A Security Enhanced Key Management Service for ARM Pointer Authentication

Liqiang Zhang, Qingsong Chen, and Fei Yan(✉)

Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University, Wuhan 430000, Hubei, China
yanfei@whu.edu.cn

Abstract. The memory-unsafe programming languages caused a pandemic of memory corruption bugs in ARM-based devices. To mitigate such threats, Control-Flow Integrity (CFI) is one of the most effective and popular solution, and integrated with the modish hardware makes it even more valuable, for instance, the ARM Pointer Authentication (PA), which can generate a message authentication code for a pointer and verify it to ensure the pointer is intact. However, according to some research, the QARMA algorithm, as a critical part of PA, is vulnerable to certain attacks, making it possible to recover the key.

In this paper, we present a key management service for PA. It utilizes the exception model of TrustZone to isolate the key generation process of PA securely, preventing the key from leaking to insecure memory; then takes advantage of a randomization scheme to dynamically derive separate keys for both kernel-space and user-space programs. Based on the scheme, we have implemented a prototype among the ARM Trusted Firmware, and also an enhanced backward-edge CFI solution. The evaluation shows that it introduces a reasonable and acceptable performance overhead, while provides better security guarantee.

Keywords: Pointer authentication · Key management · Control-flow integrity

1 Introduction

According to Google's report, memory corruption bugs generally accounted for more than 65% of High & Critical security bugs in Chrome and Android. These bugs exist in the "memory-unsafe" programming languages (e.g., C and C++), which allow developers to take fine-grained control of the memory through pointers. And security mechanisms such as Stack Canary, DEP and ASLR have indeed defeated the primitive attack patterns like code-injection, but in the meantime, code-reuse attacks gradually become the main pattern of hacking the software and system security.

Return-oriented programming (ROP) [1] is a well-known code-reuse technique which allows the attacker to reuse the original benign code snippets (called gadgets) in the memory and manipulate them to launch powerful attacks [2]. In order to defend against the ROP attack, multiple approaches have been proposed in the past decades,

B. Chen and X. Huang (Eds.): AC3 2021, LNICST 386, pp. 41–55, 2021.
https://doi.org/10.1007/978-3-030-80851-8_4

such as control-flow integrity (CFI) [3]. However, software implementations of CFI are limited by high performance overhead or significant changes to system software architecture, and hardware-assisted ones are unlikely to ever be deployed because of requiring the underlying processor architecture changes.

In recent years, major processor vendors have directly embedded security primitives into their modern processors. ARM introduced Pointer Authentication (PA) in ARMv8.3-A to detect and reject crafted pointers through a set of instructions. PA calculates a cryptographic message authentication code for 64-bit pointers, and stores it at the high bits of the pointer, referred to Pointer Authentication Code (PAC). The pointer with PAC must be verified by the hardware before it is used as a return address or function pointer. Actually, Qualcomm has proposed a backward-edge CFI scheme based on PA to protect the return address [4], then Apple suggested to sign the virtual function pointers to protect forward-edge CFI [5].

However, the default QARMA block cipher algorithm used by PA is subjected to certain attacks: Li et al. [6] applied the meet-in-the-middle attack on reduced-round QARMA-64/128; Yang et al. [7] have utilized impossible differential attack to analyze QARMA with less time and space complexity; and related-tweak statistical saturation attack are also proved to be effective on QARMA [8]. Therefore, the ability of sophisticated attackers to perform cryptanalysis on the PA mechanism cannot be ignored. The existing PA-based schemes cannot be trustworthy enough to safeguard the CFI, for the following reasons: 1) the assigned PA keys for each process are constant, and 2) the attacker may attempt cryptanalysis attacks based on the collected PAC values over a long period of time.

In this paper, we design and implement Pointer Authentication Key Management Service (PAKMS), providing security enhanced PA key management service. We also implement a backward-edge CFI solution based on it to evaluate the security and performance overhead of PAKMS. Our main contributions are:

- Design: We propose a novel scheme, leveraging the privilege and exception levels of ARMv8-A to provide pointer authentication key management service, which could dynamically generate distinct keys for both the kernel and user-space applications (Sect. 4).
- Implementation: We have implemented a prototype of PAKMS based on the ARM Trusted Firmware (ATF). We also implement a LLVM-based backward-edge CFI solution for user-space applications, which proves the practicality of PAKMS (Sect. 5).
- Evaluation: We conducted systematic analysis of PAKMS on the ARM Fixed Virtual Platform (FVP) with nbench-byte benchmark, demonstrating that it has a reasonable performance overhead (Sect. 6).

2 Background

2.1 Privilege and Exception Levels

ARMv8-A architecture enables exception levels with a different privilege of accessing system resources. The exception levels are referred as EL<x>, with number x ranging from 0 to 3. EL0 is the least privileged one, typically used for applications; and EL1 is

for the operating system. While the EL0 and EL1 are available in both the Non-Secure and Secure worlds introduced by TrustZone [9], the EL3 is the most privileged one that only resides in Secure World, reserved by low-level firmware or security code. The current exception level can only be switched when the processor takes or returns from an exception. For instance, SMC instruction can causes an exception to EL3, and SVC instruction to EL1.

