itself and its origin to a global CFI state that is verified at runtime in the operating system. A second-stage linking mechanism within the kernel dynamically applies a second link to ensure that the correct system call was selected and executed.

To automatically protect arbitrary programs, we develop an LLVM-based toolchain to provide CFI and instrument all system calls. We provide an instrumented standard library, where all system calls are instrumented with our system call protection. Furthermore, we modify the Linux kernel to dynamically verify at runtime that the correct system call was executed.

We implement SFP on top of FIPAC, a software-based CFI scheme exploiting ARM pointer authentication. We evaluate the performance of SFP based on a microbenchmark to measure the impact of SFP on the system call latency, leading to an overhead of 1.9 %. To show the applicability to real-world programs, we perform macrobenchmarks using the SPEC 2017 application benchmark. On average, we measure a runtime overhead of 20.6 % for protected applications. Summarized, we make the following contributions:

- We provide system call flow protection by linking the syscall and its origin to a global CFI state and verifying it at runtime.
- We provide a prototype implementation comprising an LLVM-based toolchain, an instrumented C-standard library, and a modified Linux kernel.
- We evaluate the performance based on a microbenchmark and on the application-grade SPEC 2017 benchmark.

# 2 Background

This section provides background to fault attacks, pointer authentication, and control-flow integrity.

## 2.1 Fault Attacks

Injecting faults into a digital circuit is a powerful threat allowing adversaries to break the security of a system entirely. The effect of an induced fault at the electrical level includes timing violations and transient voltage and current changes [42]. Typically, the effect of a fault is modeled at the bit-level with transient bit-flips and permanent stuck-at effects [52].

Common fault injection approaches include voltage or clock glitching, laser fault injection (LFI), and electromagnetic fault injection (EMFI) [54]. While these methodologies require physical access to the device, recently, new techniques relaxing this constraint have been released [11, 32, 39, 46]. E.g., in Plundervolt [32], the attacker utilizes the dynamic voltage scaling interface of the CPU to induce faults remotely in software.

Independently of the injection technique, an attacker can exploit the effects of faults in various ways. E.g., fault attacks on encryption primitives enable the attacker to leak secret keys [4, 12, 18]. Despite dedicated attacks on encryption, fault attacks are also actively used to bypass security features, such as secure boot, on embedded systems [17, 23, 35, 37, 49]. By inducing targeted faults into the program counter of a processor, faults enable an adversary to arbitrarily hijack the control-flow of a program [34, 47, 48].

## 2.2 Control-Flow Integrity

The control-flow of a program can be hijacked using software attacks, fault attacks, or combined software-fault attacks. Therefore, various countermeasures targeting different attacker models were proposed to protect programs from these attack vectors.

*Software CFI Schemes.* Software-based control-flow attacks are typically performed by exploiting a memory vulnerability. By overwriting control-flow-related data, e.g., return addresses or function pointers, the adversary can arbitrarily manipulate the execution of the program [5, 24, 45]. To mitigate these attacks, software control-flow integrity (SCFI) schemes [15, 25, 27, 31] aim to provide pointer integrity using different mechanisms. E.g., PARTS [27] uses ARM pointer authentication (PA) to cryptographically seal and verify security-sensitive pointers to protect them while stored in memory.

*Fault CFI Schemes.* Software CFI schemes only protect control-flow transfers the adversary can also manipulate in the software threat model, *i.e.*, return addresses and function pointers. Faults also allow the attacker to tamper with static control-flow data stored in the program or even skip instructions. Therefore, fault control-flow integrity (FCFI) schemes enforce their protection at a finer granularity, e.g., at the instruction level [13, 53]. However, as these schemes usually require custom hardware changes to avoid tremendous runtime overheads, software-based FCFI schemes typically operate at the function or basic block level [36, 41]. These schemes track the execution of the program using a signature and compare this running signature with a precomputed signature during runtime.

*Software Fault CFI Schemes.* As most FCFI schemes [36, 41] do not consider a software attacker in their threat model, software attacks allow the adversary to bypass most FCFI schemes. Here, the adversary uses a memory bug to overwrite the state maintained in software and arbitrarily hijacks the control-flow. Hence, mitigating software, fault, and software-fault combined attacks require even stronger countermeasures, *i.e.*, software fault CFI (SFCFI) schemes.

## 2.3 FIPAC

FIPAC [44] is a software fault CFI (SFCFI) scheme mitigating software and fault-based control-flow attacks exploiting ARM pointer authentication. Internally, FIPAC maintains a global state through the entire program execution. When entering a basic block, *i.e.*, a block of consecutive instructions without a control-flow transfer, FIPAC cryptographically updates the state. Depending on FIPAC's configured checking policy, the value of the state is compared to the expected value determined during the compilation of the program at the end of each basic block, function, or program. On control-flow merges, *i.e.*, indirect calls, the state is updated using a justification signature to ensure that different valid control-flow paths yield an identical state. To prevent a software adversary from predicting and overwriting the state using a memory bug, a MAC is utilized for the state update. Moreover, the state update and check functions cryptographically derive and verify the running signature on program execution. FIPAC uses the pointer authentication instructions of modern ARMv8.6A architectures for the MAC computation.

