PAC-PL: Enabling Control-Flow Integrity with Pointer Authentication in FPGA SoC Platforms

Gabriele Serra*, Pietro Fara*, Giorgiomaria Cicero*, Francesco Restuccia†, and Alessandro Biondi*
*Scuola Superiore Sant'Anna, Pisa, Italy
†University of California, San Diego, USA

***Abstract*—Control-flow integrity (CFI) is an effective technique to enhance the security of software systems. Processor designers recently started to provide hardware-based support to efficiently implement CFI, such as the pointer authentication (PA) feature provided by ARM starting from ARMv8.3-A processor architectures. These CFI mechanisms are also accompanied by support in the mainline codebase of popular compilers (such as GCC and LLVM) and the Linux operating system. As such, they are expected to establish as widespread security mechanisms. Nevertheless, many commercial chips still do not support hardware-assisted CFI, even some of the ones that just entered the market. This paper presents PAC-PL, a solution to enable hardware-assisted CFI on heterogeneous platforms that include a field-programmable gate array (FPGA) fabric, such as the Xilinx Ultrascale+ and Versal. PAC-PL comes with compiler- and OS-level support, is compatible with ARM's PA, and enables advanced key management and attack detection strategies. A timing analysis for PAC-PL is also presented. PAC-PL was experimentally evaluated with state-of-the-art benchmarks in terms of run-time overhead, memory footprint, and FPGA resource consumption, resulting in a practical solution for implementing CFI.**

I. Introduction

Security is a prominent requirement for modern software systems. The complexity of today's software, the integration of software components of disparate kinds, and the exposure to networks left room for several vulnerabilities in many software systems that, when properly exploited, were unfortunately used to accomplish malicious actions and cyber-crimes. Designers of embedded systems are required to pay particular attention to security for at least two major reasons. First, being embedded software typically used to control a physical system, attackers that take unauthorized control of the software can produce severe damages in the real world, e.g., think of passenger cars with automated driving capabilities that get hacked. Second, being embedded systems built with resource-constrained computing platforms, efficiency is almost always a must in the design and development of software. For this reason, languages such as C and C++ are the most used to develop embedded software due to their excellent balance between performance and programmability. As a matter of fact, for several embedded platforms, they are the only languages supported by the toolchain provided by the hardware vendor. Nevertheless, C and C++ are known to be memory-unsafe languages, thus exposing the system to a series of attacks [1] in the presence of vulnerabilities, named *memory corruption*, which are commonly inadvertently left in the code by programmers and are hard to detect at testing time.

Memory corruption vulnerabilities can be exploited to hijack the canonical execution flow of software processes by overriding part of their data in memory, such as pointers pushed into the stack. Code-reuse attacks (CRA) [2] are a modern example of attacks taking advantage of these vulnerabilities. CRA aim at manipulating the execution of a program modifying the control flow of a process by combining processor instructions already present in a system. Historically, CRA date back to 1997, when Peslyak [3] proposed the famous *return-to-libc*. Since 1997, researchers have been committed to contrast CRA by devising defense techniques.

Among the various techniques developed over the years to contrast CRA, one of the most effective is *control-flow integrity* (CFI). CFI aims to ensure that a process's execution flow always corresponds to the legal one specified at compile time. CFI is undoubtedly a powerful technique but is still scarcely applicable in practical scenarios, mainly due to the large overhead it requires to be implemented to ensure a complete CFI enforcement in any possible execution scenario. To make some CFI techniques implementable in practical applications, processor designers have started to offer hardware support. The leading example of this kind is the *pointer authentication* (PA) [4], [5] feature introduced by ARM in their ARMv8.3-A processor architectures. Intel also proposed a similar architectural extension called Control-flow Enforcement Technology (CET). Both these mechanisms are transparent to the programmer and are supported at the compiler and OS levels. For instance, the GCC and LLVM compilers and the Linux OS already support ARM's PA. Due to their availability in popular processors adopted in a huge number of systems and the corresponding compiler- and OS-level support, these CFI mechanisms are expected to establish as reference solutions for enhancing the security of software systems.

Nevertheless, to date, the number of system-on-chips (SoC) natively offering hardware-assisted CFI mechanisms is still very low. For instance, the widespread Xilinx Ultrascale+ SoC and even the newest Xilinx Versal, which has still to enter full production, do not offer the ARM's PA feature. Given that these SoC are going to stay on the market for a long time and have already ended (or will end) up in being used in many embedded systems, it is definitively essential to seek for solutions to efficiently enable CFI with these platforms.

Contribution. This paper presents PAC-PL, a solution that leverages the *field-programmable gate array* (FPGA) technology to efficiently enable CFI in FPGA-based SoC such as Xilinx Ultrascale+ and Versal. PAC-PL is compatible with ARM's PA, comes with its own compiler- and OS-level support for GCC and Linux, and can largely reuse the support for ARM's PA in Linux. Furthermore, PAC-PL enables improved key management and attack detection strategies with respect to ARM's PA. A timing analysis for PAC-PL is also proposed. PAC-PL is finally evaluated with state-of-the-art benchmarks in terms of run-time overhead, memory footprint, and FPGA resource consumption.

Paper structure. The rest of the paper is organized as follows. Section II provides background information on CFI and ARM's PA. Section III defines the problem addressed in this work. Section IV discusses the design and implementation of PAC-PL. Section V presents a timing analysis for PAC-PL. Section VI reports on our experimental evaluation. Finally, Section VII discusses the related work and Section VIII concludes the paper.

II. Background

A. Control-flow integrity

CFI is a term that is generally referred to a set of security countermeasures focused on enforcing that the execution flow of computational activities coincides with one of the legal paths defined by their canonical control-flow graph (CFG) [6]. It consists of a directed graph in which nodes represent routines or basic blocks and arcs indicate control transfer instructions. Arcs can be categorized as forward-arcs and backward-arcs. The former category encompasses control-flow transfers such as branches or function calls, while the latter incorporates returns from functions. Branches can be direct or indirect. Direct branches transfer the execution flow to a target destination computed at compile time. In contrast, indirect branches jump to a value computed at run-time and stored in a register or memory. Almost all CFI techniques watch over a program execution to ensure that the target of an indirect branch is the intended instruction. At each function return, they verify that the control comes back to the calling function. Direct branches and function calls are rarely supervised, especially if the code section is read-only. Since control-flow hijacking is essential in many exploits (e.g., those based on buffer overruns), independently of the exploited vulnerability [7], CFI techniques proved to be effective against several popular attacks and are considered among the most advanced security countermeasures.

B. CFI implementations

Until a few years ago, CFI techniques were almost considered unpractical because they were mostly implemented in software introducing considerable run-time overhead. Most recently, due to their relevance and effectiveness, leading chip manufacturers such as ARM and Intel started introducing hardware support to make CFI practically implementable with low overheads. Most relevant to us, in 2016, ARM announced the third version of their Cortex ARMv8-A microarchitecture (v8.3-A). Among other novelties with respect to the previous version, ARM introduced the support for *pointer authentication* (PA) [8]. PA is a hardware-assisted security feature that makes it harder for attackers to modify pointers in memory without being detected. Operating systems and compilers can jointly leverage PA to implement security countermeasures including CFI [5].

In a nutshell, PA works by cryptographically authenticating the content of a register before using it. Indeed, it is conceived as a protection against modification of code pointers such as return addresses stored in memory. For instance, PA represents a valuable protection mechanism to ensure that functions only return to legal locations as expected by the program according to the CFG, hence preventing stack overflow attacks.

C. Pointer authentication in ARMv8-A

1) Mechanism overview: In 64-bit architectures, not all the available bits are actually used to address memory (even the virtual one), as the available address space can be referenced with less than 64 bits. Typically, 48 bits are enough to address all the memory-mapped space. Therefore, the most significant part of 64-bit pointers is, in practice, unused. As a matter of fact, on an ARMv8-A Linux kernel (aarch64), only the least significant 48 bits are used. ARM's PA implementation uses part of the remaining (most-significant) 16 bits of memory addresses to store a *pointer authentication code* (PAC) [9], which is the means by which PA is implemented. In other words, PA works by embedding an authentication code, i.e., the PAC, within the authenticated pointer itself.

Technically speaking, the PAC is a cryptographic checksum obtained by truncating the output of the QARMA [10] algorithm, a lightweight tweakable block cipher. QARMA guarantees authenticity and integrity through tweaks, i.e., the permutation computed by the algorithm on the plaintext is determined by a secret key and an additional salt value. As such, the PAC is computed by elaborating three inputs: **(i)** the value of the memory pointer to be authenticated, **(ii)** a secret key, which is stored in dedicated processor registers, and **(iii)** a piece of context information used to discriminate where the authenticated pointer can actually be used. A relevant example of context information is the stack pointer, which binds the authenticated pointer to the stack frame of a certain function.

