Towards the Hypervision of Hardware-based Control Flow Integrity for Arm Platforms

Giulia Ferri, Giorgiomaria Cicero, Alessandro Biondi, and Giorgio Buttazzo

Scuola Superiore Sant'Anna, Pisa, Italy

Abstract. Embedded systems are being adopted in applications with mixed levels of criticality and security, thus making them more susceptible to malicious cyber-attacks. The programming languages that are typically used for these systems (e.g., C and C++) are memory-unsafe by construction, and so exposed to a large range of cyber-attacks such as Code Reuse Attacks (CRAs) and, in particular, the usage of Return-Oriented Programming (ROP) to craft exploits. Control-Flow Integrity (CFI) is one of the most popular and efficient mitigation techniques against said attacks. Recently, CFI also started to be supported with specialized hardware mechanisms in modern platforms by means of dedicated processor instructions. This paper focuses on the hardware-based CFI solution offered by latest Arm platforms, namely Pointer Authentication Code (PAC), and investigates on the possible approaches to integrate said technique with virtualization mechanisms. In particular, this work aims at (i) leveraging the hardware-based isolation offered by the ARM TrustZone technology to perform key management, (ii) providing hypervisor-centric attack detection and recovery strategies, and (iii) emulating PAC via software. Our proposals have been implemented and tested within a Type-1 Hypervisor running upon an industrial-grade emulated environment by Arm. The implementation and the investigations carried out during this work revealed interesting insights but also crucial limitations. Despite it emerged that the Armv8.3-A architecture allows hypervising PAC with limited effort thanks to the available hardware-based support, our work also revealed that a better support (in terms of virtualization extensions of the processor) is needed to properly virtualize PAC-enabled software and to implement recovery strategies in the presence of attacks.

1 Introduction

Operating Systems, leveraging memory management units (MMUs), are able to deny applications on writing/modifying text sections where the user code resides, mitigating several code-injection-based attacks. However, other classes of attacks are still possible without requiring the injection of malicious code. In particular, Code Reuse Attacks (CRAs) exploit code that is already present in the system, which is used with malicious data injected into the address space of applications and/or in the context of a non-conventional execution flow. Indeed, the underlying idea of such attacks is to deviate the normal execution flow to

a malicious path [8]: one of the most used technique is the, so-called, Return-Oriented Programming (ROP).

Historically, this approach dates back to the *return-to-libc* attack, published by Solar Designer in 1997 [7]. Here, the attacker is able to change the return address of the called function presented in the libc library by manipulating the function arguments in order to contain the address of a target routine. This attack strongly influenced the hacking community that successively improved ROP-based attacks. In general, a chain of short code sequences can be used rather than a single function. Indeed, multiple code sequences present inside benign programs, also called *gadgets*, can be combined to construct a ROP payload and induce arbitrary malicious program behaviors. A gadget can simply be a return instruction after loading, or popping, from memory the return address.

One of the biggest weaknesses of this kind of attacks is that they are mostly based on *known* fixed addresses within the code in which there are gadgets. For this reason, a well-known technique that aims at mitigating these attacks is Address Space Layout Randomization (ASLR), which allows base addresses of various segments (.text, .data, .bss, etc.) to be loaded/placed at randomized memory addresses [11]. In this way, pre-calculated fixed addresses for gadgets are not anymore consistent. The resistance of this technique is mainly based on the adopted address length, which drives the range of feasible addresses in which the code can be stretched against brute-force attacks, and on the absence of pointer leaks [13].

