KPAC: Efficient Emulation of the ARM Pointer Authentication Instructions

Illia Ostapyshyn, Gabriele Serra, *Member, IEEE*, Tim-Marek Thomas, and Daniel Lohmann

***Abstract*—ARMv8.3-A has introduced the pointer authentication (PA) feature, a new set of measures and instructions to sign and validate pointers. PA is already used and supported by the major compilers to protect the return addresses on the stack as a measure against memory corruption attacks. As more and more SoCs implement ARMv8.3-A and code compiled with PA is even fully backwards compatible on CPUs without (where the new instructions are just ignored), we can expect PA-enabled binaries to become standard in the near future. This gives rise to the question, if and how also systems without the native PA could benefit from the extra security provided by the return address protection. In this article, we explore KPAC, a set of efficient software-based approaches to bring the PA-based return-address protection onto the platforms without the hardware support in an easily adoptable (binary-compatible) and scalable manner. Technically, KPAC achieves this by either a synchronous trap-based emulation inside the kernel or an asynchronous novel memory-based invocation of a dedicated CPU core. Our experiments with the CortexSuite benchmarks, Chromium, and Memcached on a variety of platforms running Linux ranging from a Xilinx ZCU102 board over a Raspberry Pi 4 up to an 80-core Ampere Altra demonstrate the broad applicability and scalability of our approach. Furthermore, we discuss how the principles of KPAC can be generalized to the other suited problem areas.**

***Index Terms*—Computer security - application security, modeling - emulation, software - embedded software, software - system software - operating systems.**

I. Introduction

HARDWARE-BASED implementations for control-flow integrity (CFI) are becoming increasingly popular with Intel's control-flow enforcement technology (CET) [1], [2], [3] and ARM's pointer authentication (PA) [4] features being the most prominent candidates. Both provide measures to ensure the integrity of the programmer-intended control-flow by protecting the return addresses on the stack, a frequent target for the buffer-overflow attacks in combination with techniques like return- or jump-oriented programming (ROP/JOP) [5], [6], [7]. The hardware-based implementations overcome the most significant acceptance limitations of software-based CFI techniques: poor performance [8] and issues regarding the protection of the protection measure itself [9], [10], [11], [12]. While the ARMv8.3-A PA feature is long supported by standard compilers [13], [14] and the Linux kernel (in contrast to the Intel's CET, which only very recently made it into Linux [15]), for the last five years only Apple's A12/M1 actually implemented it. However, this is currently changing with Qualcomm's Snapdragon 8cx Gen 3 [16], which includes the PA support. As the PA-enabled binaries are fully backwards compatible (the special new instructions inserted by the compiler to encode/decode return addresses resolve to NOPs on CPUs without), we can expect to see a much broader adoption in the near future. Therefore, we consider it worthwhile to explore how and at what costs it would be possible to emulate the PA feature for the return address protection on the platforms without native PA.

A. About This Article

In this article, we provide, discuss, and evaluate four different approaches to emulate the PA-based return-address protection on the ARM processors without the PA support. We compare our results to the only attempt in this direction we are aware of, which is PAC-PL of Serra and colleagues [17], who employed an FPGA for the hardware-based encryption/decryption of return addresses. While PAC-PL provides an acceptable performance impact (negligible in many cases, up to 3× in some cases), it also comes with a number of drawbacks. First, PAC-PL is not binary compatible, as the code has to be compiled with a custom GCC extension. Furthermore, it requires the availability of an FPGA, which alone makes it unsuitable for many application scenarios. Consequently, their work triggered our attempt to look for more efficient software-based and, if possible, also binary-compatible approaches.

In a nutshell, we present a 2×2 matrix of software-based approaches that *either* require recompilation (like PAC-PL) *or* are binary compatible (via code patching) and *either* execute synchronously (by trapping) *or* asynchronously (by employing a dedicated CPU core) and compare them to *kpacpl*, a reimplementation of PAC-PL by its author. Our results show that with the extra core (which in real-world settings is arguably more available/affordable than the on-board-FPGA), we outperform the programmable logic (PL)-based approach in all the cases. Without the extra core, the synchronous and binary-compatible variant comes at a *worst-case* overhead of

Manuscript received 12 August 2024; accepted 12 August 2024. This work was supported in part by the German Research Foundation (DFG) under Grant LO 1719/4-1 (391395160). This article was presented at the International Conference on Embedded Software (EMSOFT) 2024 and appeared as part of the ESWEEK-TCAD special issue. This article was recommended by Associate Editor S. Dailey. *(Corresponding author: Illia Ostapyshyn.)*

Illia Ostapyshyn, Tim-Marek Thomas, and Daniel Lohmann are with the Institute of Systems Engineering (ISE-SRA), Leibniz Universität Hannover, 30167 Hannover, Germany (e-mail: ostapyshyn@sra.uni-hannover.de; thomas@sra.uni-hannover.de; lohmann@sra.uni-hannover.de).

Gabriele Serra is with the TECIP Institute, Scuola Superiore Sant'Anna, 56127 Pisa, Italy (e-mail: gabriele.serra@santannapisa.it).

Digital Object Identifier 10.1109/TCAD.2024.3443773

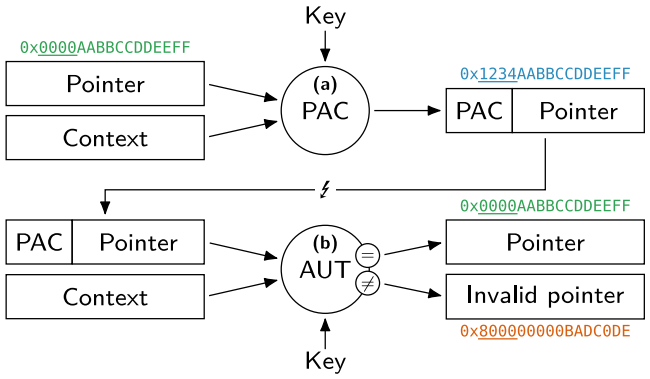

Fig. 1. PA mechanism.

17.37×, which is orders of magnitude below the costs for the software emulation reported so far [17]. For instance, Chromium on a Raspberry Pi 4 receives an actual slowdown by 7.13× in the JetStream benchmark, which could be considered as acceptable for the security-critical Web applications.

In particular, we claim the following contributions.

1) We describe KPAC, an approach for efficient software-based and optionally ARMv8.3-ABI-compatible PA as an extension to the Linux kernel.
2) We provide the remote-core system call (RCSC), a novel mechanism for efficient and safe interaction between the user-mode threads and dedicated kernel cores.
3) We explore and evaluate the design space for the KPAC on a variety of benchmarks and platforms.

The remainder of this article is organized as follows. Section II presents the ARM PA mechanism and Serra et al.'s [17] implementation based on the programmable logic. Section III describes the assumed threat model, Section IV our approach, and Section V the concrete implementation. In Section VI we evaluate the implementation variants and discuss our findings in Section VII. Finally, we review the further relevant literature in Section VIII and conclude this article in Section IX.

II. Background

A. ARMv8.3-A Pointer Authentication

pointer authentication (PA) is an approach to protect the code and data pointers with negligible footprint in performance, memory, and hardware. The key idea is to utilize the free bits in the unused upper part of pointers to store a cryptographic hash of the pointer value as its signature, so that unintended modifications can easily be detected. Typical memory configurations on AArch64 require only 48-bit virtual addresses, which leaves 16 bits for the signature, called the pointer authentication code (PAC) [4]. The signature algorithm is left to the implementation; ARMv8.3-A suggests the QARMA block cipher [18], which can efficiently be realized in hardware.

