MAFIA: Protecting the Microarchitecture of Embedded Systems Against Fault Injection Attacks

Thomas Chamelot, Damien Couroussé, Karine Heydemann

***Abstract*—Fault injection attacks represent an effective threat to embedded systems. Recently, Laurent et al. have reported that fault injection attacks can leverage faults inside the microarchitecture. However, state-of-the-art counter-measures, hardware-only or with hardware support, do not consider the integrity of microarchitecture control signals that are the target of these faults.**

We present MAFIA, a microarchitecture protection against fault injection attacks. MAFIA ensures integrity of pipeline control signals through a signature-based mechanism, and ensures fine-grained control-flow integrity with a complete indirect branch support and code authenticity. We analyse the security properties of two different implementations with different security/overhead trade-offs: one with a CBC-MAC/Prince signature function, and another one with a CRC32. We present our implementation of MAFIA in a RISC-V processor, supported by a dedicated compiler toolchain based on LLVM/Clang. We report a hardware area overhead of 23.8 % and 6.5 % for the CBC-MAC/Prince and CRC32 respectively. The average code size and execution time overheads are 29.4 % and 18.4 % respectively for the CRC32 implementation and are 50 % and 39 % for the CBC-MAC/Prince.

***Index Terms*—fault injection attacks, code integrity, control-flow integrity, control-signal integrity, code authenticity, control logic, counter-measures**

I. Introduction

CONTEXT. Fault injection attacks are an important threat to the security of embedded systems [1]. An attacker injects physical disturbances in a circuit, such as power or clock glitches, electromagnetic pulses, or laser beams, to induce a faulty behaviour. This may result at the logical level in the alteration of several bits in different ways. State-of-the-art attackers are able to control the alteration of one or few bit values [1], [2]. The attacker aims at inducing computation errors or modifying values in the circuit under attack in order to leverage fault injection for many attack objectives such as the extraction of confidential data or privilege escalation.

State-of-the-art counter-measures against fault injection attacks ensure three security properties: data integrity, code integrity, and control-flow integrity. Data integrity ensures that data in storage, in transit or manipulated by the processor are not modified by any illegitimate means, e.g. by a fault inducing a bit-flip in a register. Code integrity ensures that instructions of the program are not modified before their execution, for example a fault inducing a bit-flip in an instruction encoding. Control-flow integrity ensures that the control-flow transfers, such as branches and calls, are correct with respect to a reference control-flow graph (CFG). A full control-flow integrity also ensures the correct execution order of branchless instructions sequences, e.g. protects against a fault inducing an instruction skip. All these properties are required to ensure the correct processing of a program.

Several works study code and control-flow integrity hardware mechanisms based on the computation of an integrity signature. In [3], a hardware monitor, external to the processor, computes a code integrity signature and uses additional metadata to validate the code and control-flow integrity in separate verification mechanisms. In [4], a single signature mechanism ensures both code and control-flow integrity. Finally, recent counter-measures for code and control-flow integrity are based on the authenticated decryption of program instructions [5]–[7]. They also ensure code confidentiality and code authenticity in addition to control-flow integrity. Note that code confidentiality prevents non-authorized entities from reading the program instructions thanks to encryption. Code authenticity ensures that the binary program is emitted by an authorized entity, and if based on sound cryptographic mechanisms, also implies code integrity.

Problem. Recently, Laurent et al. have reported that attacks can leverage faults inside the microarchitecture [8]. For example, a fault corrupting the write-back control signals after the decode stage will change the instruction behaviour. State-of-the-art code and control-flow integrity counter-measures fail to catch such fault injection attacks because the fault does not modify the binary encoding of the instruction nor the control flow. We argue that integrity of the control logic in the processor is required, together with data integrity, code integrity and control-flow integrity, to protect against fault injection attacks. We call *control-signal integrity* the security property ensuring the integrity of the control logic in the processor.

Goal & Challenges. Our goal is to design a counter-measure against fault injection attacks simultaneously supporting control-flow integrity, code authenticity, and control-signal integrity. Control-signal integrity protects the whole instruction path of the processor microarchitecture against fault injection attacks. The first challenge is to implement a control-signal integrity mechanism, that is, to protect the whole control signals in the processor microarchitecture against fault injection attacks. The second challenge is to combine control-signal integrity with a code and control-flow integrity approach that

This work was partially funded by the French National Research Agency (ANR) under grant agreement ANR-18-CE39-0003. Thomas Chamelot and Damien Couroussé are with the Univ. Grenoble Alpes, CEA, List, F-38000 Grenoble, France (e-mail: thomas.chamelot@cea.fr; damien.courousse@cea.fr). Karine Heydemann is with the Sorbonne Université, CNRS, LIP6, 75005 Paris, France and with Thales DIS France (e-mail: karine.heydemann@thalesgroup.com).

is robust against fault injection attacks. Our last challenge is to implement the counter-measure in an embedded system with complete hardware and software support while maintaining a minimum overhead.

Contributions. This paper presents MAFIA, the first counter-measure of our knowledge to ensure control-signal integrity against fault injection attacks, in combination with control-flow integrity and code authenticity.

MAFIA is designed around the concept of *pipeline state*, which is a selection of control signals representative of the current state of the processor. An integrity signature is derived from the pipeline state, and any deviation from the expected signature values can be detected, highlighting a fault injection. This approach ensures the integrity of all the control signals monitored upstream from the pipeline state. Downstream from the pipeline state, the monitored control signals are protected by a redundancy scheme, typical of counter-measures against fault injection attacks. The combination of an integrity signature derived from the pipeline state with a redundancy-based protection ensures a full protection coverage of the control signals in the processor microarchitecture.

We detail the properties of the function signature required to ensure code integrity and control-flow integrity in our attacker model. Code authenticity is also ensured when the function signature provides message authentication.

MAFIA is extended with support for indirect control-flow transfers and interrupts, which provides full support of software used in embedded systems.

MAFIA is implemented as an extension of the CV32E40P RISC-V in-order processor, and is supported by a dedicated compiler toolchain. We describe how MAFIA is integrated to the processor architecture, and we describe the modifications required for the compiler toolchain to fully support the counter-measure.

The signature function at the core of MAFIA supports many possible implementations. We evaluate two implementations with different security/overhead trade-offs: one with a CBC-MAC integrating the Prince block cipher providing code authenticity, and another one with a CRC32 error detection code providing code integrity only. Notably, the integration of MAFIA in the microarchitecture of the CV32E40P does not impact the design critical paths, allowing to maintain the target frequency of the reference ASIC implementation, at 400 MHz in the GF-22FDX FDSOI technology. We report a hardware area overhead of 23.8 % and 6.5 % for CBC-MAC/Prince and CRC32 respectively. The average code size and execution time overheads are 29.4 % and 18.4 % respectively for CRC32 and are 50 % and 39 % for CBC-MAC/Prince.

This paper is an extension of the work published in [9], in particular it presents support for indirect branches, branch prediction and interrupts. It also gives more details regarding the hardware and software implementations, and it provides an analysis of MAFIA's security.

Outline. Section II illustrates why control-signal integrity is necessary. Section III introduces our threat model and then gives some background on code and control-flow integrity. Section IV details the design of MAFIA, Section V details our implementation. Section VI provides a security analysis of MAFIA, and Section VII presents an evaluation of the resulting hardware and software overheads. Finally, Section VIII discusses related work and Section IX concludes.

```
loop:
  ff f2 82 93   addi t0, t0, #-1
  fe 04 9e e3   bne t0, zero, loop
```

Listing 1. Example of RISC-V instructions sequence implementing a loop. Binary code on the left, assembly machine instructions on the right.

```
loop:
  ff f2 82 93   addi t0, t0, #-1
  fe 04 8e e3   beq t0, zero, loop
```

Listing 2. Instructions sequence from Listing 1 with a single bit-flip applied on bit 15 of the second binary instruction (in bold face). The faulted `bne` instruction is decoded as a `beq`.

II. Motivating Example

We illustrate the necessity of protecting the control signals in the microarchitecture with control-signal integrity, and of combining this security property with code and control-flow integrity. Listing 1 is a small piece of RISC-V assembly code implementing a loop that exits when register `t0` equals `0`.

A single bit-flip applied on the binary encoding of the instructions, for example in program memory, could lead to the replacement of instruction `bne` by an instruction `beq`, as illustrated in Listing 2, leading to an inversion of the branch conditions. Counter-measures ensuring code integrity would detect such a fault [3], [4], [6], [7], [10], [11].

If a fault with a similar branch inversion effect occurs in the microarchitecture during or after instruction decoding, code integrity counter-measures fail to detect the fault because it does not modify the instruction encoding. Moreover, regarding the control flow, the fault only appears as a branch inversion and does not alter the original program CFG. Therefore the fault can only be detected by control-flow integrity counter-measures tracking the integrity of branch conditions [12].

Other faults in the microarchitecture can have harmful effects. For example in Listing 1, a single bit-flit in the control signal of the forwarding mechanism can prevent forwarding of the `addi` instruction result in register `t0` to the `bne` instruction. If such a fault is injected during the last loop iteration, the `bne` instruction uses the previous value in `t0`, leading to an additional iteration instead of exiting the loop. This is why ensuring control-signal integrity in the microarchitecture is required to ensure the correct execution of a program. Note that a fault applied before instruction decoding, i.e. into program memory or during instruction fetch, may be detected by code integrity but not by control-signal integrity. Therefore, it is necessary to ensure code, control-flow and control-signal integrity and to cover the entire instruction path.