There are two types of privilege relevant to the exception levels. One is memory privilege controlled by Memory Management Unit (MMU) and isolated by the TrustZone Address Space Controller (TZASC) in hardware. The other is privilege of accessing processor resources, i.e. System registers. Generally, the name of the System register indicates the lowest exception level required by a successful access. For example, the register *TTBR0_EL1* holds the base address of the translation table of applications at EL0, which requires privilege of EL1 at least.

2.2 Pointer Authentication

In 2016, ARM introduced an ISA extension in ARMv8.3-A, named Pointer Authentication (PA) [4]. The main purpose of PA is to provide a fast and security mechanism against exploitation of memory corruption bugs, by the pointer authentication codes (PACs). PAC is a short Message Authentication Code (MAC) which can be generated through a single instruction and embedded into the high bits of 64-bit pointers.

PA mainly introduces three types of instructions: (1) *PAC** instructions are used to generate and insert PAC into the pointer; (2) *AUT** instructions are designed to authenticate PAC, in other words, if PAC matches, the original pointer will be returned, or else it returns an invalid pointer that would trigger a fault upon use; (3) *XPAC** instructions also yield the original pointers, but without the authentication routine. As illustrated in Fig. 1, a *PAC** instruction calculates PAC over a 64-bit virtual address, a 64-bit modifier as a customized context and a 128-bit secret PA key stored in System registers, using Hash function (QARMA [10] by default). Since the PAC is embedded in unused bits of the pointer, its size is restricted both by the virtual address size and the memory tagging extension (MTE) introduced by ARMv8.5-A. For example, when the *VA_SIZE* is 48-bit and MTE is enabled, the PAC size is merely 7 bits.

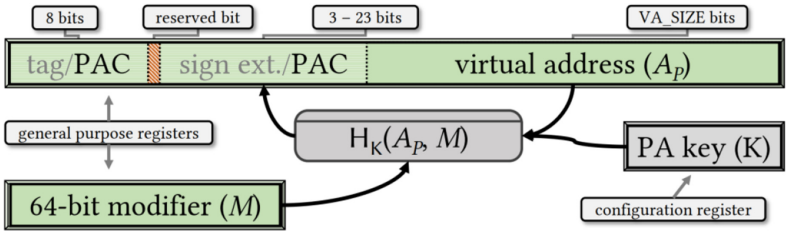

Fig. 1. PAC is computed on the pointer's address, a modifier and a key (from [17]).

Furthermore, PA also introduces a set of System registers to hold the PA keys used by the PAC cryptographic algorithm. There are five different keys in total, *IA* and *IB*

keys are used for code pointers, *DA* and *DB* are for data pointers and *GA* is for general purpose. Take the *PACIA Xd, Xn* instruction as an example, it computes the corresponding authenticated pointer (returned by register *Xd*) over an original pointer kept in *Xd*, a 64-bit modifier kept in register *Xn* and the 128-bit IA key composed of the System register *APIAKeyHi_EL1* and *APIAKeyLo_EL1*.

3 Threat Model

3.1 Attacks

PA prevents the attacker from forging and utilizing pointers to hijack the control flows, relying on the signing and authenticating instructions. Hence, to obtain a legal PAC of the target address becomes attackers' primary goal. And PAC is calculated from a context-associated modifier and a confidential PA key, using QARMA algorithm by default, which consequently exists three possible attack channels as follows.

First of all, the modifier value is not confidential in principle, and can be either static (e.g., a hard-coded value) or dynamic (e.g., the stack pointer). On the one hand, reverse engineering could reveal the static value, and on the other hand, take the stack pointer as an example, it is not required to be unique for a specific function, therefore, attackers could reuse the signed pointer in one function when another vulnerable function executes with the same stack pointer.

Secondly, the management of PA keys could under attack. PA keys used by EL0 applications are generated and switched by the kernel. And the kernel tracks them in the *thread_struct* for each application, therefore, the keys remain constant throughout its lifetime and are shared by all threads in a single address space. A constant PA key is fragile to cryptanalysis attack based on known PAC values, and shared keys expand the search space for attackers to find a usable signing or authenticating gadget.

Lastly, PA also depends on the security of the underlying cryptographic primitives, i.e., the QARMA algorithm. For QARMA, researchers have proposed several cryptanalysis methods, such as meet-in-the-middle and impossible differential attacks. Thus, a sophisticated attacker could observe and collect adequate pairs of the original pointer and the signed one to attempt cryptanalysis attacks. If the attacker could recover the PA keys, all the integrity protections within the scope could be useless.

3.2 Assumptions

In line with the threat model of pointer authentication, we consider a powerful attacker who has the capability to read and write almost the entire address space, constrained only by DEP memory policies, for instance, the attacker cannot write to the read-only memory and execute codes in writable memory. In addition, the attacker cannot read or write the System registers directly, especially the registers that hold PA keys, although he/she may obtain the capability to read and write arbitrary kernel memory via kernel vulnerabilities [11]. The attacker's goal is to subvert the control flow of the process. Thus, he/she would attempt to corrupt any return addresses or function pointers in the data section, stack and heap region. Furthermore, with the understanding of cryptanalysis

attacks, the attacker could also try to guess the PAC value of a fake pointer, even to infer the PA keys used in some context.