Signed pointers, i.e., pointers that include both the actual memory pointer and the PAC, must be authenticated before being used. Note that signed pointers cannot be directly used as they are because they would not be recognized as valid memory addressed due to the presence of the PAC in the most significant bits. The authentication process works by recomputing the PAC for the pointer, using the key and context information, and comparing the resulting PAC with one stored in the signed pointer. The creation of PACs and their authentication are illustrated in Figure 1.

2) ISA extension: A set of processor registers has been introduced in the ARMv8.3-A microarchitecture to store the keys required to create PACs. The current specification defines

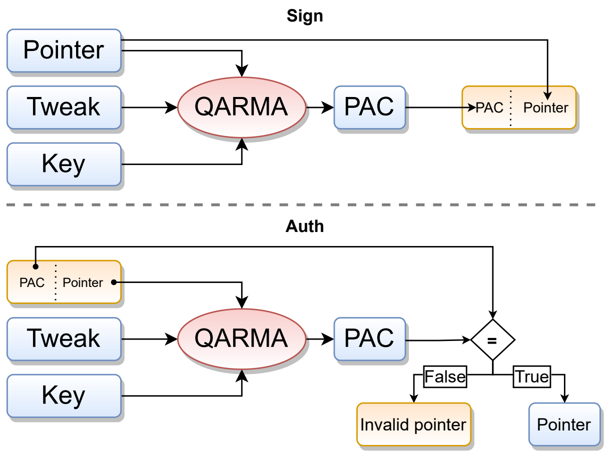

Fig. 1. Flowcharts of pointer signing and authentication processes.

five 128 bit registers for this purpose. Keys are also assigned a type named A or B. The actual semantic of a type-A or type-B key is left to the programmer. These registers are not accessible at EL0 but are also not tied to specific exception levels. Hence, to hide keys for certain exception levels, the software running at EL1, EL2, or EL3 is required to explicitly clear the content of the key registers [11]. In general, the whole key management process is left to the programmer. The keys must be ephemeral (for instance, the operating system can generate a key for each process at EL0).

The PA feature requires the addition of several instructions to the instruction set architecture (ISA) to accomplish the creation and authentication of PACs. Two sets of instructions have been introduced in ARMv8.3-A to serve this purpose. They are denoted by `PAC*` and `AUT*` for the creation and authentication of PACs, respectively.

`PAC*` and `AUT*` are sets of instructions that comprehend specialized opcodes to create PAC for instruction and data pointers using a certain key register. As an example, given an instruction address stored in the `Xd` register, the `PACIA Xd, Xn` instruction computes the PAC using the A key. The value contained in `Xn` is used as a QARMA tweak and constitutes the context information. The instruction also has the side effect of placing the resulting PAC in the upper part of `Xd`, which will hence contain the signed pointer. The signed pointer contained in `Xd` must then be authenticated using the `AUTIA` instruction. When the authentication succeeds, `AUTIA` restores the original pointer into `Xd`. Otherwise, if the verification fails, the PAC is replaced with a specific pattern that alters the pointer value to turn it into an illegal one that, if used, will generate a memory access exception [5]. Unfortunately, a PA-specific exception to be generated when the authentication of a pointer fails has been only recently introduced as part of the ARMv8.6-A architecture [12]. Without this exception, it is hence hard to distinguish a CFI violation from any other non-security-related illegal memory access, e.g., those due to a software bug.

3) PA to enforce CFI: The sets of `PAC*` and `AUT*` instructions can be used to enforce CFI. As a relevant example, note that the PA mechanism can be used to sign and authenticate the return address stored in the stack frame of a function. The function call convention on ARMv8-A works as follow: branch-and-link instructions (`BL` and `BLR`) allow transferring the execution to a given address and setting the link register (LR) to the address of the next instruction, i.e., the so-called return address. If the called function, in turn, calls another function, the return address is pushed to the stack (as it is typical in Intel x86 systems), and so on for the next functions. In other words, only leaf functions can avoid storing the link register value in the stack.

```
paciasp
stp     fp, lr, [sp, #-FRAME_SIZE]!
mov     fp, sp

; function body

ldp     fp, lr, [sp], #FRAME_SIZE
autiasp
ret
```

Fig. 2. Example of function with link register protected by ARM's PA.

PA instructions can hence be used to sign the return address stored in the link register before being pushed to the stack. The current stack pointer can be used as a context information, otherwise the same return address, which may be computed several times, would result in the same signed pointer that can be reused to accomplish attacks.

The ARMv8.3-A instruction set provides two instructions for this purpose: `PACIASP` and `AUTIASP`. They implicitly use LR as the register that contains the pointer to be signed and the stack pointer (SP) as the context information. For this reason, they are expected to be the most used PA instructions to implement CFI for standard function calls.

4) Compiler- and OS-level support: Both GCC and LLVM already support PA. For instance, with GCC 10 and Clang 12, the option `-mbranch-protection=standard` turns on PA for signing the return address of functions stored in the stack (i.e., leaf functions are not protected). Type-A keys (stored in the `AP*Key_EL1` registers) are used. Other options to tune the protection level are available. With this option enabled, the compiler produces a specific prologue and epilogue for non-leaf functions using PA instructions: an example is shown in Listing 2.

The Linux kernel offers a basic support for ARM's PA, both at user and privileged levels. When the kernel runs on a CPU that offers the PA instructions, the kernel assigns a set of random keys to each process. Specifically, each process has five random user keys (as mandated by the ARM's PA specification) and a random kernel key (stored in the A key). The set of keys is assigned when a process is created. All threads of a process share the same keys, which are also preserved after a `fork()`. When the scheduler of the kernel decides to preempt a process, its keys are stored into the process control block. On kernel entry, the kernel switches the user A key with the kernel A key and, viceversa, on kernel exit. Each PAC occupies 55−`VA_SIZE` bits, where `VA_SIZE` denotes the virtual address size configured by the kernel. The kernel can decide to disable the protection: in this case, `PAC*` and `AUT*` instructions are treated as NOPs.

III. Problem definition

Pointer authentication is an extremely relevant technique to enhance the security capabilities of software systems. Nevertheless, the hardware support to efficiently implement PA is available on ARMv8.3-A architectures only and even as an optional feature. Today, there exist many commercial-off-the-shelf (COTS) system-on-chips (SoC) that include processors based on older versions of the ARMv8-A architecture and, as such, do not dispose of hardware support for PA. These platforms are going to stay on the market for a long time and are already employed in several systems, which hence lack of a key feature to enhance the security of the software they execute. For instance, even the latest SoC by Xilinx, i.e., Versal, which has still to enter in full production, includes Cortex-A72 processors based on the ARMv8.0-A architecture only. In this work, we aim at supporting PA in ARMv8-A platforms that do not dispose of the corresponding hardware support. Fortunately, in heterogeneous platforms that couple processor cores with programmable logic (PL) implemented with field-programmable gate arrays (FPGA) technology, it is possible to deploy custom hardware devices to efficiently implement PA with minimal support from the software side. In this paper we explore this idea while also proposing enhancements to the official ARM's PA support.

A. Platform model

An FPGA SoC platform combines a *Processing System* (PS), including one or more processors (generally ARM-based), with a *Field-Programmable Gate Array* (FPGA) fabric. While software tasks (SW-tasks) are executed on processors, the FPGA fabric can host custom hardware devices such as *hardware accelerators* (HAs). The internal architecture of a typical FPGA SoC platform is illustrated in Figure 3. HAs are typically controlled by SW-tasks. HAs and processors share an off-chip DRAM memory, which is accessed through a memory controller embedded in PS. The communication between the processors and the HAs is allowed by the FPGA-PS and PS-FPGA interfaces (see the figure).

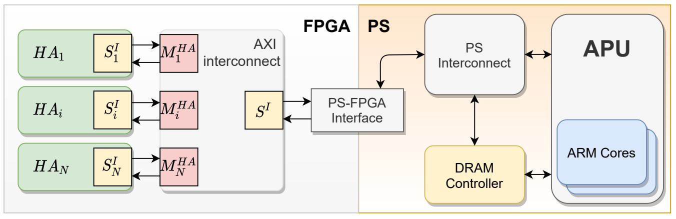

Fig. 3. Typical high-level internal architecture of FPGA SoC platforms. Only the components relevant to this work are illustrated.

B. Challenges in implementing PA using PL

Efficiently implementing PA employing PL requires facing with the following major challenges: (i) the FPGA clock frequency is slower than the one of the PS (100-300 MHz vs. 1-2 GHz): as such, the response times of the devices deployed in PL must be carefully optimized to ensure a proper cooperation between PL and PS; (ii) data transfers between PS and PL (e.g., pointers to be authenticated) require synchronized bus transactions that, if not properly handled, may introduce a considerable delay in the execution of SW-tasks; (iii) bus transactions may suffer interference from other bus masters (e.g., DMA, other cores).