Control-flow Integrity (CFI) is another promising and effective solution to mitigate such attacks. Note that the execution flow of a program can be represented by a Control Flow Graph (CFG) in which nodes are parts of code and edges are jumps/returns. CRAs involve the addition of a new path within the Control Flow Graph (CFG) corresponding to the regular execution of a program. In short, CFI enforces a pre-determined, trusted CFG, without paying care at mitigating some specific attack [1, 6]. CFI is powerful but non-trivial to implement, especially if a complete enforcement of a CFG is desired. To overcome this limitation, industrial players are starting to push for hardware-based supports that allow implementing CFI. One of the most relevant example is the Pointer Authentication Code (PAC) feature, which has been recently introduced in the latest Arm architectures—specifically, since version Armv8.3A. Intel architectures also offer a hardware-based control-flow integrity technology named Control-flow Enforcement (CET).

Contribution. This work specifically focuses on the PAC technology by Arm and investigates on possible approaches to integrate PAC-based CFI with virtualization stacks. In particular, this work aims at (i) leveraging the hardware-based isolation offered by the ARM TrustZone technology to perform key management, (ii) providing hypervisor-centric attack detection and recovery strategies, and (iii) emulating PAC via software. Our proposals have been implemented and tested within a Type-1 Hypervisor running upon an industrial-grade emulated environment by Arm. The implementation and the investigations carried out

during this work revealed interesting insights but also crucial limitations of the support provided by Arm in terms of virtualization extensions.

2 Essential background

This section introduces the main technologies used in this work. First, the Armv8-A architecture is briefly described; then, an overview of the Pointer Authentication Code feature is presented.

2.1 The ARMv8 architecture in a nutshell

Arm is a family of Reduced Instruction Set Computers (RISC) processors that are becoming the reference architecture for modern embedded systems, especially for the industry of mobile phones due to their low-power characteristics. The Arm version 8 (Armv8 for short) [9] architecture implements two execution states, namely *Aarch32* and *Aarch64*, respectively based on 32-bit and 64-bit registers (and transactions) with the corresponding specific instruction sets. Each execution mode can dispose of up to four privilege levels, called Exception Levels (EL): EL0 is the least privileged one, typically used for applications; EL1 is the first privileged level that is typically used for OSes; EL2, more privileged, is typically used for Hypervisors; while EL3 is the most privileged one and is used for trusted bare-metal firmware (e.g., Arm Trusted Firmware). Arm also introduced a security-oriented processor extension called TrustZone [3], which natively provides the foundations to implement a Trusted Execution Environment via hardware by splitting the processor into two similar execution modes denoted Secure and Non-Secure worlds. As it is illustrated in Figure 1, EL1 and EL0 are available in both the worlds. EL2 in Secure world is only available on the latest Arm architectures, i.e., since version Armv8.4A [2], while EL3 only resides in Secure world.
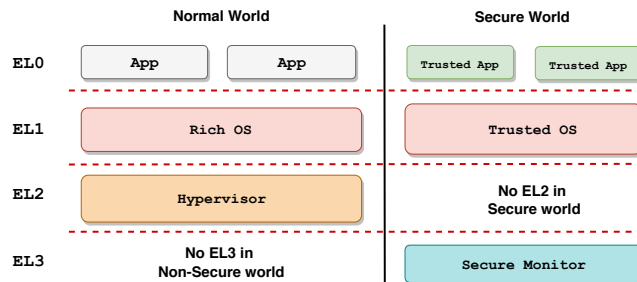

Fig. 1. Armv8-A Exception Levels and Security States.

2.2 Pointer Authentication Code

In 2016, ARM released the version 8.3-A of its application processor architecture. One of the main changes introduced in this version is a mechanism able to *authenticate* the content of a physical register before it is used as the address for an indirect branch or as a data reference. This mechanism is called Pointer Authentication Code (PAC) [9, 12].

PAC is a protection mechanism against exploitation of memory corruption bugs and prevents the modification of function pointers and generic code pointers, such as return addresses stored in memory or registers. The mechanism is able to authenticate pointers and embed a short Message Authentication Code (MAC) within 64-bit memory address. This implies that the mechanisms can be enabled only in the Aarch64 state of Arm platform (i.e., the 64-bit support) and that the actual address space is less than 64 bits because a part of the address needs to be reserved to the MAC. The MAC is built from a pointer (a 64-bit value), a 128-bit secret key stored in system registers, and a 64-bit modifier. Technically, the code is a cryptographic checksum on data, obtained by the QARMA [4] algorithm, which guarantees authenticity and integrity of pointers.