In addition to the ability of the attacker, we made some assumptions about the system as well. We assume that the firmware or secure monitor is trusted, and the TZASC is configured correctly to separate the physical memory space. The hypervisor (if existed) and the kernel boot-up process is also trusted, since the bootloader in EL3 can verify their cryptographic hash of the binary. And then, we further assume that the random number generation used in this work has the expected random entropy. The hardware circuitry relevant to the PA mechanism should work as defined by the ARM specification, leaking no private and confidential information.

4 Design

4.1 Architecture

In this section, we present a novel scheme named PAKMS to provide PA keys for both kernel and user-space applications. Before illustrating it in details, we need to address security challenges arising from the contradiction between the existing management pattern and the threat model mentioned in the previous section:

Confidentiality. The kernel stores PA keys in *thread_struct* to support each process signing and authenticating pointers correctly, which is not feasible for an attacker with the capability to read and write to arbitrary kernel memory. In order to maintain the confidentiality of PA keys, no keys should reside in the kernel memory.

Variability. This is caused by the immutability of the PA keys, which ensures that all signed pointers remain verifiable in their lifetime, which is vulnerable to the attacker with the knowledge of cryptanalysis.

Diversity. The diversity means the PAC of the same pointer can be distinct in different context. Since the process' PA keys remain constant, the diversity mainly relies on the modifier. However, whether the modifier is static or dynamic, there still exists potential risk of pointer reuse attacks, as we discussed in Sect. 3.1.

We propose the PAKMS to provide PA keys for the kernel and user-space applications, which ensures confidentiality, variability and diversity of PA keys (see Fig. 2). In order to satisfy the demand for confidentiality, PAKMS holds a set of *master-keys* in the privileged EL3, which are designed to derive PA keys for the upper exception levels (i.e., the kernel and the application). On the one hand, the *master-keys* are kept in EL3 memory, and securely isolated from other exception levels through the TZASC, at the same time, the derived PA keys are always kept in System registers and never banked to the kernel memory. On the other hand, the PA keys are also derived on a dynamic parameter, namely the Generation (*Gn*), which ensures that the attacker cannot obtain adequate pairs of the original pointer and the signed one to infer the PA keys by cryptanalysis attacks. Besides, *Gn* further offers variability and diversity of PA keys thanks to that it changes based on system time or access frequency. What's more, to enlarge diversity of PA keys, we have adopted distinct parameters for the granularity of process level, thread level and function level. In the followings, we will detail the PAKMS in accordance with the hierarchical structure.

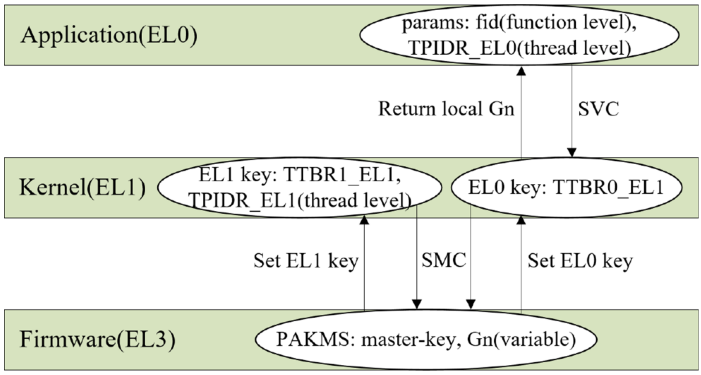

Fig. 2. PAKMS ensures confidentiality, variability and diversity of PA keys.

1) EL3 Isolation: naturally, PAKMS in EL3 can guarantee good confidentiality of PA keys through TZASC configurations. Resided in the system bus and the memory chip, TZASC can support multiple memory regions with different access-control settings (e.g., TZC-400 supports up to 9 memory regions), so that PAKMS can preserve the *mater-keys* and *Gn* in the EL3 memory which is exclusively accessible by the secure world. Besides, as an EL3 runtime service, PAKMS is both versatile and extensible, not only can it provide PA keys for the kernel and the application in Non-secure world, but also can serve the Hypervisor and the Trusted OS in Secure world. What's more, EL3 can provide a more reliable source of randomness for initializing the *master-keys* and *Gn*, and updating the *Gn* with a random strategy.
2) Kernel in the Middle: in our design, the kernel plays two roles on the whole. For one thing, to protect the integrity of kernel-space pointers, the kernel can invoke the PAKMS through a Secure Monitor Call (SMC). According to the SMC Calling Convention [12], an SMC64 call can pass six parameters at most, using registers *X1–X6*, and return results in registers *X0–X3*. Thus, while invoking the PAKMS, we can pass some unique parameters to ensure a rich diversity of PA keys. Optional parameters are the values of register *TTBR1_EL1* and *TPIDR_EL1*. The former holds the base address of the translation table of kernel address space, which can be used to separate keys of the kernel from the user-space applications; and the latter, storing thread identifying information, can be used to produce distinguishable PA keys for kernel threads. And further, function-level parameter can be passed to produce fine-grained PA keys for each function context.
   For another, the kernel plays the role of a bridge between PAKMS and the user-space applications. Since SMC is not available in EL0, the kernel receives a request for PAKMS from applications and passes it to the EL3. Except for the parameters from the application (the function-level parameter), analogously, the register *TTBR0_EL1* and *TPIDR_EL0* can be used to produce PA keys. A local *Gn* will be returned by PAKMS, and can be stored on the stack, in order to correctly authenticate signed pointers later, because *Gn* in EL3 is variable throughout the system.
3) Application Strategies: for SMC instruction is undefined in EL0, the user-space applications have to invoke a Supervisor Call (SVC) to access PAKMS via the