*ARM Pointer Authentication.* ARM pointer authentication is a hardware feature introduced with ARMv8.3A [29] and updated in ARMv8.6A [30]. This extension provides new instructions to cryptographically sign and authenticate data. These instructions derive a message authentication code (MAC) using a secret key, a 64-bit modifier, and the value of a provided register, e.g., an address stored in a pointer. A fraction of this MAC, called the pointer authentication code (PAC), is then stored in the upper bits of the provided register. By using the authentication instructions, the authenticity of the MAC and the data in the register can then be verified.

## 2.4 Linux and the System Call Interface

Linux [50] is a monolithic kernel used in billions of devices [51] and embedded systems. To retrieve a particular service or get a specific resource, e.g., reading and writing a file, or to get dynamic memory, the user program needs to request this from the kernel, *i.e.*, via a system call. A system call changes the privilege and transfers the execution from the user-space program to the kernel of the operating system, which then grants or denies the requested service. A user-space program aiming to execute a certain system call invokes the corresponding system call wrapper routine provided by a library. This wrapper then initiates a control-flow and privilege transfer into the kernel space by using a dedicated instruction, *i.e.*, the `svc` instruction for AArch64. The system call instruction requires the system call number of the requested service and additional optional parameters as arguments.

# 3 Threat Model and Attack Scenario

Our threat model considers a powerful adversary capable of performing software attacks, fault attacks, or combined software and fault attacks. This attacker can exploit memory vulnerabilities to arbitrarily read or modify data in memory. However, we assume that the code segment of the program cannot be modified by a software adversary by, for example, exploiting memory vulnerabilities. Nevertheless, by inducing faults, the attacker can flip bits in memory, the registers, the code segment, or the instruction pipeline of the processor. We assume that the control-flow of executed programs *and* the kernel is protected using an SFCFI scheme, such as FIPAC.

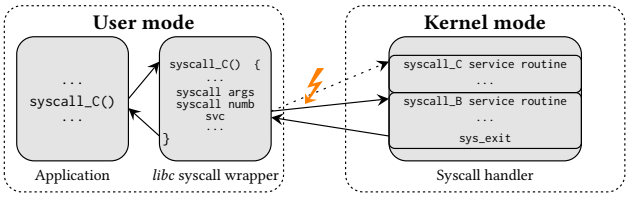


**Figure 1: Redirecting a system call using fault attacks.**

```
1 basic_block:
2   ...
3   ldr w8, memAddress  ; load data from memAddress to w8
4   ...
5   mov x0, #...        ; arguments for C system call
6   mov w8, #syscall_C  ; system call number for C
7   svc #0
```

**Listing 1: Invoking system call C on AArch64.**

Note that faults on the data, except the syscall register, are out of the scope of this work. It requires orthogonal schemes, e.g., redundancy encoding schemes for data [6], for their protection. We assume ARM PA to be cryptographically secure, and the attacker does not have access to the encryption keys. Furthermore, the operating system is assumed to be secure, providing isolation of the kernel task structure to the user program.

## 3.1 Attack Scenario

Within this threat model, the adversary aims to hijack the program's interface to the Linux kernel. In the example shown in Figure 1, the user program invokes the system call **C** using the Linux system call interface. However, by using a fault attack or a software-fault combined attack, the adversary can either *(i)* redirect the system call to **B** or *(ii)* entirely skip the system call.

Listing 1 shows the instruction sequence to invoke the system call **C** on AArch64. The system call number is stored in register `w8`, and the system call arguments are stored in the remaining registers. By flipping bits in register `w8` using faults, the adversary can redirect *(i)* the execution to a different system call.

Moreover, the syscall gadget in Listing 1 is susceptible to combined attacks. A software-fault combined attacker utilizes a memory vulnerability to overwrite data at address `memAddress`. Afterward, in Line 4, the adversary hijacks the execution of the program by flipping bits in the program counter to redirect the control-flow to the `svc` instruction in Line 7, responsible for switching to the kernel. This attack enables the adversary to invoke arbitrary system calls. In addition to these attacks, a fault attacker can also corrupt the `svc` instruction to skip *(ii)* the execution of the entire syscall.

SCFI schemes, such as FIPAC, currently *cannot* mitigate these attacks as these countermeasures do not consider transitions between user-space and kernel space in their threat model. While they only protect the user-space application, they fail to provide protection for the kernel interface, posing a large threat surface for critical vulnerabilities. Furthermore, current SCFI protection schemes use static control-flow instrumentation, which is the same for subsequent calls to the program. As a result, an attacker with access to the code segment or to general-purpose registers can learn from subsequent program executions. Thus, it would be possible

for an attacker to attempt multiple control-flow attacks until the hijack succeeds.

### 3.2 FIPAC Intra Basic Block Protection

The authors of FIPAC describe a mechanism to extend the protection guarantees of FIPAC from inter to intra-basic block security [44]. By applying a state update after every instruction within a basic block, the subsequently also update the CFI state continuously. Although this mechanism can be applied around syscalls, it does not add any protection. With a state update before and after the system call, an attacker can still fault the syscall number or manipulate the svc instruction to perform a *nop* instruction. Although this attack manipulates the execution of the system call, FIPAC's extended intra-basic protection does *not* detect these attacks. Consequently, it requires a different protection scheme to provide call flow protection for system calls.