In practice, PAC introduces some additional instructions to the Instruction Set Architecture (ISA) for **(i)** calculating an authentication code and embedding it into a pointer; and **(ii)** verifying an authentication code and restore the original pointer.

Furthermore, a set of system registers have been introduced to store the keys used by the PAC cryptographic algorithm.

The PAC specification provided by Arm defines five keys:

– two pairs of keys named `A` and `B`, where each pair includes a key for creating PACs of instruction addresses and a key for creating PACs of data addresses, for a total of four keys; and
– one key named `G` for general purpose usages.

The size of the PAC depends on size of (virtual) memory address spaces, which determines how big is the unused part of memory address that can hence be used to host the PAC. For instance, given an instruction address contained in the `Xd` 64-bit general-purpose register, the `PACIA Xd, Xn` instruction computes the corresponding authenticated pointer (i.e., the address and its PAC) using the key `A`. The value stored in `Xn` before calling the instruction is used as a modifier. The authenticated pointer will be stored in `Xd`, which will hence contain the concatenation of the original address in `Xd` with its PAC.

Analogously, `AUTIA Xd, Xn` authenticates the instruction address stored in `Xd` by re-calculating and verifying its PAC. If the authentication succeeds, the instruction automatically restores the original pointer into `Xd`. If it fails, the instruction restores the original pointer into `Xd` by adding a *dirty bit* in the most significant part of the address. In this case, the usage of the latter address will result in a *translation fault*. This way for signaling the authentication failure suffers a severe problem: it is not easy to distinguish a PAC failure—i.e., a

possible CFI violation, and hence an attack—from a common exception resulting from an illegal access to memory. This aspect is discussed in details in Section 4.2.

The PAC instructions are typically used for authenticating the return address stored in the link register `lr` and using the value of the stack pointer as a modifier. Other relevant instructions are `PACIASP` and `AUTIASP`, which perform the same tasks of `PACIA` and `AUTIA` with the difference of implicitly using `lr` as `Xd` and the stack pointer `sp` as `Xn`. For this reason, they are expected to be most used PAC instructions in standard software.

Support for PAC is also available at the level of compilers. For instance, since version 7 of the GCC compiler, by enabling the option `-msign-return-address` it is possible to automatically protect every function with PAC (adequate assembly code is generated at the beginning and at the return points of each function).

As an example, Figure 2 represents the classical prologues and epilogues for building an activation record of a function (on the left), and its modification with the PAC-related instructions introduced by the compiler (on the right).

```
// Create frame and push lr, sp
SUB sp, sp, #0x40
STP x29, x30, [sp,#0x30]
ADD x29, sp, #0x30
<function_body>
// Restore frame
LDP x29,x30,[sp,#0x30]
ADD sp,sp,#0x40
RET
```

```
// Compute PAC of lr with sp as modifier
PACIASP
// Create frame and push lr(with PAC), sp
SUB sp, sp, #0x40
STP x29, x30, [sp,#0x30]
ADD x29, sp, #0x30
<function_body>
// Restore frame
LDP x29,x30,[sp,#0x30]
ADD sp,sp,#0x40
// Authenticate lr with sp as modifier
AUTIASP
RET
```

Fig. 2. Example of prologue and epilogue of a function with (right) and without (left) PAC.

The PAC keys are stored into system registers and are hence accessible by the privileged `mrs/msr` instructions (the standard Arm instructions to manipulate system registers), which are not executable at EL0. The key management is the only part of the PAC support that has to be controlled by software: this aspect motivated our investigations.