kernel. Then, the kernel would invoke the PAKMS with some context-sensitive parameters, that is the register *TTBR0_EL1*, *TPIDR_EL0* and the function-level parameter. In addition, the PAKMS should return a signed/authenticated pointer directly rather leave the *PAC*/AUT** operations in the application context. Because the user-space tasks can be preempted by other high-priority tasks, which may invoke the PAKMS again, leading to the former task cannot sign/authenticate the pointer rightfully. Hence, the signed/authenticated pointers are returned and *BR** instructions are recommended to make up a function epilogue. Lastly, there should be no way for the application developers to create a signing or authentication gadget manually using the *PAC*/AUT** instructions, for that may form attackable code sequences.

4.2 Algorithm

We illustrate the signing and authenticating routines of PAKMS in Algorithm 1.

Algorithm 1 PAKMS signing and authenticating routines

Input: *SID*: service identifier; *IAP*: input address pointer; IGn_l: input local Generation of PA keys; *FLP*: function-level parameter.
Output: *OAP*: output address pointer; OGn_l: output local Generation of PA keys.

1. **initialize** master-keys (*IA, IB, DA, DA, GA*) and global Generation of PA keys (Gn_g) randomly at boot-up time
2. $Gn_g \leftarrow$ update with random value by EL3 physical timer
3. **switch** *SID*:
4. *Km* ← select a master-key from (*IA, IB, DA, DA, GA*)
5. *PLP* ← select a process-level parameter, either *TTBR0_EL1* or *TTBR1_EL1*
6. *TLP* ← select a thread-level parameter, either *TPIDR_EL0* or *TPIDR_EL1*
7. **case** signing:
8. set the related PA keys, take *APIAKey_EL1* for example:
9. *APIAKeyLo_EL1 = QARMA(Km, PLP⊕FLP, Gn_g)*
10. *APIAKeyHi_EL1 = QARMA(Km, TLP⊕FLP, Gn_g)*
11. set $OGn_l = Gn_g$
12. **if** preemptive **then**:
13. signing *IAP*, for example: *OAP = PACIA(APIAKey_EL1, IAP, FLP)*
14. **return** *OAP*, OGn_l
15. **else** non-preemptive:
16. **return** OGn_l
17. **end if**
18. **case** authenticating:
19. set the related PA keys, take *APIAKey_EL1* for example:
20. *APIAKeyLo_EL1 = QARMA(Km, PLP⊕FLP, IGn_l)*
21. *APIAKeyHi_EL1 = QARMA(Km, TLP⊕FLP, IGn_l)*
22. **if** preemptive **then**:
23. authenticating *IAP*, for example: *OAP = AUTIA(APIAKey_EL1, IAP, FLP)*
24. **return** *OAP*
25. **else** non-preemptive:
26. **return**
27. **end if**
28. **end switch**

After the firmware accomplishes the necessary architectural and platform initialization, including the setup of exception vectors and control registers (specifically, to enable PA instruction in EL3, the register *SCTLR_EL3.{EnIA, EnIB, EnDA, EnDB}* field need to be set up), the PAKMS is initialized with five random *master-keys* and one Gn_g on behave of the Generation of PA keys (Line 1). Then setup the EL3 physical timer to update Gn_g with random value periodically. According to the SMCCC, the framework schedules every SMC call to the corresponding handler through the function identifier, which determines the service and function to be invoked. As shown in Algorithm 1, the service identifier (*SID*) is actually the lower 16 bits of the function identifier, indicating the function number of PAKMS. When PAKMS is invoked, firstly, some context-sensitive parameters are selected according to the value of *SID*, e.g., the *master-key* to use (Line 3 to 6). As there are five *master-keys* in total, it is recommended to use different master-key for code pointers and data pointers. In order to distinguish between the kernel and user-space processes, the process-level parameter can take the register value of *TTBR1_EL1* or *TTBR0_EL1*. And the thread-level parameter, which makes use of the register *TPIDR_EL0* or *TPIDR_EL1*, is related to the thread local storage. After that, the signing and authenticating key generation processes are introduced separately.

In the signing procedure, apart from the parameters mentioned above, the function-level parameter passed from the kernel is also utilized to generate a unique key, using some specific algorithm. In this paper, we choose QARMA for the reason that the *PAC** instructions have provided an efficient implementation (Line 8 to 10). In practice, we take advantage of the *PACGA* instruction, using *Km* as the PA key, $PLP \oplus FLP$ or $TLP \oplus FLP$ as the pointer, and Gn_g as the modifier. Analogously, in the authenticating procedure, the same algorithm are used. However, Gn_g is changeable and may be different from the signing procedure, hence it needs to be backed up through OGn_l (Line 11). To generate the correct key, the authenticating procedure uses IGn_l instead (Line 19 to 21). And for the preemptive situation, we need to return the signed/authenticated pointer calculated by *PAC*/AUT** instructions directly.