## 4 Design of SFP

In this section, we present SFP, a mechanism that provides system call flow protection by exploiting a stateful CFI protection scheme. While SFP is generic and compatible with different CFI protection schemes, our design exploits FIPAC as the underlying CFI protection scheme. Section 7.3 discusses the compatibility aspects and how SFP can be applied to different CFI schemes.

### 4.1 Requirements for System Call Protection

The goal of SFP is to protect the system call interface to the kernel against software, fault, and combined attacks. Based on the attack scenario from Section 3, the protection of SFP must fulfill the following requirements.

- **R1** *System Call Number.* Prevent an attacker from manipulating the system call number to a different system call.
- **R2** *System Call Execution.* Ensure that a syscall cannot be skipped.
- **R3** *System Call Protection.* Ensure the system call dispatcher in the kernel executes the correct system call function.
- **R4** *Dynamic CFI Instrumentation.* Provide a dynamic CFI instrumentation to ensure protection between consecutive program executions.

### 4.2 System Call Protection

To fulfill requirements **R1** to **R3**, SFP introduces a two-step approach cryptographically linking the syscall to the state of the deployed SCFI scheme. First, at the system call caller site, we cryptographically link the system call origin and which system call we want to execute to the cryptographic CFI state. Second, at runtime, we perform a second-stage linking operation during the system call operation, confirming that the correct syscall gets executed.

*First-Stage System Call Linking.* We statically identify at compile-time which system call is getting executed for all locations in the program. To protect the system call, SFP binds the syscall to the CFI state, *i.e.*, to perform a CFI state update with the system call number. The system call number is a monotonically increasing number, thus not providing a significant Hamming distance between different system calls. A single bit-flip on the system call number changes the system call to a different one. As a result, the system call number cannot safely be used to bind it to the CFI state since faults can easily manipulate the system call to a different one.

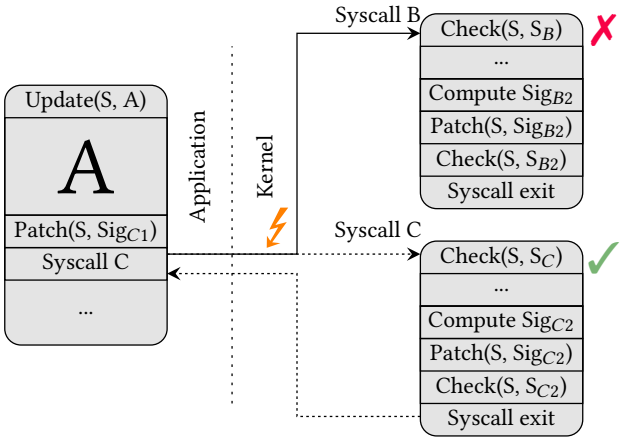


**Figure 2: System Call protection in SFP. Before a syscall, we cryptographically bind the syscall to the CFI state for later verification and second-stage linking in the kernel.**

To overcome this limitation and perform a safe and secure state update, we need to compute a system call-dependent update value with a sufficiently large Hamming distance. In SFP, we exploit the cryptographic properties of ARM PA for this purpose. We use computation of a `PACIA` operation, with the system call and a random modifier as input, and compute a cryptographic 15-bit patch value for the particular system call. Due to the cryptographic MAC operations of ARM PA, the patch values for subsequent system call numbers have a large Hamming distance and cannot be computed without having access to the secret ARM PA key. The computation of those patch values occurs at compile-time or load time and replaces the empty patch values in the binary.

Before executing a system call and jumping to the kernel, we patch the CFI state with the statically computed system call patch, thus performing the *first-stage* linking. At this point in time, we bind the future execution of the particular system call to the CFI state ahead of executing it. Performing first-stage linking already provides protection for requirements **R1** and partly **R2**.

*Second-Stage System Call Linking.* After linking the system call to the CFI state in the user-space of the program, the system call is executed, and the execution switches into the kernel. Via dispatching code and the selected system call in the general-purpose register `w8`, the kernel selects the correct system call function and executes it. At the end of each system call function, we apply a second patch, *i.e.*, the *second-stage* linking to the CFI state, confirming that the previously selected system call was really executed. This patch value is computed dynamically during the execution of the syscall. The second linking step ensures that both requirements **R2** and **R3** are fulfilled.

In Figure 2, we summarize SFP's system call protection. A user program performs the first-stage linking and patches the CFI state with a statically computed syscall patch to link the execution of a system call. The execution transitions to the kernel, which executes the desired system call function. At the end of the system call, the kernel performs the second-stage linking operation, followed by a CFI check operation. The later second-stage linking operation only

succeeds when the correct system call is linked to the CFI state. As a result, SFP's approach translates system call errors, independent of how they occur, to CFI state errors, which eventually are detected through the checking policy of the selected CFI protection scheme. Note, Figure 2 includes CFI checks at the beginning and end of the syscall to immediately detect a wrong syscall when entering the kernel and after the syscall's execution.

### 4.3 Dynamic Instrumentation

Existing SFCFI protection schemes [13, 33, 44, 53] use a static post-processing or encryption phase. A dedicated post-processing tool recovers the control-flow, computes the patch and check values, and modifies the program. The static approach with a single encryption key leads to the fact that all executions of the same program use the same CFI values, e.g., patches, updates, or checks. By observing the used CFI-related values, attackers can more easily craft valid CFI states to bypass the control-flow protection.