3   Objectives

This work aims at a detailed analysis of the PAC support offered by Arm platforms to the end of realizing the following new features by leveraging the adoption of a Hypervisor:

- **Improved key management.** One of the most vulnerable parts of a system with cryptographic functions is the way it handles secrets, which most times are keys. Concerning the PAC support for Arm architectures, this aspect is totally left to the programmer. As mentioned in Section 2, PAC keys are stored in system registers that are accessible by privileged instructions. In this way, a kernel running at EL1 (such as Linux, Android, or a RTOS) can use one or more keys to control the flow of the applications and/or the

kernel itself. For instance, the PAC support for Linux uses a key for each process [10], context-switching the value of the PAC key registers to kernel data structures. If assuming an attack model in which the attacker is able to read kernel data, e.g., due to a data leakage vulnerability, the PAC keys can be retrieved and used to craft PACs to alter the control flow. To mitigate such attacks, this work investigates the possibility of implementing a PAC key management by leveraging the TrustZone technology, i.e., employing virtualization to intercept the accesses to the PAC key registers and then configure the actual PAC keys by using software running in Secure world. Under this approach, the OS running in Non-Secure world can be unaware of virtualization, and can continue using its own PAC keys, which will be associated to the actual keys by the software running in Secure world. As it will be detailed in the next section, a hash function has been adopted to implement said key association.

– **Attack detection and recovery.** Hypervisors are typically adopted in embedded systems to implement mixed-criticality systems. Domains of the hypervisor can host the execution of (i) a rich, low-critical OS such as Linux, which is notoriously prone to bugs and typically serves the execution of complex software that are also prone to attacks; or (ii) a safety-critical OS such as a real-time OS, which typically implements crucial functions. In such a kind of settings, the hypervisor can be used to implement recovery strategies in the presence of attacks. That is, when an attack in the low-critical domain is detected, the hypervisor can shutdown this domain and trigger a fail-safe/fail-operational recovery strategy in the safety-critical domain. Our aim is to investigate how a violation of the CFI, i.e., a failure of a PAC authentication, can be used to trigger such a recovery strategy.

– **PAC emulation.** Another aspect addressed in this work is the evaluation of the possibility of using PAC without the presence of the corresponding hardware support. In other words, we studied how to emulate via software the PAC mechanism in order to let programs compiled for Armv8.3 (including PAC-related instructions) to be compatible with older Arm processors. When adopting a Hypervisor, this can be achieved with para-virtualization.

4 Investigations

This section presents the main investigations performed in this work. Our proposals have been implemented in a custom Type-1 hypervisor that is under development in our laboratory and integrated with Arm Trusted Firmware, a reference low-level, open-source firmware for managing TrustZone-enabled processors. It is important to note that, at the time of writing of this paper and to the best of our knowledge, PAC is available in *only two system-on-chips* (Kirin 980 from HiSilicon and Apple A12 Bionic), which however *do not dispose of off-the-shelf evaluation boards*. Hence, an evaluation of our proposals on an actual platform was not possible. Our implementations have been tested upon Arm Fixed Virtual Platform, a processor emulator offered by Arm. The emulator

aims at reproducing the functional behavior of real processors, but is not cycle-accurate: for this reason, it was not possible to extract meaningful measurements to be presented in this paper.

4.1 PAC key management with TrustZone

Among the protection features related to the PAC mechanism, the Arm architecture allows configuring the processor to trap the accesses to the registers holding the PAC keys in a higher Exception Level. In particular, accesses to said registers from EL1 can be trapped in EL2, while the accesses from both EL2 and EL1 can be trapped in EL3. This feature is particularly useful to manage the PAC mechanism at the hypervisor level.

The solution studied in this work leverages the Arm Trustzone technology in order to (i) deny the access to the keys from each Exception Level belonging to the Non-Secure world, and (ii) manage keys only from the Secure world without any form of awareness for the OS running in the Non-Secure world. In this way, if an attacker is able to exploit a vulnerability within the Non-Secure world, hence gaining privileges for reading (or even writing) the key registers at EL1/2, the accesses to the key registers will be trapped and virtualized.