5 Implementation

5.1 EL3 Runtime Service

ARM Trusted Firmware (ATF) provides a reference implementation of ARM interface standards, including SMC Calling Convention (SMCCC), System Control and Management Interface (SCMI), the initialization of Generic Interrupt Controller (GIC) and TrustZone Controller (TZC). By providing the EL3 runtime service framework, ATF makes it convenient to integrate services provided by different developers into the firmware. And it also supports multiple toolchains, such as GCC and LLVM, making it easier to customize.

Our prototype is based on the ATF runtime service framework, making use of the signing and authenticating schemes illustrated in Algorithm 1. Firstly, to enable PA instruction in EL3, the register *SCTLR_EL3.{EnIA, EnIB, EnDA, EnDB}* field need to be set up, then PAKMS need to initialize five *master-keys* and one Gn_g randomly. In order to update Gn_g periodically, we also need to configure the EL3 physical timer and the GIC. Specifically, we need to setup the *CNTPS_CTL_EL1* register fields: *ENABLE*

to 1 and *IMASK* to 0. The *IMASK* field controls the interrupt generation. When the timer fires (the value of *CNTPS_CVAL_EL1* <= *CNTPCT_EL0*), an interrupt is asserted to the interrupt controller. The interrupt *ID* used for EL3 physical timer is 29, as recommended by the Server Base System Architecture (SBSA). Then we can update Gn_g and *CNTPS_CVAL_EL1* register in the interrupt handler.

5.2 EL1 Kernel Module

```
MACRO  movFLP  XX
   movk   XX,  #flp00
   movk   XX,  #flp16,  lsl  #16
   movk   XX,  #flp32,  lsl  #32
   movk   XX,  #flp48,  lsl  #48
ENDM

Function:
   ldr    X0,  =#SID              ; ① service identifier
   movFLP  X1                     ;    function-level parameter
   smc    #0                      ; ② invoke PAKMS
   str    X0,  [SP,  #-16]!       ; ③ store local Gn
   paciasp                        ; ④ signing pointer
   stp    FP,  LR,  [SP, #-32]!   ;  store LR
   <function-body>
   ldp    FP,  LR,  [SP], #32     ;  load LR
   ldr    X2,  [SP],  #16         ; ⑤ input local Gn
   movFLP  X1                     ;    function-level parameter
   ldr    X0, =#SID               ;    service identifier
   smc    #0                      ; ⑥ invoke PAKMS
   autiasp                        ; ⑦ authenticating pointer
   ret
```

Fig. 3. A non-preemptive kernel can use PAC*/AUT* instructions in its own code sequences (④, ⑦), after switching PA keys by PAKMS (②, ⑥).

Typically, a non-preemptive kernel can use *PAC*/AUT** instructions directly to protect the integrity of the kernel pointers, as illustrated in Fig. 3. Firstly, at the function entry, kernel sets up the necessary parameters and invokes PAKMS to generate and switch the associated PA keys (Fig. 3, ①②). Next, in order to authenticate the signed pointer without error later, a local *Gn* needs to be preserved properly (Fig. 3, ③). Then, the kernel can use any *PAC** instruction to do the signing procedure. For example, *PACIASP* is utilized to sign the return address using stack pointer as a modifier (Fig. 3, ④). Before the function returns, a similar authenticating procedure needs to be executed to get a functional return address. The differences are that the local *Gn* needs to be passed to the PAKMS and the corresponding *AUT** instruction is used to calculate a correct return address (Fig. 3, ⑤⑥⑦). And finally, the function returns to the rightful place using *RET* instruction.

Another important function of the kernel is to redirect the PAKMS requests from the applications in EL0. The kernel only needs to set the proper service identifier according to specific system call, without considering other parameters which are set by the applications in EL0. Furthermore, before returning to the EL0, the kernel can execute security checks to detect and trap on authentication failures, rather return an illegal pointer to the applications. Then the kernel uses *RET* instruction to return to the application, with return values kept in specific registers.

5.3 EL0 Calling Convention

Based on LLVM 10.0, we add some new passes to the optimizer for generating function-level parameters and to the AArch64 backend for emitting suitable low-level instructions. We use optimizer passes to generate function-level parameters based on the pointer's LLVM *ElementType*, and the backend passes to modify the function prologues and epilogues for making up the parameters and invoking system call for PAKMS. Here we only show the sample code listing of the return address signing and authenticating, but assuredly PAKMS can also provide protections for both code pointer and data pointer.

```
Function:
  mov   X8, #pac_no          ; ① syscall number for pac*
  movFLP  X1                 ;    function-level parameter
  mov   X2, LR               ;    input address pointer
  svc   #0                   ; ② invoke kernel proxy
  str   X19, [SP, #-16]!     ; ③ store local Gn
  stp   FP, X0, [SP, #-32]!  ; ④ store signed pointer
  <function-body>
  ldp   FP, X3, [SP], #32    ; ⑤ input address pointer
  ldr   X2, [SP], #16        ;    input local Gn
  movFLP  X1                 ;    function-level parameter
  mov   X8, #aut_no          ;    syscall number for aut*
  svc   #0                   ; ⑥ invoke kernel proxy
  br    X0                   ; ⑦ return authenticated pointer
```

Fig. 4. The application in EL0 invokes PAKMS indirectly through SVC instruction.

