Confidaent: Control flow protection with instruction and data authenticated encryption

Olivier Savry, Mustapha El-Majihi, Thomas Hiscock

CONFIDAENT : CONtrol FLow protection with Instruction and Data Authenticated ENcrypTion

Olivier Savry, Mustapha El-Majihi, Thomas Hiscock
Univ. Grenoble Alpes, CEA, LETI
MINATEC Campus, F-38054 Grenoble, France
Email: firstname.name@cea.fr

***Abstract*—Computing devices became part of our daily world. But being physically accessible they are exposed to a very large panel of physical attacks, which are most of the time underestimated. These systems must include protections against these attacks in order to keep user data secret and safe.**

In this work, we argue that addressing the security requirements of embedded processors with independent countermeasures is not the most efficient strategy and may introduce security flaws in the process. Instead, we suggest a more monolithic approach to security design. Following this idea, we propose a new efficient and flexible memory encryption & authentication mechanism called CONFIDAENT, that can protect code and data in embedded processors. On the top of this primitive, we build a strong Control Flow Integrity (CFI) countermeasure. We describe a RISC-V instruction set extension to support these mechanisms and the compiler support needed in the LLVM framework. This new countermeasure is developed on a modified RISCY RISC-V core and its performances are evaluated on a FPGA target. We conclude that a truly high-security can be achieved, with an overhead factor of $\times 2.66$ up to $\times 3.73$ on execution time of benchmarks programs.

I. Introduction

Embedded systems are nowadays omnipresent in our lives. They can be found in many areas from the autonomous car to the industrial Iot where safety and security are essential. This is nevertheless a difficult problem to solve for objects which have strong cost constraints and which are physically accessible to any attacker. The past has shown with the Miraï attack [1] for example that simple connected objects such as IP cameras could cause a global denial of service on the Internet.

A. Security Challenges

Physical access to the devices requires a strong protection of both the confidentiality and integrity of the code as well as the data. Memory tampering allows the recovery of the firmware, which opens the door to reverse engineering, the first step towards an attack allowing copyright infringement, or the simple understanding of proprietary algorithms. The code can also be modified during a physical access to the memory. This modification can be done during execution with fault injection (by voltage or clock glitches), on instructions or registers such as the Program Counter (PC) which points to the next instruction to be executed. The code reuse attack and its many variants (such as ROP [2], JOP [3] or return to libc [4]) allow to modify the Control Flow Graph (CFG) and to reuse parts of the original code in a malicious way. Stack overflows also allow the CFG to be compromised with a data buffer whose boundaries are not checked. It is then possible to overwrite a return address of a function on the stack to send the CFG back to a shellcode injected in the faulty buffer. We can see that a real protection of the control flow cannot be done by simply protecting the instruction sequences, but that a protection of the integrity of the stack data, which constitutes the real hinge of the CFG, is essential.

B. Existing Countermeasures

All these attacks are addressed in the literature one by one with specific countermeasures. Thus, stack overflows are prevented more or less successfully by Data Execution Prevention (DEP) (e.g. the well-known NX bit of the x86 architecture), or by Address Space Layout Randomisation (ASLR) [5]. These countermeasures have been quickly bypassed by code reuse attacks which propose to use legitimate code parts (called gadgets) to execute any instruction sequences. The response has been to further monitor the CFG to catch deviation from the normal behavior by using Control Flow Integrity (CFI) mechanisms. Abadi et al [6] first propose to apply CFI by replacing vulnerable indirect jumps by code snippets. Then solutions involving compilers protect the forward edges by addition of appropriate protections at direct branch site [7]. The return instructions are traditionally verified by a shadow stack which stores a duplication of return addresses to be compared with usual ones [8]. These fine-grained CFIs, supported by hardware to maintain acceptable performances, are the most promising solutions. However, these CFIs have shown their limitations particularly because they do not really address the protection of the stack content which is an essential element of the CFG and the protection against faults on code and data.

In order to overcome both fault injections and code reuse attacks, it has been proposed to combine a CFI with an encryption of the instructions [9], [10] and finally with authenticated encryption. This approach first appeared with SOFIA [11], which proposes to encrypt instructions by associating them with their PC and the previous PC in the CFG. A Message Authentication Code (MAC) is added to this encryption to ensure integrity. This integrity is verified in the stages following the Decode stage in the pipeline and validated by a single signal. Unfortunately, this signal is vulnerable to a fault injection which can validate an illegal instruction. Another more recent

978-1-7281-9535-3/20/$31.00 ©2020 IEEE
DOI 10.1109/DSD51259.2020.00048

approach called SCFP [12] proposes to link the internal state of a cryptographic sponge function with the CFG. At each branch or function return a patch is added to the internal state allowing a single collision and thus a unique change of direction of the CFG. Indirect jumps are also allowed but to a restricted set of locations and require 4 patches per call. The interrupts are managed by saving the internal state of the AE primitive which can potentially be cumbersome. At the end, the performance of the solution is attractive but it is shown on an example without authentication or even sponge function.

We can see that if SOFIA and SCFP are looking to effectively protect the CFG against modifications or reuse, they give up protecting data-driven attacks such as stack overflows even though the stack is the hub of the CFG, especially during function calls. Moreover, their approaches are purely static and do not handle real indirect jumps where the destination address is only known at runtime, such as when calling a dynamic library.