Note that a hypervisor running at EL2 is free to implement its own policy to virtualize the accesses to the PAC key registers performed by the OSes running at EL1. This would guarantee a form of protection enabled by the ring-based privilege levels of Arm platforms, but keys are still manged in Non-Secure world. A more secure solution can be realized by exploiting the TrustZone technology. The idea is to dispose of a secure service running on a Trusted OS within the Secure World that acts as the only software layer in charge of handling the PAC keys, i.e., modifying the corresponding physical registers.

Specifically, as soon as an OS (from Non-Secure EL1) or the Hypervisor (from Non-Secure EL2) perform a read/write operation to the PAC key registers, a trap is triggered and managed by the Secure Monitor at EL3. The Secure Monitor will then (i) perform a world context switch, (ii) give access to the Secure World on accessing the keys, and finally (iii) awake the Trusted OS to perform the actual access to the PAC keys. At this point, the Trusted OS running in Secure EL1 is free to implement any policy to virtualize accesses to the target PAC key register. Here, the challenge is to act as a transparent layer, i.e., the OS in EL1 should be oblivious of this virtualization chain.

The realized policy to handle write accesses to the PAC key registers works according to the following steps:

– retrieve and save in a secure storage the key that the Non-Secure World was trying to set (referred to as the *virtual key*);
– generate a new key by using a cryptographic hash function that takes as input the virtual key and a secret; and
– write the generated key into the physical PAC key register.

In dual manner, when a read operation is trapped, the secure service just returns the virtual key to the Non-Secure World.

Note that the function that generates the key cannot be a simple hash function; otherwise, even if the used hash algorithm is secret, a brute-force attack can easily allow guessing the algorithm. A cryptographic hash function is hence needed. The secret key used by the function is randomly generated at each system boot for making brute-force attack still more difficult. In the presence of warm resets, the secret can be also periodically or on demand re-generated by a secure service running in the Secure world.

To implement these features, the Arm Trusted Firmware (ATF) running at EL3 has been patched in order to (i) configure the processor such that the Non-Secure World cannot access the PAC key registers, and (ii) give the possibility to the Secure World to manage the context of the Non-Secure World while virtualizing the read/write operations. Additionally, a basic bare-metal firmware has been developed to implement a simple secure service running at Secure EL1 to handle key management.

4.2 Detection and recovery

The PAC support is not designed to detect attacks in a direct manner. If an instruction for authenticating a PAC fails, the processor replaces the PAC with a specific pattern that makes the corresponding pointer an *illegal* address, while no PAC-specific exceptions are generated. An exception is generated only when the pointer is used, e.g. for a branch. For example, consider the epilogue of the function with PAC support shown in Figure 2 where the `AUTIASP` instruction verifies whether the PAC stored in the `LR` register can be authenticated. If the authentication fails, it does not generate an exception, but it replaces the PAC with an illegal address that will be stored in the link register `LR`. Then, when the return instruction (`RET`) is executed, the value in `LR` is used and a translation fault is generated.

Unfortunately, the information provided by the processor about this exception through state registers in not enough to recognize that a failed authentication happened. As a consequence, it is difficult to distinguish an attack with respect to a common illegal memory access performed by a lower level of privilege, e.g., due to a harmless bug in a user application. In other words, the PAC mechanism does not provide explicit support (e.g., a processor flag) for a rapid attack detection—to the best of our knowledge, the only option is to parse the faulty address and check if the PAC-specific illegal pattern is present.

This work investigated possible solutions to overcome this limitation. A first solution considered in this work consists in trapping *all* translation faults to a higher level of privilege, and then verify if the address that generated the exception includes the specific pattern of a failed authentication. Unfortunately, the architecture does not help in implementing this approach as it is not possible to trap translation faults only. The only available option is to trap all exceptions, hence leading to a high overhead.