III. Background

A. Threat Model

We consider an attacker that only has physical access to the device under attack. The attacker is supposed to use fault injection on the device. They can arbitrarily inject two

kinds of faults in the memory or in the processor logic: either a fault with full control over a few bits (typically less than 8 bits), or a fault altering many bits but without any control on the faulted value (random bit-flips). They can inject multiple faults at different time locations. Note that state-of-the-art attackers are able to selectively inject up to 4 bit faults thanks to laser illumination [2]. We consider fault injections targeting the instruction path only; faults targeting the data path are assumed to be covered by a complementary dedicated mechanism ensuring data integrity, typically, error detection code in internal data registers and data memory. Besides, the attacker does not have logical access to the device, and therefore cannot perform common software attacks, nor cannot modify the memory contents through logical access, e.g. by reprogramming it. Moreover, side-channel analysis and invasive attacks such as micro-probing are out of scope.

B. Signature-Based Code and Control-Flow Integrity

A program can be decomposed in maximal instruction sequences with a single entry instruction and a single exit instruction, commonly called basic blocks. A standard technique to ensure code integrity is to compute a runtime signature for each basic block from the binary encoding of its instructions [3], [10]. The signature S_i associated to a basic block B_i composed of instructions $I_0, \ldots, I_n$ is computed using a signature function f and an initialization vector IV_i (1). Note that fine-grained signature mechanisms are required in the context of fault injection attacks in order to detect any alteration of instructions. Hence, the signature is usually computed from the binary encoding of every machine instruction executed.

$$s_{i_0} = f(IV_i, I_0), \quad s_{i_n} = f(s_{i_{n-1}}, I_n), \quad S_i = s_{i_n} \tag{1}$$

The runtime signature is updated each time a new instruction or sequence of instructions (e.g. basic block) is processed. The runtime signature is regularly verified, for example during control-flow transfers. Verification is usually performed by checking the signature for equality with a reference value, thereafter called *reference signature*. Reference signatures are precomputed offline, they are either stored in a dedicated memory or embedded in the program memory, e.g. at the end of basic blocks.

Generalized path signature analysis (GPSA) ensures a fine-grained code and control-flow integrity by computing signatures that depend on the control-flow graph [13]. Typically, the signature of the basic block B_{i-1} is used as the initialization vector IV_i of the successor basic block B_i. Each basic block (and each instruction in a basic block) is associated with a single and distinct signature value. As a consequence, if several execution paths merge into a basic block, patch values are applied to the signature of all but one of each predecessor basic blocks $B_j, B_k, \ldots$: an update function u generates a unique initialization vector IV_i for every tuple of signatures $S_j, S_k, \ldots$ and patch values $P_j, P_k, \ldots$ (2):

$$IV_i = u(S_j, P_j) = u(S_k, P_k) = \ldots \tag{2}$$

GPSA requires that reference signatures are accessible to the signature verification mechanism. Similarly to code integrity presented above, such signatures are intertwined with program instructions, or stored in a separate memory section. Additionally, GPSA requires to instrument the program for the application of patch values.

C. Indirect Branch Integrity

Control-flow integrity (CFI) was first studied to prevent control-flow attacks on indirect branches such as ROP or JOP attacks [14], [15]. The main bottleneck lies in the precise identification of the possible targets of indirect branches. As a consequence, CFI techniques rely on some over-approximations, for example *equivalence classes*, to regroup targets reachable from the same indirect branch [16]. Equivalence classes can be defined by various means, but usually exploit some type information associated with the target functions. From a security perspective, the equivalence classes need to be as small as possible, because their size define the number of targets reachable by permitted control-flow transfers [16].

CFI techniques typically associate a unique label to each equivalence class, and an equivalence class to each indirect branch. The label is verified at runtime before the control flow transfer [17]. Similarly, in GPSA, all the basic blocks belonging to the same equivalence class are associated to the same entry signature (i.e. IV value). We call this *signature confusion*.

Other techniques protect indirect branches by replacing them with sequences of direct branches [18], which removes the need for shared labels in classical CFI approaches or signature confusion in GPSA. Note that, this approach does not prevent control-flow hijacking resulting from an alteration of the stack or of the register stroing the branch target address.

IV. MAFIA Concepts

A. MAFIA Overview

MAFIA combines GPSA with a redundancy-based mechanism to ensure control-flow integrity, code authenticity, and control-signal integrity. Fig. 1 illustrates how MAFIA would typically be integrated into a 5-stage in-order pipeline processor architecture. MAFIA is composed of two modules: The Code Authenticity and Control-Flow Integrity module (CACFI) implements the hardware support for GPSA and ensures control-signal integrity up to the decode stage. The Control Signal Integrity module (CSI) completes the coverage of control-signal integrity through a redundancy-based mechanism. The two modules run in parallel with the pipeline stages and therefore do not modify the information flow within the pipeline. On the software side, MAFIA requires modifications of the compiler backend to insert GPSA signature verifications and patch values.

Instead of using binary encoding of program instructions to compute a signature, CACFI uses signals coming from the decode pipeline stage, called the *pipeline state*. CSI checks that signals from the pipeline state are correctly propagated up to their consumption in the subsequent pipeline stages. The selected signals are duplicated into CSI at the output of the decode stage. Then, for each subsequent pipeline stage, CSI checks the original control signals against their duplicates. Therefore, the CSI module can detect any fault on control

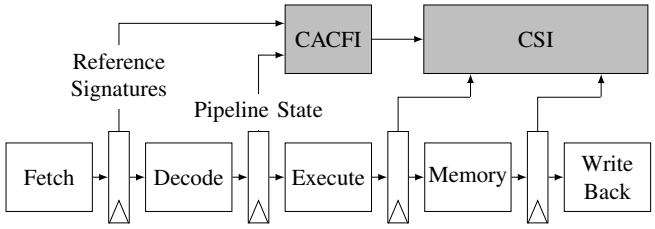

Fig. 1. Illustration of a 5-stage processor extended with MAFIA (CACFI and CSI modules, in grey)

signals included in the pipeline state after the decode stage up to the pipeline end. Control-signal integrity of the whole instruction path is ensured by the combination of the CACFI and CSI modules: CACFI ensures the integrity of the pipeline state, and CSI then ensures the integrity of control signals up to their consumption stage.

We argue that the design of a single module dealing with the control signals in all the pipeline stages, instead of two separate modules as presented in our approach, would be increasingly more complex, if not impossible. Indeed, many dynamic events (e.g. stalls due to memory latencies or jumps) may make the computation of reference signatures and the design of the hardware module more complex. Our decomposition into two coordinated modules avoids such complexity: the control signals selected in the decode stage are not impacted by the execution of instructions in later stages. Moreover, it allows different implementations of the two modules as they are independent.

B. Pipeline State

The pipeline state is a bit vector composed of control signals coming from the decode stage. To ensure that each instruction is associated with a single signature independently of the previously executed instructions, each instruction must also be associated with a unique pipeline state value. We call this property *pipeline state uniqueness*. In order to ensure code integrity, the pipeline state must include the control signals that deterministically result from the decoding of binary instructions. Also, GPSA requires that the reference signature is computed ahead of program execution (i.e. by static analysis), which implies that the value of the signals monitored by the signature (and hence included in the pipeline state) can also be computed ahead of program execution. We discuss below which control signals can be included in the pipeline state.

There are two kinds of control signals: the static ones and the dynamic ones. The static control signals only depend on the instruction currently in the decode stage. These signals can be integrated in the pipeline state, since their value can be computed from the only knowledge of the related instruction. For example, the signals for selecting the source and destination operands are directly linked to the binary encoding of instructions. The binary encoding also contains opcode fields which control the operation to perform in the execute and memory stages. To ensure full code integrity, the pipeline state can in addition include the contents of any immediate field in instruction encodings.

The dynamic control signals depend on processed data or on other processed instructions. Data-dependent control signals, such as branch decision, cannot be integrated into the pipeline state because their values cannot be statically computed. Dynamic control signals that depend on other instructions in flight in the pipeline can be integrated to the pipeline state under certain conditions. In the context of the processor architectures targeted by our counter-measure, that is, simple in-order processors targeting embedded systems, this restricts to forwarding control signals. A forwarding mechanism enables to bypass the write-back stage when there is a data dependency between two instructions. The computation of the forwarding signal is implementation-dependant, but without loss of generality we assume that forwarding is computed in the decode stage, and hence that its control signals can be integrated to the pipeline state. Note that forwarding control signals that are computed after the decode stage can be protected by the CSI module. Figure 2 illustrates cases where forwarding is involved. Figure 2a illustrates a basic block where the forwarding is enabled between the two successive `add` instructions. The sequence of instructions is invariable (program-dependant), the forwarding control signal can be statically determined, and hence can be safely integrated into the pipeline state. In Figure 2b, forwarding is enabled in the transition $\texttt{B}_1 \rightarrow \texttt{B}_2$ between the `mov` and `add` instructions, but is disabled in the transition $\texttt{B}_2 \rightarrow \texttt{B}_2$ between the `bneq` and `add` instructions. This case illustrates that forwarding may be involved at the transitions between basic blocks. As a consequence, the value of the forwarding control signal cannot be statically computed. In such case, the forwarding dependency must be broken to ensure the pipeline state uniqueness property, for example by the insertion of additional instructions (Section V-D1). Such modification is not required when the forwarding mechanism is placed after the decode stage as its control signals cannot be included in the pipeline state.