The mechanism features instructions for the creation and validation of these signatures. Fig. 1(a) visualizes the signing instructions using the mnemonic `PAC`. These instructions take three values, the 64-bit pointer itself, a 128-bit key (implicitly), and a 64-bit context information to produce a pointer with a PAC in its upper bits. The ARM implementation features five keys: two for instructions and data pointers each, and one general-purpose key. They are stored in the system control registers that are accessible only by higher privilege levels (i.e., the operating system kernel) and, thus, kept secret from the user applications requesting authentication. Linux, Windows, and XNU [16], [19], [20] manage these keys on a perprocess basis for the systems with the PA extension; Linux 5.7+, and XNU even support PA inside the kernel itself [21].

After the pointer has been signed using `PAC` instructions, their counterparts based on the mnemonic `AUT` are responsible for verification of signature before usage [Fig. 1(b)]: The `AUT` instructions take the authenticated pointer, recompute the PAC, and compare the result with the code stored in the signed pointer. If the signature matches, the PAC is stripped from the pointer. Otherwise a trap will occur, either immediately (ARMv8.6-A) or upon dereferencing of the pointer.

GCC and LLVM compilers already employ PA [13], [14] to protect the function return addresses (the *backward edges* of the control flow) that might be stored on the stack, where they would become vulnerable for the buffer overflow attacks. This is done by inserting `PACIASP` and `AUTIASP` instructions, operating on the link register (`LR`/`X30`) with the stack pointer (`SP`) as the context value in the function prologues and epilogues, respectively. In the ARM ISA, these instructions are located in the NOP instruction space, which ensures the backward compatibility of newly compiled programs with CPUs lacking the PA extension. As leaf functions never push their return address onto the stack, the standard setting is to omit the PA instructions in them.

However, half a decade following the introduction of the PA mechanism in the ARM specification and despite ubiquitous compiler and OS support, only few systems are readily available that implement it in hardware. Most notably, the A12 chip presented by Apple in 2018 and all its successors come with PA [22] mechanism. This has only recently be complemented by Qualcomm's Snapdragon 8cx Gen 3 SoC [16], which brings the PA also to the Windows and Android domains. Nevertheless, we face a plethora of systems with no support for the PA and adoption will continue to be slow, especially in the embedded domain.

B. PA Using FPGA: The PAC-PL Approach

As a solution for this, Serra et al. [17] implemented the PA mechanism on an SoC featuring an field programmable gate array (FPGA). Since, we base our work on theirs and use a reimplementation as a comparison point, we briefly present and discuss it here.[1]

The main idea of PAC-PL is to perform the signing and authentication of pointers using PL on the SoC. Its architecture consists of two components: 1) a QARMA block cipher [18] crypto engine and 2) an AXI subordinate device, which handles interaction between the crypto engine and the host over the AXI bus via the memory-mapped registers, which are mapped into the kernel- and user-level address spaces,

[1] Unfortunately, the original PAC-PL code underlies the IP restrictions, but its author provided us with a personal reimplementation of its core features.

respectively. Instead of using the ARMv8.3-A PAC/AUT instructions, the signing/authentication of pointers is triggered by writing into the corresponding registers, which lets the PAC-PL accelerator generate, remove, and check the PAC. Hence, the approach is not binary-compatible: the software has to be compiled with a custom GCC plugin that generates the necessary instructions.

Since, QARMA is designed to be particularly fast in hardware, the overheads are dominated by the communication latency, which is costly due to the mismatch in the clock frequencies between the FPGA and the host CPU. While calculating the cipher itself only requires ten host cycles, a complete PAC/AUT operation takes at least 426 cycles. In our measurements on a Xilinx ZCU102 at 1.2 GHz this approach leads (with the most strict PA application mode all explained later) to an average overhead of 34% for the CortexSuite [23] benchmarks. The operation time is bounded, making it suitable for the real-time systems.

In the paper [17], the utilization of an FPGA is partly justified by comparing it to the performance results from a software-based emulation of their approach, which bears much higher (up to five orders of magnitude!) overheads. However, this extremely high overhead is likely caused by the employed user-kernel interface, which induces two page faults per PAC/AUT transaction (hence, the four page faults per protected function) to emulate a PAC-PL device. Furthermore, while QARMA is optimized for the hardware implementations, another cipher might be more suitable for a CPU-based software implementation. Last but not least, the work does not evaluate nor mention support of the multithreaded applications. In the remainder of this article, we explore the options for more efficient and optionally binary-compatible PA emulation that scales well in concurrent environments.

III. Security Objectives and Threat Model

ARMv8.3 pointer authentication was developed to accomplish the pointer integrity. Intuitively, pointer integrity seeks to prevent the alterations to pointers while residing in memory, ensuring that the value of a pointer at the time of its use (i.e., dereferencing) remains consistent with the value intended during its creation or storage. Control-flow attacks and numerous other data-oriented attacks hinge on manipulating the susceptible pointers. Consequently, the enforcement of pointer integrity defends against these attacks. The security objective of ARMv8.3 PA, therefore, consists of preventing the attacker from forging pointers used by a vulnerable program.

Likewise, KPAC pursues the same security guarantees. Our approach shall satisfy the following functional requirements.

1) *Pointer Integrity:* Prevent and detect the use of the corrupted code or data pointers.
2) *Attack Resistance:* Resist attempts to forge the valid pointers and resist pointer reuse attacks.

Further, we identify nonfunctional requirements, which allow wider compatibility as follows.

1) *Compatibility:* Enabling pointer integrity protection of existing programs without interfering with their operation even without dedicated hardware support.
2) *Performance:* Minimize run-time overhead by providing configurable protection scopes as a tradeoff between the hardening and performance.

The following assumptions define the attacker's capability, consistent with the prior works in this area ([24], [25]). Our adversary model reckons with an attacker as follows.

1) With unrestricted user-space memory read and write capabilities, constrained exclusively by the data execution prevention (DEP) mechanism, therefore with the ability to read any program memory and write to the nonexecutable segments exploiting the input-controlled memory corruption errors in the victim process (e.g., controlling return addresses, function pointers, or VTable pointers).
2) Disposes of a full knowledge of the process memory layout and has successfully bypassed the address space layout randomization (ASLR), if present.
3) With no control over privilege levels higher than the user level, meaning without the ability to access the kernel space or higher privilege levels.

Note that, assumptions 1 and 2 rule out the feasibility of randomization-based defenses susceptible to the information disclosure, such as stack canaries, ASLR, or software shadow-stacks. KPAC was designed to maintain its effectiveness even when the complete memory layout of the victim process is disclosed as long as the assumption 3 holds. Therefore, the attacker cannot deduce the keys, which are located in memory not directly readable from the user space.

According to the presented threat model, KPAC is as secure as ARMv8.3 PA. The PAC-PL Serra et al. [17] have obsoleted the assumption 3 by employing ARM TrustZone, which creates isolated secure environments to protect the sensitive data for the key management. This extra protection is applicable to KPAC as well, but not further explored in this article.