C. Threat model

From the attacker perspective, we assume that an attacker (i) can read/write memory with a successful stack/heap overflow attack (e.g., controlling return addresses, function pointers or VTable pointers, etc.), (ii) disposes of a full knowledge of the process memory layout, (iii) has successfully bypassed address space layout randomization (ASLR), if present, and (iv) optionally, can even leak data from the task control block (TCB) stored into kernel memory in which the PAC key is saved. Note that assumption (iv) makes PA not secure when adopting the official ARM's PA support: this is because attackers capable of leaking the key can arbitrarily forge PACs and hence bypass CFI. The solution proposed in this work uses an improved key management strategy based on ARM TrustZone and hypervisor technologies, which is provided as an optional feature to strengthen the capabilities of PA.

IV. PAC-PL: Design & Implementation

This section presents PAC-PL, the proposed solution to enable PA in FPGA-based SoC with processors that do not include built-in hardware support for PA.

A. Design principles

PAC-PL has been designed according to the following design principles.

High efficiency. PAC-PL is conceived for resource-constrained embedded systems in which it is extremely important to contain run-time overhead and memory footprint. As such, our solution must be capable of (i) supporting low-latency data transfers between PL and PS, (ii) ensuring short response times in signing and authenticating pointers, and (iii) necessitating a minimal set of processor instructions to be controlled.

Lock-step execution. When dealing with software security, it is almost impossible to predict all possible ways with which a system can be attacked. For instance, even executing a few dozen of malicious instructions could lead to a privilege escalation. As such, PAC-PL must be capable of stopping a hijacked process as soon as a CFI violation is detected.

Key protection. Keys must be stored in such a way that it is not possible to access them from user-space so that they can be controlled only from privileged exception levels (the OS or a hypervisor). Finally, the writing of a new key or the reading of the generated PAC from the PL device must reset the last generated PAC, hence preventing data leakage.

Seamless integration with OS and compilers. To make PAC-PL practically usable in real-world systems, its design must be suitable for integration with popular OS such as Linux and compilers such as GCC. Considering that OS- and compiler-level support for PA has already been proposed, PAC-PL aims at retaining at much as possible the interface of the official ARM's PA mechanisms, hence requiring modicum changes to the software support that is already available. In particular, PAC-PL preserves the same strategy adopted by Linux for managing PA by assigning a different key to each process, having care of changing key at each context switch.

B. Hardware accelerator design and workflow

PAC-PL is based on a hardware accelerator deployed in PL that is composed of **(i)** *AXI-conf*, a device to interact with the system bus that exposes an AXI-lite subordinate interface to interact with the PS; and **(ii)** *QARMA-crypto*, a cryptographic core that implements the QARMA algorithm. Cores in the PS can interact with AXI-conf by simply acting on memory-mapper registers exposed as part of the AXI-lite interface. AXI-conf also exposes an interrupt signal to notify the PS. QARMA-crypto is then commanded by AXI-conf according to the configurations set in its registers. A graphical representation of the accelerator is reported in Figure 4.

TABLE I
REGISTERS OF THE PAC-PL INTERFACE.

Register	Offset	Bits	Kernel access	User access
KEY_LOW	0x0	64	R/W	-
KEY_HIGH	0x8	64	R/W	-
CTRL	0x10	64	R/W	-
PLAIN	0x1010	64	R/W	R/W
TWK	0x1018	64	R/W	R/W
CIPH	0x1020	64	R/W	R/W

For simplicity, PAC-PL only provides one key (the extensions required to support the fifth keys of the ARM's PA specification are trivial and not discussed here to unnecessarily complicate the presentation). The interface of the PAC-PL device in terms of memory-mapped registers is reported in Table I. The registers are split into privileged and non-privileged ones and defined as follows. **Privileged registers)** `KEY_LOW` and `KEY_HIGH` hold bits [63:0] and [127:64], respectively, of the 128-bit key used to generate PACs. `CTRL` is the control register of the accelerator. An interrupt is generated when `CTRL`[63] == 1. `CTRL`[63] has to be set to 0 to mark the interrupt as handled. The six least significant bits (`CTRL`[5:0]) specify the size of the generated PACs. **Non-privileged registers)** `PLAIN` and `TWK` hold the plaintext and tweak operands of the QARMA algorithm, respectively. `CIPH` provides the signed pointer after executing a signing operation and holds the signed pointer to be authenticated before executing an authentication operation. If the authentication failed, it holds zero. Note that privileged registers are mapped in a different memory page with respect to the non-privileged ones (with 4KB memory pages the displacement of the offsets is 0x1000). This is because privileged registers are intended to be mapped in the address space of the OS to control the configuration of the PAC-PL accelerator, while the others can be directly mapped in the address space of processes. In this way, the most frequent PAC-PL-related operations (signing and authenticating pointers) can be performed without the intervention of the OS (process can directly work on the PAC-PL accelerator registers), hence fostering efficiency.

The accelerator offers the same functionality provided by both `PAC*` and `AUT*` instructions from the ARM's official PA support. The interaction between the PS and the hardware accelerator for signing and authenticating pointers is illustrated in Figure 7 (for the case in which LR and SP processor registers are used) and discussed next.

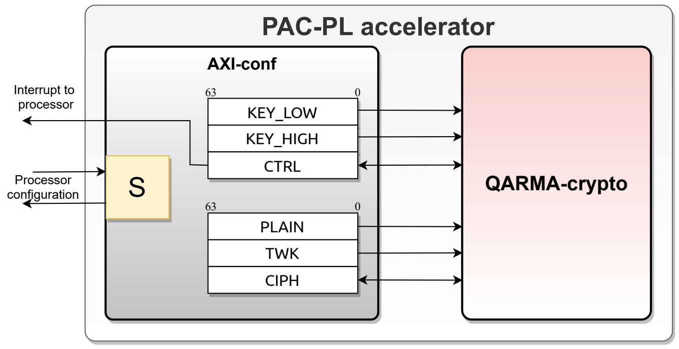

Fig. 4. The architecture of the PAC-PL accelerator.

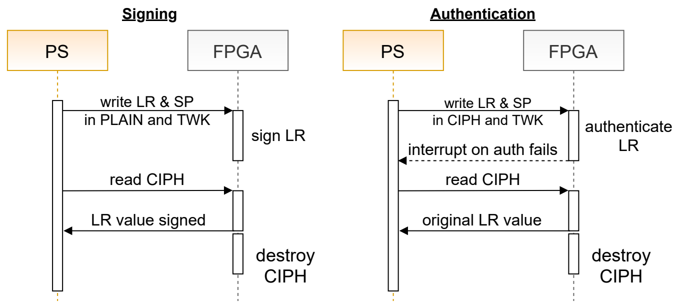

Fig. 5. Sequence diagrams of signing and authentication operations.

Signing a pointer. The processor writes the pointer to be signed and the context information in the plaintext and tweak registers of the PAC-PL accelerator, respectively. These operations trigger QARMA-crypto to start the generation of the PAC. The signed pointer (i.e., the pointer plus its PAC in the most significant bits) will be eventually made available to the processor in the cyphertext register together with the pointer. The reading of the latter register also clears the register. Note that a context switch may occur right after generating a PAC. In this case, the generated PAC will be reset to a null value and the processor will have to retry to sign the pointer.

When protecting the return address of functions, these operations have to be performed at the prologue of each non-leaf function. The corresponding ARMv8 assembly code is reported in Figure 6 for the case in which the stack pointer (SP) is used as a context information. The workflow of the signing process is also reported in the state machine of Figure 7(a).

```
 mov x10, #DEV_BASE
 mov x9, sp
0:
 ; write lr and sp in the device
 stp lr, x9, [x10, #DEV_PLAIN_OFFSET]
 ; read the signed pointer from the device
 ldr x11, [x10, #DEV_CIPH_OFFSET]
 ; retry if the signed pointer is null
 cbz x11, 0b
 ; place the signed pointer in the link register
 mov lr, x11

 ; former prologue
 stp fp, lr, [sp, #-FRAME_SIZE]!
 mov fp, sp
```

Fig. 6. Source code of a function prologue protected with PAC-PL. DEV_BASE denotes the base address of the PAC-PL accelerator registers, while DEV_PLAIN_OFFSET and DEV_CIPH_OFFSET denote the offsets of the plaintext and cyphertext registers, respectively.

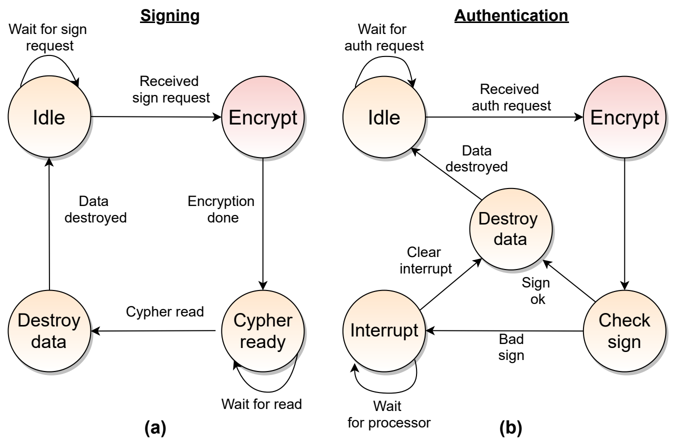

Fig. 7. State machines of the working flow of PAC-PL.