C. Contributions

In order to address all of the above vulnerabilities, we propose, in line with SOFIA and SCFP, an approach that breaks with the stacking of countermeasures and is more holistic. Our solution is based on authenticated encryption of both instructions and data. This has the merit of first detecting fault injections such as DRAM rowhammers [13] or glitches on the code and data and in particular on the stack. The integrity check is no longer done at boot time like classical secure boots which are long for small processors (it's always annoying to wait at boot time!) and then leaves the code vulnerable to modifications. Integrity checking is now done at runtime just before the Fetch stage for instructions and before the Load/Store Unit (LSU) level for data. The CFI that we call CONFIDAENT is done in the form of xored masks with each instruction and which are specific to each Basic Block. New instructions added to the RISC-V ISA ensure the change of the mask's root at the moment of a branch. To manage indirect jumps, an encrypted mask is added in the header of the first Basic Block called. Thus, there are no restrictions on the destination of indirect jumps which can be totally dynamic (switches, callbacks, or dynamic libraries are managed). Finally, during function calls, the return address and the mask necessary for decryption at this address are stored on the stack in an encrypted way. An attacker who wants to exploit a buffer overflow will need to know the mask used where he wants to jump to. The guess will be difficult since masks are secret.

We demonstrate the feasibility of this solution on FPGA with a RISCY processor of the Pulpino platform [14] which uses the RISC-V ISA [15]. An LLVM compilation toolchain including the new instructions added at the backend has been developed. The authenticated encryption step is done after the link, which makes our solution compatible with binaries that are already compiled (they will however not benefit from the protections provided by our CFI). Overheads in code size and CPU cycles number are in the same order of magnitude as SCFP: from 0 to 36%. To this must be added a doubling of the memory occupation for the storage of meta-data and integrity tags.

D. Outline

In this paper, we first define a security framework with a threat model and targeted attacks. Then, the architecture of our solution will be described followed by a presentation of the CFI and the required adaptation of the LLVM compiler. Finally, we will discuss its performances and analyze its security properties.

II. Security Framework

A. Threat Model

As we have seen, we assume that the attacker has physical access to the device but is limited to elements that are potentially external to the CPU such as RAM. He can then dump and manipulate these memories at will, he will not have access to the data in plain text. The inputs and outputs are accessible and faults can be injected in the form of voltage or clock glitches for example. On the other hand, we do not consider in our model an invasive intrusion on the core. Similarly, we consider side-channel attacks outside the scope of this study. Finally, we place ourselves in the Dolev-Yao [16] threat model where the attacker is capable of performing protocol-level attacks but cannot break cryptographic primitives.

B. Targeted Security Properties

Faced with these threats, we define a set of security properties that our device will have to verify in the model described in the previous section:

- **Entry Protection**: An attacker must not be able to jump to any address in the code.
- **Fault Attack**: An attacker must not be able to inject faults in the code or data.
- **Stack Overflow**: An attacker must not be able to inject his own code.
- **Code/Data integrity**: An attacker must not be able to change the code or data.
- **Code reuse attack**: An attacker cannot reuse existing pieces of code.
- **Code/Data confidentiality**: An attacker must not be able to decipher the code and the data.
- **Replay attack**: An attacker must not be able to replay data already processed.

III. Architecture

Encryption-based CFI hardware solutions generally consider the encryption of instructions and not data. A joint encryption of the two represents a challenge for the choice of the primitive and its mode. Many constraints have to be taken into account such as lightness in particular with the possibility of encrypting/decrypting with the same hardware block, low latency, random access to memory byte by byte which requires a nonce, iv or tweak and memory alignment for fast addressing. Our encryption method must also have the ability to encrypt

already compiled binaries. With these constraints in mind, we looked for 128-bit security for confidentiality and 64-bit tags for integrity. The recent NIST competition for lightweight AE has resulted in many primitives that meet our requirements. Most of them are defined by the update of an internal state with which the plaintext is xored. This scheme matches well with the encryption of a sequence of instructions. We have chosen one of the finalists, ASCON [17], which fills our criteria. However, if encryption must not to be the limiting factor of CPU speed, operating instruction by instruction and data by data is not realistic.

The data or instructions are therefore grouped in blocks of 16 bytes (or 4×32-bit words) on our 32-bit architecture (i.e. 128-bit ciphered text) associated with a 64-bit integrity tag. Nevertheless the structure of the AE primitives obliges to renew the nonce and thus requires that a part of it is stored (Cf Figure 1). We thus put 32 bits of the nonce in a 64 bits

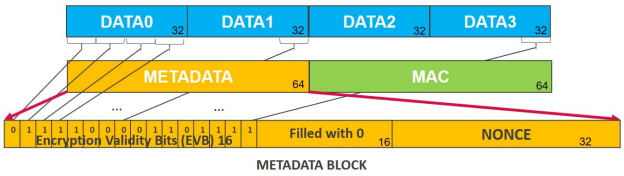


Fig. 1: Format of the Code or Data authenticated encryption with a zoom on the metadata sub-block

part of metadata which is seen as authenticated data (AD i.e. not ciphered but authenticated with the integrity tag). At each store instruction of a new data, the nonce is changed by taking 31 bits of the previous integrity tag. This saves an implementation of an RNG. The missing bit is a NX bit (Non Executable) which is set by the compiler to 1 for data and to 0 for instructions. This bit is checked by the Fetch stage of the processor and in case this bit is equal to 1, an exception is raised. This is a real simple proven hardware mechanism to avoid stack overflow attack. The Nonce being on 128 bits, we also have 32 bits which are defined by the address of the block allowing us a random access to each data and a diversity of encryption. The remaining 64 bits of the Nonce are considered secret to contribute reinforcing security. With this management by block, the writing of one byte of data can pose integrity problems because it involves, by the decryption of a 128-bit block (before being encrypted again), the validation of all the other bytes. Indeed, they may not have been initialized and therefore they must not have a valid integrity tag. To avoid this inconvenience, we have used 16 bits (1 per byte) in the authenticated data part that we call *Encryption Validity Bit* (EVB) which goes from 0 to 1 during a valid writing of a byte. Consequently, on reading only the bits set to 1 allow the associated bytes to be considered as valid (provided that the block integrity tag is also valid).