IV. KPAC Approach

A key point of the ARMv8.3-A PA (also mimicked by PAC-PL) is that it delegates key management to the OS running on the EL1 privilege level (the supervisor mode) ensuring higher protection. In order to stay true to this property, KPAC delegates key management and exception handling to the OS kernel. As a corollary, the partial interpretation of the PAC and AUT operations by software has to take place inside the kernel, which generally induces the significant overhead, as every operation thereby comes with a minimum of two user–kernel context switches. Mitigating this overhead as far as possible is one key to an efficient software implementation. The other key is the overhead of the signing algorithm itself.

The central component of KPAC is a Linux kernel extension, which implements the PA backend. Since, QARMA is not suitable for the fast software implementation, *SipHash* [26] has been selected as the cryptographic hashing algorithm instead. It is designed to be efficient and secure with short inputs to compute a 64-bit message authentication code, which is truncated to the unused bits of the pointer.

The kernel extension exposes two interfaces for the user-space applications to request the PA (Fig. 2) as follows.

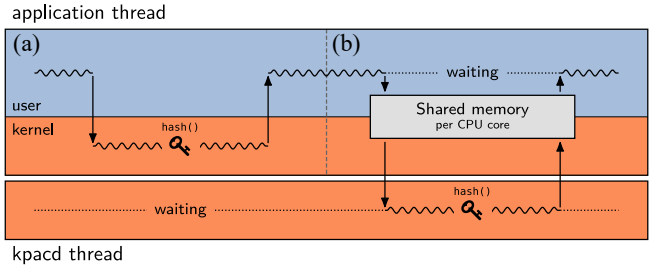

Fig. 2. Application requesting PA by making (a) an *svc* request followed by (b) a *kpacd* request.

TABLE I
PROPERTIES OF PRESENTED EMULATION APPROACHES

	Approach	Hardware requirements	Multi-threaded	Binary compat.	Bounded WCET	Average overhead
	native	ARMv8.3-A	✓	✓	✓	0 %
(C)	*kpacpl-static*	FPGA			✓	34 %
(AC)	*svc-static*		✓		✓	88 %
(BC)	*kpacd-static*	extra core	✓		✓	17 %
(D)	*kpacpl-libkpac*	FPGA		✓	✓	44 %
(AD)	*svc-libkpac*		✓	✓	✓	87 %
(BD)	*kpacd-libkpac*	extra core	✓	✓	✓	31 %

1) *Synchronous system calls* (*svc* requests), which execute the PA within the invoking thread. This is the canonical way to implement an user kernel interaction.
2) RCSC, a novel asynchronous communication protocol based on the per-CPU shared memory, which executes the PA on a dedicated kernel core running the *kpacd* daemon. This (kind of) mimics the idea of PAC-PL to use extra hardware (here a CPU core instead of an FPGA) for the PA.

To instrument applications with either invocation scheme, two methods have been investigated as follows.

1) *Static instrumentation* by a compiler plugin (as in PAC-PL). This is, assumingly, the most run-time efficient way, as it facilitates static optimization of the code and also provides configurable protection scopes for the overhead mitigation.
2) *Load-time instrumentation* by a dynamic library (*libkpac*) that is applied by the `LD_PRELOAD` feature of the system's dynamic loader and patches at load time all `PACIASP`/`AUTIASP` instructions in the code to invoke KPAC instead. This provides full binary compatibility for ARMv8.3-A binaries that were compiled with PA support.

Table I briefly summarizes the resulting four KPAC variants, together with PAC-PL and a native ARMv8.3-A processor. The given overhead numbers should be considered as a ballpark figure only. They describe the geometric mean over all the CortexSuite benchmarks on a Xilinx ZCU102 at 1.2 GHz. Our experiments with Apple's M1 Ultra did not yield any measurable overhead for the native ARMv8.3-A PA.

V. IMPLEMENTATION

We integrated KPAC into the Linux kernel version 6.1. The compiler support for the static instrumentation is provided as a GCC 12.2 plugin.

A. *svc*: The Synchronous System Call Interface

The AArch64 instruction set defines the `SVC` (supervisor call) instruction, which transfers the control flow to the EL1 privilege level running the OS kernel. This instruction is used across the operating systems to implement the system calls and has a 16-bit immediate argument. On Linux, the system call number is passed in the `W8` register and the immediate argument of the `SVC` instruction is ignored.

We extend the Linux system call interface to emulate the ARMv8.3-A `PACIASP` and `AUTIASP` instructions by reserving two values of the `SVC` instruction's immediate argument. These new emulation calls thereby require a single instruction in the code that only alters the `LR`, making them semantically equivalent to the `PACIASP`/`AUTIASP` instructions emitted by the standard compilers. As the execution time of the `SVC` instruction and the SipHash algorithm takes bounded time [26], the emulation is also suitable for the hard real-time settings that demand bounded WCETs.

B. *kpacd*: The Remote-Core System Call Interface

On many platforms, context switches into the OS kernel induce a high overhead for changing the privilege level and the address space. Furthermore, the executed kernel code may put extra pressure to the CPU-local caches and the TLB, significantly impairing the performance [27]. An alternative approach is to run the kernel services asynchronously on a dedicated core [28], [29] that always stays in the kernel mode. The services are invoked by a shared-memory interface between both the cores, omitting the above overhead altogether. Our rcsc implements this idea for the Linux, while additionally providing for the lock-free per-core separation.

With RCSC, one or several CPU cores are reserved for the KPAC and execute the *kpacd (Kernel PAC Daemon)* in the kernel mode, which polls a shared memory page for the PA requests. This is comparable to the PAC-PL, where the service core acts as the accelerator instead of an FPGA.

Sacrificing a full core just for the PA purposes might appear as an odd design decision, given that such core induces a *much* higher hardware overhead than a small FPGA. However, in practice, an unused core is way more often available and actually cheaper for many embedded systems than an FPGA.

Invocation of kpacd: Listing 1 demonstrates the assembly code corresponding to an authentication RCSC to the 2kpacd thread. After storing the pointer and the context value at the respective offsets in the RCSC page (L3), the application hands off the request by writing the operation code into the first *status word* of the page (L6), which wakes the remote *kpacd* to perform the requested operation. The status word is then checked in a loop (L9–11) by loading the first word of the page with the exclusive load (`LDXR`) instruction, branching to `WFE` if the value is not zero (zero signals completion). The `WFE` instruction hints the CPU core to enter a low-power state, until a wake-up event occurs [4]. A remote store (by the *kpacd* core) to this location, which was recently read using an exclusive load (`LDXR`), generates such an event. Hence, on both the sides the polling does not come with an extra energy/heat overhead. The combination of `LDXR` and `WFE` is

```
1       mov     x9, #KPAC_BASE
2       mov     x10, sp
3       stp     lr, x10, [x9, #REG_PLAIN] // store pointer and context
4
5       mov     x10, #OP_PAC
6       stlr    x10, [x9]                 // request operation
7
8       sevl
9 1:    wfe                               // sleep for event  <--+
10      ldxr    x10, [x9]                 //                    |
11      cbnz    x10, 1b                   // until completion ---+
12      ldr     lr, [x9, #REG_CIPHER]     // obtain result
```

Listing 1. Assembly code of a function prologue requesting a signed pointer from *kpacd*.

also used in the AArch64 `__CMPWAIT_CASE` macro of the Linux kernel.

Multithreading Support: On multiprocessor systems, multiple threads from within the same or different processes might invoke *kpacd* simultaneously. These concurrent requests need to be isolated and coordinated. RCSC solves this by providing an individual RCSC page for each core, which is (implicitly) used by the thread currently executing on this core. Hence, no synchronization is required when accessing the RCSC page, enabling scalability. As each core executes exactly one thread at a time, the per-core pages also ensure isolation. Upon a switch to another thread, the scheduler completes any pending RCSC requests and saves the relevant content (24B) of the shared page in the thread control block.