Authenticating a pointer. The processor writes the context information and the signed pointer in the tweak and cyphertext registers of the PAC-PL accelerator, respectively. These operations trigger the QARMA-crypto to start the re-generation of the PAC that, once computed, is eventually compared against the one in the signed pointer. If they correspond, the clean pointer (i.e., without the PAC) is made available in the cyphertext register. Otherwise, a null pointer is written in the cyphertext register, an interrupt signal is sent to the processor to notify a failed pointer authentication, and the accelerator is disabled (new pointers cannot be signed or authenticated). The processor finally reads the cyphertext register. In the case of a failed pointer authentication, the processor can detect the null pointer and busy wait to avoid making progress with a corrupted execution until the interrupt is served.

```
 ; former epilogue
 ldp fp, lr, [sp], #FRAME_SIZE
 ; end of former epilogue

 mov x10, #DEV_BASE
 mov x9, sp
1:
 ; write sp and lr in the device
 stp x9, lr, [x10, #DEV_TWK_OFFSET]
 ; read the authenticated pointer
 ldr x11, [x10, #DEV_CIPH_OFFSET]
 ; retry if the authenticated pointer is null
 cbz x11, 1b
 ; place the auth. pointer in the link register
 mov lr, x11

 ret
```

Fig. 8. Source code of a function epilogue protected with PAC-PL. DEV_BASE denotes the base address of the PAC-PL accelerator registers, while DEV_TWK_OFFSET and DEV_CIPH_OFFSET denote the offsets of the tweak and cyphertext registers, respectively.

The corresponding ARMv8 assembly code is reported in Figure 8 for the case in which the stack pointer (SP) is used as a context information. The workflow of the authentication process is also reported in the state machine of Figure 7(b).

The sign and authentication operations do not interfere. One PAC-PL accelerator per core must be provided. The virtual addresses of the accelerator's registers are always the same. When a process is migrated to another core, the virtual address space of the process must be reconfigured to match the corresponding PAC-PL accelerator.

C. Compiler-level support

1) Overview: We implemented compiler-level support for PAC-PL as a *plugin* for GCC, which offers a way to load custom modules that interact with the main module without modifying the core compiler source code. Likewise, the support for the ARM's PA, our plugin provides security protection transparent to the programmer, i.e., no code modifications are required. The plugin has the purpose of generating instructions at the prologues and epilogues of functions to sign return addresses. Technically speaking, our plugin works as an additional (late) register transfer logic (RTL) pass that analyzes the low-level RTL representation and protects a set of functions. The plugin works with all the latest GCC versions and will be released under a GPL license. The plugin makes heavy use of the API offered by GCC and consists of approximately 600 physical source lines of code (SLOC) written in C++ and ARMv8-A assembly.

2) Compiler passes and optimizations: GCC works by analyzing code in subsequent phases called passes and our plugin registers itself as a compiler pass. At each pass, the compiler performs some actions, such as abstracting the representation of a program, optimizing data structures, etc. Our plugin operates as a back-end pass at the very last stages of compilation where GCC builds the low-level intermediate representation of the program using the RTL. For each function produced by the compiler, the plugin decides if the function needs to be protected with PAC-PL according to two optimizations. The first one consists in excluding leaf functions from being protected. As highlighted in Section II, on ARMv8-A systems, return addresses are pushed on the stack only by non-leaf functions; hence, only return addresses of these functions end up being stored in memory and are therefore prone to corruption without hijacking the control flow. The second optimization consists in excluding functions that do not make use of arrays and that do not contain the `alloca` standard library function. This optimization is inspired by the logic of the GCC support for stack canaries, i.e., the one enabled by the popular option `-fstack-protector`[1]. Indeed, if a function does not use any array, the probability that it can be exploited to trigger a buffer-overflow is near zero.

D. Improved key management

Key management is one of the most vulnerable aspects of systems based on cryptography. In ARM's PA, key management is left to the programmer, which can hence introduce vulnerabilities in the security mechanism. For instance, in the current implementation of ARM's PA in Linux, whenever a process is created, the kernel generates a random key and stores it into the process control block. The process can sign and authenticate pointers using the key, but it cannot read the key itself (key registers are non-accessible from EL0). Assuming an attack scenario in which the attacker can leak kernel data (e.g., think of the famous Spectre and Meltdown attacks), the PA keys can be retrieved and used to craft valid PACs, hence opening for hijacks of the control flow.

To address this issue and provide a more robust structural support for key management, we also provide, as an optional feature, a software mechanism to more securely manage the keys of PAC-PL. Instead of storing PA keys in the process control blocks residing in the OS memory, we leverage the ARM TrustZone technology, which provides a hardware-isolated execution environment and memory storage named Secure World. Note that ARM TrustZone is available on all modern ARM processors. To ensure transparency in the implementation of the Linux PAC-PL support, which we want to be same with or without this optional key management feature, we require the presence of a hypervisor configured to trap all accesses to the PAC-PL key registers (this is easily possible because, by the design of PAC-PL, they are in a separate memory page with respect to the other registers). The mechanism works as follows (see also Figure 9): (i) when the kernel wants to write a new key K in the key register of the PAC-PL accelerator the access is trapped and emulated by the hypervisor running at EL2, (ii) the hypervisor then forwards the request to the Secure Monitor at EL3 through a secure monitor call (SMC), (iii) the Secure Monitor calls a trusted firmware running in Secure World (S-EL1) also forwarding K, and finally (iv) the trusted firmware associates K to the actual key K' according to a hash function writes K' in the key register of the PAC-PL accelerator, which can be directly accessed only from Secure World (the access is not trapped).

[1]This option protects functions with buffers larger than or equal to 8 bytes or containing the `alloca` function call [13].

Note that, in this way, any data leak in Non-Secure World cannot allow retrieving the actual key. To implement this improved key management, a trusted firmware that runs in S-EL1 is needed. The PAC-PL accelerator may be used to generate the hash of K: for this purpose, the accelerator allows a configuration to obtain a 64-bit hash. In our implementation, adding this feature resulted in a trusted computing base enlarged by about 80 lines of code.

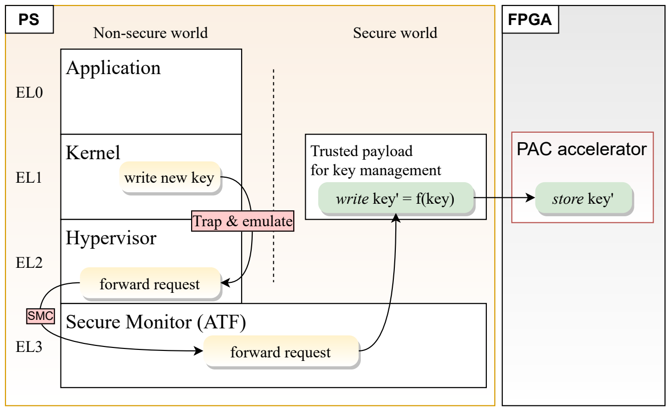

Fig. 9. Illustration of improved key management workflow of PAC-PL.

E. Attack detection

Until the recent introduction of the ARMv8.6-A architecture, the official ARM's PA support was not designed to easily detect attacks. Indeed, when the authentication of a pointer fails, the ARM's PA support modifies signed pointers to make sure that a translation fault (i.e., an illegal memory access) occurs when dereferencing the pointer. The corresponding processor exception is hence hard to be distinguished from any other illegal memory access, e.g., due to a software bug.

PAC-PL overcomes this issue by explicitly generating an interrupt signal whenever the authentication of a pointer fails. The kernel is then in charge of handling the interrupt by a corresponding service routine and taking decisions about the compromised process (e.g., to terminate the process, which is our default option). This feature has been implemented in Linux with a custom kernel module. Note that the availability of this interrupt signal is also helpful to take higher-level decisions in a system. For instance, if a hypervisor is present, it is possible to monitor the frequency of the PAC-PL interrupts and decide to disable or reset a virtual machine under attack to avoid more severe consequences.

V. Timing analysis

This section presents a timing analysis to bound the time required to sign and authenticate pointers with PAC-PL, hence favoring its adoption in real-time embedded systems. Due to space limitations, the analysis considers a single PAC-PL accelerator deployed in PL. Its extension to multiple accelerators is left to future work and can be performed by building upon state-of-the-art bus analysis techniques such as [14] and [15]. The analysis also does not consider the time to fetch the instructions to interact with the PAC-PL accelerator, i.e., it refers to the case of hot caches. This is because bounding (cache-related) instruction fetching delays is a general problem that is independent of this work: the interested reader can refer to the survey of Maiza et al. [16].