In SFP, we overcome this limitation by splitting up the toolchain and integrating the CFI instrumentation into the kernel. When starting a program, the ELF loader of the OS identifies a CFI instrumented program. It generates a random ARM PA encryption key and stores it in the process task structure. The ELF loader then performs the per-program call unique CFI instrumentation and computes the expected CFI state and all patch values needed to handle the control-flow. The CFI states are stored along with the process task structure within the kernel. With this mechanism, subsequent calls to the same program create different encryption keys. As a result, it guarantees that different CFI values are generated on each new program start, *i.e.*, fulfilling requirement **R4**.

*Kernel Checking Policy.* In SFP, we develop a novel CFI checking policy at the edge of the operating system. Due to dynamically instrumenting the program when starting it, the operating system exactly knows the expected CFI state for every location of the program. When a user program now enters the kernel, e.g., due to a system call instruction, the kernel, which has access to both the user program state and the expected CFI states, can verify them. If the current CFI state matches the expected state, the system call continues. However, if the CFI state deviates from the expected state, a CFI error is detected, and the operating system aborts the program execution. A CFI check at the end of the syscall confirms the execution of the right syscall. Apart from system calls, a user program can enter the operating system also via different execution paths. We include the same checking policy when a timer interrupt is raised, and the kernel is entered.

## 5 Implementation

The prototype implementation of SFP consists of two parts. First, we develop a toolchain to automatically compile and instrument arbitrary C-programs with CFI, including a custom runtime library. Second, we modify the kernel of the Linux operating system to include the system call verification, the new checking policy, and the dynamic instrumentation on the program start.

### 5.1 Toolchain

*Compiler.* We base the toolchain on the modified compiler of FIPAC [43], which is based on the LLVM [26] compiler framework. We adapt the AArch64 backend of the compiler to instrument the control-flow and embed control-flow meta information in a custom

```
1 basic_block:
2   ...
3   mov x0, #...        ; arguments for B system call
4   mov x15, #0         ; Zero system call patch
5   eor x28, x28, x15   ; Perform a CFI state update
6   mov w8, #syscall_B  ; system call number for B
7   svc #0              ; Jump to kernel
```

**Listing 2: Patched system call in the musl standard library.**

section of the ELF binary. The compiler inserts the updates for every basic block, inserts patches for control-flow merges, and also deals with call instructions. Our modified compiler emits a running ELF binary but leaves all patch values for control-flow merges and system calls to be zero. The necessary post-processing step is shifted to the operating system, which computes all patches at the program start. Note that the instrumented program does not contain any check instructions as they are part of the transition to the operating system and are performed in the kernel.

*C Standard Library.* System calls are typically invoked via wrapper functions provided by the standard library of the programming language. This prototype toolchain uses a CFI-instrumented version of the *musl* [19] C standard library. The standard library provides wrapper functions for all system calls or uses system calls directly in different library functions. We identify every system call in the musl standard library and insert the necessary patch sequence containing an immediate load and the xor-based state update ahead of executing the system call. Listing 2 summarizes the first-stage linking, where the immediate value for the `mov` instruction is zero. When starting the binary, the operating system computes the actual patch value for this system call and fills out the correct load value.

### 5.2 Kernel Support

SFP requires minor modifications to the operating system. We base the prototype of SFP on the Linux kernel in version 5.15.32 [3].

*Dynamic Instrumentation on Program Start.* On program start, when an instrumented ELF binary is started, SFP performs the per-program instrumentation of the program. First, the kernel generates a random encryption key used for the PA instrumentation. With the help of control-flow metadata, which is stored along with the ELF binary in a metadata section, we compute the CFI state throughout the program and fill the necessary patch values for justifying signatures. Furthermore, we compute the syscall- and key-dependent patch values that are used to protect the system call interface. For every system call in the program, we compute its PAC based on the system call number and user-space program unique modifier. The resulting PAC value, which is not guessable by the attacker, is filling out the immediate patch value before the syscall.

As discussed, the instrumented program does not contain dedicated CFI check operations as they are performed when entering the kernel. Instead, we store the expected CFI state for each program location in the task's kernel structure. To reduce the storage overhead, we use a `RangeMap`, to only have one entry for a contiguous range of states, where it does not change.

*System Call Verification.* During the system call, the user program updates the CFI state with a statically computed cryptographic patch value that depends on the system call number. The verification that the correct system call gets executed happens in the kernel. After the system call jumps into the kernel, a dispatcher code selects

```
1 syscall_A:
2   ...
3   mov x16, #1           ; Load kernel modifier
4   pacia x8, x16         ; Compute system call patch
5   eor x28, x28, x15     ; Perform 2nd stage linking
6   and x28, x28, #0xffffffff00000000 ; Clear syscall
7   ret                   ; number from CFI state
```

**Listing 3: Dynamically computing the system call patch and removing it from the CFI state at the system call end.**

the correct system call function to be executed. At the end of every system call function in the kernel, we perform the second-stage linking. Based on the system call number, we dynamically compute a second patch value dependent on the currently executed system call. In Listing 3 , we summarize this operation sequence, where we perform the second-stage linking within the kernel. To retrieve a cryptographically secure patch value, we exploit ARM PA's `PACIA` instruction, which takes the system call and a modifier as input operands. Note that the modifier used for the kernel update of the CFI state is different from the one used for the first-stage linking in the user program. This property is essential to avoid attackers being able to skip system calls entirely since patching the CFI state twice with the same value would cancel out and has no permanent effect on the CFI state. We finally apply the computed patch to the CFI state and clear the lower bits from the system call.