Technically, the provision of per core pages (which we consider a general mechanism) has to be integrated with the virtual memory subsystem. For this, the data structure representing the address space and containing the pointer to the top-level page directory (*page global directory* and *PGD*), is extended to support a different PGD per core. As illustrated in Fig. 3, all the entries in these PGDs are kept synchronized except for one. The entry leading to the core-local RCSC page. Thereby, all cores use the same virtual address to access their core-specific memory. This comes at the cost of duplicated PGDs for processes using the *kpacd* service. Moreover, when a PGD entry is modified, the changes have to be mirrored into the PGDs of other CPU cores. The performance overhead of this is negligible, since the top-level page-directory entries are only populated at the process start and rarely modified during the execution. As the underlying page tables are shared, all the changes in them (e.g., induced by an `mmap()`) are immediately seen by other CPU cores and require no further mirroring nor synchronization.

For load balancing in larger multicore systems, an arbitrary number of cores could be assigned to *kpacd*. Each *kpacd* core services a fixed set of application cores in a round-robin manner. Hence, the worst-case service time of *kpacd* is also bounded in multicore settings.

C. Static Instrumentation via Compiler Plugin

Applying either of the KPAC invocation approaches for the return-address protection requires adding the signing and authentication code in the prologues and epilogues of functions. One way to achieve this, also taken by Serra and associates [17], is to employ a compiler plugin and add an additional pass working on the register-transfer language representation of the program.

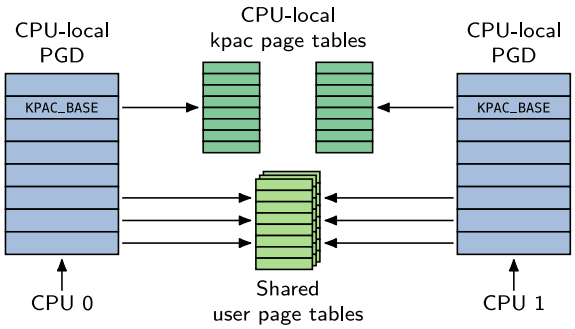

Fig. 3. Page table arrangement introduced by CPU-local top-level page directories (PGDs).

Protection Scopes: If compiling code for ARMv8.3-A with `-mbranch-protection`, GCC would apply the PA-based return address protection on the prologues and epilogues of all the nonleaf functions (which push the return address to the memory). As the PA-based return address protection basically just adds two instructions to a protected function, this does not induce any measurable overhead. In contrast, a PA emulation induces a much higher overhead, so it might be worthwhile to explore different protection scopes and let the compiler plugin only instrument the most vulnerable functions.

Our compiler plugin therefore resembles the protection levels of GCC's and Clang's `-fstack-protector` feature [13], [14], a purely compiler-based CFI measure that comes with the common limitations regarding performance and actual protection (cf. Section I). However, its defined protection levels are established among the developers who have to trade between the hardening and performance of their software. The three different protection scopes are referred to in the following as *char*, *strong*, and *all*.

- *char* protects the nonleaf functions that place char arrays of at least size 8 (*ssp-buffer-size*) on the stack and nonleaf functions, that perform dynamic stack allocation with `alloca()`. This protection scope bears the lowest performance impact, while already providing some protection for simple but common buffer-overflow attacks.
- *strong* extends the scope of protected functions to the nonleaf functions that accommodate any arrays or variables that have their address taken on the stack. This further mitigates the range of some advanced attack techniques based on the ROP at a moderate performance impact.
- *all* extends the scope even further to all the (nonleaf) functions, which provides the highest protection level, but also induces significant performance costs.

Our plugin supports all the optimization levels, but automatically disables the *ipa-ra* and *shrink-wrap* optimizations, as we depend on the caller-saved registers to be actually saved and the function prologue at the beginning of a function.

D. Load-Time Instrumentation via *libkpac*

While recompiling the existing applications might be feasible in some settings (e.g., embedded applications), this is often

```
paciasp // PAC lr
sub  sp, sp, #0x60
stp  x20, x19, [sp, #32]
stp  fp, lr, [sp, #16]

// function body omitted

ldp  fp, lr, [sp, #16]
ldp  x20, x19, [sp, #32]
add  sp, sp, #0x60
autiasp // AUT lr
ret
```

(a)

```
sub  sp, sp, #0x60
stp  x20, x19, [sp, #32]
stp  fp, lr, [sp, #16]
bl   kpacd_pac_24 // PAC [sp+24]

// function body omitted

bl   kpacd_aut_24 // AUT [sp+24]
ldp  fp, lr, [sp, #16]
ldp  x20, x19, [sp, #32]
add  sp, sp, #0x60
ret
```

(b)

Listing 2. Example function from Memcached patched by *libkpac* for *kpacd* invocation. The `kpacd_{pac,aut}_24` trampoline operates on the return address at the offset 24 from the SP (a) Before patching (b) After patching.

not the case, especially in end-user environments. Thus, we propose a binary-compatible method of adding the software-emulated PA to the programs already compiled for ARM PA by providing a run-time library (*libkpac*). *libkpac* patches the program at load time and can be applied selectively to the whole system or single application processes.

Technically, *libkpac* is injected by setting the `LD_PRELOAD` environment variable, which causes the system's dynamic loader to additionally load the library and execute its constructor function. The `LD_PRELOAD` mechanism provides for maximum flexibility on the user's side. For example, the user might run one instance with KPAC support to improve the security and another one without, for the performance-sensitive activities.

The constructor function parses the memory map of the process, exposed by Linux in the *procfs* file system, and takes note of the executable memory areas in the address space. It then iterates over these areas and searches for the `PACIASP` and `AUTIASP` instructions. At these places, the code needs to be patched to invoke KPAC by either the synchronous *svc* or the asynchronous RCSC mechanism.

svc-Only Mode: In this mode, the `PACIASP` and `AUTIASP` instructions are simply replaced by their respective *svc* equivalents. As this takes only a single opcode and clobbers the same set of registers (just the `LR` register), this is trivially possible in all the cases.

kpacd and kpacpl Modes: Invoking *kpacd* or *kpacpl* via their shared-memory interface requires inserting additional branches to a subfunction, which is more complicated and not (safely) possible in all the cases. The general idea of patching a function for such invocation is demonstrated in Listing 2. Fundamentally, it is not possible to just replace `PACIASP` and `AUTIASP` by a call to the *kpacd*/*kpacpl* invocation, as this would overwrite the return address stored in the `LR` to be protected. Instead, the invocations have to be put at the end of the prologue (beginning of the epilogue), when `LR` has been saved onto the stack. The required space for these calls is created by shifting the stack frame (de)allocation sequences into the `PACIASP`/`AUTIASP` instructions. However, as compilers might do arbitrary things in their function pro/epilogues (e.g., reordering), the patching falls back to the *svc* mechanism if no familiar stack frame (de)allocation sequence is detected. When instrumenting binaries for *kpacpl*, the *svc* fallback uses the accelerator for QARMA computation from the kernel space.