C. CACFI – Code Authenticity and Control-Flow Integrity

The CACFI module implements the hardware support for GPSA. It requires two functions, for the signature computation and for the application of patch values, with specific properties summarized in this section. Cf. Werner et al. [4] for a detailed discussion. Note that most cryptographic functions intrinsically support all these properties.

1) The signature function: The signature function f is the core of GPSA. In CACFI, f computes the runtime signature from the pipeline state and the previous runtime signature. The runtime signature is stored in the *signature register* within CACFI. The signature register should not be directly accessible from any instruction to limit the attack surface on CACFI. The GPSA fault detection capabilities depend on f's properties.

1) *Collision resistance*: prevents an attacker from forging a faulted basic block presenting the same signature as the signature of the original basic block (also known as second-preimage resistance). This property also prevents the attacker from reverting the signature to a valid value after the introduction of one or many faults.
2) *Error preservation*: alterations of signature values are not cancelled by any following fault-free sequence. This

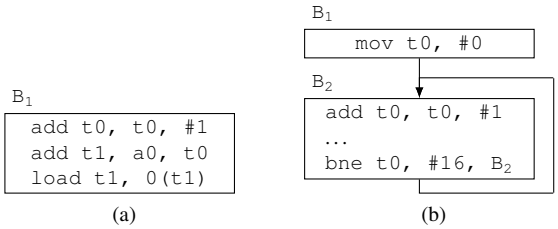

Fig. 2. Illustration of forwarding: intra (left) and inter basic block (right)

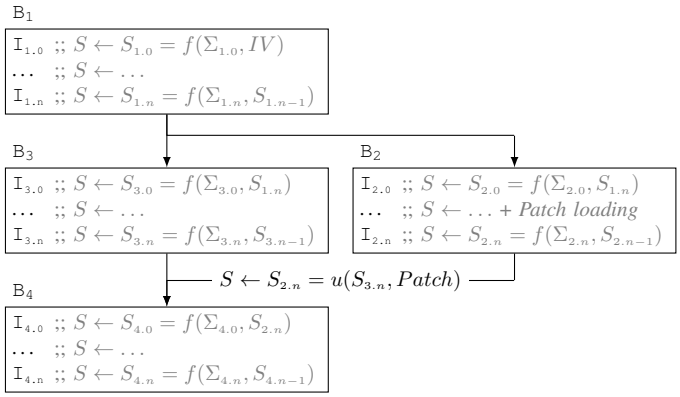

Fig. 3. Illustration of the application of GPSA to a small program sequence of 4 basic blocks. The application of a patch is required in basic block B_2 because of the merging of two execution paths.

property, in combination with collision resistance, allows for the arbitrary placement of signature verifications.

3) *Non associativity*: sequences of instructions with different orderings produce different signatures. This property ensures control-flow integrity at the granularity of machine instructions.
4) *Invertibility*: is introduced by [4] to compute patch values. However, it is not required by our approach because patch values are applied on the basic block signature instead of being applied on intermediate signatures (see below).

Note that many function signatures can support the properties listed above, which allows for many implementation trade-offs. For example, key-based cryptographic signature functions such as Message Authentication Codes (MAC) are good candidates. MACs require a secret key, which prevents the generation of valid reference signatures without knowledge of the secret key. Using such function signatures, MAFIA ensures code authenticity in addition to code integrity. In Section V, we present two implementations, one using CBC-MAC with the Prince block cipher and another one, only supporting code integrity and not code authenticity, using an error detection code.

2) The update function: GPSA requires the application of an update function u before merging execution paths. This function must support the following properties:

1) *Full control*: given a signature, there exists a patch value for any target IV.
2) *Error preservation*: any fault previously introduced in the signature cannot be reverted by applying an error-free update.
3) *Invertibility*: a patch value can be computed from an initialization vector and a signature.

The update function is triggered at each control-flow transfer (i.e. taken branch, call and return). The runtime signature in the signature register is updated using function u and the current patch value. The patch value is stored in a *patch register* in CACFI, and can be updated by a dedicated instruction that loads a patch value from memory. Additionally, the patch register is reset to a default, constant patch value after the processing of each control-flow instruction (taken or not). This default patch value must be known at compile time to compute the reference signatures, and the identity element of u, if it exists, can be used as the default patch value.

When several basic blocks $B_i, B_k, \ldots$ have the same successor B_s, there is at most a single basic block B_f falling into B_s (i.e. the basic block immediately preceding B_s in the memory layout). If B_f exists, its signature S_f is used as the initialization vector IV_s of B_s: $IV_s = S_f$. Otherwise, IV_s is chosen randomly among the signatures of $B_i, B_k, \ldots$. Knowing IV_s and u^{-1}, a patch value is computed for all the other predecessors of B_s. Fig. 3 shows a simple example of a CFG that requires an update on one of its edges. The instructions in basic blocks $\mathtt{B}_1$, $\mathtt{B}_2$ and $\mathtt{B}_3$ do not require an update because they all have only one predecessor. The instruction $\mathtt{I}_{4.0}$ has two predecessors, $\mathtt{I}_{2.n}$ and $\mathtt{I}_{3.n}$, and therefore requires an update. $\mathtt{B}_3$ falls into $\mathtt{B}_4$ which means that if $\mathtt{I}_{3.n}$ is a branch, then it is not taken on this execution path. Therefore, it is not possible to have an update on the execution path $\mathtt{B}_3$–$\mathtt{B}_4$. This is why the update is applied in $\mathtt{B}_2$ during the control transfer to the taken branch, i.e. $\mathtt{B}_4$.

3) Signature verification: A runtime signature is computed for every instruction in the program. Thanks to the properties of the functions f and u, any fault captured in the signature will be forwarded into the next ones (cf. IV-C1). Therefore, it is possible to insert verifications anywhere in the program.

MAFIA uses custom control-flow transfer instructions, thereafter called *verification instructions*, which have the same semantics as their original counterpart. Verification instructions load a reference signature immediately following in the program memory, and trigger the signature verification. Then, they proceed similarly to other control-flow instructions: if the branch is taken, the runtime signature is updated with the current patch value. Finally, the current patch value is reset to its default value.

The substitution of a control-flow instruction by a verification instruction impacts code size, as a reference signature is inserted after each verification instruction, and potentially execution time if the delay due to the loading of the reference signature is not masked. When the verification fails, it triggers an exception that calls a software user-designed fault handler.

Thanks to the use of verification instructions as control-flow instructions, our approach provides great flexibility in the insertion of signature verifications, which allows to fine-tune the trade-off between the detection delay and the overheads due to code size and execution time. Similarly to GPSA, it is possible to use a single verification instruction at the exit point of a secured function to minimize the performance

overheads without reducing the detection coverage of the counter-measure. We discuss the security impact of such trade-offs in Section VI.

D. CSI – Control Signal Integrity

The CSI module ensures control-signal integrity for the pipeline stages following the decode stage. The principle is to use a redundancy scheme to detect any change in the control signals constituting the pipeline state, from their emission to their consumption stage. This approach is lightweight because it involves only a small part of the pipeline's control logic. The CSI module duplicates the propagation of selected signals between the different stages in the pipeline. In each pipeline stage, the duplicated signals are checked against the original ones. The duplication can use any redundancy scheme, potentially with several duplicates, e.g. a simple copy, a complementary copy or the initial value `xor`ed with an arbitrary value.

E. Indirect Control-Flow Handling

In this section we focus on the protection of indirect function calls and function returns. Other indirect branches can be removed using a compiler option. We discuss this in Section V-D.

MAFIA uses equivalence classes derived from function prototypes to identify indirect call targets. Equivalence classes regroup functions with identical function prototypes (return type, number of arguments and type of each argument).

MAFIA combines GPSA with indirect call elimination to remove signature confusion, hence reducing the attack surface. Each indirect function call is replaced by a *dispatcher* (illustrated in Listing 4), that is, a sequence of direct branch instructions that forwards the control flow to the target function. With this approach, each function remains associated with a unique IV even if the function is the target of indirect branches.

For function returns, which are also indirect branches, MAFIA uses signature confusion only, although it would be possible to use dispatchers. All the basic blocks that follow the calls to a given function belong to the same equivalence class, and hence share the same IV. When a function has several exit points, MAFIA assumes a constant signature value at each of the exit points. As a consequence, patch values are applied to all but one of the exit points. Note that without indirect call elimination, all the function sharing the same indirect call site would also share the same signature at their exits, hence increasing the attack surface. To increase the protection level of function returns, it is possible to extend MAFIA with a shadow stack.