A. Architecture model

1) Processing system: The PS-FPGA interface exports multiple ports enabling the PS to communicate with the PL. The interfaces are based on the AMBA AXI standard, which is the de-facto standard for communications in modern FPGA SoCs [17]. Typically, communications within the PS are handled by a multi-level AXI-based interconnect that connects the resources in the PS with the DRAM memory and the FPGA fabric. Taking the Xilinx Ultrascale+ SoC as a reference platform, we identified the path for the communication between the processor and the PAC-PL accelerator and analyzed the delay components along that path. To cope with these delays, we define: **(a)** $d^{\text{PS}}_{\text{AW}}$, the worst-case propagation time from the issue of an address write request by the processor interacting with the PAC-PL accelerator and the arrival of the address request to the PS-FPGA interface; **(b)** d^{PS}_{W}, the worst-case propagation time experienced by a write data word issued by the processor to reach the PS-FPGA interface; **(c)** d^{PS}_{B}, the worst-case propagation time experienced by a write response from the PS-FPGA interface to the processor issuing the corresponding write request; **(d)** $d^{\text{PS}}_{\text{Int}}$, the worst-case propagation time required to propagate the interrupt generated by the PAC-PL accelerator from the PS-FPGA interface to the processors; **(e)** $d^{\text{PS}}_{\text{AR}}$, the worst-case propagation time elapsed for the propagation of an address read request from a processor to the PS-FPGA interface; and **(f)** d^{PS}_{R}, the worst-case propagation time experienced in the propagation of a read data word from the PS-FPGA interface to a processor.

2) FPGA interconnect: Once propagated through the PS, the transactions directed to the PAC-PL accelerator issued by a processor reach the PS-FPGA interface. From this point on, the requests are propagated by the FPGA interconnect, until reaching the accelerator. The FPGA interconnect influences the worst-case response time of transactions. Similarly to what introduced before, it is required to define other delay terms: **(a)** $d^{\text{FPGA}}_{\text{AW}}$, the worst-case propagation time of an address write request from the PS-FPGA interface to the PAC-PL accelerator; **(b)** $d^{\text{FPGA}}_{\text{W}}$, the worst-case propagation time experienced in the propagation of a write data word from the PS-FPGA interface to the PAC-PL accelerator; **(c)** $d^{\text{FPGA}}_{\text{B}}$, the worst-case propagation time for the propagation of a write response from the PAC-PL accelerator to the PS-FPGA interface; **(d)** $d^{\text{FPGA}}_{\text{Int}}$, the worst-case propagation time to propagate the interrupt generated by the PAC-PL accelerator to the PS-FPGA interface; **(e)** $d^{\text{FPGA}}_{\text{AR}}$, the worst-case propagation time for the propagation of an address read request from the PS-FPGA interface to the PAC-PL accelerator; **(f)** $d^{\text{FPGA}}_{\text{R}}$, the worst-case propagation time to propagate a read data word from the PAC-PL accelerator to the PS-FPGA interface.

3) PAC-PL accelerator: The requests issued by the processor to the accelerator are managed by AXI-conf (see Section IV). Following the AXI standard, each received write request is replied with a write response acknowledging the processor. AXI-conf requires a few clock cycles to generate such a response: we define t^{Conf}_{W} as the maximum delay it introduces to generate such a write response after receiving a write request and the corresponding data word. Each sign and authentication operation requires the execution of the QARMA algorithm on the input data. This phase is called *QARMA-crypto execution* and is performed in at most t^{QARMA} time units. The processor can read results from the PAC-PL accelerator by issuing read requests. Such requests are managed by AXI-conf, which, after receiving the request, provides the requested data word on the external bus of the accelerator after at most t^{Conf}_{R} time units. The time required to compare PACs during a pointer authentication is bounded by t^{Check}, while the time to generate the interrupt to notified a failed authentication is bounded by t^{Int}.

B. Bounding the response time of the PAC-PL accelerator

Lemma 1. ***(Write transactions)*** *The delay experienced by a processor to write a register in the PAC-PL accelerator is bounded by:*

$$t^{PAC}_{W} = max(d^{PS}_{AW}, d^{PS}_{W}) + max(d^{FPGA}_{AW}, d^{FPGA}_{W}) + t^{Conf}_{W} + d^{FPGA}_{B} + d^{PS}_{B}. \quad (1)$$

Proof. The interaction starts with the issue of a write request AW from the processor dropped in the PS interconnect. Following the AXI standard, AW is immediately followed by the data word W to be written. From the model of Sec. V-A, the worst-case propagation time for AW and W to reach the FPGA subsystem are $d^{\text{PS}}_{\text{AW}}$ and d^{PS}_{W}, respectively. Since AXI-based interconnections propagate address and data in parallel, both of them are available at the PS-FPGA interface in at most the maximum delay between $d^{\text{PS}}_{\text{AW}}$ and d^{PS}_{W}. The same consideration follows for the FPGA interconnect – after at most $max(d^{\text{FPGA}}_{\text{AW}}, d^{\text{FPGA}}_{\text{W}})$ the address request and the corresponding data word are available at the AXI port of the PAC-PL accelerator. Still from the model of Sec. V-A, once the request and data are received, the PAC-PL accelerator replies with a write response B after at most t^{Conf}_{W} time units. B is propagated through the FPGA interconnect in at most $d^{\text{FPGA}}_{\text{B}}$ time units and eventually propagated through the PS interconnect in at most d^{PS}_{B} time units, finally reaching the processor. The lemma follows by summing up these contributions. □

Lemma 2. ***(Read transactions)*** *The delay experienced by a processor to read a register in the PAC-PL accelerator is bounded by:*

$$t^{PAC}_{R} = d^{PS}_{AR} + d^{FPGA}_{AR} + t^{Conf}_{R} + d^{FPGA}_{R} + d^{PS}_{R}. \quad (2)$$

Proof. The interaction starts with the issue of a read request AR from the processor dropped in the PS interconnect. The worst-case propagation time for AR to reach the FPGA subsystem is $d^{\text{PS}}_{\text{AR}}$. Once AR is available at the PS-FPGA interface, the propagation time in the FPGA interconnect to reach the AXI port of the PAC-PL accelerator is upper bounded by $d^{\text{FPGA}}_{\text{AR}}$. Once AR reaches the accelerator, the requested data word R is available at its external interface after, at most, t^{Conf}_{R} time units. R is propagated through the FPGA interconnect in at most $d^{\text{FPGA}}_{\text{R}}$ time units and through the PS interconnect in at most d^{PS}_{R} time units, until reaching the processor. The lemma follows by summing up these contributions. □

By profiling our implementation, we found that the following inequalities hold (details are provided in Section VI-A):

- $t^{\text{QARMA}} + t^{\text{Check}} < d^{\text{PS}}_{\text{AR}} + d^{\text{FPGA}}_{\text{AR}}$;
- $t^{\text{Int}} + d^{\text{FPGA}}_{\text{Int}} + d^{\text{PS}}_{\text{Int}} < t^{\text{PAC}}_{R}$.

These observations allow bounding the worst-case time required to either sign or authenticate a pointer as follows.

Lemma 3. *The response time of a PAC-PL operation, i.e., either signing or authenticating a pointer, is bounded by:*

$$t^{OP} = 2 \cdot t^{PAC}_{W} + t^{PAC}_{R}. \quad (3)$$

Proof. **Signing:** To issue a request for signing a pointer, the processor needs to write the plaintext and the tweak, each written to a corresponding register of the PAC-PL accelerator. From Lemma 1, each write transaction is upper bounded by t^{PAC}_{W}. Thus, in the worst-case scenario, the setup of the PAC-PL is bounded by $2 \cdot t^{\text{PAC}}_{W}$. From the model of Sec. V-A, QARMA-crypto executes in t^{QARMA} time units. The processor reads the result by issuing a read request. As from lemma 2, the response time of such a request is bounded by t^{PAC}_{R}. Since the propagation time of the read request ($d^{\text{PS}}_{\text{AR}} + d^{\text{FPGA}}_{\text{AR}}$) is larger than t^{QARMA} and noting that the read request is issued after the end of the two write transactions, the processor is guaranteed to read the signed pointer after at most t^{PAC}_{R} time units.