Upon invocation, the actual return address to be en/decoded resides on the stack, but at an varying offset that depends on the function-specific stack frame. To deal with this, *libkpac* provides trampoline functions for the offsets from 0 to 504 bytes (at machine word granularity). As the AArch64 BL instruction can jump only in the region of ±128 MiB, *libkpac* furthermore places the trampolines in neighboring address-space holes in the case of large text sections. For example, this is required to fully patch the Chromium's 161.02 MiB executable segment.

VI. Evaluation

In our evaluation, we 1) demonstrate the latency of a single `PAC`/`AUT` transaction; 2) show that our approaches efficiently implement PA in software; 3) illustrate the multicore scalability using the memory caching system Memcached; and 4) showcase the ease use of the binary-compatible approaches using the *Chromium browser*.

We have integrated our mechanism into Linux 6.1 on the three systems: 1) Xilinx Zynq UltraScale+ ZCU102 evaluation board with XCZU9EG MPSoC at 1.2 GHz; 2) Raspberry Pi 4 single-board computer with Broadcom BCM2711 at 1.8 GHz; and 3) Gigabyte R152-P31 rack server with 80-core Ampere Altra Q80-30 CPU at 3 GHz. The ZCU102 evaluation board allows us to directly compare our approaches with the PAC-PL reimplementation as the SoC features an FPGA fabric on the chip. The Raspberry Pi resembles a typical medium-end hardware used in embedded appliances. This does obviously not hold for the 80-core Ampere Altra/Memcached setup, which we include for the sole purpose of stressing the multicore scalability of our approach.

As the baseline, we chose to run the targets without any enabled PA. This corresponds to our measurements on the Apple's M1 Ultra (see Table I), which did not yield any measurable overhead for the native PA.

A. Cost of User–Kernel Interaction and Hashing

First, we evaluate the cost of the user kernel interaction methods introduced in Section IV by measuring the end-to-end latency of the transactions on the mentioned systems. Simultaneously, we demonstrate the high overhead of the QARMA hashing algorithm when implemented in software and motivate the choice of SipHash for fast hashing. Table II demonstrates the 99th percentile latencies for the PA requests in clock cycles over 32 million samples. The cycles are measured using the PMU's cycle counter `PMCCNTR_EL0`. Consequently, the *kpacd* spin loop does not use `WFE`, as it is undefined whether the counter continues to increment in low-power state [4].

The measurement in the *None* row of Table II does not perform any hashing and thus represents the raw communication overhead for the *svc*- and RCSC-based transactions. Comparing the raw communication overhead across the systems, both the ZCU102 and Raspberry Pi 4 require over 2000 cycles for an NOP system call. The *kpacd* request on these systems is much faster, by factor 4.99 on the Raspberry Pi 4 and by factor 9.31 on the ZCU102 evaluation board.

TABLE II
99TH PERCENTILE ROUND-TRIP LATENCY OF PA REQUESTS IN CLOCK CYCLES. (A) AMPERE ALTRA Q80, 3 GHZ. (B) BROADCOM BCM2711, 1.8 GHZ. (C) XILINX XCZU9EG, 1.2 GHZ

(A)

Hashing alg.	*svc*	*kpacd*
None	389	476
SipHash	434	488
QARMA	3903	3947

(B)

Hashing alg.	*svc*	*kpacd*
None	2095	420
SipHash	2170	508
QARMA	6779	5332

(C)

Hashing algorithm	*kpacpl*	*svc*	*kpacd*
None	643	2020	217
SipHash	—	2122	339
QARMA	650	11608	8231

On the Xilinx ZCU102 evaluation board, a ***kpacd*** transaction takes only 217 clock cycles, which amounts to 180.8 ns. On the same system, a round trip to the PL takes 643 cycles or 535.8 ns. In contrast, the Ampere Altra Q80 system shows a different picture: a system call is 18.28% faster than a round trip over the shared memory and takes only 389 clock cycles or 129.7 ns. This demonstrates that high-end systems might implement the system calls more efficiently and could profit from the fast software PA without an additional accelerator core.

Moving onto the hashing algorithms, the ARM-suggested QARMA cipher comes with an overhead of at least 3000 cycles on all the systems even when subtracting the raw communication costs. For instance, a system call calculating the QARMA cipher takes 9.67 μs on the ZCU102 evaluation board. Having an FPGA at its disposal, this is the only system providing a low latency for QARMA with ***kpacpl*** taking 650 clock cycles to authenticate a pointer. In fact, the QARMA calculation costs are fully amortized by the communication overhead, as the round trip latency is nearly equal to the latency without hashing. This stems from the mismatch in the clock frequencies between the FPGA and the host CPU, together with the costs for the data transfer.

SipHash offers a practical alternative to the QARMA cipher. Subtracting the communication overhead, its calculation takes roughly 100 cycles on all the systems. This results in the round-trip latency of 339 cycles (282.5 ns) on the ZCU102 evaluation board, 434 cycles (144.7 ns) on the Ampere Altra machine via ***svc***, and 508 cycles (282.2 ns) on the Raspberry Pi 4. Therefore, we use SipHash in the remainder of evaluation.

B. Approach Comparison

For the comparison of the KPAC approaches, we chose the CortexSuite [23]. It is a representative embedded workload, as it consists of machine learning, computer vision, language processing, and IoT tasks. As the baseline, we compile all the benchmarks without any protection using GCC 12.2 with the `-O2` optimization level. For the static instrumentation, the benchmarks are compiled with our compiler plugin with the same optimization level.[2] For the binary-compatible ***libkpac*** evaluation, the compilation flags are complemented with `-mbranch-protection=pac-ret` to add the return-address protection using ARMv8.3-A PA. The benchmarks are executed on the Xilinx Zynq UltraScale+ ZCU102 evaluation board, allowing us a direct comparison to the PL-based approach ***kpacpl***.

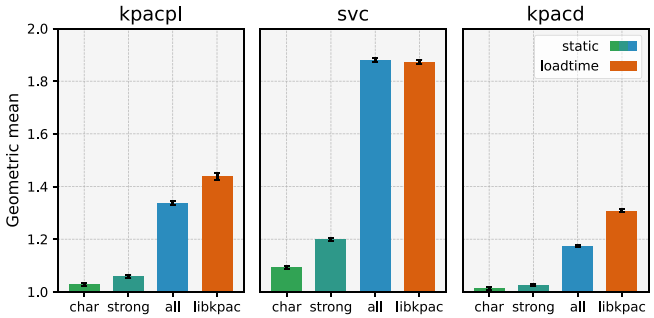

Fig. 4. Geometric mean of CortexSuite benchmark run durations normalized to the baseline run on Xilinx ZCU102.

Static Instrumentation: The advantage of instrumenting applications statically using the compiler plugin lies in the ability to mitigate the performance overhead using the protection scope heuristics introduced in Section V-C. Thus, we evaluate the three protection scopes and compare the communication approaches to each other. Fig. 4 provides a high-level overview of the overhead over all the CortexSuite benchmarks (summarized using the geometric mean) and Table III breaks down the figure for the individual benchmarks.

The highest overhead is measurable for the most secure protection scope ***all***, where all the nonleaf functions are protected. The ***svc***-based instrumentation has the highest average overhead of 1.88× and is outperformed by ***kpacpl*** with the average overhead of 1.34×. The ***kpacd*** approach leads in this category with the average overhead of 1.17×.