F. Branch Prediction

When the pipeline implements branch prediction, the control flow might roll back to the previous branch in case of a misprediction. The pipeline then flushes the speculatively executed instructions and the execution resumes at the correct address. In order to be compatible with branch prediction, CACFI saves the signature register after a branch in order to support

```c
void bar();
void baz();

void foo(void (*fptr)()) {
  /* fptr is either assigned to &bar or to &baz */
  fptr();
}
```

Listing 3. Example of indirect function call in language C

```
foo:
  call        dispatcher_EC0_a0_0
  ret

dispatcher_EC0_a0_0:
  push        ra
  push        s11
dispatcher_EC0_a0_0_bar:
  li          s11, bar
  bne         s11, a0, dispatcher_EC0_a0_0_baz
  load_patch  PATCH_bar
  call        bar
  load_patch  PATCH_ret_dispatcher
  jmp         dispatcher_EC0_a0_0_ret
dispatcher_EC0_a0_0_baz:
  li          s11, baz
  bne         s11, a0, error_handler
  load_patch  PATCH_target1
  call        baz
dispatcher_EC0_a0_0_ret:
  pop         ra
  pop         s11
  ret
```

Listing 4. Protection of the source-code example from Figure 3 with MAFIA (in RISC-V pseudo assembly). The indirect function call is replaced by a dispatcher. In the original code of function `foo`, register `a0` stores the branch target address.

signature roll back. On misprediction, the signature register is restored to the saved signature. When a branch is predicted not taken, the update function is applied to the signature register before saving it so that in case of misprediction the restored signature is the one that would have been computed if the branch had been taken.

After a misprediction, the instructions in the pipeline are invalidated but may impact the value of the dynamic control signals of the next pipeline state, which may impact the pipeline state uniqueness (Section IV-B). Since MAFIA already requires breaking dependencies such as forwarding dependency at basic block transitions, branch prediction does not add more constraints to the CACFI module. In Section V-D1, we propose to break forwarding dependency at basic block boundaries using a dedicated compiler pass.

To protect speculatively executed instructions and invalidated instructions, the CSI module should also cover all the control signals related to branch prediction.

In conclusion, branch prediction can be supported by MAFIA with a negligible increase of the complexity of the CACFI and CSI modules.

Note that the branch prediction mechanism itself remains sensitive to some fault injection attacks. MAFIA is not able to detect a fault targeting the misprediction control signal that would change the branch decision. To protect against such case, it would be necessary to ensure data integrity or/and to duplicate the misprediction control signal. Any other fault

affecting the control flow is detected by MAFIA.

G. Interrupts Handling and Protection

Interrupts can occur at any time during program execution and hence interrupt handlers cannot be associated with a set of predecessor instructions. As a consequence, a dedicated mechanism is required to handle interrupts and to protect interrupt handlers. MAFIA is designed to fully protect the execution of interrupt handlers and to increase the difficulty to leverage interrupts in an attack scenario.

Each interrupt handler is associated to a different IV, and all the IVs are stored in a table similar to the interrupt vector table. Upon triggering of an interrupt, the signature register is saved in a dedicated register, called the *context register*. The CACFI module selects the IV corresponding to the triggered interrupt to reset the signature register, and the processor starts the execution of the interrupt handler. A verification instruction can be placed at the end of the interrupt handler to ensure its integrity. Similarly to other sequences of code, verification instructions can be added inside the interrupt handler if needed to reduce the delay between verifications. When the interrupt handler returns, the signature register is restored to the value saved in the context register.

After the interrupt handler has returned, the last instructions of the interrupt handler are still in the pipeline. This might impact the pipeline state uniqueness the same way as forwarding dependency between basic blocks (Section IV-B). To avoid this, MAFIA delays interrupt processing until the end of a basic block, since forwarding dependencies are already broken at basic block transitions.

In our design, the signature register is not saved in memory, which prevents attacks on the saved signature, e.g. during memory transactions. This helps reduce the attack surface of interrupts, for a negligible hardware overhead, and allows for the use of dedicated protections on the context register if need be. To support nested interrupts, a *context stack*, internal to the processor, can be used in place of the context register.

V. Implementation

We integrate MAFIA to the CV32E40P processor [19], a 32-bit, in-order, 4-stage RISC-V core implementing the RV32I base instruction set version 2.1. We select the CV32E40P because such a small in-order core is representative of typical fault injection targets, and because a 4-stage pipeline is representative of the main challenges of microarchitectural design due to control and data hazards such as forwarding mechanisms. The integration to more complex processors is left for future work.

A. Pipeline State

We manually select the control signals to integrate to the pipeline state. The pipeline state consists of 64 bits composed as follows:

1) All the non-redundant control signals internal to the decode stage that are involved in the operand selection and the forwarding mechanisms: 23 bits from the operand selection multiplexers; 4 bits from the operand forwarding multiplexers.
2) All the control signals produced by the decode stage and transmitted to the next stages and that deterministically result from the decoding of the instruction opcode: 7 bits to control the arithmetic and logic unit; 2 bits to control the read and write enable of the load store unit; 10 bits to control the registers to write in the write-back stage.
3) All the signals derived from the immediate data fields: 10 bits from the binary instruction's immediate fields. Note that RISC-V ISA supports up to 20 bits immediate, but the remaining immediate fields overlap with the operand selection fields that are already included in the pipeline sate.
4) 8 bits of padding to fill the 64 bits of the pipeline state.

The control signals outputted by the decode stage and that will go through subsequent stages are duplicated in the CSI module. The remaining ones are directly used after the decode stage and do not go through more than one other stage. We use a simple duplication scheme to implement the CSI redundancy.

B. Signature and Update Functions

MAFIA is implemented with two different single cycle signature functions for the CACFI module:

- a CBC-MAC based on a fully unrolled hardware implementation of the Prince block cipher, which is selected for its small silicon area and for its capability to deliver output within one CPU clock cycle. Prince is a symmetric cipher using 64-bit blocks and a 128-bit key, and the CBC-MAC therefore generates 64-bit tags. In order to limit the code size and runtime overheads, the CACFI signature is composed of the 32 lowest significant bits of the CBC-MAC output tag. We discuss the security impact of this design choice in Section VI.
- a CRC32 designed to detect up to 8 bit-flips per basic block. Because CRC32 functions do not use any secret to compute the signature, MAFIA only ensures code integrity and not code authenticity in this case.

We select the exclusive or (XOR) for the update function in the two implementations.

C. MAFIA RISC-V ISA Extension

The CV32E40P is modified as follows. All the control-flow instructions update the runtime signature with the current patch value if the branch is taken (Section IV-C2). The core is also extended with custom instructions: we implement a verification instruction (Section IV-C3) for each control-flow instruction in the RV32I instruction set (`MAFIA.beq`, `MAFIA.bne`, `MAFIA.blt`, `MAFIA.bge`, `MAFIA.bltu`, `MAFIA.bgeu`, `MAFIA.jal`, `MAFIA.jalr`). We also add a load patch instruction (`MAFIA.ldp`) that fetches a patch value from memory, in the `.patches` section; the base address is stored in a new Control Status Register (CSR). This CSR is set during the core bootstrap to point to the memory section of the binary program that gathers all the patches. The patch value offset in the memory section is encoded as a 20-bit

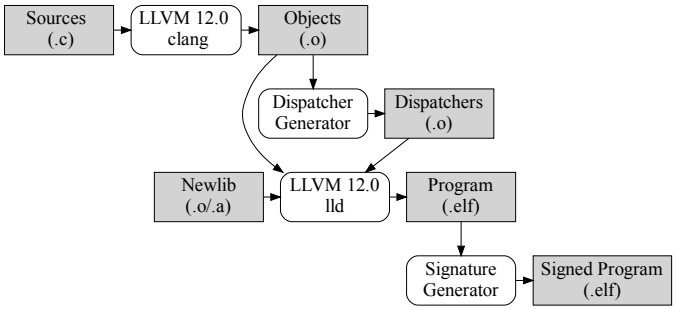

Fig. 4. MAFIA compiler toolchain with the files in light gray and the tools in rounded boxes

immediate value in the `MAFIA.ldp` instruction, which limits the number of patch values accessible to 2^{20}. If patch values are aligned on 4-byte boundaries in memory, it is possible to fix to 0 the two least significant bits of the offset, and increase the number of accessible patch values to 2^{22}.

Note that it is not desirable to store patch values in the immediate fields of instruction encoding, because this would introduce a circular dependency between the signature values and the patch values. A solution to avoid this dependency is to remove the immediate fields from the pipeline state for `MAFIA.ldp` instructions, hence increasing the complexity of the decoding logic. Furthermore, locating patch values in a dedicated section offers the possibility to store the values in a separate, secured memory.

D. Software Support

Fig. 4 presents the complete MAFIA compilation process. We select LLVM version 12.0 to build MAFIA compiler toolchain. We extend the RISC-V backend with several passes to apply the code modification required by MAFIA.

1) Forwarding dependency elimination pass: This pass ensures the pipeline state uniqueness property at basic block transitions (Section IV-B), which is mandatory to support basic blocks with multiple predecessors, branch prediction and interrupts. For the first instructions of a basic block, this pass ensures that the forwarding mechanism is deactivated for all the predecessors and if need be inserts a `nop` instruction to break the forwarding dependency. The insertion of a single instruction is sufficient for the 4-stage CV32E40P core, but longer instruction sequences may be required for more complex pipeline architectures.

2) Dispatcher pass: This pass replaces each indirect call by a direct call to a dispatcher (Section IV-E). The code for each dispatcher is generated separately by the Dispatcher Generator before the linking process.

Indirect branches that do not represent function calls are eliminated using the `-fno-jump-table` option.