We then have a block size of 256 bits (128 bits of data/instructions, 64 of AD including 16 bits of EVB, 32 bits of nonce and 16 useless bits set to 0, and 64 bits of integrity tag) which allows a simple memory alignment to be addressed but at the price of doubling the memory. The size of this block can be lowered with the presence of a cache. For example, we can have a 64 byte block of encrypted data with 16 bytes of metadata and 16 bytes of MAC for an overhead of only 50% of the memory occupancy but requiring the implementation of an addition for its addressing. Figure 2 shows the position of the encryption/decryption block between the instruction memory and the data memory for a RISCY core. It provides an average of just over 1 de/encryption per CPU clock cycle thanks to two 128-bit buffers at the memories input/output. The loading of the key and the secret part of the Nonce is done through the debug port which can be locked by a 128-bit password.

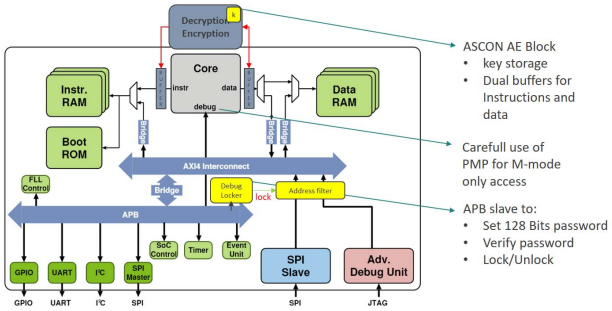


Fig. 2: RISCY Core with ASCON AE block and the two buffers enabling a sharing between Instruction and Data encrypted memories. Debug lock mechanism with 128-bit password to store the cryptographic key.

IV. Control Flow Integrity

A. Overview

Control Flow Integrity is ensured by a lightweight masking scheme applied to the instructions, the latter can be built on top of the memory encryption mechanism described previously. In a nutshell, the fetch logic of the processor holds a mask register named `m`, which is updated at each cycle by applying a function called `next` (see Definition IV.1) to the current value of `m`. The processor mask `m` is combined at each cycle with data coming from instruction memory using a XOR to produce the plain instruction. While our proof of concept implements the `next` function as a very simple permutation, more complex functions may be selected, such as a LFSR.

Definition IV.1. The function `next` : $\{0,1\}^{32} \rightarrow \{0,1\}^{32}$ is defined as the 32-bit DES permutation whose bit association is shown in Table I. Note that the hardware implementation of such permutation has almost no cost.

TABLE I: Bit permutation, adapted from the DES cipher

	0	1	2	3	4	5	6	7
0	16	7	20	21	29	12	28	17
8	1	15	23	26	5	18	31	10
16	2	8	24	14	32	27	3	9
24	19	13	30	6	22	11	4	25

For a given sequence of instructions $(i_0,\ i_1,\ \ldots,\ i_n)$, the masked instructions stored in memory (before being encrypted) will be: $(i_0 \oplus \mathtt{m}_0, i_1 \oplus \mathtt{next}(\mathtt{m}_0), \ldots, i_n \oplus \mathtt{next}(\mathtt{m}_{n-1}))$. Where the initial mask $\mathtt{m}_0$ is picked randomly and the rest of the sequence $\mathtt{m}_i = \mathtt{next}(\mathtt{m}_{i-1})$ is computed by updating the previous mask. Branchless programs can be easily executed using this scheme: the instructions are stored in memory in masked form, and at each cycle the processor unmasks the instruction using `m` and updates it. The processor only needs the initial mask of the sequence $\mathtt{m}_0$. However, when a branch is taken, the value of `m` must change to the mask of the target location. This simple observation implies two important requirements:

1) There should be a mechanism to change the mask on branches to a value other than $\mathtt{next}(\mathtt{m})$
2) The mask used at the target address of any branch (both direct and indirect) must be known

To address these requirements, we describe a RISC-V [15] ISA extension that allows programmers to manipulate masks using additional instructions and registers. This extension was designed to be highly flexible and be compatible with existing programs. Thus, this extension allows to apply CFI on very complex programs, for example we successfully compiled the C library Newlib [18] with CFI and run programs that use it. While programmers can use these extra instructions explicitly, manually managing masks can be cumbersome. A better approach that we develop in this paper is to leverage compiler to handle the mask management for us.