The protection scope ***strong*** reduces the overhead to a lower figure for all the approaches, while keeping the security guarantees high as the likelihood of a stack-buffer overflow occurring in a function that never exposes the addresses to its stack is exceedingly low. There, ***kpacd*** still outperforms all the approaches with an average overhead of 1.88×. The worst-case overhead for ***kpacd-strong*** is observed in the *sphinx* benchmark with 1.20× or 20%. The respective ***kpacpl*** figures are 1.06× for the average and 1.48× for the worst case in *sphinx*. Even for ***svc***, which is inherently suboptimal on this system due to the cost of the system calls, the average overhead is reduced to 1.20× with the worst case of 3.34×.

Depending on the application, the security can be traded for the performance by reducing the protection scope to the ***char***: functions that allocate the character arrays on the stack. Note that, this is the default mode of GCC's stack protector and the only one evaluated by Serra and associates for PAC-PL originally. This reduces the worst-case overhead to 1.26× for ***kpacpl***, 2.25× for ***svc***, and 1.11× for ***kpacd***. On average, ***char*** yields the lowest overhead regardless of the approach.

[2]This excludes optimizations incompatible with current compiler plugin prototype discussed in Section V-C.

TABLE III
RUN TIMES OF CORTEXSUITE BENCHMARKS NORMALIZED TO THE BASELINE RUN WITHOUT PROTECTION ON XILINX ZCU102

Benchmark	Baseline run duration [s]	*kpacpl*				*svc*				*kpacd*			
		char	*strong*	*all*	*libkpac*	*char*	*strong*	*all*	*libkpac*	*char*	*strong*	*all*	*libkpac*
lda	18.25	1.01	1.01	4.37	7.53	1.04	1.04	17.38	17.37	1	1	2.38	5.37
sphinx	12.39	1.26	1.48	3.21	3.6	2.25	3.34	11.78	11.43	1.11	1.2	1.87	2.3
rbm	21	1	1.08	1.17	1.17	1	1.4	1.8	1.8	1	1.03	1.06	1.07
srr	29.12	1.01	1.04	1.04	1.11	1.01	1.16	1.16	1.16	1	1.02	1.02	1.08
svd3	14.55	1	1	1.02	1.03	1	1	1.14	1.14	1	1	1.01	1.01
motion-est.	9.42	1.03	1.04	1.05	1.03	1.03	1.08	1.13	1.1	1.02	1.03	1.03	1.01
spectral	6.96	1	1	1	1	1	1	1	1	1	1	1	1
pca	3.29	1	1	1	1.01	1	1	1	1.01	1	1	1	1
liblinear	25.75	1	1	1	1	1	1	1	1	1	1	1	1
kmeans	33.74	1	1	1	1	1	1	1	1	1	1	1	1
Geometric mean		1.03	1.06	1.34	1.44	1.09	1.2	1.88	1.87	1.01	1.03	1.17	1.31

TABLE IV
LOAD-TIME STATISTICS FROM LIBKPAC PATCHING ROUTINE

Benchmark	Text section size [KiB]	Patched locations	Patching time [µs]	
			svc	*kpacd*/*kpacpl*
lda	9.11	53/54	455	640
sphinx	284.65	1529/1555	7928	8275
rbm	3.46	18/26	288	469
srr	12.45	30/39	490	654
svd3	25.15	145/145	872	1049
motion-est.	4.01	20/22	300	491
spectral	5.71	17/18	377	559
pca	5.40	18/18	293	496
liblinear	37.90	119/128	1060	1262
kmeans	2.09	6/8	286	473
Average	38.99	97.12 %	1235	1437

Load-Time Instrumentation: Next, we examine the binary-compatible approach based on the load-time patching using *libkpac*. In terms of the security, load-time patching is tantamount to the protection scope *all*, as GCC hardens all the nonleaf functions with `PACIASP`/`AUTIASP` instructions.

The average overheads are 1.44 and 1.31× for *kpacpl* and *kpacd*, respectively. The worst overhead can be observed in *svc*-only mode (*svc-libkpac*), where the load-time instrumentation yields 1.87× overhead on average. The is slightly better than the respective static approach (*svc-all* with 1.88×) as the dynamic instrumentation (unlike the compiler plugin) does not require disabling any optimizations. Here, *kpacd-libkpac* represents the middle ground between *kpacd-all* and *svc-all*, since *libkpac* replaces `PACIASP`/`AUTIASP` conservatively with a call to the optimized *kpacd* routine, resorting to costly *svc* where no familiar prologue/epilogue sequences are recognized (due to the instruction reordering).

Table IV provides the additional statistics on the load-time patching. Overall, *libkpac* in the *kpacd*/*kpacpl* modes manages to successfully patch 97.12% of prologues and epilogues in CortexSuite. The time required for patching does not exceed 10 ms for any of the benchmarks and correlates roughly with the size of the executable segment. The required average time per KiB is 32 μs for *svc*-only and 37 μs for *kpacd*/*kpacpl*.

C. Case Study: Memcached

Despite the fact that *kpacd* requests do not require synchronization with the other threads, there is a risk of high contention on a single service core when serving the multiple application cores. To accommodate highly parallel applications, the KPAC kernel allows flexibly configuring the amount of *kpacd* service cores and the mapping to the application cores that they serve. The synthetic benchmarks from the CortexSuite are single-threaded and do not assess the multithreading aspect of KPAC. Hence, we deploy Memcached 1.6.22 on a Gigabyte R152-P31 rack server featuring the Ampere Altra Q80-30 at 3 GHz with 80 Neoverse-N1 cores and 256 GiB of DRAM. The machine offers uniform memory access (UMA) latencies for all the cores.

We chose Memcached for several reasons. The code base is written in plain C, making it easy to deploy it with custom CFI techniques like our software-emulated PA. Furthermore, benchmarking tools are readily available. Also, Memcached is a realistic use-case for the PA as it is used in security relevant environments, for example in combination with an LDAP service for the user authentication.

Workload: Memcached server is compiled with the default compiler flags, including the `-O2` optimization level. It is complemented with PA-based return address protection and ran with 32 threads pinned to 32 CPU cores. For the client side, we use the *memtier_benchmark* [30], which is developed by Redis specifically for benchmarking the key-value databases. The Memtier benchmark starts 32 threads on another 32 cores of the same machine. Each threads opens 50 connections (resulting in 1600 active connections) and records latencies of SET and GET requests with the default SET:GET ratio of 1:10 for 100 s. We vary the amount of *kpacd* service threads on the remaining 16 cores of the machine.

Results: Fig. 5 displays the average latency in milliseconds as well as the 99th-percentile tail latency for the baseline reference without PA, *svc*-, and *kpacd*-instrumented runs (statically via the compiler plugin and load time via *libkpac*). For *kpacd*, the latencies are measured for different amounts of the service cores (*kpacd-x*).