We illustrate part of the function code listing to do the return address signing and authenticating, including the necessary prologues and epilogues (see Fig. 4). First, we set up the special system call number for signing procedure and other essential parameters, then we use the *SVC* instruction to request the kernel proxy as mentioned above (Fig. 4, ①②). After the kernel invokes the PAKMS in EL3, it returns the local *Gn* in register *X19* and the signed pointer in *X0*, which will be preserved on the stack (Fig. 4, ③④). Similar to the signing procedure, the authenticating procedure invokes system call to request the kernel proxy, and the kernel also execute the check routine to detect authenticating failures as early as possible before returns to the application (Fig. 4, ⑤⑥). Finally, the application utilizes the *BR* instruction to return to the location pointed by the register *X0*, which holds the authenticated return address, since the application can be preempted when it returns from the kernel.

6 Evaluation

In this section, we will analyze the security and performance of our proof-of-concept implementation, which is developed and integrated with ARM Trusted Firmware. At the time of writing this paper, an evaluation on an actual platform was not possible, because the PA-capable SoCs (e.g., Apple A12 and Kirin 980) do not dispose of the off-the-shelf development boards. Therefore, we evaluate PAKMS upon the ARM Fixed Virtual Platform (FVP), an emulator supporting the ARMv8.3-A features.

6.1 Security Analysis

The security problems of existing PA-based schemes are stemmed in the immutability of each process' PA keys managed by the kernel. PAKMS, however, can provide different keys to the same process or even the same function context in different time partitions, thanks to the sink of the key management and the variable *Gn* in EL3, which prevents the following security issues to a certain extent.

A signing gadget is a code fragment that can be exploited to sign an arbitrary pointer, which could be utilized to substitute all the pointers signed by the same PA keys. Such gadgets could probably compromise the effectiveness of PA. Another similar attack discussed above, reuse attack, is also a kind of substitute attack reusing the existing authenticatable pointer to construct a loop to perform malicious calculations. Instrumented by LLVM passes, however, all the signing and authenticating code pieces are automatically generated, without certain treacherous patterns of code. Even if the attacker could take advantage of the signing procedure to forge a pointer, he/she still cannot use the pointer in another context, since PAKMS provides rich diversity of PA keys which could be different between function-level contexts. In addition, the keys used in the same context will also change over time, because a dynamic parameter (*Gn*) is used by PAKMS to generate PA keys.

Authentication gadgets, on the other hand, can be exploited to authenticate any forged pointers. And the attacker could verify the correctness of the forged PAC without triggering an exception, which means online guessing attacks are possible. This is because PA mechanism only replaces the PAC with an invalid bit if the authenticating procedure failed, and returns an illegal pointer which causes a translation fault only when it's used. PAKMS solves this problem through a check routine added to the kernel proxy, which can do the early detection on authentication failures, so that all the failures can be recorded to restrict an attacker on the number of guessing.

Table 1. CPU cycles needed by pac* and pakms* instructions to calculate PACs.

Instructions	Execution times & CPU cycles			
	2^8	2^{10}	2^{14}	2^{16}
pac*	3342	13326	229391	917519
pakms*	18702	74766	1212431	4849679

For a sophisticated attacker, cryptanalysis attacks could also be launched to recover the PA keys. Either by the controlled signing procedure, or just by waiting and observing the signing results patiently, the attacker could collect numerous pairs of the original pointer and the signed one. When sufficient data is obtained (only need a set of 65536 chosen plaintexts at least [8]), the attacker could perform offline cryptanalysis on the QARMA to recover the PA keys. We evaluated how many CPU cycles needed by PAC* instruction and PAKMS to calculate PACs, the result is shown in Table 1. PAKMS attempts to mitigate such attacks with a dynamic parameter *Gn* in EL3, and it varies according to the EL3 physical timer, so does the keys based on it. Hence, according to Table 1, we can set the timer less than 4849679 CPU cycles to update *Gn*. So that the attacker cannot collect enough data to recover a specific PA key before other keys are generated for this context. Therefore, PAKMS can prevent such advanced attacks effectively.

6.2 Performance Analysis

We choose nbench-byte 2.2.3 benchmark [13] to evaluate the performance over-head of PAKMS, for the reason that nbench-byte is designed to measure CPU and memory subsystem performance with real-world algorithms, and it is easy to be migrated to new processors or operating systems quickly, allowing us to measure PAKMS instrumentation with scalable and dynamic workload adjustment on FVP. Simultaneously, we build two root file systems based on Yocto Project, us-ing the Linux kernel v5.4.23, and compiled by the customized Clang/LLVM com-piler, one enabled by PAKMS extension and the other not, and the binaries of nbench do the same.