3) Patch placement pass: This pass performs a control-flow analysis to insert the `MAFIA.ldp` instructions. As described in Section IV-C2, MAFIA requires a patch for all but one predecessor for each basic block. A `MAFIA.ldp` is also inserted before each call instruction and all but one functions returns. Loops require a dedicated analysis regarding the update function to prevent circular dependencies during the reference signature computation. The simple rule of placing a `MAFIA.ldp` in all but one predecessor can fail to break such a circular dependency. In this case, the pass adds a `MAFIA.ldp` in one of the loop's basic blocks. Note that this pass disables tail call optimization only when a function has multiple exit points so that the caller function and the tail-called function do not share the same signature.

The patch values and offsets in the `.patches` section are computed later by the Signature Generator tool.

4) Reference signature placement pass: The third pass identifies all the functions annotated with the dedicated attribute, `MAFIA_secured`, and replaces the control-flow instructions by the MAFIA equivalent ones that trigger the signature verification. This pass inserts a signature placeholder after each branch that performs a signature verification.

5) Dispatcher Generator: Before the link process, this tool builds the dispatchers by leveraging debug information from the `clang` compiler. The Dispatcher Generator computes the equivalence classes through a context-insensitive analysis from the type information of the target function prototypes (Section IV-E). Each equivalence class is associated with one dispatcher for each register storing a target address in an original function call (e.g. register `a0` in Figure 4). From the perspective of an attacker, this design choice increases the difficulty to fault the target function address because the address can be stored in different registers, depending on the register allocation policy used by the compiler.

6) Instrumentation of the Newlib C-library: We use LLVM infrastructure to link the sources object files with the dispatchers and the Newlib C-library. The C-library is not protected with verification instruction, but it is instrumented with signature updates (`MAFIA.ldp` instructions) so that the signatures are correctly propagated through the library. Yet, extending the C-library with signature verifications only requires minimum changes by adding the `MAFIA_secured` attribute to the desired function prototypes.

7) Signature Generator: Post link, the Signature Generator extracts the CFG by static analysis. Then, it computes the reference signatures and patch values. The computation is done by exploring the CFG recursively basic block per basic block. Each basic block is processed through a stateful signal-accurate model of the CV32E40P's decode stage to extract the pipeline state and derive the signature. When a `MAFIA.ldp` is present in a basic block, the Signature Generator attributes it a unique offset in the `.patches` section. The associated patch value is computed from the basic block signature and its successor signature. Finally, the Signature Generator creates a new ELF file with the reference signature placeholder and the `MAFIA.ldp` offset filled and the additional `.patches` section containing the patch values.

VI. Security Analysis

This section presents a security analysis of MAFIA regarding our threat model. MAFIA is designed to ensure code authenticity or code integrity, control-flow integrity and control-signal integrity. The data integrity property is not supported by MAFIA and is supposed to be ensured by a complementary dedicated mechanism.

A. Pipeline State Verification

The pipeline state is the cornerstone of MAFIA as the control signals included in the pipeline state feed both the CACFI and CSI modules. The construction of the pipeline state determines the capability of MAFIA to ensure code integrity and control-signal integrity. As presented in Section V, in our implementation, we build the pipeline state through a manual analysis of the control signals emitted by the decode stage. Such analysis and the resulting implementation are prone to errors, which could lead to vulnerabilities.

1) Verification Workflow: We perform a formal verification of our implementation of the pipeline state and of the control-signal integrity property with the workflow of Tollec et al. [20], which targets the formal vulnerability analysis against fault injection attacks from both a hardware (RTL) and a software (binary code) description. The workflow works as follows. The processor implementation, in SystemVerilog, is translated using the Yosys tool [21] into a formal model in the Satisfiability Modulo Theory Language (SMT-LIB), which represents at the RTL level the combinatorial and sequential logic of the CPU, memories, and peripherals. This translation preserves a complete correspondence between RTL signals and SMT variables. The binary program is expressed, in the Yosys SMTC constraints language, as constraints applied to the SMT model of the processor memories (typically, the RAM). The fault model, specified by the user in SMTC, defines: the target SMT variables, the effect of fault injection, the maximum number of injections, and the timing constraints (CPU cycles where fault injection is possible). The fault model is automatically instantiated as controllers applied to all the target SMT variables. The verification engine performs Bounded Model Checking (BMC), leveraging the Yosys-SMTBMC tool for BMC and Yices 2 for satisfiability queries. During verification, the BMC engine drives the fault controllers, and verification targets a property φ specified by the user. When $\neg\varphi$ is satisfiable (i.e. φ does not hold), the workflow generates a VCD trace as a counter-example.

2) Verification Use Case and Verified Properties: We verify the security of MAFIA running a VerifyPIN program under fault injection. VerifyPIN is an authentication procedure where an input (user) PIN code is compared to a secret (card) code. This small program includes control flow (conditional and unconditional branches, function calls), memory accesses, etc., and is similar to the `memcmp` procedure widely used in software. Hence, it is representative of the many ways to leverage fault injection in an attack of the authentication procedure, e.g. bypassing authentication, bypassing the PIN comparison, modifying the status value returned, altering the computation of PIN comparison. Data integrity is not verified (although it can be) because it is not included in our threat model. The formal verification targets $\neg\varphi := \psi \wedge \neg\phi_1 \wedge \neg\phi_2$, where properties ψ, ϕ_i are informally described as follows:

- ψ Authentication succeeds with different user and card PIN codes.
- ϕ_1 Fault injection leads to an alteration of the pipeline state.
- ϕ_2 Fault injection is detected by the CSI module.

If ϕ_1 holds, we know that the CACFI module ensures code and control-signal integrity, since any alteration of the pipeline state is captured by the signature function (Section VI-B). If ϕ_2 holds, we know that the CSI module ensures control-signal integrity for the pipeline stages after decode.

The verification considers a single fault (mono- or multi-bit) applied to any control signal of size smaller or equal to 8 bits, at each of the CPU cycles in a 60-cycles window overlapping with the full execution of the VerifyPIN procedure. Note that, in our target implementation, the control signals of size larger than 8 bits don't need to be considered since they drive unused features, e.g. multiplier, performance monitoring module.

3) Verification Results: The verification fails to find any fault leading to an invalidation of property φ. Furthermore, MAFIA eliminates all the vulnerabilities identified by the same workflow on the original (unprotected) CV32E40P processor running the VerifyPIN procedure. Note that, due to the error preservation property of the signature function, multiple fault injections that do not exceed the attacker capabilities in our threat model (e.g. 8 cumulative bit flips for the CRC32 implementation) are also detected by the CACFI module. Multiple fault injections may not be captured by our implementation of the CSI module, but can be supported by other redundancy schemes for a negligible increase in overhead (Section IV-D).

This brings confidence in the pipeline state construction and in MAFIA's capacity to protect the execution of an application against fault injections in the microarchitecture.

B. Signature Functions

The CBC-MAC/Prince signature function uses a secret key, which prevents an attacker from inverting the signature function to identify collision values. Furthermore, CBC-MAC with Prince provides strong resistance to collision attacks because in our threat model the attacker cannot control the whole contents of a basic block. In our implementation, only 32 bits of the 64-bit signature are verified, which reduces the probability to find a collision to $1/2^{16}$ because of the birthday paradox. Yet, a successful collision attack is unlikely because the complexity of this attack combines with the complexity of fault injection. Last, the known weaknesses of CBC-MAC, such as message forgery for variable length messages or variable initialization vectors, are not relevant here because our threat model assumes that it is not possible to modify the memory contents except by the use of fault injection.

The CRC32 signature function protects against a weaker attacker model, because it is designed to ensure code and control-flow integrity only. Moreover, CRC functions are invertible, meaning that an attacker could identify the fault to inject to create a signature collision. Yet, CRC functions are designed so that collisions are possible only above a fixed amount of bit-flips. Therefore, instead of relying on direct collision resistance, CRC functions rely on the detection capabilities to increase the fault injection complexity.

For CRC32, We determine the best candidate function according to our security model. To do so, we search, in a list of polynomials known for providing good detection capabilities [22], the polynomial that requires the highest minimal

number of bit-flips to create a collision in the signature. The search tests exhaustively all the basic block lengths up to 40 instructions, and all the collision vectors with a Hamming Weight value smaller than 11. The best generator polynomial identified is `0xFA567D89`, which detects up to 8 bit-flips in the input sequence. Thus, in order to create a collision in the signature, an attacker has to control precisely the alteration of at least 8 bits in the pipeline state. Note that such collision can be obtained in one or several fault injections. It can also be obtained indirectly by faults targeting the update function (including the patch value) or the runtime signature value, since these values are in the end combined with subsequent values of the pipeline state. However, such fault targets do not reduce the security level of the candidate CRC32 function.

C. Signature Verification

A possible attack is the case where a fault triggers a jump outside the program sections instrumented with signature verifications. This attack is equivalent to the case where several faults target all the subsequent verifications after a first fault. In such case, the attack is undetected by the counter-measure, and the security level of MAFIA is determined by the time intervals between verifications. As the substitution of control-flow instructions by verification instructions is performed at compile time, it is possible to determine the maximum delay between successive verifications, or to constrain it by inserting extra verification instructions (i.e. direct branch instructions jumping to the instruction following in program memory). A watchdog could then ensure that this maximum delay is never reached, and so detect an attacker jumping outside the program section instrumented with signature verification.