A second solution investigated in this work consists in exploiting the architectural support to trap PAC instructions. Instead of reacting to translation faults, the idea is to prevents the usage of faulty addresses by executing said

instructions at the level of the hypervisor and then checking the results of PAC authentications to detect failures. To develop a proof-of-concept implementation of this approach, only a subset of PAC instructions has been virtualized; specifically, those that are natively supported by the GCC compiler: `PACIASP` to create a PAC in the prologue of a function, and `AUTIASP` to authenticate a PAC in the epilogue. As stated before, as soon as software running in EL1 (or EL0) executes a PAC instruction, a trap exception is generated and routed to the hypervisor (EL2). Note that, to virtualize `PACIASP` and `AUTIASP`, the hypervisor should create or authenticate a PAC with values that reside in the Exception Level that generated the trap. Furthermore, to implement a transparent virtualization, their results must be stored into the corresponding registers of the Exception Level that generated the trap exception. The following subsections detail the implemented mechanisms to provide a taste of the complexity and the issues that arise when adopting this approach.

Virtualizing `PACIASP`. First note that the `PACIASP` cannot be used within the hypervisor, as it would use the `LR` and `SP` registers of the EL2 context and not those of the Exception Level that generated the trap. For this reason, the `PACIA Xd, Xn` instruction has to be used by moving to `Xd` and `Xn` the values of `LR` and `SP`, respectively, from the context of EL1 or EL0. Furthermore, if the PAC support is enabled in EL2, the Hypervisor should temporarily replace the physical key with the one belonging to the EL1/0 context. A crucial issue is that PACs have a length that is dependent on (i) the size of virtual addresses and (ii) whether address tagging is enabled or not. During the execution of each PAC instruction, this information is implicitly extracted by system registers of the current Exception Level. As a consequence, when PACs are computed at EL2, it is not guaranteed to dispose of the same configuration of EL1/EL0, hence risking to generate incompatible PACs (i.e., by losing useful parts of the original pointer). For this reason, after EL2 computes the PAC, it is necessary to perform some checks and, in some cases, to fix the results of the PAC generation. For example, if tagging is not enabled in EL2, but it is enabled in the Exception Level that caused the trap exception, then the first byte (which must remain equal as in the original pointer) is replaced by the byte of the PAC created by EL2. This byte is then non-recoverable. In this case, the Hypervisor should truncate the PAC created in EL2 by replacing the first byte with the original pointer, hence replicating the behavior of the instruction to create the PAC when executed at the Exception Level that caused the trap exception. The same thing happens if the size of virtual addresses in EL2 is smaller than the one of virtual addresses in EL1. Here, a PAC created in EL2 would override some relevant bits of the original pointer. Clearly, this does not happen when the size of the virtual addresses in EL2 is greater than the size of those in EL1 (i.e., a smaller PAC is created).

Virtualizing `AUTIASP`. The same problems encountered in virtualizating `PACIASP` also hold for the case of `AUTIASP`. In addition, since PACs created

with virtualization could have been modified to handle the problems discussed above, it is not possible to use a standard authentication proccess in every case. For this reason, to authenticate a PAC, the original pointer has to be rebuilt to compute its PAC, which can then be authenticated. At this point, our implementation verifies if the two PACs are the same, so finally allowing for attack detection.

4.3   PAC Emulation

Considering its native support in compilers, PAC is likely to establish as a reference solution to implement CFI. For this reason, it is relevant to explore possibilities to provide support for PAC also in architectures that do not dispose of specialized instructions. Para-virtualization is a powerful technique to offer this support.