Authentication: To issue a request for authenticating a pointer, the processor needs to write the cyphertext and the tweak to the corresponding registers of the PAC-PL accelerator. Again, the time for each write transaction is bounded by t^{PAC}_{W}. Following the two write transactions, the processor issues a read transaction to read the result of the authentication. The response time of the read transaction is bounded by t^{PAC}_{R}. Since ($t^{\text{QARMA}} + t^{\text{check}}$) is lower than $d^{\text{PS}}_{\text{AR}} + d^{\text{FPGA}}_{\text{AR}}$, and noting that the read request is issued after the completion of the write request, it is guaranteed that, when the read request reaches the PAC-PL accelerator, the authentication is completed. Moreover, since $t^{\text{Int}} + d^{\text{FPGA}}_{\text{Int}} + d^{\text{PS}}_{\text{Int}}$ is lower than t^{PAC}_{R}, it is guaranteed that, when the data word corresponding to the read transaction reaches the processor, the interrupt has already reached the interrupt controller. Hence the latter operations last at most t^{PAC}_{R} in total. The lemma follows. □

VI. Evaluation

This section presents the results of a set of experiments that were conducted to assess the performance of PAC-PL in terms of run-time overhead, memory footprint, and resource consumption of the accelerator. The ZCU102 evaluation board by Xilinx, which is equipped with Zynq UltraScale+ XCZU9EG SoC, was taken as a reference platform. A security evaluation of PAC-PL is also briefly discussed at the end of the section.

At the time of writing, and to the best of our knowledge, there are only two SoC on the market that provide hardware support for ARM's PA: A12/A13 Bionic by Apple and Kirin 980 from HiSilicon. Unfortunately, no off-the-shelf evaluation boards for these SoC are available. Hence, a comparison between PAC-PL and the stock ARM's PA support was not possible. To evaluate PAC-PL, we used a set of applications from the TACLeBench suite [18] and the SanDiego CortexSuite [19]. From both benchmark suites we excluded those applications that were not suitable for being directly compiled for ARMv8-A. From the TACLeBench suite we excluded those applications that do not include functions to be protected. Furthermore, from the CortexSuite we excluded those applications that make use of randomized algorithms to reach convergence, which were not suited to enable a performance comparison by repeating their execution. Each selected benchmark was cross-compiled using the GCC ARM 10.2 compiler, with and without our plugin for PAC-PL. The plugin was also responsible for mapping the portion of physical memory that includes the non-privileged PAC-PL registers in the process virtual address space.

PAC-PL has been implemented using the VHDL language. The QARMA-crypto Core makes use of a VHDL implementation of the QARMA algorithm developed by Werner et al. [20]. The synthesis of the PL accelerator was carried out using Xilinx Vivado 2020.2. The resource consumption of the PAC-PL accelerator is reported in Table II. As it can be noted, the accelerator requires a very limited amount of resources.

As a baseline for comparison, we compared our approach against a software-only CFI implementation. In order to perform a fair evaluation that considers the same threat model and the implied security level, we compared against a software-only PA implementation, referred to as PAC-SW in the following, which provides the same security capabilities of PAC-PL. It was realized as a pluggable kernel module that implements the sign and authentication state machines. The module and the software version of QARMA were written using the C language.

When the module is loaded into the kernel, a new misc-device is made available under `/dev`. Processes cannot directly read or write in the device; rather, they need to map the device in their virtual address spaces as an additional *virtual-memory area* by means of the `mmap` syscall. As for PAC-PL and the ARM's PA implementation, note that PAC-SW still guarantees that user-space processes cannot access PA keys.

The benchmarks were executed on Petalinux 2021.1. Their execution times were collected using a custom tool written in C that forks its execution to run each benchmark (*fork()* and *execvp()* were used). The tool gets the system timestamp by *clock_gettime()* from the *time.h* (with `CLOCK_MONOTONIC`) library just before and after the fork is performed. Each bench-

TABLE II
Resource consumption of PAC-PL compared with the cost of an FPGA interconnect and the total available resources on the Zynq UltraScale+ XCZU9EG (ZCU102 Board) and the AES-ZU3EG (Ultra96 Board)

Target: ZCU102	**TOTAL**	**PAC-PL**	**FPGA INTC**
CLBs	274080	2068 (0.8%)	2617 (1%)
FFs	548160	919 (0.1%)	3049 (0.6%)

Target: Ultra96	**TOTAL**	**PAC-PL**	**FPGA INTC**
CLBs	70560	2068 (2.9%)	2617 (3.7%)
FFs	141120	919 (0.7%)	3049 (2.2%)

mark was executed 1000 times, dropping out the outlier samples with values over the 95^{th} percentile (due to benchmark-unrelated interference introduced by Linux services).

The results for the TACLeBench suite are reported in Table III. The second and third columns of the table report the mean execution times without and with the PAC-PL protection, respectively. The fifth column reports the number of protected functions executed per second when continuously running the benchmark. Percentage overheads are reported in the sixth column of the table. Interestingly, the overhead introduced by PAC-PL in some benchmarks is so small that it was not detected by our measurements (same execution times for the protected and non-protected versions). These benchmarks are assigned a 0% run-time overhead in the table. The last column reports the percentage overhead (OH) in terms of memory footprint. Notably, we found that a function protected by our plugin increases its footprint by 48 bytes. Figure 11 graphically illustrates the percentage run-time overhead of the benchmarks: note that 21 out of 25 of them have overhead below 10% and the average overhead introduced by PAC-PL is about 16.65%. The overhead clearly depends on the number of functions that are protected by PAC-PL: the more the larger. Table III also reports the mean execution times with PAC-SW (fourth column) alongside the corresponding overhead, expressed with a slowdown factor, with respect to the case without PA (seventh column). As it can be noted from the table, the implied execution times are definitively prohibitive.

Table IV reports the same kind of results for the San Diego CortexSuite, where PAC-PL exhibits even better results (although three applications do not include functions to be protected).

The improved key management strategy presented in Section IV-D was implemented using CLARE, a type-1 real-time hypervisor [21]. The round-trip time that elapses from the trap of the access to the PAC-PL key registers, passing by the execution of the trusted payload in Secure World, and coming back to the execution in Linux was profiled on the Zynq Ultrascale+. The results are reported in Table V. As it can be noted from the table, the introduced overhead is compatible with the sporadic usage of key registers (remember that they are accessed at context switches).

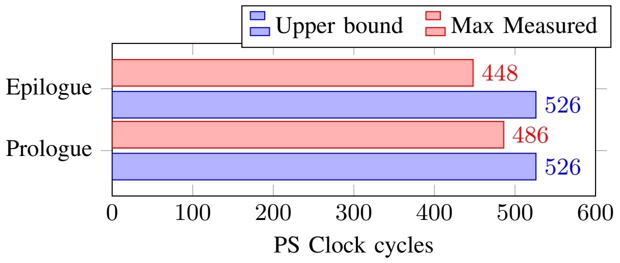

Fig. 10. Comparison of maximum measured times to execute the function prologues and epilogues with respect to the bound by analysis.

A. Evaluation of response-time bounds

The delays introduced by the PAC-PL accelerator and the FPGA interconnect can be effectively analyzed by means of a System Integrated Logic Analyzer (System ILA) [22] deployed in PL to probe a set of relevant signals. Such technology cannot, unfortunately, be leveraged for profiling the propagation delays introduced by the PS. This is because the PS is made of hard silicon, hence no custom logic can be deployed to probe signals. To face this challenge, we deployed a custom hardware profiler in PL conceived to introduce a known and constant delay in serving each of the transactions issued by the processors. We then issued 1000000 read requests and 1000000 write requests to the profiler and measured the response times. Thanks to the predictability in the response time guaranteed by our profiler and knowing the profiled propagation delays introduced by the FPGA interconnect measured with the System ILA, we were able to profile the overall maximum propagation delays introduced by the PS, defined as $d^{\text{PS}}_{\text{Write}} = max(d^{\text{PS}}_{\text{AW}}, d^{\text{PS}}_{\text{W}}) + d^{\text{PS}}_{\text{B}}$ and $d^{\text{PS}}_{\text{Read}} = d^{\text{PS}}_{\text{AR}} + d^{\text{PS}}_{\text{R}}$, respectively, for read and write transactions. The profiled results are summarized in Table VI and allowed instantiating the timing analysis of Section V.

The execution times of both the function prologue and epilogue required by PAC-PL (as generated by our GCC plugin) were also profiled by running them 1000000 times. These measurements were performed from a bare-metal firmware (still running on one of the Cortex-A cores of the XCZU9EG) to avoid the typical interference generated by Linux. The measurements were collected with hot caches (i.e., pre-fetching the prologue and epilogue instructions) as considered by our analysis. Figure 10 compares the maximum measured execution times to the upper bounds given by the analysis of Section V. As it can be noted from the figure, the analysis correctly bounds all the collected measurements.

B. Security evaluation

PAC-PL preserves the same security capabilities of the ARM's PA implementation. Attacks based on guessing, forging, or substituting PACs that are effective against ARM's PA may hence also be effective against PAC-PL. Nevertheless, it is important to note that the improved key management strategy presented in Section IV-D and the attack detection mechanism discussed in Section IV-E improve the security capabilities of

TABLE III
TIMING PERFORMANCE OF THE BENCHMARKS FROM THE TACLEBENCH SUITE WITH AND WITHOUT THE CFI SUPPORT.