B. Instruction Set Extension

The RISC-V ISA extension we proposed is called ECR, which stands for Encryption Control Registers. It adds a new register bank (holding up to 16×32-bit registers) and four new instructions: `ecr.lui`, `ecr.addi`, `ecr.lw` and `ecr.sw`. An additional instruction `ecr.enter` is also reserved. It should never be executed, its sole purpose is to reserve a mask slot in code. Because this instruction has a unique encoding, the masking software can detect them and replace them through the relocation mechanism (see subsection IV-E). The instructions semantics are described in Table II. Four important ECR registers are:

- `m`, the current mask value used by the Fetch stage to retrieve the plaintext of the current instruction. This register is implicit, its value is not directly accessible to the programmer.
- `emb`, the mask value to be used as `m` if the next branch is taken.
- `emr`, a return mask which contains the value of $\mathtt{next}(\mathtt{m})$ when the last `jal` instruction was executed.
- `eme`, which contains the value of $\mathtt{next}(\mathtt{m})$ when the last exception was taken. It is the analogous of `emr` for exceptions.
- `emt`, a configuration registers which defines the masks to be used when jumping to an exception.

The registers `emr` and `eme` are updated automatically by the processor when executing any jump (for `emr`) or jumping to an exception handler (for `eme`).

TABLE II: ECR instructions semantic

Instruction	Pseudo-code
`ecr.lui ecrd` , $imm20$	$ECR[\mathtt{ecrd}] \leftarrow imm20 \ll 12$
`ecr.addi ecrd , ecrs` , $imm12$	$ECR[\mathtt{ecrd}] \leftarrow ECR[\mathtt{ecrs}] + \text{signext}(imm12)$
`ecr.lw ecrd` , $offset$(`rs1`)	$ECR[\mathtt{ecrd}] \leftarrow MEM[\mathtt{rs1} + \text{signext}(offset)]$
`ecr.sw ecrt` , $offset$(`rs1`)	$MEM[\mathtt{rs1} + \text{signext}(offset)] \leftarrow ECR[\mathtt{ecrt}]$

C. Transformation of common code structures

As we have seen, each Basic Block has its own $\mathtt{m}_0$ mask which must be changed at each direct or indirect branches. Conditional branches and direct jumps are managed by loading in the `emb` register the new $\mathtt{m'}_o$ mask of the Basic Block the CPU is jumping to. As the mask is on 32 bits, the loading of an immediate value of a register requires 2 instructions:

$$\mathtt{ecr.lui\ emb}\ ,\mathtt{m'}_{0,\,20} \qquad\qquad \mathtt{ecr.addi\ emb}\ ,\mathtt{m'}_{0,\,12}$$

where $\mathtt{m'}_{0,\,20}$ are the 20-bit MSB of $\mathtt{m'}_0$ and $\mathtt{m'}_{0,\,12}$ are the LSB 12-bit of $\mathtt{m'}_0$ (Cf Figure 3). The next instruction is therefore a direct jump or a conditional branch that will automatically change the current mask register `m` (`m` ← `emb`). This instruction also puts $\mathtt{next}(\mathtt{m})$ in the emr register to keep track of the mask necessary to decipher the sequence of instructions in case of a non-leaf function return (`emr` ← $\mathtt{next}(\mathtt{m})$).

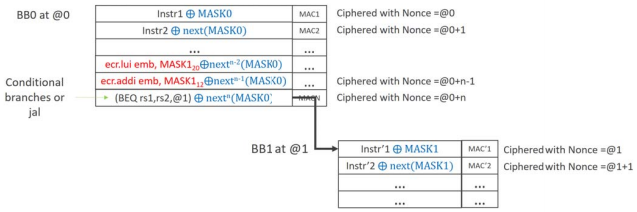


Fig. 3: Modification of the mask during a conditional branch or a direct jump. For the ease of understanding, each instruction has a MAC whereas it is only 1 MAC for 4 instructions in practice

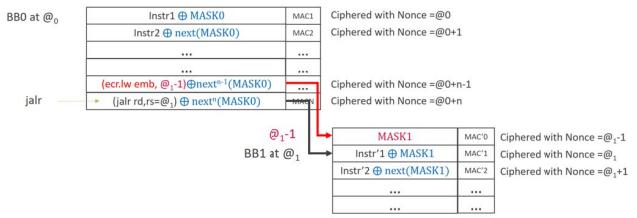


Fig. 4: Modification of the mask during an indirect jump. The first Basic Block of the called function has a ciphered header which is the mask required to decipher it. For the ease of understanding, each instruction has a MAC whereas it is only 1 MAC for 4 instructions in practice

For nested function calls, the return mask is stored on the stack along with the return address. This is done by the

instruction ecr.sw emr , $offset$(rs1) which loads the mask into emr at $offset$(rs1). Since the data is encrypted, the masks stored in main memory are encrypted as well. The attacker is then forced to make a brute force attack to find the right mask. This structure allows us to easily manage indirect jumps. Indeed, the instruction jalr behaves as a jal with respect to the registers m, emb and emr. It also writes next(m) into emr. The ecr.lw emb , $offset$(rs1) instruction loads the mask stored on the stack in the register emb before any jalr instruction. Any function can also have a mask as the header of its first Basic Block, which can then be retrieved by the ecr.lw instruction and thus be called indirectly (Cf Figure 4). This approach has the advantage over the state of the art to allow dynamic calls to libraries for example, calls to C++ class methods via vtables, switch tables and many other things. This mechanism does not compromise protection against code reuse attacks because the masks remain secret and must be found by brute force by the attacker if he wants to use parts of the legitimate code.