*Checking Policy at the Kernel Boundary.* Whenever a user program enters the kernel, SFP performs a CFI check to validate if the current CFI state still matches the expected state. We perform CFI checks on two entering points: During a system call and when a timer interrupt is raised. With the help of the CFI states stored in a `RangeMap` within the process structure and the knowledge of the program's current program counter, we look up the expected CFI state for the program location. If the current CFI state, stored in the register `x28` of the user program state, diverges from the expected state, a CFI error is raised, and SFP stops the program execution. For syscalls, we perform a second CFI check at the end of the syscall function in the kernel to ensure the syscall was really executed.

## 6 Evaluation

In this section, we first evaluate the security of SFP and show how it provides protection and the defined threat model. Second, we evaluate the functionality and the performance overhead of the prototype implementation.

### 6.1 Security Evaluation

We analyze the security guarantees of SFP and show how different attacks within the threat model are mitigated.

*Control-Flow Hijacks in the User-Space or Kernel.* SFP provides CFI protection for the user-space application based on the selected underlying CFI protection scheme. The prototype uses FIPAC, a basic-block granular CFI scheme, protecting all direct/indirect branches as well as direct/indirect calls. The protection domain includes the C standard library, which is fully CFI instrumented. Consequently, an attacker cannot redirect syscalls in the user-space application by redirecting the control-flow to a different wrapper function of the standard library. Control-flow attacks in the kernel are detected via the kernel internal CFI protection scheme.

*Skipping a System Call.* When skipping a system call instruction, *i.e.*, the `svc` instruction, the first-stage linking already occurred. Subsequently, the skipped system call misses the second-stage linking from the kernel, which yields a wrong CFI state, which is detectable through the CFI checking policy. However, if the entire system call instruction sequence is skipped, *i.e.*, first-stage patching and the syscall instruction are omitted, the hijack is still detectable. As both patch operations are missing on the CFI state, the state is wrong again, and a subsequent CFI check, e.g., when the program gets scheduled, detects the invalid state. In both cases, SFP transforms the skipped system call into a CFI error, which manifests itself in a wrong CFI state, which is detectable.

*Changing a System Call.* A fault on the register containing the system call number, or a combined attack, in which the attacker controls the register used to execute the system call, redirects the system call to a different one. SFP protects against both attacks. By applying the first-stage linking to the CFI state, the correct system call is already bound to its future execution. Manipulating the system call register, e.g., due to a fault or software vulnerability, leads to applying the wrong system call patch to the CFI state. When the system call is executed, the CFI state for that program differs from the expected state, and the CFI check in the kernel detects the problem and aborts the program.

To bypass a system call, the attacker only has a single chance to change the system call number and manipulate the previous system call patch to correct one for this location. However, the system call patch is protected via the secret ARM PA key, which the attacker cannot access. Guessing the PAC leads to a probability of $p = \frac{1}{2^{15}} = 0.0031\,\%$ for getting the correct patch value, where 15 is the configured PAC size of our prototype implementation. Furthermore, due to the dynamic instrumentation on the program startup, the system call patches always differ between subsequent calls of the same program. As a result, the attacker cannot learn new patch information between subsequent program calls.

### 6.2 Functional Evaluation

To validate the functional correctness of SFP, we emulate the execution on the functional simulator QEMU [38] in version 7.0.0. Since this simulator currently only supports ARM PA from ARMv8.3-A, we extend it to include ARM PA of ARMv8.6-A to support the CFI protection. The functional evaluation runs the modified Linux kernel from the prototype and can start and execute instrumented programs, where all system calls are protected. Within the kernel, the functional simulator performs the second-stage linking and a CFI check to verify the execution of the correct syscall.

To verify the functionality of the countermeasure, we emulated skipping a system call and modifying the system call number. In both cases, SFP detects the attack through the next CFI check since the CFI state became invalid and stops the program execution.

### 6.3 Performance Evaluation

At the time of evaluation, there is no publicly available system supporting ARMv8.6-A needed to run FIPAC. However, to conduct the performance evaluation and to measure the performance impact of SFP, we emulate the runtime overhead of PA instructions. Therefore, we base the performance evaluation on a Raspberry Pi

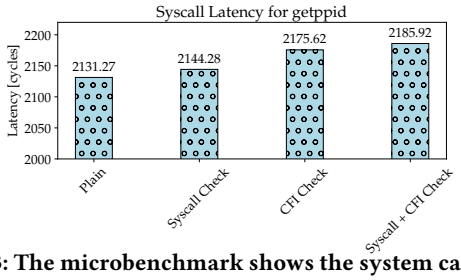


**Figure 3: The microbenchmark shows the system call latency of the `getpid` system call for different kernel configurations. SFP increases the system call latency by 1.9 %.**

4 Model B [20] with 8 GB RAM configured with a fixed CPU frequency of 1.5 GHz. The Raspberry Pi contains an ARM Cortex-A72 CPU based on ARMv8-A but without Pointer Authentication. To emulate the overhead of PA instructions, we replace them with a PA-analogue instruction sequence, *i.e.*, four consecutive XORs. Related work [27, 28] evaluated this instruction sequence to mimic the timing behavior of a PA instruction.