D. Control Signal Integrity

The CSI module covers all the control signals transmitted from the decode stage to the next stages. An attack targeting the pipeline stages after the decode stage can be effective if it simultaneously faults the original signal and its duplicate in the CSI module. If such an attack is relevant, it can be mitigated by using redundancy schemes with better detection capabilities, and we believe that the implementation of such schemes will have a negligible impact on the hardware area overhead, since the number of control signals monitored by CSI is low. Additionally, the comparison result is encoded as a single bit connected to the exception mechanism. A single fault could then prevent the detection propagation and the software handler triggering. A typical protection is the use of specific encoding (e.g. differential encoding) for the connection to the exception mechanism.

E. Control-Flow Integrity

MAFIA ensures a static CFI policy, which ensures that: i) for indirect branches, the target address is part of the identified equivalence class; ii) for returns, the target address follows a valid call site of the current function for return. MAFIA thus considerably reduces the number of reachable addresses that would not be detected. Thanks to the combination of CFI with code authenticity and control-signal integrity, the replacement of a control-flow target address by a valid address is restricted to data corruption, which is outside our threat model.

TABLE I
ANALYSIS OF THE BENCHES INCLUDING INDIRECT BRANCHES: NUMBER OF DISPATCHER FUNCTIONS, NUMBER AND SIZES OF CLASSES, AND NUMBER OF NON-LEGITIMATE FUNCTIONS IN EACH CLASS.

Bench	Nb. dispatchers	Nb. eq. classes	Classes size	Nb. non-leg. fun. in classes
picojpeg (`Os`)	1	1	[1]	[0]
picojpeg (`O2`)	2	1	[1]	[0]
sglib-combined (`Os`)	2	2	[1, 3]	[1, 3]
sglib-combined (`O2`)	1	1	[3]	[3]
wikisort (`Os`)	9	2	[1, 9]	[0, 0]
wikisort (`O2`)	3	2	[1, 9]	[0, 0]

In the following, we proceed with an in-depth security analysis of our design, considering the possibility of attacks outside our threat model. Regarding the protection of indirect branches, the security level is impacted by the precision of the analysis of indirect branch targets, e.g. the size of equivalence classes (Section III-C). Table I reports an analysis of the evaluated benches with indirect branches (Section VII), showing the number of dispatchers inserted, the number of equivalence classes, the size of each equivalence class, and the number of non-legitimate functions per class. The number of dispatchers depends on the number of indirect call sites in the original bench. The classes size reports the number of reachable functions in each equivalence class. In the evaluated benches, the number of equivalence classes does not exceed 2 and the largest class is limited to 9 elements, which fairly restricts the possibility of attacks. Note that several dispatchers can correspond to the same equivalence class (but using different registers for the target branch address): for `wikisort` and for `sglib-combined -Os` there are more dispatchers than equivalence classes. Finally, non-legitimate functions are functions that are not reachable from a given call site. In an equivalence class, the non-legitimate functions correspond to functions unreachable from an indirect call site but that share the same property (e.g. prototype) with the other functions. Out of 3 benches, `sglib-combined` is the only one to present equivalence classes that include non-legitimate functions, and furthermore each class is composed of non-legitimate functions only. Here, the context-insensitive analysis performed by the Dispatcher Generator is not able to detect that there is a function pointer set to `NULL` in the source code, resulting in one or two useless dispatchers depending on the optimization level. Such issue could be avoided by manually selecting the dispatchers to include into the program after a cross-analysis of the source code and of the metrics reported by the Dispatcher Generator. In conclusion, even against an attacker able to control data, MAFIA only leaves a small attack surface against attack such as ROP or JOP.

VII. Experimental Evaluation

To evaluate the hardware overhead due to MAFIA, we synthesize the modified CV32E40P into an Application Specific Integrated Circuit (ASIC). The ASIC is designed for a frequency of 400MHz, in the GF-22FDX FDSOI technology,

TABLE II
EMBENCH-IOT RESULT WITH THE SIZE (IN BYTES, AND OVERHEAD WRT. UNPROTECTED VERSION), SIGNATURES AND PATCHES AND THE EXECUTION TIME (IN CPU CYCLES, AND OVERHEAD WRT. UNPROTECTED VERSION) FOR MAFIA WITH THE CRC SIGNATURE FUNCTION

Bench	O2				Os			
	Size	Signatures	Patches	Exec. time	Size	Signatures	Patches	Exec. time
aha-mont64	6204 (×1.30)	124	106	75335 (×1.38)	4720 (×1.28)	92	70	67461 (×1.18)
crc32	824 (×1.34)	8	21	380997 (×1.21)	856 (×1.35)	9	21	381006 (×1.21)
cubic	97460 (×1.16)	98	1611	14787617 (×1.16)	96920 (×1.16)	95	1608	14802517 (×1.16)
edn	3564 (×1.29)	53	63	1157814 (×1.22)	3944 (×1.25)	53	64	1157585 (×1.22)
huffbench	4028 (×1.39)	90	90	382404 (×1.20)	3548 (×1.33)	61	73	383163 (×1.16)
matmult-int	3376 (×1.26)	29	70	1119255 (×1.21)	2588 (×1.23)	10	54	1200057 (×1.21)
minver	21472 (×1.24)	56	489	151685 (×1.18)	21404 (×1.24)	59	474	178843 (×1.17)
nbody	20576 (×1.20)	52	404	55767175 (×1.18)	19944 (×1.20)	42	387	55780098 (×1.18)
nettle-aes	5600 (×1.14)	46	63	136279 (×1.10)	5512 (×1.14)	44	59	136447 (×1.10)
nettle-sha256	7864 (×1.07)	39	44	9573 (×1.03)	7720 (×1.08)	46	46	10239 (×1.03)
nsichneu	25204 (×1.45)	654	565	3693 (×1.44)	25152 (×1.45)	648	546	3685 (×1.44)
qrduino	21840 (×1.39)	554	455	1605197 (×1.18)	18304 (×1.37)	448	357	1610103 (×1.16)
slre	7604 (×1.55)	242	211	–	6760 (×1.52)	214	168	45480 (×1.17)
st	21964 (×1.20)	58	420	6618952 (×1.17)	21764 (×1.19)	44	409	6626187 (×1.17)
statemate	8956 (×1.29)	203	141	1573 (×1.09)	9116 (×1.28)	198	138	1672 (×1.08)
ud	3456 (×1.25)	39	62	23674 (×1.16)	3232 (×1.27)	37	62	24125 (×1.16)
picojpeg	31116 (×1.47)	881	665	2403701 (×1.21)	22548 (×1.49)	642	485	2424312 (×1.16)
sglib-combined	8332 (×1.47)	197	219	409637 (×1.18)	8736 (×1.46)	206	214	466871 (×1.31)
wikisort	32340 (×1.30)	354	647	28465027 (×1.38)	31956 (×1.29)	318	625	24817089 (×1.20)
geometric average	**(×1.30)**			**(×1.19)**	**(×1.29)**			**(×1.18)**

and the target frequency is not impacted by the addition of MAFIA. The core occupies 64 kGE with CBC-MAC/Prince and 55 kGE with CRC32, which represents an area overhead wrt. the unmodified core of 23.8% and 6.5% respectively.

The software evaluation is carried out through HDL cycle-accurate simulations of the modified CV32E40P with CRC32. We benchmark our implementation with the Embench-IoT [23] test suite, which targets embedded systems without operating system. All the test programs are compiled with the MAFIA toolchain, with optimization levels `-Os` and `-O2`, and are linked with the Newlib C-library and the LLVM multiplication and soft float libraries. The benches are compiled with the option `-ffunction-section` to eliminate any dead code. Embench-IOT contains 4 benches with indirect branches. We prevent the compiler from using indirect branches with the `-fno-jump-table` compiler option, which leaves 3 benches with indirect function calls (`picojpeg`, `sglib-combined` and `wikisort`). Note that `slre` could not be compiled with the `-O2` optimization level because our reference signature generation does not support the inter-procedural loop caused by a recursive call.

We add the attribute `MAFIA_secure` to the benchmarked functions only, meaning that only those functions contain signature verifications. The C-library and the LLVM library do not contain any signature verification but are still instrumented with `MAFIA.ldp` instructions.

The code size evaluation considers only the sections impacted by MAFIA (`.text` and `.patches`), which provides a pessimistic, upper bound of the overall code size overheads for a complete firmware image.

Table II and Figure 5 summarize the results of our experimental evaluation. The results show that MAFIA can handle different kind of software as selected by Embench-IOT. Execution time overheads range between 2.5% and 44.0% with a geometric average of 18.4%. The code size overheads range between 7.3% and 55.4% with a geometric average of 29.4%. For the majority of the benches, the difference between the optimization levels `-Os` and `-O2` is negligible, except for `aha-mont64`, `sglib-combined`, and `wikisort`. The difference is explained by the execution of instruction sequences where a `MAFIA.ldp` is immediately followed by a branch. In this case, the pipeline controller stalls the processor so that the memory stage can fetch the patch before the branch is taken. For those 3 benches, this pattern is present in a loop nest increasing considerably the number of stalled cycles between optimization level `-O2` and `-Os`.