On the other hand, a stack overflow attack is still possible because an attacker can take advantage of a buffer whose bounds are not checked to overwrite not only the return address of the function on the stack as usual but also the return mask with a mask of his choice with which he will have instrumented his shellcode. This masked shellcode is then put in the faulty buffer. It is only the hardware NX bit mechanism, of negligible cost, that avoids the execution of this shellcode and thus the exploitation of the vulnerability.

The proposed CFI scheme allows to handle exceptions without trouble. Indeed, exceptions can be viewed as jumps (to a specific handler) that can be taken before executing any instruction. Thus, in order to take an exception the processor needs 1) to setup the mask associated with the current exception handler and 2) when done, to restore the source instruction mask in order to resume execution. Fortunately, the exception return mask is already available in the ecr.eme register. Configuring the mask of the exception handler is done by using the ECR register emt presented previously. Thus, a typical boot code would now have to setup the handler appropriately using a code snippet such as (given using RISC-V GNU assembly syntax):

```
_start: # Setup trap handler
        la t0, handler
        csrw mtvec, t0
        ecr.lw emt, -4(t0)
        # ...
        j main

ecr.enter   # Allocate a mask for the handler
handler: # Handler code
         # ...
         eret
```

This scheme can also support vectored exceptions. For that, the processor will set the mask to emt $+ 4 *$ exception$_{number}$. Of course, the exception table entries must be masked appropriately on the compiler side.

D. Compiler Support

The CFI mechanism proposed relies on a compiler for inserting appropriate mask manipulation instruction. We extended the LLVM [19] compiler RISC-V backend to support this extension. The modifications are done in 3 passes:

1) A first pass allocates for each non-leaf function stack slots for return masks
2) A second optional pass takes care of splitting basic blocks
3) A third pass inserts the appropriate ecr.$\star$ instructions, mask saving instructions in the prologue epilogue and pairs of ecr.addi, ecr.lui before direct jumps.

E. Applying CFI on Executable Files

While the algorithm for performing the masking is conceptually simple, it has lots of subtle corner cases. Thus, we make the effort to formalize the data structures and algorithms used.

The CFI algorithm workflow is depicted in Figure 5. The masking algorithm takes as input a sequence of *CFI regions*, which are continuous regions of memory. An ELF file for instance may contain different segments (.text, .data) that resides at completely different memory addresses, a CFI region is associated to each memory segment to be analyzed.

1) Analysis: A first pass performs a whole program analysis in order to build CFI chunks and find relocations. A CFI chunks is defined as a sequence of instructions whose masks are only updated using the next function. Given a CFI chunk, a single mask is generated for the first instruction, then all other masks are derived. While creating CFI chunks, CFI relocations are also collected. CFI relocations are the analogous of relocations used in linkers (such as GNU ld). Namely, a relocation is made of a target address (or equivalently a mask, since a single mask is associated with each program address) and a relocation type. Three types of relocations are supported:

1) RELOC_IMM20: 20 bit relocations, the upper 20 bits of the instruction will be replaced by the 20 upper bits of the target mask. This relocation appears in ecr.lui instructions (for example, ecr.lui emb, %hi(loop) involves such a relocation).
2) RELOC_IMM12: 12 bit relocations, the lower bits of the instructions will be replaced by the 12 lower bits of the target mask. This relocation appears in ecr.addi instructions, as in ecr.lui emb, %lo(loop).
3) RELOC_IMM32: 32 bit relocations, the instruction is completely replaced by the target mask. This type of relocation is emitted for each ecr.enter instructions encountered.

The algorithm for performing the analysis is shown in algorithm 1. It disassembles all CFI regions and looks for ecr.$\star$ instructions to build appropriate relocations. Furthermore, it uses a very reduced RISC-V interpreter (called scpu in algorithm 1) to reconstruct branch target values. Indeed, we need to detect reliably direct jumps to the next instructions (such as j 4), which are interpreted as explicit mask change request. Then, CFI chunks are divided by the algorithm in two cases: if the current instruction is an ecr.enter or when a direct jump to the next instruction is found.

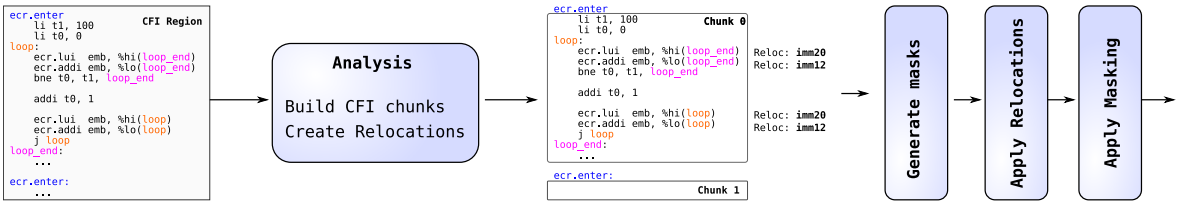


Fig. 5: Data flow of the CFI masking algorithm

Algorithm 1: CFI Analysis Algorithm

Input: CFI Regions to be analyzed
Output: $\mathcal{C}, \mathcal{R}$, respectively a set of CFI chunks and relocations