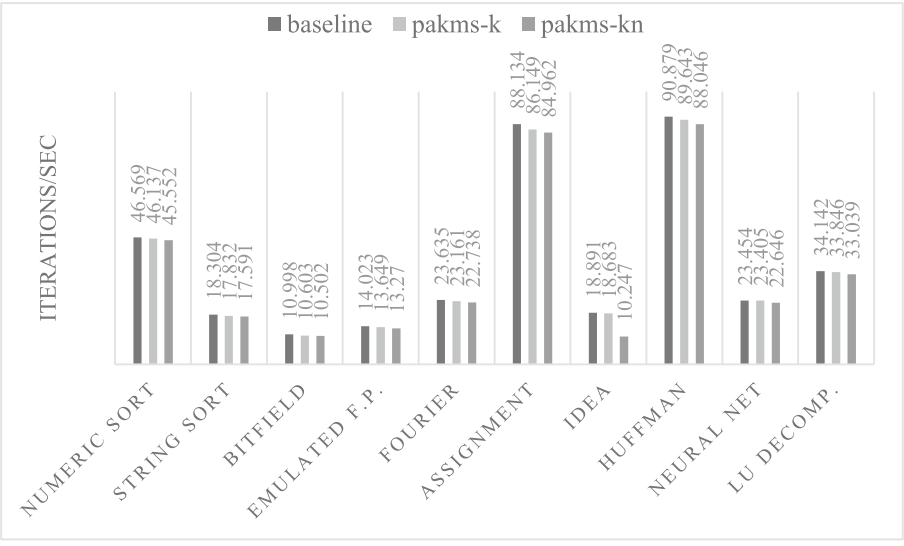

Fig. 5. Performance evaluations of PAKMS using nbench-byte benchmark.

Since PAKMS-enabled nbench cannot run on the original kernel, there are three sets of evaluations in total, as illustrated in Fig. 5, a) the baseline is evaluated with the original kernel and nbench, b) and the pakms-k uses the pakms kernel and the original nbench, c) and the pakms-kn equips both the kernel and nbench with PAKMS extension. In order to analyze and compare experiment results of each algorithm conveniently, we just modify the configuration of each algorithm, but make each evaluation set with the same configurations. Compared with the baseline, both pakms-k and pakms-kn introduce some reasonable overhead, i.e., the iterations per second reduce a little for each algorithm. Except for the IDEA algorithm, which causes 45.7% of performance overhead, most of the other algorithms have an overhead of less than 3%. This is because when the IDEA performs the encryption iterations, all the operations (like ADD, MUL and XOR) need to call its internal functions separately, which leads to frequent calls to PAKMS for signing and authenticating pointers. By optimizing the implementation of IDEA, the PAKMS call frequency can be reduced. In general, for pakms-k, the geometric mean of the performance overhead of all algorithm tests is 1.39%, while pakms-kn is 4.62%.

According to the evaluations, we believe that the performance overhead of PAKMS is reasonable and acceptable. Even in the worst case, IDEA, can reduce the overhead by optimizing the algorithm with less function calls inside, not to mention other more common algorithms. For the ARM Fast Models documentation states that "all instructions execute in one processor master clock cycle" [14], in the future work, we are planning to evaluate PAKMS on the PA-capable hardware and support more signing and authenticating patterns.

7 Related Works

Most of the existing works on ARM Pointer Authentication can be divided into two categories: 1) the usages of PA for pointer protection; 2) the schemes to enhance the security of PA mechanism itself.

The Qualcomm whitepaper [4] came out first with simple backward-edge CFI based on PA, using the SP value as the only modifier, and Apple also proposed forward-edge protection by signing the vtable pointers [5], using zero as the modifier, so that they are both susceptible to reuse attacks, which do no harm to our design due to the rich diversity of PAC. PARTS [15] was the first academic presentation on this subject, which not only implemented a more fine-grained CFI solution for both the return address and code pointer, but also provided the data pointer protection. Our function-level parameter generation required linkage time optimization, as well as its unique function identifiers, and our solution can also offer the comparable security as PARTS. In addition, we further enhanced the security of PA mechanism itself by dynamically changing the PA keys for the kernel and the applications. Other researches also applied the PA to realize a more secure stack canaries [16], and to chain the call stack [17] for fully precise verification of return addresses, however, either of them considered the kernel as a target, which may not readily acceptable for the kernel protection. PTAuth [18] proposed a PA-based heap memory protection system, which can detect and prevent heap-based temporal memory corruptions. However, PTAuth currently does not support multi-threading, and due to shared PA keys, there is still the risk of being bypassed.

PACKER [19] proposed a design for protecting the CFI of the kernel pointers, providing both backward-edge and forward-edge solutions. However, it was also confronted with the problem of sharing the same PA keys in a thread, and it replaced all *BLR* instructions with *BLRAA* instruction, lack of certain compatibility. Based on XOM, Denis-Courmont et al. [20] presented a secure architecture for kernel PA management, which did not depend on traps to higher exception levels. Although the attacker can't access the key stored in the XOM, but in practice, with adequate information of the key, it is possible to recover the key. Ferri et al. [21] took advantage of the hypervisor mode (EL2) and dedicated traps to manage PA keys of applications, certainly the kernel could also be managed this way. However, the traps that the scheme relied on can also be utilized for lazy initialization of PA for hypervisor itself, and were not intended for the key management. As a comparison, PAKMS makes use of the EL3 runtime service framework, which is hardly conflicted with other system services, and besides, it also has the ability to generate PA keys for hypervisor.