The baseline average and tail latencies are 1.09 and 1.66 ms, respectively. When using one service core there is a high latency for all the protection scopes except for *char*. In fact,

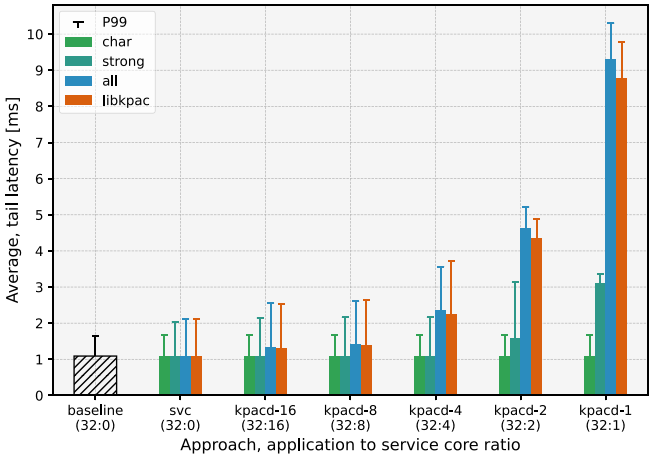

Fig. 5. Average latencies and 99th percentiles for the Memtier benchmark with 0–16 service cores (*svc* and *kpacd-x*).

due to the low amount of protected functions, *char* shows no measurable change for all the approaches both in average and tail latencies. For all, the average latency increases almost tenfold to 9.30 ms. This figure halves as we double the amount of service cores: it amounts to 4.62 ms (4.24×) for the two service cores and 2.37 ms (2.17×) for the four service cores. Distributing the load over eight *kpacd* threads and above, they are no longer saturated, and the latency does not exceed 1.43 ms (1.31× of the baseline). The figures are similar to all for *libkpac* instrumented *kpacd* experiments as *libkpac* manages to patch 91.78% of locations with a *kpacd* invocation. Looking at the *strong* protection scope, the average latency increase is low for the four service cores and above. For two service cores the increase is 1.45× or 1.58 ms.

Interestingly, *svc*-instrumented servers (including *all* and *libkpac* variants) demonstrate the same average latency as the baseline run. The tail latency, however, shows a minor increase of 28.52% for *libkpac* and 23.20% for *strong*. This stems from the architecture of the used machine. As demonstrated in Section VI-A, the Ampere Altra Q80 CPU features particularly the fast system calls. Combined with the fact that the *svc* approach does not induce contention in multithreaded scenarios, this results in fast return-address protection. This highlights that the underlying hardware and architecture needs to be taken into account when applying the mechanism. In this case, the *svc-libkpac* approach can be easily applied to an already compiled Memcached server (e.g., from the distribution's repository) without significantly affecting the performance of the database. This comes at a cost of relatively short 1.55 ms patching time for the 147 KiB executable segment of the Memcached binary and all the libraries it links with.

D. Case Study: Chromium

Investigating the binary-compatible approach further, we concentrate on its ease of use with the already existing software and toolchains. We demonstrate this by applying *svc*- and *kpacd*-based PA using *libkpac* to the *Chromium browser*. For this we use the Raspberry Pi 4 single-board computer featuring a quad-core Cortex-A72 64-bit SoC clocked at 1.8 GHz. We chose this system, as it represents a small lightweight ARM desktop PC.

The *Chromium browser* is a long-existing project with a large code base, leading the browser market with the usage share of 63% on all the platforms (September 2023) [31]. Moreover, the binary Chromium package of the AArch64 Debian distribution is already hardened with the ARMv8.3-A return address protection using `PACIASP`/`AUTIASP` instructions. However, this protection has no effect on systems without PA. This makes Chromium a prime target for our evaluation, as we want to showcase the ease of use and the low adoption hurdle for the end-users. Security in Web browsers is highly relevant in general, as the users use them for the online banking, healthcare information, and a wide range of other sensitive tasks. The Chromium project itself states that around 70% of their security bugs are the memory safety problems [32]. The severity of those bugs would be alleviated by enabling PA.

Workload: To stress test our mechanism and give an intuition on how usable the binary-compatible approaches are for the user-oriented applications, we execute the *Speedometer 2.1*, *Jetstream 2.1*, and *Motionmark 1.2* browser benchmarks from the WebKit's Browserbench suite [33]. Speedometer emulates user input by adding, changing and removing to-do items from a Web application, evaluating the browser's responsiveness. Jetstream, on the other hand consists of the Web assembly and JavaScript benchmarks (64 in total), which are then scored using the geometric mean. These benchmarks consist of several cryptography algorithms, data processing tasks, parsers, and so on. The third Browserbench benchmark, Motionmark, puts the browser's graphics engine to the test by animating complex scenes.

Results: Our library manages to patch 95:85% of prologues and epilogues of the *Chromium browser* binary with the optimized *kpacd* invocations. As this binary is quite large (161.02 MiB executable segment), we need 549.89 ms to patch it. This corresponds to the rate of 3.42 ms/MiB. Extending the patching onto the libraries that link with *Chromium browser*, the patching takes 571.13 ms. Out of those libraries, only *libffmpeg.so* (part of the Chromium package) and *libgnutls.so* (distribution's version) are compiled with ARMv8.3-A PA. All the experiments ran without any changes to the source code and without any crashes or errors.

Fig. 6 displays the reached scores for both the benchmarks. The optimized *kpacd*-based instrumentation reduces the scores by a factor of 4.43× for the speedometer benchmark and by a factor of 3.54× for the Jetstream benchmark. On the other hand, the *svc* approach reduces the scores by a factor of 10.63 and 7.13× for the speedometer and Jetstream, respectively. The Motionmark benchmark shows only minor difference between the three browser variants, with a worst-case score reduction of 14.05% for *svc*.

The results are consistent with the transaction latencies measured for this system in Section VI-A, where performing a system call calculating SipHash PAC has the quadruple latency of an RCSC transaction. The high overall overhead can be attributed to the fact that the browsers spend significant amount

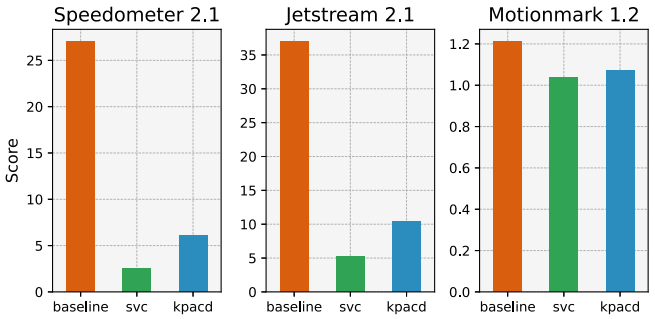

Fig. 6. Scores as reported by the Browserbench benchmarks for *libkpac*-instrumented browser. Higher scores are better.

of their time interpreting the JavaScript code. If one of the interpreter's hot functions is nonleaf and thus authenticates its return address, this results in a high performance impact when compared to the CortexSuite and Memcached figures. However, even with *svc-libkpac*, the browser remains usable and responsive, which suggests restricting this protection technique for the security-critical applications.

VII. Discussion

General Applicability: Given that the vast majority of even recent ARMv8.3 designs do not yet include the PA extensions, its efficient software-based emulation in the kernel will probably be useful for several years but (hopefully) eventually become obsolete. However, the four techniques and their tradeoffs presented in this article for such emulation are not restricted to PA. They could most probably be applied also to the future security/safety-related ISA extensions. The RCSC mechanism is furthermore usable for the easy offloading of any kind of performance- or security-critical service to a dedicated core. By its CPU-local page-tables, it provides seamless integration into multithreaded applications without extra synchronization efforts.

Hardware Costs: Offloading kernel tasks to the dedicated cores has been shown to be effective in improving the performance in many settings [29], but to the best of our knowledge not yet as an alternative to a relatively simple FPGA-based solution. We consider this as a question of pragmatics. Technically, (i.e., with respect to the HW overhead), the FPGA-based solution is undoubtedly a lot cheaper. However, actual availability and market prices often tell a different story. While SoCs, including an FPGAs are still a development niche, multicore CPUs are prevalent on the market and benefit from competitive pricing and mass production for procurement, the SoC including an extra core is often cheaper than the one with the FPGA. Besides, these multicore CPUs are rarely utilized to their full potential due to the limited parallelism within the software. This warrants considering dedicating one or multiple cores to a service like KPAC for increased security or performance or employing them instead of an FPGA accelerator. In the end, the question of spending an extra core or not comes down to the actual performance cost tradeoff, as developers can always opt for one of the synchronous emulation variants.