The evaluation results show that the update function (patch values and `MAFIA.ldp`) is responsible for the largest part of the code size overhead. The number of reference signatures is dependent of the number of branches in the functions annotated with `MAFIA_secured`. In our evaluations, reference signatures are only inserted for the benches core functions. But, all the functions need `MAFIA.ldp` independently whether they come from the benchmark core or from libraries.

The forwarding dependency elimination pass and the patch placement pass (Section V-D) also contribute to the code size overhead ("Other" in Fig. 5). Those passes insert `nop` instructions to break some forwarding dependencies and disable some tail call optimization respectively. Also, MAFIA LLVM passes insert new instructions in the code (e.g. `MAFIA.ldp` instructions). This requires to adapt direct branch offsets. The new offsets might not fit in the original branch binary encoding anymore. In such a case, an additional direct branch is inserted which increases the code size.

Table III reports the overhead induced by the dispatchers. In the worst case, the dispatchers contribute to 15.3% of the total overhead for `wikisort -Os`. The absolute overhead induced is only 4% at worst. For all the other benches, the dispatchers contribute to less than 10% of the total overhead. Note that this overhead could even be reduced by replacing

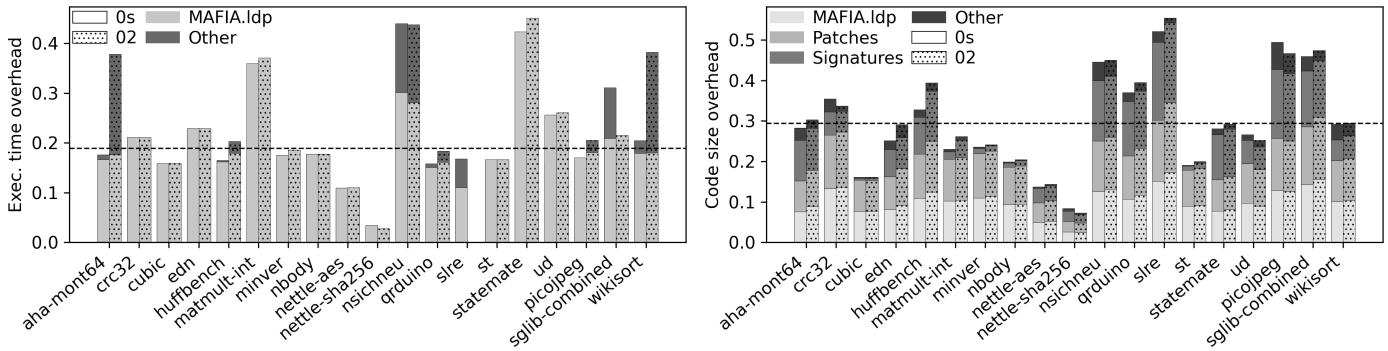

Fig. 5. Execution time (left) and code size (right) overheads for the Embench-IoT benchmarks, with the `-Os` and `-O2` compiler optimization levels for MAFIA with the CRC signature function.

TABLE III
CONTRIBUTION OF DISPATCHERS TO THE TOTAL CODE SIZE OVERHEAD: IN BYTES, NUMBER OF ADDED PATCHES AND ADDED SIGNATURES

Bench	Code Size (Bytes, %)	Patches	Signatures
picojpeg (`Os`)	84 (1.1%)	4	2
picojpeg (`O2`)	168 (1.7%)	4	4
sglib-combined (`Os`)	256 (9.3%)	8	6
sglib-combined (`O2`)	172 (6.4%)	6	4
wikisort (`Os`)	1108 (15.3%)	34	26
wikisort (`O2`)	604 (8.2%)	22	14

dispatcher targeting equivalence classes with a single element to a direct call to the target function. This demonstrates that using dispatchers is a practical solution to avoid GPSA signature confusion for a small additional overhead.

We observe that the compiler optimization level has a moderate impact on the code size and execution time overheads, but that there is a large variation of overheads, which are due to the different code structures used in the benches. Furthermore, smaller basic block sizes are more impacted by the instrumentation with MAFIA instructions. The maximum code size overhead is 400% ($\times 4$) for a single basic block composed of a single branch instruction, as MAFIA's code instrumentation requires the addition of 3 memory words: a `MAFIA.ldp` instruction, the corresponding patch value, and the reference signature. Our results show that the average code size overhead is far less because basic blocks have greater sizes. Some optimizations (such as tail duplication or loop unrolling) could increase the size of basic blocks or reduce the number of branches to reduce the execution time overheads.

It is possible to get an approximation of the CBC-MAC/Prince overheads from the CRC32 results. Both CBC-MAC/Prince and CRC32 compute a signature in a single cycle and verify 32-bit reference signature. However, CBC-MAC/Prince works with 64-bit blocks and therefore requires 64-bit patch values. To handle 64-bit patch values MAFIA uses two update instructions, one for the 32 most significant bits and the other for the 32 least significant bits. Such implementation leads to double the software overheads induced by the update function (patch values and `MAFIA.ldp`). Therefore, with the CBC-MAC/Prince implementation, MAFIA induces an average execution time overhead close to 39% and an average code size overhead close to 50%, but it also ensures code authenticity. Replacing Prince with a 32-bit block cipher, such as Simon [24], could close the performance gap between the CBC-MAC and the CRC32 implementations of MAFIA.

VIII. RELATED WORK

Counter-measures ensuring code and control-flow integrity are often implemented as hardware components external to the processor microarchitecture [3], [10]. Such counter-measures are easier to integrate in a processor design but are intrinsically blind to faults targeting the microarchitecture. For these reasons, they are not further discussed in this section.

Table IV provides a comparison between MAFIA and the related counter-measures. It reports the claimed security properties ensured by the counter-measures, the estimated hardware area, code size execution time overhead. Regarding the hardware area overhead, note that Table IV is only indicative of a trend because each work is based on different processor architectures and different technologies.

Werner et al. use GPSA in the context of fault injection [4]. The signature is derived from the binary encoding of program instructions using a CRC32 signature function. In MAFIA, the CRC32 implementation presents slightly larger hardware overhead due to the additional CSI module. Actually, the CACFI module, which handles GPSA in MAFIA, is equivalent to the GPSA monitor in [4]. The main difference is the origin of the signature input which in MAFIA is the pipeline state instead of the binary encoding of program instructions. On the software side, MAFIA induces half less code size overhead and execution time overhead. The reason is that MAFIA handles the loading of patch values in a single dedicated `MAFIA.ldp` instruction while [4] requires several standard loads and stores to place the patch values in a memory mapped register. Regarding the security properties, both MAFIA and [4] ensure code and control-flow integrity. Additionally, MAFIA supports control-signal integrity, meaning that it can detect fault targeting the control signals after the fetch stage.

[5]–[7] are code and control-flow integrity counter-measures based on authenticated decryption. MAFIA ensures code authenticity, but is not designed to ensure code confidentiality as is. [6] is the closest to MAFIA implemented with a CBC-MAC signature function. Both designs have similar hardware overheads. They are both based on the Prince

TABLE IV
SECURITY AND OVERHEAD COMPARISON OF CODE AND CONTROL FLOW INTEGRITY PROTECTION TARGETING FAULT INJECTION ATTACKS

	Security	Target	Technology	Area overhead	Exec. time overhead	Code size overhead
Arora et al. [10]	CI/CFI	ARM9TDMI ARM920T	FPGA Virtex 2	13.7%	100%	NA
Werner et al. [4]	CI/CFI	ARMv7-M Cortex-M3	ASIC UMC 130nm	4%	32%	57%
Danger et al. [3]	CI/CFI	RISC-V PicoRV32	FPGA Artix 7	20%	2% to 63%	118% to 160%
Clercq et al. [5]	CC/CA/CFI	SPARC LEON3	FPGA Virtex 6	28.2%	13.7%	140%
Werner et al. [6]	CC/CA/CFI	RISC-V CV32E40P	ASIC UMC 65nm	28.8%	9.1%	19.8%
Savry et al. [7]	CC/CA/CFI/DC/DA	RISC-V CV32E40P	–	–	167%	24%
MAFIA CRC	CI/CFI/CSI	RISC-V CV32E40P	ASIC FDSOI 22nm	6.5%	18.4%	29.4%
MAFIA CBC-MAC	CA/CFI/CSI	RISC-V CV32E40P	ASIC FDSOI 22nm	23.8%	39%	50%

CI: Code Integrity, CA: Code Authenticity, CC: Code Confidentiality, CFI: Control Flow Integrity, DC: Data Confidentiality, DA: Data Authenticity, CSI: Control-Signal Integrity, NA: Non-Applicable, –: Not provided

cryptographic primitive, which is responsible for the most important usage of extra silicon area. However, the software overheads of [6] are approximately 30% smaller for two reasons: (i) faults are detected in case of bad instruction decoding, which alleviates the need for reference signatures; (ii) control-flow instructions simultaneously load patch values, whereas MAFIA requires a dedicated instruction.