$\mathcal{C}, \mathcal{R} \leftarrow \emptyset, \emptyset$

for *(vma, instruction)* $\in$ *CFI Region* **do**
  $opcode \leftarrow$ `disassemble`$(instruction)$
  `scpu.eval`$(instruction)$
  if $opcode =$ `ecr.enter` **then**
    $\mathcal{R}+=$ RELOC_IMM32$(vma + 4,\ instruction)$
    Start New Chunks
  else if $opcode =$ `ecr.lui` **then**
    $\mathcal{R}+=$ RELOC_IMM20$(vma,\ instruction)$
  else if $opcode =$ `ecr.addi` **then**
    $\mathcal{R}+=$ RELOC_IMM12$(vma,\ instruction)$
  Append $instruction$ to the current chunk
  if `scpu.is_direct_branch`$(instruction)$ *and* $target = vma + 4$ **then**
    `/* Explicit mask change */`
    Start New Chunk
end
return $\mathcal{B}$

2) Masking: Once CFI chunks are properly constructed, the rest of the algorithm is straightforward. First, the masks for the whole program are computed. More precisely, a RNG is used to sample a 32-bit mask for the first instruction of each CFI chunk. Then, the `next` function is called repeatedly to generate the masks for the chunk. It is important that all masks are computed first, as relocations might refer to mask at any point of the program. For example, the mask of `loop` in Figure 5 must be known for resolving relocations before the `loop_end` label, generating only masks for chunk entry is not sufficient since the loop label appears in the middle of a block.

All relocations are iterated in order and applied: some bits of the instructions are modified according to the relocation type and the target mask. A last pass applies masks to the whole program by doing a bitwise exclusive OR (XOR) with the masks generated and the instructions. Obviously, the relocations must be patched before doing the masking.

3) Use Shorter Masks: A slight modification of the algorithm described above allows to use shorter masks, for instance only represented on 20 bits. This is interesting, since shorter masks means less size overhead and better performances (fewer instructions are needed to modify the mask value). In order to support shorter masks, the following modifications are required:

- Enable Basic Block boundary splitting in the compiler discussed in the subsection IV-D.
- Emit only `ecr.lui emb` for defining the mask (the `ecr.addi` instruction is not emitted).
- In the analysis pass of the masking program (algorithm 1), create a chunk for each direct branch entry

A full example is shown on Figure 6. Even if the masking program generates much more CFI Chunks (4 instead of 2 for 32-bit masks), the overall number of instructions is not highly affected. On this small example, the number of instruction remains the same. Interestingly, the number of instructions in the critical path (the body of `loop`) has two less instructions, which will make the code faster.

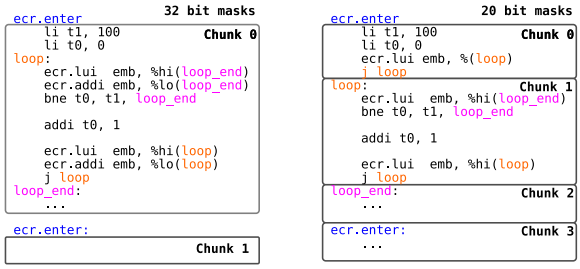


Fig. 6: Difference between a program compiled with 32-bit (left) and 20 bit masks

V. Security Analysis

The security of the proposed solution is ensured by a first layer of data and code authenticated encryption based on the ASCON primitive with a 128-bit key associated with a 32-bit part of secret nonce which ensures a complexity of 160 bits (**Code/data confidentiality**). The integrity tags are 64-bit long which, with the birthday paradox, reduces the collision search to 32 bits. However for each address, the nonce changes and thus any brute force attack has to start again for different addresses. The security of the instructions also relies on a second level of encryption provided by the secret masks of each basic block. In the case of a fault injected on a mask, the tag will not detect an error and the CPU will continue to execute false instructions until it encounters an illegal instruction. This happens very quickly (usually 2 to 3 instructions) given the sparse ISA of RISC V. This concern has been well documented in the SCFP [12] solution. Changing the mask at each instruction prevents an attacker from jumping indiscriminately at any instruction of a basic block and from permuting line of code(**Entry Protection**). On the other hand, the integrity tag allows to efficiently avoid data faults and thus

protect indirect jumps (**Fault Attack**, **Data/Code Integrity**). To carry out a code reuse attack, the attacker will have to guess blindly (because of the ASCON encryption) the 32-bit mask allowing the CPU to correctly decipher the instruction he wants to jump on. It is therefore difficult to generalize ROP type attacks requiring several gadgets (**Code Reuse Attack**). We have seen that the authenticated encryption of code and data alone does not prevent stack overflows from being exploited. On the other hand, the encryption structure used with metadata allows a simple use of a hardware NX bit which makes the stack un-executable and therefore prevents the execution of shellcode (**Stack Overflow**). Finally, data encryption requires protection against replay attacks. Data memory integrity is generally provided by very heavy Merkle tree structures. The implementation we have done does not take into account this security objective for the moment. Nevertheless, the authenticated encryption structure described in this article opens the way to much simpler possibilities to deal with this problem. This will be the subject of further work (**Replay attack**).