In this work, PAC instructions have been emulated via software. A proof-of-concept implementation has been developed, again by only supporting the main PAC instructions managed by GCC (`PACIASP` and `AUTIASP`). A tool to patch the binary code produced by GCC for ARM has been implemented: the tool replaces PAC instructions (which would take no effect in architectures that do not offer hardware-support for PAC) with *secure monitor calls*—i.e., `SMC` instructions. Approaches similar to those discussed in the previous subsection have been adopted. Here, the main difference is that we do not dispose of PAC instructions, hence every creation and authentication of PACs must be performed in software. The QARMA algorithm has been re-implemented in C for this purpose. Not surprisingly, this solution led to a very high run-time overhead, mostly related to the software execution of QARMA, which jeopardized the system performance. Our preliminary measurements revealed that, besides introducing considerable overhead, the intensive usage of `SMC` instructions to emulate PAC can still lead to reasonable performance. Future work should investigate on the use custom hardware accelerators to implement the QARMA algorithm. For instance, this approach could be viable in FPGA-based system-on-chips such as those of the Zynq and Zynq Ultrascale families by Xilinx.

5   Conclusion

This work studied a brand-new security mechanism to implement Control-Flow Integrity (CFI) that is available in the latest Arm platforms, namely Pointer Authentication Code (PAC), and its possible integration with Hypervisors. Specifically, the paper focused on investigations to the end of (i) improving the PAC key management by leveraging the Arm TrustZone technology, (ii) implementing Hypervisor-centric attack detection and recovery strategies, and (iii) emulating via software the PAC mechanism.

Overall, the result of our investigations can be summarized with three take-away messages. First, TrustZone-assisted key management can be realized with limited effort thanks to the virtualization extensions offered by Arm, provided

that accurate care is given to the structure of PACs. Second, attack detection with PAC is difficult to realize due to architectural limitations. A solution with limited overhead has been proposed to solve this problem, but it would make much more sense to dispose of a simple flag in a status register of Arm processors to signal a failed PAC authentication. Third, emulating the PAC mechanism by only using software approaches is not a practically-viable approach as it leads to low performance, but the usage of custom hardware accelerators may solve this issue.

Our investigations will hopefully lead to interesting future works aimed at improving the usability, the performance, and the security of PAC, which include a deeper investigation of the aspects addressed in this paper, support for dual-hypervisor designs [5], and accurate performance evaluations with optimized implementations in hypervisors.

References

1. Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.
2. ARM. Introducing 2017s extensions to the arm architecture. https://community.arm.com/processors/b/blog/posts/introducing-2017s-extensions-to-the-arm-architecture.
3. ARM. Trustzone. https://developer.arm.com/technologies/trustzone.
4. Roberto Avanzi. The qarma block cipher family – almost mds matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes. Cryptology ePrint Archive, Report 2016/444, 2016. https://eprint.iacr.org/2016/444.
5. G. Cicero, A. Biondi, G. Buttazzo, and A. Patel. Reconciling security with virtualization: A dual-hypervisor design for arm trustzone. In *2018 IEEE International Conference on Industrial Technology (ICIT)*, pages 1628–1633, Feb 2018.
6. Ruan de Clercq and Ingrid Verbauwhede. A survey of hardware-based control flow integrity (cfi). *CoRR*, abs/1706.07257, 2017.
7. Solar Designer. lpr LIBC RETURN exploit. https://insecure.org/sploits/linux.libc.return.lpr.sploit.html. 1997.
8. M. Kayaalp, M. Ozsoy, N. A. Ghazaleh, and D. Ponomarev. Efficiently securing systems from code reuse attacks. *IEEE Transactions on Computers*, 63(5):1144–1156, May 2014.
9. ARM Ltd. Arm architecture reference manual (armv8, for armv8-a architecture profile). Technical report, 2017.
10. LWN.net. ARM pointer authentication. https://lwn.net/Articles/718888/.
11. Pax Project. Address space layout randomization (aslr). https://pax.grsecurity.net/docs/aslr.txt.
12. Inc. Qualcomm Technologies. Pointer authentication on armv8.3. 2017.
13. Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.