*Microbenchmark.* To evaluate the overhead of SFP executing system calls, we perform a simple microbenchmark. Our benchmark measures the syscall latency of the `getpid` system call, which is a side-effect-free syscall and is used in related works to benchmark the syscall execution path [7, 8]. We execute the system call 10 million times and measure the system call latency via the processor's inbuilt cycle counter. Figure 3 summarizes our evaluation results, showing the syscall latency in different kernel configurations. On the plain unmodified Linux kernel, we measure an average system call latency of 2131 cycles. When integrating the system call verification alone, the latency rises to 2144 cycles. Furthermore, with the CFI checks alone enabled, the latency increases to 2175 cycles. When both are active, we measure a system call latency of only 2185 cycles, impacting the system call latency by only 1.9 %.

*Macrobenchmark.* To demonstrate the applicability of SFP on a larger scale, we perform a macrobenchmark on real-world applications. We compiled the SPECspeed 2017 [14] benchmark with our toolchain, including only C-based programs. In Figure 4, we plot the runtime overheads in two different configurations compared to the plain uninstrumented code. First, we only include the dynamic verification, including the new CFI checking policy, that verifies the CFI state of user programs when entering the kernel. Second, we include the syscall protection based on the two-stage linking approach together with the previously evaluated CFI checking policy.

During the evaluation, we measure a geometric mean overhead of 18.8 % for the new CFI checking policy and 20.6 % with the system call protection and CFI checking policy in place. Based on the evaluation of the SPEC 2017 benchmark, we only measure a difference in the overhead of 1.8 % between the pure CFI protection and the full system call protection of SFP. This result shows that the dominating part of the overhead comes from the CFI instrumentation, not from the system call protection. Thus, reducing the overheads of the CFI protection directly influences the performance of SFP.

# 7 Discussion

This section discusses prototype limitations and shows how SFP is compatible with other CFI protection schemes.

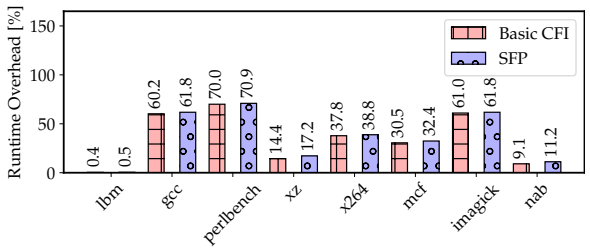


**Figure 4: Macrobenchmark shows the performance impact of SFP on the SPEC 2017 benchmark. We evaluate the impact of CFI only and SFP, including the system call protection.**

## 7.1 Dynamic System Call Instrumentation

In our prototype, we manually instrument all syscalls of the C standard library with the necessary patch instructions, consisting of a load of an immediate patch value followed by applying the patch value to the CFI state. The immediate value is zero and is set to its concrete value during the dynamic instrumentation of the startup phase of the program. In a future version of SFP, we could instrument the compiler to detect syscall instructions, *i.e.*, `svc`, and then automatically insert the necessary patch sequence. This enhancement would also include cases where syscalls are invoked manually without the wrapper functions of the standard library.

## 7.2 CFI Checking Policy Extension

SFP currently performs CFI checks when entering the kernel through a syscall or a timer interrupt. A future version of this work can extend the CFI checking policy to include all interrupts of the system. Our microbenchmark shows adding new CFI checks adds minimal overhead to the syscall latency. Thus, adding additional CFI checks for all interrupt handlers are expected to have minimal impact on the system performance.

## 7.3 Compatibility

Although SFP uses FIPAC as the underlying CFI protection scheme, the design or the protection mechanism of SFP is generic and compatible with different CFI schemes. To apply the protection of SFP to a different protection scheme, two things are required. First, the CFI protection scheme must be stateful, and there must be a possibility to manipulate the state, e.g., via standard or custom instructions, to inject the system call patch. Second, it is necessary to be able to dynamically compute a second system call patch required for the second-stage linking in the kernel. With these requirements, SFP is compatible with existing CFI protection schemes such as SCFP, SOFIA, or any other state-based CFI protection scheme.

# 8 Related Work

SCFP [53] and SOFIA [13] are hardware-assisted control-flow integrity schemes on the instruction level. They encrypt the program's instruction stream at compile-time, and perform a fine-granular decryption during runtime to retrieve the correct instruction sequence. In order to deal with the performance penalty, both protection schemes require intrusive hardware changes. This limits their applicability to small custom embedded processing cores but does not provide protection on a larger scale.

FIPAC [44] is a software-based FCFI protection scheme that exploits the architectural features of recent ARM processors. This protection scheme instruments all basic blocks of a user program

with a running CFI signature, thus providing control-flow integrity at that granularity. They present three checking policies, *i.e.*, where to check whether the running CFI signature still matches the expected one. However, FIPAC only protects the control-flow of the user-space part of the program. Although FIPAC is developed for being used with operating systems, they miss the protection of the system call interface to the kernel.