VI. Performance Evaluation

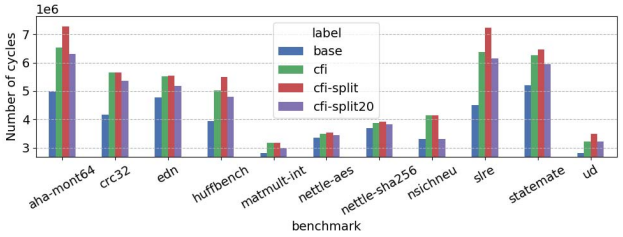
A. Control Flow Integrity

The CFI countermeasure presented in section IV was implemented in a RISCY core from the Pulpino project [14]. We compare the cost of the different CFI variants in terms of execution time and program size. The RISCY core being tailored for IoT applications, we chose a subset of the Embench-IoT benchmarks [20] for our evaluation. Some of these benchmarks make calls to the C-library. To support them, we successfully compile and link a Newlib [18] C-library with CFI enabled. This highlights the robustness of the proposed mechanism. The benchmarks are all compiled in `-O2` mode, which is the common compilation mode for general-purpose applications. The following CFI variants are compared:

- *base*: the program is compiled without CFI.
- *cfi*: the program is compiled with CFI.
- *cfi-split*: the program is compiled with CFI and the mask is changed for each basic block of the program.
- *cfi-split20*: same as the *cfi-split* mode, but only the 20 bits upper bits of the masks are used (lower bits are set to zero).

Each benchmark is executed on the FPGA implementation of the RISCY core, from which we retrieve the exact number of clock cycles. The size of the executable is obtained by measuring the size of the `.text` section given by GNU binutils `size` tool. The binary is stripped from any uncalled functions (using `gc-sections`), thus the `.text` section only includes code that will be executed and no "dead" sections from the C-library. We then compute the relative overhead in terms of cycle and size.

The cycle overhead measurements are shown in Figure 7. The most performant is the *cfi-split20* variant, which is not obvious since it inserts additional direct jumps to any basic block. However, having the 20 bit masks reduces the number instructions to setup the mask in loops and hence, improves performances compared to the base CFI version. The overhead for this variant ranges from 0.08% to 36% in the worst case, which is pretty good comparing current state-of-the art solutions. Interestingly, the cryptographic benchmarks (nettle-aes and nettle-sha256) have a very low overhead (lower that 4%). This is due to a low number of basic blocks, as most of the code is unrolled in such benchmarks.



(a) Graphical results

benchmark	base	cfi	cfi-split	cfi-split20
aha-mont64	4,983,621	6,536,843 (+31.17%)	7,272,503 (+45.93%)	6,311,849 (+26.65%)
crc32	4,159,221	5,645,549 (+35.74%)	5,646,133 (+35.75%)	5,348,729 (+28.60%)
edn	4,778,450	5,510,499 (+15.32%)	5,545,619 (+16.05%)	5,171,863 (+8.23%)
huffbench	3,931,101	5,028,320 (+27.91%)	5,491,008 (+39.68%)	4,786,819 (+21.77%)
matmult-int	2,808,107	3,161,533 (+12.59%)	3,166,417 (+12.76%)	2,990,772 (+6.50%)
nettle-aes	3,342,817	3,478,669 (+4.06%)	3,527,257 (+5.52%)	3,448,319 (+3.16%)
nettle-sha256	3,692,382	3,880,247 (+5.09%)	3,907,251 (+5.82%)	3,813,947 (+3.29%)
nsichneu	3,304,837	4,131,404 (+25.01%)	4,131,404 (+25.01%)	3,307,563 (+0.08%)
slre	4,510,608	6,376,640 (+41.37%)	7,225,528 (+60.19%)	6,144,376 (+36.22%)
statemate	5,212,495	6,261,268 (+20.12%)	6,458,060 (+23.90%)	5,939,445 (+13.95%)
ud	2,803,095	3,224,836 (+15.05%)	3,487,836 (+24.43%)	3,214,037 (+14.66%)

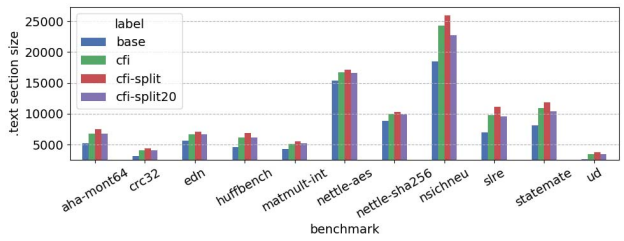
(b) Raw results

Fig. 7: Evolution the number of cycles for different benchmarks

The size overhead measurements are shown in Figure 8. Unfortunately, the embench benchmarks remain "small" benchmarks by design, thus these results should be interpreted with care. The overall CFI overhead lies in the range 0%-40% range. Here, the *cfi* and *cfi-split20* variants are the best and only differs by a few percents.

B. Memory Confidentiality and Integrity

CFI and memory encryption must be used conjointly for the countermeasure to be secure. However, regarding performance evaluation, the memory encryption mechanism can be studied separately. The current memory encryption engine implemented on FPGA uses a single ASCON instance shared between the instruction and data memory. It can perform both encryption and decryption and has a latency of 4 CPU clock cycles. This is achieved by dividing the ASCON clock (250 MHz) 10 times to obtain the CPU clock (25 MHz) (Cf Figure 2). In results shown in Table III, we only provide results for the most efficient CFI profile which uses 20-bit masks. In our current proof of concept, the memory encryption overhead lies between $\times 2.32$ and $\times 3.27$ in the worst case.