Bench. name	no PAC (s)	PAC-PL (s)	PAC-SW (s)	Prot. Fun/s	OH PAC-PL (%)	OH PAC-SW (x)	Footprint overhead
ammunition	0,187299298	0,264061311	8763,164772	439758	40,98%	46.787,0x	0,10%
anagram	0,005106490	0,011227707	441,5337322	520320	119,87%	86.465,2x	0,02%
audiobeam	0,001933759	0,001950937	1,169154285	8714	0,89%	604,6x	0,03%
basicmath	0,003566405	0,003843802	19,60050957	67381	7,78%	5.495,9x	0,02%
bitcount	0,000588526	0,000586522	0,173536739	3410	0,00%	294,9x	0,00%
cjpeg_wrbmp	0,000764096	0,000764105	0,190194407	1309	0,00%	248,9x	0,01%
crc	0,000619830	0,000620693	0,173483114	3222	0,14%	279,9x	0,01%
filterbank	0,013577159	0,013581470	0,110793535	74	0,03%	8,2x	0,01%
fmref	0,001157648	0,003543533	171,2919319	641732	206,10%	147.965,5x	0,05%
gsm_dec	0,003692272	0,003839493	10,13933127	36463	3,99%	2.746,1x	0,05%
gsm_encode	0,010304174	0,010574756	19,68282736	24587	2,63%	1.910,2x	0,00%
huff_dec	0,000932836	0,000941550	0,190190033	1062	0,93%	203,9x	0,01%
huff_enc	0,001847069	0,001832258	0,174772603	1092	0,00%	94,6x	0,75%
insertsort	0,000557924	0,000557362	0,097826635	1794	0,00%	175,3x	0,00%
lms	0,000944936	0,000945996	0,098110963	1057	0,11%	103,8x	0,01%
ludcmp	0,000571426	0,000575620	0,097950857	1737	0,73%	171,4x	0,01%
md5	0,030401856	0,039176703	638,1477887	215638	28,86%	20.990,4x	0,03%
minver	0,000565801	0,000568674	0,097864848	1758	0,51%	173,0x	0,75%
mpeg2	0,313437533	0,312925333	27,10033014	1125	0,00%	86,5x	0,01%
ndes	0,000666498	0,000685293	1,303094601	24807	2,82%	1.955,1x	0,02%
powerwindow	0,005396462	0,005393379	0,329849098	742	0,00%	61,1x	0,04%
rijndael_dec	0,010042055	0,010033722	0,181708348	199	0,00%	18,1x	0,02%
rijndael_enc	0,009535291	0,009530081	0,182435988	210	0,00%	19,1x	0,02%
sha	0,004567239	0,004562490	0,101900501	219	0,00%	22,3x	0,01%
susan	0,149121310	0,140817085	7,283510958	28	0,00%	48,8x	0,02%

TABLE IV
TIMING PERFORMANCE OF THE BENCHMARKS FROM THE SAN DIEGO CORTEXSUITE WITH AND WITHOUT THE CFI SUPPORT.

Bench. name	no PAC (s)	PAC-PL (s)	PAC-SW (s)	Prot. Fun/s	OH PAC-PL (%)	OH PAC-SW (x)	Footprint overhead
liblinear_small	1,030671923	1,038614622	2,426655875	47	0,77%	2,4x	0,07%
motion-est_small	0,200724591	0,199959897	203,3306641	5606	0,00%	1.013,0x	0,07%
pca_small	1,623647403	1,623836179	1,623926245	0	0,00%	1,0x	0,09%
rbm_small	0,068965434	0,068953107	0,068952105	0	0,00%	1,0x	0,09%
sphinx_small	0,237910051	0,304823469	18343,94837	850541	28,13%	77.104,6x	0,02%
svd3_small	0,310233613	0,310193739	9,233728778	10	0,00%	29,8x	0,00%
kmeans_small	0,626327233	0,626401080	1,798776152	2	0,01%	2,9x	0,00%
spectral_small	41,45806728	41,45453931	42,10983737	0	0,00%	1,0x	0,00%

TABLE V
MAXIMUM AND AVERAGE ROUND-TRIP DELAYS WITH HYPERVISOR-BASED KEY MANAGEMENT (IN NANOSECONDS)

Max	Max 99th perc.	Max 95th perc.	Avg
8102	2248	2188	2056

TABLE VI
MAXIMUM DELAYS PROFILED ON THE ZYNQ ULTRASCALE+ XCZU9EG (IN PS CLOCK CYCLES).

$d_{\mathbf{AW}}^{\mathbf{FPGA}}$	$d_{\mathbf{W}}^{\mathbf{FPGA}}$	$d_{\mathbf{B}}^{\mathbf{FPGA}}$	$d_{\mathbf{AR}}^{\mathbf{FPGA}}$	$d_{\mathbf{R}}^{\mathbf{FPGA}}$	$d_{\mathbf{Write}}^{\mathbf{PS}}$	$d_{\mathbf{Int}}^{\mathbf{FPGA}}$
29	20	20	29	20	116	5

$d_{\mathbf{Read}}^{\mathbf{PS}}$	$t_{W}^{\mathbf{Conf}}$	$t_{R}^{\mathbf{Conf}}$	$t^{\mathbf{QARMA}}$	$t^{\mathbf{Check}}$	$t^{\mathbf{Int}}$	$d_{\mathbf{Int}}^{\mathbf{PS}}$
92	20	15	10	5	5	40

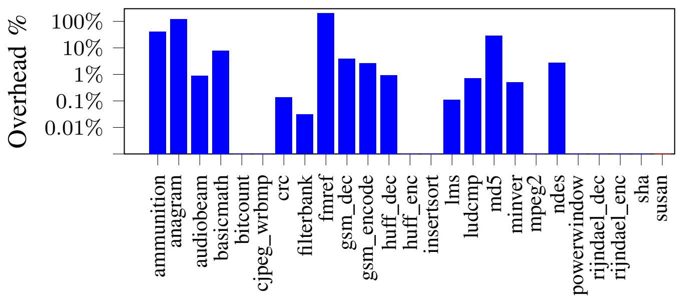

Fig. 11. Percentage run-time overhead for each benchmark. Note that no measurable overhead was recorded for some benchmarks.

today's ARM's PA implementation and its software support in Linux.

Protection against kernel data leakages. In the Linux support for ARM's PA, a key is generated for each process and stored into the process data structure. If an attacker is capable of reading the kernel memory the PAC keys could be stolen. With the improved key management strategy of PAC-PL, a kernel data leakage is not sufficient to stole the PAC keys. Indeed, the attacker must have access to the Secure World to either retrieve PAC keys as a result of the hash function running at S-EL1 or access the PAC-PL device registers.

Protection against attempts at forging PACs. Attacks that aim at forging PACs may require a high number of attempts before guessing the right PAC. For instance, when using a software model in which a master process spawns several workers by means of the `fork()` system call, the kernel creates an exact copy of the process, which in the today's Linux support for ARM's PA also means transferring the key to the new process. If an attacker controls the worker, the attacker can try to forge PAC with unlimited trials (e.g., workers reply to remote requests and, when they crash, are replaced by newer workers). In this scenario, PAC-PL easily allows detecting an excessive number of failed PAC authentications by means of the interrupt signal that is generated for each failed authentication. For instance, the signal can be used by a hypervisor to shutdown a Linux virtual machine under attack, hence avoiding that the attack can propagate to more critical software modules.

Limitations of this work. The PAC-PL accelerator can be used to implement all the functionality available in ARM's PA implementation for ARMv8.3-A architectures. The prototype GCC plugin developed for this work is however limited to the protection of return addresses only, but it can be easily extended to cope with other attack means such as function pointers.

VII. Related work

Until a few years ago, CFI techniques were limitedly considered due to their significant overhead when lacking adequate hardware support. Nevertheless, CFI techniques have been investigated and implemented with a pure-software approach since years. One of the first implementations within an OS was released for Windows 8.1 in 2014 by Microsoft [23], which focused on protecting indirect calls on invalid targets only. This solution, however, is entirely realized in software; consequently, the performance of a protected program experiences a considerable slowdown. Furthermore, their implementation was not robust versus a series of attacks [24]. Targeting RTOSes, Walls et al. [25] proposed a software solution named RECFISH that, differently from our solution, targets Cortex-R microcontrollers and FreeRTOS and implements CFI through the use of a privileged shadow-stack. RECFISH patches precompiled binaries to add security instrumentation and inserts a trampoline to jump to checks to enforce CFI. However, being the shadow-stack privileged, performing operations onto it requires RECFISH to jump into a privileged mode. Benchmarks show that the execution time of indirect branches considerably increases, with an approximately 30% increase in the total execution time [25].