8 Conclusion

In this paper, we present an enhanced scheme for PA key management, named PAKMS, which takes advantage of the privilege and exception levels in ARMv8-A to protect the key generation process. PAKMS can dynamically generate PA keys for both the kernel and applications during their lifetime, thanks to the special variable in EL3. According to our evaluation, not only can PAKMS thwart conventional attacks against PA (like substitution or reuse attack), but also elaborate and advanced cryptanalysis attacks. And the prototype of PAKMS based on the ARM Trusted Firmware and Clang/LLVM instrumentation, introducing an acceptable performance overhead.

In the future, we are planning to evaluate the performance of PAKMS comprehensively on the PA-capable hardware and support more signing and authenticating patterns, and also we need to make PAKMS to setup a sufficiently secure and efficient threshold for each system automatically. In addition, the integration with hypervisor and the migration of PAKMS-based process between them are also under consideration.

Ackonwledgements. This research is supported by the Prospective Applied Research Projects of Suzhou City (Grant No. SYG201845), the National Natural Science Foundation of China (Grant No. 61272452) and the National Program on Key Basic Research Project (973 Program) (Grant No. 2014CB340601).

References

1. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 552–561 (2007)
2. Roemer, R., Buchanan, E., Shacham, H.: Return-oriented programming: systems, languages, and applications. ACM Trans. Inf. Syst. Secur. (TISSEC) **15**(1), 1–34 (2012)
3. Abadi, M., Budiu, M., Erlingsson, U.: Control-flow integrity principles, implementations, and applications. ACM Trans. Inf. Syst. Secur. (TISSEC) **13**(1), 1–40 (2009)

4. Qualcomm Technologies: Pointer authentication on ARMv8.3 (2017). https://www.qualcomm.com/media/documents/files/whitepaper-pointerauthentication-on-armv8-3.pdf
5. Apple Inc.: Pointer Authentication (2019). https://github.com/apple/llvm-project/blob/apple/main/clang/docs/PointerAuthentication.rst
6. Li, R., Jin, C.: Meet-in-the-middle attacks on reduced-round QARMA-64/128. Comput. J. **61**(8), 1158–1165 (2018)
7. Yang, D., Qi, W.F.: Impossible differential attack on QARMA family of block ciphers. IACR Cryptology ePrint Archive 2018/334 (2018)
8. Li, M., Hu, K., Wang, M.: Related-tweak statistical saturation cryptanalysis and its application on QARMA. IACR Trans. Symmetric Cryptol. **2019**(1), 236–263 (2019)
9. Hua, Z., Gu, J., Xia, Y.: vTZ: virtualizing ARM trustzone. In: 26th USENIX Security Symposium (USENIX Security 2017), pp. 541–556 (2017)
10. Avanzi, R.: 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. IACR Trans. Symmetric Cryptol. **2017**(1), 4–44 (2017)
11. Zhang, T., Shen, W., Lee, D.: PeX: a permission check analysis framework for Linux kernel. In: 28th USENIX Security Symposium (USENIX Security 2019), pp. 1205–1220 (2019)
12. ARM Limited: SMC calling convention (2019). http://infocenter.arm.com/help/topic/com.arm.doc.den0028b/ARM_DEN0028B_SMC_Calling_Convention.pdf
13. BYTE Magazine (2011). http://www.math.utah.edu/~mayer/linux/byte/bdoc.pdf
14. ARM Limited: Fast Models User Guide (2019). https://static.docs.arm.com/100965/1111/fast_models_ug_100965_1111_00_en.pdf
15. Liljestrand, H., Nyman, T., Wang, K.: PAC it up: towards pointer integrity using ARM pointer authentication. In: 28th USENIX Security Symposium (USENIX Security 2019), pp. 177–194 (2019)
16. Liljestrand, H., Gauhar, Z., Nyman, T.: Protecting the stack with PACed canaries. In: Proceedings of the 4th Workshop on System Software for Trusted Execution, pp. 1–6 (2019)
17. Liljestrand, H., Nyman, T., Ekberg, J.E.: Authenticated call stack. In: Proceedings of the 56th Annual Design Automation Conference 2019, pp. 1–2 (2019)
18. Farkhani, R.M., Ahmadi, M., Lu, L.: PTAuth: temporal memory safety via robust points-to authentication. CoRR (2020)
19. Yang, Y., Zhu, S., Shen, W.: ARM pointer authentication based forward-edge and backward-edge control flow integrity for kernels. arXiv preprint arXiv:1912.10666 (2019)
20. Denis-Courmont, R., Liljestrand, H., Chinea, C.: Camouflage: hardware-assisted CFI for the ARM Linux kernel. In: 2020 57th ACM/IEEE Design Automation Conference (DAC), pp. 1–6. IEEE (2020)
21. Ferri, G., Cicero, G., Biondi, A.: Towards the hypervision of hardware-based control flow integrity for arm platforms. In: ITASEC (2019)