(a) Graphical results

benchmark	base	cfi	cfi-split	cfi-split20
aha-mont64	5,264	6,756 (+28.34%)	7,548 (+43.39%)	6,776 (+28.72%)
crc32	3,152	4,120 (+30.71%)	4,384 (+39.09%)	4,096 (+29.95%)
edn	5,624	6,652 (+18.28%)	7,060 (+25.53%)	6,680 (+18.78%)
huffbench	4,584	6,216 (+35.60%)	6,900 (+50.52%)	6,156 (+34.29%)
matmult-int	4,296	5,180 (+20.58%)	5,504 (+28.12%)	5,204 (+21.14%)
nettle-aes	15,360	16,708 (+8.78%)	17,092 (+11.28%)	16,580 (+7.94%)
nettle-sha256	8,855	9,955 (+12.42%)	10,351 (+16.89%)	9,935 (+12.20%)
nsichneu	18,440	24,312 (+31.84%)	25,980 (+40.89%)	22,696 (+23.08%)
slre	7,002	9,834 (+40.45%)	11,130 (+58.95%)	9,542 (+36.28%)
statemate	8,092	10,976 (+35.64%)	11,900 (+47.06%)	10,420 (+28.77%)
ud	2,632	3,504 (+33.13%)	3,804 (+44.53%)	3,484 (+32.37%)

(b) Raw results

Fig. 8: Evolution of the size of the `.text` section

TABLE III: Cycle cost of memory encryption. The overhead is computed between cfi-split20 and cfi-split20-xmem.

benchmark	base	cfi-split20	cfi-split20-xmem
aha-mont64	4,983,621	6,311,849	15,316,543 (+142.66%)
crc32	4,159,221	5,348,729	13,370,535 (+149.98%)
edn	4,778,450	5,171,863	13,764,446 (+166.14%)
huffbench	3,931,101	4,786,819	14,124,502 (+195.07%)
matmult-int	2,808,107	2,990,772	7,946,961 (+165.72%)
nettle-aes	3,342,817	3,448,319	8,560,447 (+148.25%)
nettle-sha256	3,692,382	3,813,947	9,253,494 (+142.62%)
nsichneu	3,304,837	3,307,563	9,847,562 (+197.73%)
slre	4,510,608	6,144,376	16,942,249 (+175.74%)
statemate	5,212,495	5,939,445	19,456,443 (+227.58%)
ud	2,803,095	3,214,037	7,461,564 (+132.16%)

VII. Conclusion

In this paper we show a new way to deal with the stack of countermeasures thanks to authenticated encryption of both code and data for assurance of confidentiality and integrity at runtime, protection against code reuse attacks and stack overflows. This encryption structure allows data writes at byte granularity and the scheme enables efficient CFI by adding a second level of masks on the instructions and new instructions to manage them. In particular, data encryption supports indirect jumps without restrictions allowing all the dynamic programming subtleties offered by today's languages and OSes. An LLVM RISC-V compiler has been modified in this sense with an encryption step pushed back after the linker. This allows easy encryption of already compiled libraries. The FPGA proof of concept on a RISCY core which shows that the overhead in terms of code size and number of cycles is really dependent of the program and falls between 0% to 36% in our benchmark. The integrity of data and code forces us to double the memory occupancy. The increase in computation times by a factor of 2 to 3 is due to the latency introduced by the sole ASCON instance switching between data and code. The addition of caches or the duplication of ASCON will drastically improve this drawback. This encryption structure of both data and instructions gives us a glimpse of further possibilities for lightweight protection against replay attacks or spatial and temporal memory safety concerns.

Acknowledgments

The authors would like to thank the reviewers for their helpful comments. This work was funded thanks to the French national program "Programme d'Investissement d'Avenir IRT Nanoelec" ANR-10-AIRT-05.

References

[1] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, and M. Kallitsis, "Understanding the mirai botnet," in *26th USENIX Security Symposium*, 2017.

[2] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, 2012.

[3] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, 2011.

[4] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the ACM conference on Computer and communications security*, 2007.

[5] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the ACM conference on Computer and communications security*, 2004.

[6] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, 2009.

[7] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *USENIX Security Symposium*, 2014.

[8] M. Frantzen and M. Shuey, "Stackghost: Hardware facilitated stack protection.," in *USENIX Security Symposium*, vol. 112, 2001.

[9] T. Hiscock, O. Savry, and L. Goubin, "Lightweight instruction-level encryption for embedded processors using stream ciphers," *Microprocessors and Microsystems*, vol. 64, pp. 43–52, 2019.

[10] R. Lashermes, H. Le Bouder, and G. Thomas, "Hardware-assisted program execution integrity: Hapei," in *Nordic Conference on Secure IT Systems*, pp. 405–420, Springer, 2018.

[11] R. de Clercq, J. Götzfried, D. Übler, P. Maene, and I. Verbauwhede, "SOFIA: Software and control flow integrity architecture," *Computers & Security*, vol. 68, pp. 16–35, July 2017.

[12] M. Werner, T. Unterluggauer, D. Schaffenrath, and S. Mangard, "Sponge-based control-flow protection for iot devices," in *European Symposium on Security and Privacy (EuroS&P)*, IEEE, 2018.

[13] M. Seaborn and T. Dullien, "Exploiting the DRAM rowhammer bug to gain kernel privileges," *Black Hat*, vol. 15, p. 71, 2015.

[14] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-threshold RISC-V core with dsp extensions for scalable iot endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2017.

[15] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi, "The RISC-V instruction set manual, volume I: Base user-level isa version 2," *University of California, Berkeley.*, 2014.

[16] D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Transactions on information theory*, 1983.

[17] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, "Ascon v1. 2," *Submission to the CAESAR Competition*, 2016.

[18] "Newlib C library." https://sourceware.org/newlib/.

[19] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[20] "Embench: Open benchmarks for embedded platforms." https://github.com/embench/embench-iot.