Christou et al. [26] extended the ISA of the Leon3 SPARC V8 processor, a 32-bit open-source synthesizable processor, by adding several instructions to check addresses during function calls and returns. Their approach was implemented and tested over SpecInt2000 benchmarks and was found to introduce a limited runtime overhead (under 5%). Nevertheless, their approach requires a hardware extension at the level of the processor, which is practically possible only when using particular soft cores. Differently, our approach complements the processor capabilities with a module in PL.

Another remarkable work that leveraged FPGAs to implement CFI was carried out by Maunero et al. [27]. They presented a mechanism to secure bare-metal programs running on FPGA-based platforms. In their approach, each program to be executed is analyzed to extract the CFG. Then, a compiler is required to generate a unique basic-block ID for each control-flow transfer to be sent to the FPGA during the execution where a device monitors each ID received, setting a timeout. The processor is interrupted if the subsequent ID is not expected or arrives after the timeout. While the performance is outstanding, this solution comes with some notable disadvantages. First, the configuration of the timer may be complex to balance false and true positives. Second, this approach is unsuitable for dynamic tasks because the edge table must be present in the FPGA memory before the process creation. Third, differently from our approach, it is a custom technique that is not compatible with any PA support in OS and compilers. From the introduction of PA support in ARM processors, several works leveraged this security mechanism. Denis-Courmont et al. [28] jointly applied PA and a key management algorithms for protecting the Linux kernel compiled for ARM platforms. Liljestrand et al. [29] proposed a scheme for signing pointers that enforces integrity for code and data pointers coupled with a runtime safety mechanism that contrasts pointer-replacement attacks. The work was later extended [30] to include a mechanism based on chained message authentication codes to improve the system security without requiring additional hardware support. Ferri et al. [31] targeted virtualized systems and proposed a hypervisor-based solution to improve key management and detect attacks using ARM's PA. To some extent, this work tries to extend the preliminary results of Ferri et al., providing an FPGA-based hardware support for some of their original proposals and an evaluation.

VIII. Conclusions and future work

This paper presented PAC-PL, a hardware-assisted solution to enable CFI in FPGA-based platforms that lack of the ARM's PA support. PAC-PL is accompanied by compiler- and OS-level support for GCC and Linux and enables improved key management and attack detection strategies, with respect to ARM's PA, using ARM TrustZone and hypervisor technologies. A timing analysis for PAC-PL was also presented. Experimental results with state-of-the-art benchmarks showed that the approach can practically enable CFI with small runtime overheads (85% of the tested benchmarks showed a percentage overhead less than 10%) and negligible additional footprint. Furthermore, experiments demonstrated the quality of the analytical bounds and showed that PAC-PL accelerator has minimal FPGA resource requirements ($< 1\%$ on a Xilinx XCZU9EG). Future work includes the enhancement of the software ecosystem for PAC-PL and the support of other CFI mechanisms such as Branch Target Identification (BTI) introduced in ARMv8.5-A processors [32].

References

[1] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos, "Memory Errors: The Past, the Present, and the Future," in *Research in Attacks, Intrusions, and Defenses*, D. Balzarotti, S. J. Stolfo, and M. Cova, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 86–106.

[2] M. Kayaalp, M. Ozsoy, N. A. Ghazaleh, and D. Ponomarev, "Efficiently Securing Systems from Code Reuse Attacks," *IEEE Transactions on Computers*, vol. 63, no. 5, pp. 1144–1156, May 2014.

[3] S. Designer, "Getting around non-executable stack (and fix)," 1997. [Online]. Available: https://seclists.org/bugtraq/1997/Aug/63

[4] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "CCFI: Cryptographically Enforced Control Flow Integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 941–951. [Online]. Available: https://doi.org/10.1145/2810103.2813676

[5] I. Qualcomm Technologies, "Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions," Qualcomm Technologies, Inc., Tech. Rep., 2017.

[6] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-Flow Integrity Principles, Implementations, and Applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, Nov. 2009. [Online]. Available: https://doi.org/10.1145/1609956.1609960

[7] J. Pincus and B. Baker, "Beyond stack smashing: recent advances in exploiting buffer overruns," *IEEE Security Privacy*, vol. 2, no. 4, pp. 20–27, 2004.

[8] D. Brash, "Armv8-A architecture: 2016 additions," Arm Limited, Tech. Rep., 2016. [Online]. Available: https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/armv8-a-architecture-2016-additions

[9] C. Marinas, *Memory Layout on AArch64 Linux*, 2021. [Online]. Available: https://www.kernel.org/doc/html/latest/arm64/memory.html

[10] R. Avanzi, "The QARMA Block Cipher Family." *IACR Transactions on Symmetric Cryptology*, vol. 2017, no. 1, pp. 4–44, Mar. 2017. [Online]. Available: https://tosc.iacr.org/index.php/ToSC/article/view/583

[11] A. Limited, *Arm® Architecture Reference Manual*, jul 2021.

[12] N. Stephens, "Developments in the Arm A-Profile Architecture: Armv8.6-A," Arm Limited, Tech. Rep., 2019. [Online]. Available: https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/arm-architecture-developments-armv8-6-a

[13] G. O. Docs, *Program Instrumentation Options*. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html\#index-fstack-protector

[14] F. Restuccia, M. Pagani, A. Biondi, M. Marinoni, and G. Buttazzo, "Modeling and Analysis of Bus Contention for Hardware Accelerators in FPGA SoCs," in *32st Euromicro Conference on Real-Time Systems (ECRTS 2020)*, 2020.

[15] F. Restuccia and A. Biondi, "Time-predictable acceleration of deep neural networks on fpga soc platforms," in *2021 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2021, pp. 441–454.

[16] C. Maiza, H. Rihani, J. M. Rivas, J. Goossens, S. Altmeyer, and R. I. Davis, "A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems," *ACM Comput. Surv.*, vol. 52, no. 3, 2019. [Online]. Available: https://doi.org/10.1145/3323212

[17] *AMBA® AXI™ and ACE™ Protocol Specification*, ARM, aRM IHI 0022D.

[18] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener, "TACLeBench: A benchmark collection to support worst-case execution time research," in *16th International Workshop on Worst-Case Execution Time Analysis*, 2016.

[19] S. Thomas, C. Gohkale, E. Tanuwidjaja, T. Chong, D. Lau, S. Garcia, and M. B. Taylor, "CortexSuite: A Synthetic Brain Benchmark Suite," in *International Symposium on Workload Characterization (IISWC)*, Oct. 2014.

[20] M. Werner, T. Unterluggauer, R. Schilling, D. Schaffenrath, and S. Mangard, "Transparent memory encryption and authentication," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–6.

[21] CLARE-Hypervisor by Accelerat. [Online]. Available: https://accelerat.eu

[22] *System Integrated Logic Analyzer v1.0*, Xilinx, 2017, pG261.

[23] Microsoft, "Control Flow Guard," 2015. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard

[24] J. Tang and T. M. T. S. Team. (2015) Exploring Control Flow Guard in Windows 10.

[25] R. J. Walls, N. F. Brown, T. L. Baron, C. A. Shue, H. Okhravi, and B. C. Ward, "Control-Flow Integrity for Real-Time Embedded Systems," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Quinton, Ed., vol. 133. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, pp. 2:1–2:24. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2019/10739

[26] G. Christou, G. Vasiliadis, E. Athanasopoulos, and S. Ioannidis, "Hard edges: Hardware-based Control-Flow Integrity for Embedded Devices," in *To be appear in 21th Proceedings of SAMOS International Conference on Embedded Computer Systems*, 2021.

[27] N. Maunero, P. Prinetto, G. Roascio, and A. Varriale, "A FPGA-based Control-Flow Integrity Solution for Securing Bare-Metal Embedded Systems," in *2020 15th Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, 2020, pp. 1–10.

[28] R. Denis-Courmont, H. Liljestrand, C. Chinea, and J.-E. Ekberg, "Camouflage: Hardware-Assisted CFI for the ARM Linux Kernel," in *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*, ser. DAC '20. IEEE Press, 2020.

[29] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan, "PAC It up: Towards Pointer Integrity Using ARM Pointer Authentication," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC'19. USA: USENIX Association, 2019, p. 177–194.

[30] H. Liljestrand, T. Nyman, L. Gunn, J.-E. Ekberg, and N. Asokan, "PACStack: an Authenticated Call Stack," in *Proceedings of the 30th USENIX Security Symposium*. United States: USENIX : THE ADVANCED COMPUTING SYSTEMS ASSOCIATION, aug 2020.

[31] G. Ferri, G. Cicero, A. Biondi, and G. C. Buttazzo, "Towards the Hypervision of Hardware-based Control Flow Integrity for Arm Platforms," in *ITASEC*, 2019.

[32] M. Gretton-Dann, "Arm A-Profile Architecture Developments 2018: Armv8.5-A," Arm Limited, Tech. Rep., 2018. [Online]. Available: https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/arm-a-profile-architecture-2018-developments-armv85a