SFIP [9] implements coarse-grained syscall flow protection for user-space applications. They statically identify the possible transitions between different syscalls at compile-time and then enforce that at runtime. Since SFIP only considers software attackers in their threat model, they fail to protect against fault attacks.

## 9 Conclusion

In this work, we presented SFP, a protection mechanism that provides system call flow protection on top of ordinary CFI, protecting the interface to the kernel against both software and fault attacks. We show that an already employed CFI protection scheme can be used as a versatile tool to protect the system call interface to the kernel. Furthermore, we present a new CFI checking policy at the edge of the kernel to verify the CFI state for all transitions to the kernel. Combined with a dynamic CFI instrumentation on program startup, the attacker cannot learn CFI or system call-related information from subsequent program executions. We showed a prototype implementation comprising an LLVM-based toolchain to automatically instrument arbitrary programs and protect all system calls. A modified Linux kernel running on a Raspberry Pi evaluation setup is used to show the applicability of SFP to real-world programs. Our evaluation based on a microbenchmark and on the SPEC 2017 application benchmark shows an average runtime overhead of 20.6 %, which is only an increase of 1.8 % compared to plain CFI protection. This slight increase in the performance impact shows the effectiveness of SFP for protecting all system calls of a program.

## Acknowledgments

This work has been supported by the Austrian Research Promotion Agency (FFG) under grant number 888087 (SEIZE).

## References

[1] Abadi et al. 2005. Control-flow integrity. In *CCS'05*. https://doi.org/10.1145/1102120.1102165
[2] Alkhalifa et al. 1999. 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] Kernel Authors. 2022. Linux Kernel. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tag/?h=v5.15.32. Accessed 2022-06-11.
[4] Eli Biham and Adi Shamir. 1997. Differential Fault Analysis of Secret Key Cryptosystems. In *CRYPTO'97*. https://doi.org/10.1007/BFb0052259
[5] Bletsch et al. 2011. Jump-oriented programming: a new class of code-reuse attack. In *AsiaCCS'11*. https://doi.org/10.1145/1966913.1966919
[6] David T. Brown. 1960. Error Detecting and Correcting Binary Codes for Arithmetic Operations. *IRE Trans. Electron. Comput.* (1960). https://doi.org/10.1109/TEC.1960.5219855
[7] Davidlohr Bueso. 2019. tools/perf-bench: Add basic syscall benchmark. https://lore.kernel.org/patchwork/patch/1048777. Accessed 2022-06-01.
[8] Canella et al. 2021. Automating Seccomp Filter Generation for Linux Applications. In *CCSW@CCS '21: Proceedings of the 2021 on Cloud Computing Security Workshop, Virtual Event, Republic of Korea, 15 November 2021*. https://doi.org/10.1145/3474123.3486762
[9] Canella et al. 2022. SFIP: Coarse-Grained Syscall-Flow-Integrity Protection in Modern Systems. *CoRR* (2022). https://arxiv.org/abs/2202.13716
[10] Canonical. 2022. Ubuntu Core - The operating system optimised for IoT and Edge. https://ubuntu.com/core. Accessed 2022-06-11.
[11] Chen et al. 2021. VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface. In *USENIX'21*. https://www.usenix.org/conference/usenixsecurity21/presentation/chen-zitai
[12] Chien-Ning Chen and Sung-Ming Yen. 2003. Differential Fault Analysis on AES Key Schedule and Some Coutnermeasures. In *ACISP'03*. https://doi.org/10.1007/3-540-45067-X_11
[13] Clercq et al. 2017. SOFIA: Software and control flow integrity architecture. *Comput. Secur.* (2017). https://doi.org/10.1016/j.cose.2017.03.013
[14] Standard Performance Evaluation Corporation. 2019. SPEC CPU 2017. https://www.spec.org/cpu2017. [accessed 2022-06-01].
[15] Cowan et al. 2003. PointGuard™: Protecting Pointers from Buffer Overflow Vulnerabilities. In *USENIX'03*. https://www.usenix.org/conference/12th-usenix-security-symposium/pointguard%E2%84%A2-protecting-pointers-buffer-overflow
[16] Criswell et al. 2014. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *S&P'14*. https://doi.org/10.1109/SP.2014.26
[17] Ang Cui and Rick Housley. 2017. BADFET: Defeating Modern Secure Boot Using Second-Order Pulsed Electromagnetic Fault Injection. In *WOOT'17*. https://www.usenix.org/conference/woot17/workshop-program/presentation/cui
[18] Dobraunig et al. 2018. SIFA: Exploiting Ineffective Fault Inductions on Symmetric Cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* (2018). https://doi.org/10.13154/tches.v2018.i3.547-572
[19] Rich Felker. 2022. musl libc. https://musl.libc.org/. Accessed 2022-06-01.
[20] Raspberry Pi Foundation. 2020. Raspberry Pi 4 Model B. https://www.raspberrypi.org/products/raspberry-pi-4-model-b. Accessed 2022-06-01.
[21] Ge et al. 2016. Fine-Grained Control-Flow Integrity for Kernel Software. In *EURO S&P'16*. https://doi.org/10.1109/EuroSP.2016.24
[22] Goloubeva et al. 2003. Soft-Error Detection Using Control Flow Assertions. In *DFT'03*. https://doi.org/10.1109/DFTVS.2003.1250158
[23] Herrewegen et al. 2021. Fill your Boots: Enhanced Embedded Bootloader Exploits via Fault Injection and Binary Analysis. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* (2021). https://doi.org/10.46586/tches.v2021.i1.56-81
[24] Hu et al. 2016. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *S&P'16*. https://doi.org/10.1109/SP.2016.62
[25] Kuznetsov et al. 2014. Code-Pointer Integrity. In *OSDI'14*. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov
[26] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO'04*. https://doi.org/10.1109/CGO.2004.1281665
[27] Liljestrand et al. 2019. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *USENIX'19*. https://www.usenix.org/conference/usenixsecurity19/presentation/liljestrand
[28] Liljestrand et al. 2019. PACStack: an Authenticated Call Stack. *CoRR* (2019). http://arxiv.org/abs/1905.10242
[29] ARM Limited. 2017. Arm Architecture Reference Manual for A-profile architecture, v8.3A. https://developer.arm.com/documentation/ddi0487/ca. Accessed 2022-06-01.
[30] ARM Limited. 2020. Arm Architecture Reference Manual for A-profile architecture, v8.6A. https://developer.arm.com/documentation/ddi0487/fa. Accessed 2022-06-01.
[31] Mashtizadeh et al. 2015. CCFI: Cryptographically Enforced Control Flow Integrity. In *CCS'15*. https://doi.org/10.1145/2810103.2813676
[32] Murdock et al. 2020. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *S&P'20*. https://doi.org/10.1109/SP40000.2020.00057
[33] Nasahl et al. 2021. Protecting Indirect Branches Against Fault Attacks Using ARM Pointer Authentication. In *HOST'21*. https://doi.org/10.1109/HOST49136.2021.9702268
[34] Pascal Nasahl and Niek Timmers. 2019. Attacking AUTOSAR using Software and Hardware Attacks. In *escar USA*.
[35] Colin O'Flynn. 2020. BAM BAM!! On Reliability of EMFI for in-situ Automotive ECU Attacks. *IACR Cryptol. ePrint Arch.* (2020). https://eprint.iacr.org/2020/937
[36] Oh et al. 2002. Control-flow checking by software signatures. *IEEE Trans. Reliab.* (2002). https://doi.org/10.1109/24.994926
[37] Ramiro Pareja and Santiago Cordoba. 2018. Fault injection on automotive diagnostic protocols. *escar USA* (2018).
[38] QEMU. 2020. QEMU the FAST! processor emulator. https://www.qemu.org. [accessed 2022-06-01].
[39] Qiu et al. 2019. VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-core Frequencies. In *CCS'19*. https://doi.org/10.1145/3319535.3354201
[40] Qualcomm. 2022. IoT begins with Qualcomm. https://www.qualcomm.com/products/internet-of-things. Accessed 2022-06-11.
[41] Reis et al. 2005. SWIFT: Software Implemented Fault Tolerance. In *CGO'05*. https://doi.org/10.1109/CGO.2005.34
[42] Richter-Brockmann et al. 2021. Revisiting Fault Adversary Models - Hardware Faults in Theory and Practice. *IACR Cryptol. ePrint Arch.* (2021). https://eprint.iacr.org/2021/296
[43] Stefan Mangard Robert Schilling, Pascal Nasahl. 2022. FIPAC LLVM Project. https://github.com/Fipac/llvm-project. Accessed 2022-06-11.