Regarding indirect control flow, [5] also relies on indirect branch elimination. By design, the basic blocks cannot have more than 4 or 5 instructions, and cannot have more than 2 predecessors. Those limitations considerably increase the complexity of the dispatchers, which leads to larger software overheads as compared MAFIA. In [6] all the functions reachable from the same indirect branch share the same state, which is similar to signature confusion. However, the state is also used for code encryption. To avoid possible cryptographic vulnerabilities, an additional patch value is inserted at the beginning of the target functions, which restricts possible signature confusions to the function entry points. In [7], indirect branch patch values are stored in a word placed before the indirect branch target. This approach prevents any signature confusion and allows more flexibility to support software constructs such as C++ `vtables`. However, this mechanism requires confidential patch values to prevent the forging of new indirect branches. MAFIA relies on indirect branch elimination to prevent signature confusion. We evaluated that the overheads due to the use of dispatchers are low, and that they could be further reduced with additional code optimizations in the toolchain. Finally, all the related works in the context of control-flow integrity against fault injection attacks implicitly assume that indirect branch targets can be identified by external means. This is a major bottleneck for the use of any counter-measure in practice. It implies either that the burden is moved to another tool or to an application designer, or in the worst case the use of a single equivalence class containing all the targets of indirect branches. Our work is the only one to provide a fully automated solution for indirect branch target identification.

While MAFIA has comparable overheads to the related counter-measures, it is still possible to reduce those overheads. First, the area overhead is estimated considering the overhead on the processor core only. However, the CV32E40P is a small processor core and we expect the contribution of the core to the area of a full system to be small, even in the case of IoT devices. Hence, MAFIA's relative area overhead measured in a complete system, including memory and peripherals, would be much smaller. It is also possible to reduce MAFIA's area overhead and code size overhead, at the expense of security, by selecting a more lightweight signature function such as CRC8. Finally, our work reports MAFIA's code size and execution time overheads when applied globally to the application. By extending MAFIA with a secure on/off mechanism, it would be possible to enable MAFIA protection for sensitive code only. The local overheads on the sensitive code would be the same as the ones reported in Section VII, but the global overheads on the application would be much smaller.

MAFIA is, to the best of our knowledge, the only counter-measure to ensure control-signal integrity against fault injection attacks. Kim and Somani propose an on-line integrity monitoring of the microprocessor control logic for safety-critical systems sensible to soft errors [25]. They use a non-secure function (XOR) to derive a signature from the static control signals in every pipeline stages, and dynamic control signals are protected by duplication. A caching mechanism is used to store reference signatures from the first execution (cache miss), and verifies the runtime signatures in the subsequent executions (cache hit). However, such a caching mechanism does not protect against attacks targeting executions leading to a signature cache miss (e.g. first program execution) because the reference signature is not available. Moreover, this solution does not detect attacks targeting the program memory, hence does not ensure code integrity. Another independent mechanism ensures control-flow integrity. In MAFIA, a unique signature is derived from static and dynamic control signals of the decode stage. This signature ensures simultaneously execution, code and control-flow integrity.

IX. CONCLUSION

This paper presents MAFIA, a counter-measure extending the state-of-the-art against fault injection attacks by combining control-signal integrity with code integrity, code authenticity and control-flow integrity. MAFIA articulates two security mechanisms to protect the control logic of the processor against faults targeting the processor microarchitecture. MAFIA also protects against faults injected outside of the processor that have an impact on the processor control logic. A first module implements generalized path signature analysis (GPSA). The signature is computed from the pipeline state, a

set of data-independent control signals that deterministically result from the decoding of the binary instruction. This module ensures simultaneously control-flow integrity, code authenticity, and control signal integrity from the fetch stage to the end of the decode stage. A second module, implementing a redundancy-based mechanism, ensures the integrity of the same control signals in the subsequent pipeline stages, which completes the full protection coverage of the processor microarchitecture.

The flexibility of the design allows for trade-offs between security and overheads. The paper presents two implementations of MAFIA based on the CV32E40P RISC-V processor, with different signature functions: one with CBC-MAC and Prince, and another one with a CRC32 error detector code. CBC-MAC/Prince makes use of the full capabilities of MAFIA. It induces a hardware area overhead of 23.8 %, and average code size and an execution time overheads of 50 % and 39 % respectively. CRC32 detects a minimum number of 8 bit-flips and ensures code integrity only instead of code authenticity; it induces a hardware area overhead of 6.5 %, and average code size and an execution time overheads of 29.4 % and 18.4 % respectively. On the software side, the compiler extension offers a complete automatic processing of the program source code to generate the MAFIA executable program. Moreover, thanks to the support of indirect branches and interrupts, MAFIA is fully compliant with software stacks used in embedded system.

Acknowledgements

We thank Mikael Le Coadou and Juan Suzano Da Fonseca for their contributions to the hardware evaluation, and Simon Tollec for his contribution to the formal verification.

References

[1] B. Yuce, P. Schaumont, and M. Witteman, "Fault Attacks on Secure Embedded Software: Threats, Design, and Evaluation," *J Hardw Syst Secur*, 2018.
[2] B. Colombier, P. Grandamme, J. Vernay, É. Chanavat, L. Bossuet, L. de Laulanié, and B. Chassagne, "Multi-Spot Laser Fault Injection Setup: New Possibilities for Fault Injection Attacks," in *CARDIS*, 2022.
[3] J.-L. Danger, A. Facon, S. Guilley, K. Heydemann, U. Kühne, A. Si Merabet, and M. Timbert, "CCFI-Cache: A Transparent and Flexible Hardware Protection for Code and Control-Flow Integrity," in *DSD*, 2018.
[4] M. Werner, E. Wenger, and S. Mangard, "Protecting the Control Flow of Embedded Processors against Fault Attacks," in *CARDIS*, 2015.
[5] R. de Clercq, R. D. Keulenaer, B. Coppens, B. Yang, P. Maene, K. de Bosschere, B. Preneel, B. de Sutter, and I. Verbauwhede, "SOFIA: Software and control flow integrity architecture," in *DATE*, 2016.
[6] M. Werner, T. Unterluggauer, D. Schaffenrath, and S. Mangard, "Sponge-Based Control-Flow Protection for IoT Devices," in *EuroS&P*, 2018.
[7] O. Savry, M. El-Majihi, and T. Hiscock, "Confidaent: Control FLow protection with Instruction and Data Authenticated Encryption," in *DSD*, 2020.
[8] J. Laurent, V. Beroulle, C. Deleuze, F. Pebay-Peyroula, and A. Papadimitriou, "Cross-layer analysis of software fault models and countermeasures against hardware fault attacks in a RISC-V processor," *Microprocessors and Microsystems*, 2019.
[9] T. Chamelot, D. Couroussé, and K. Heydemann, "SCI-FI: Control Signal, Code, and Control Flow Integrity against Fault Injection Attacks," in *DATE*, 2022.
[10] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Hardware-Assisted Run-Time Monitoring for Secure Program Execution on Embedded Processors," *IEEE Trans. on VLSI Systems*, 2006.
[11] R. de Clercq and I. Verbauwhede, "A survey of Hardware-based Control Flow Integrity (CFI)," *arXiv:1706.07257*, 2017.
[12] R. Schilling, M. Werner, and S. Mangard, "Securing Conditional Branches in the Presence of Fault Attacks," *arXiv:1803.08359*, 2018.
[13] K. Wilken and J. P. Shen, "Continuous signature monitoring: Low-cost concurrent detection of processor control errors," *IEEE TCAD*, 1990.
[14] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," in *CCS*, 2007.
[15] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *ASIACCS*, 2011.
[16] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-Flow Integrity: Precision, Security, and Performance," *ACM Comput. Surv.*, 2017.
[17] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, 2009.
[18] W. Arthur, B. Mehne, R. Das, and T. Austin, "Getting in control of your control flow with control-data isolation," in *CGO*, 2015.
[19] OpenHW Group, "CV32E40P," https://github.com/openhwgroup/cv32e40p, 2021.
[20] S. Tollec, M. Asavoae, D. Couroussé, K. Heydemann, and M. Jan, "Exploration of Fault Effects on Formal RISC-V Microarchitecture Models," in *FDTC*, 2022.
[21] Yosys, "Yosys – Yosys Open SYnthesis Suite," Yosys, 2023, https://github.com/YosysHQ/yosys.
[22] P. Koopman, "32-bit cyclic redundancy codes for Internet applications," in *DSN*, 2002.
[23] "Embench™: Open Benchmarks for Embedded Platforms," https://github.com/embench/embench-iot, 2021.
[24] P. Maene and I. Verbauwhede, "Single-cycle implementations of block ciphers," in *LightSec*. Springer, 2015.
[25] S. Kim and A. K. Somani, "On-line integrity monitoring of microprocessor control logic," *Microelectronics Journal*, 2001.

Thomas Chamelot received the M.S. degree in Cybersecurity from the University Toulouse III, Toulouse, France in 2019, and the Ph.D. degree from Sorbonne Univeristé, Paris, France, in 2022. His current research interests include hardware security and computer architecture.

Damien Couroussé is with CEA-List since 2011, as a Research Engineer and Senior Expert. He received the Ph.D. from the Institut National Polytechnique de Grenoble in 2008. His research interests include embedded software and its interaction with hardware, compilation and runtime code generation for performance and security, with a recent focus on hardware security.

Karine Heydemann is an Associate Professor at Sorbonne University / LIP6 since 2006 and a Senior Expert Architect at Thales DIS since September 2022. She received the Ph.D. degree in Computer Science from the University of Rennes 1 in 2004. Her areas of expertise encompass hardware microarchitecture, compilation, code optimization, and physical attacks, including modelling of hardware fault injection effects, automated code hardening and robustness analysis.