[44] Schilling et al. 2022. FIPAC: Thwarting Fault- and Software-Induced Control-Flow Attacks with ARM Pointer Authentication. In *COSADE'22*. https://doi.org/10.1007/978-3-030-99766-3_5
[45] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *CCS'07*. https://doi.org/10.1145/1315245.1315313
[46] Tang et al. 2017. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In *USENIX'17*. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang
[47] Timmers et al. 2016. Controlling PC on ARM Using Fault Injection. In *FDTC'16*. https://doi.org/10.1109/FDTC.2016.18
[48] Niek Timmers and Cristofaro Mune. 2017. Escalating Privileges in Linux Using Voltage Fault Injection. In *FDTC'17*. https://doi.org/10.1109/FDTC.2017.16
[49] Niek Timmers and Albert Spruyt. 2016. Bypassing secure boot using fault injection. *Black Hat Europe* (2016).
[50] Linus Torvalds. 2022. The Linux Kernel. https://kernel.org/. Accessed 2022-06-11.
[51] Raj Vardhman. 2022. How many Linux users are there? https://findly.in/how-many-linux-users-are-there. Accessed 2022-06-11.
[52] Verbauwhede et al. 2011. The Fault Attack Jungle - A Classification Model to Guide You. In *FDTC'11*. https://doi.org/10.1109/FDTC.2011.13
[53] Werner et al. 2018. Sponge-Based Control-Flow Protection for IoT Devices. In *EURO S&P'18*. https://doi.org/10.1109/EuroSP.2018.00023
[54] Ziade et al. 2004. A Survey on Fault Injection Techniques. *Int. Arab J. Inf. Technol.* (2004). http://www.iajit.org/ABSTRACTS-2.htm#04