VIII. Related Work

Software-Based Pointer Protection: Before ARMv8.3-A PA, the idea of adding a cryptographic MAC to the code pointers has been explored in a technique called *CCFI* [34]. To keep the key secret, CCFI reserves 11 XMM registers on x86-64, which constitutes a change to the ABI, requiring the recompilation of the program and all its dependencies. Its predecessor, *PointGuard* [35] introduces a compiler extension that instruments programs to encrypt the pointers when storing them into memory using simple XOR with a key stored in the same address space. Another approach, called *CPI* [36], protects pointers by storing them in a secret location along with metadata. However, Evans [37] demonstrated an attack that is able to bypass CPI and argued that the security mechanisms relying on the information hiding are ineffective.

Compared to these approaches, KPAC maintains higher security guarantees by computing the cryptographic signature in the kernel space, which allows it to reliably hide the secret key from the attackers.

Applications of PACs: The ARM PA mechanism is not limited to the return address protection. In the recent years, many CFI mechanisms employing PA codes in the user space have been proposed. Liljestrand et al. [24] have presented several works on this subject. *PARTS* [24] is an instrumentation framework, which extends the set of protected pointers to the local, global, and static pointers as well as the pointers in C structures. *PCan* [38] revisits the concept of stack canaries by dynamically generating their value for each function call using PA instructions, eliminating the need to hide their value in memory. *PACStack* [39] upgrades the return address protection by cryptographically binding its value to all the previous return addresses in the call stack, preventing the pointer reuse attacks. *PTAuth* [40], *PACMem* [41], and *CryptSan* [42] are the sanitizers that detect the spatial and temporal memory bugs by leveraging PACs. Schilling, Nasahl, and colleagues utilize PACs to thwart not only the software but also the fault attacks by 1) ensuring CFI at the basic block granularity [43] and 2) protecting indirect branches by encoding them at compile time and verifying them at run time [44]. The work of Fanti et al. [45] generalizes PA by protecting not only the pointers but all the spilled registers.

Given the scarcity of systems with hardware PA, many of these works have resorted to emulating PA instructions using a rudimentary XOR "encryption" as a proof-of-concept. With KPAC, all these PA-based techniques could be seamlessly integrated with our cryptographically secure approaches, extending the CFI guarantees beyond the backward edge protection for the systems without hardware-assisted PA.

Dedicated Service Cores: Several other works have explored the possibility of dedicating CPU cores of the system for some specific service in order to avoid the context switching overhead. For example, Lozi et al. [29] replaced lock acquisitions with remote calls to a dedicated core executing a critical section and observe the performance increase, attributing it to the data locality. The technique of offloading network packet processing to a separate CPU core has been repeatedly proposed since the early days of consumer grade multicore

CPUs [46], [47]. *IsoStack* [48] and *Shenago* [49] implement that kind of network stack and demonstrates significant performance improvements. A similar technique has also been successfully applied to speed up virtualization [50], [51].

The novelty of our approach lies in the idea of modifying the virtual memory layer to present each application thread with the page private to the CPU core it is running on. This forms a framework for the secure communication with the dedicated service core without requiring any synchronization.

IX. Conclusion

ARMv8.3-A pointer authentication is a promising CFI mechanism, which is expected to gain more traction in the following years. It provides significant security gains for minimal performance impact, owing to its hardware implementation. However, CPUs implementing this feature are still rare and we face many systems without PA support.

In this work, we explore how PA can be emulated in software, while maintaining the low performance overhead. For this, we extended the Linux kernel with a PA service and exposed two communication interfaces for the user applications: 1) the classical synchronous system call and 2) a shared memory page for the asynchronous communication. We also investigated two instrumentation methods for the existing applications: 1) statically, by recompiling them with our compiler plugin and (2) in the ARMv8.3-ABI-compatible way, by patching them at load time. In the static case, we employ several heuristics inspired by GCC's stack protector feature [13] to limit protection to the vulnerable functions, offering a flexible balance between the performance and security.

We combined all the aspects into a total of eight approaches and evaluated their run-time impact using the CortexSuite benchmarks and the Memcached key-value database. We also assessed the ease of use in the end-user environments by applying our approaches to the *Chromium browser* without recompilation. For the best of our approaches, we observed low overheads: a worst-case run duration increase of 20% for the CortexSuite benchmarks when using *kpacd-strong* and a modest 29% increase in tail latency for the Memcached with *svc-libkpac*.

The source code and evaluation artifacts are available at: https://github.com/luhsra/kpac

Acknowledgment

The authors thank the anonymous reviewers for their feedback and fruitful comments.

References

[1] *Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1–4*, Intel, Santa Clara, CA, USA, Apr. 2022.

[2] T. Garrison (Intel, Santa Clara, CA, USA). *Intel CET Answers Call to Protect Against Common Malware Threats*. 2020, [Online]. Available: https://www.intel.com/content/www/us/en/newsroom/opinion/intel-cet-answers-call-protect-common-malware-threats.html.

[3] V. Shanbhogue, D. Gupta, and R. Sahita, "Security analysis of processor instruction set architecture for enforcing control-flow integrity," in *Proc. 8th Int. Workshop Hardw. Archit. Support Security Privacy*, 2019, pp. 1–11, doi: 10.1145/3337167.3337175.

[4] *Arm Architecture Reference Manual for A-Profile Architecture*, Arm Limited, Cambridge, U.K., 2022.

[5] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Security*, vol. 15, no. 1, pp. 1–34, Mar. 2012, doi: 10.1145/2133375.2133377.

[6] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-Libc without function calls," in *Proc. 14th ACM Conf. Comput. Commun. Security*, New York, NY, USA, 2007, pp. 552–561, doi: 10.1145/1315245.1315313.

[7] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *Proc. 6th ACM Symp. Inf., Comput. Commun. Security*, New York, NY, USA, 2011, pp. 30–40, doi: 10.1145/1966913.1966919.

[8] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *Proc. 10th ACM Symp. Inf., Comput. Commun. Security*, Singapore, 2015, pp. 555–566, doi: 10.1145/2714576.2714635.

[9] C. Zou, X. Wang, Y. Gao, and J. Xue, "Buddy stacks: Protecting return addresses with efficient thread-local storage and runtime re-randomization," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 2, pp. 1–37, Apr. 2022, doi: 10.1145/3494516.

[10] C. Zou, Y. Gao, and J. Xue, "Practical software-based shadow stacks on x86-64," *ACM Trans. Archit. Code Optim.*, vol. 19, no. 4, pp. 1–26, Dec. 2022, doi: 10.1145/3556977.

[11] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity: Principles, implementations, and applications," in *Proc. 12th ACM Conf. Comput. Commun. Security*, New York, NY, USA, 2005, pp. 340–353, doi: 10.1145/1102120.1102165.

[12] N. Burow, X. Zhang, and M. Payer, "SoK: Shining light on shadow stacks," in *Proc. IEEE Symp. Security Privacy*, San Francisco, CA, USA, 2019, pp. 985–999, doi: 10.1109/SP.2019.00076.

[13] (GNU project, Boston, MA, USA). *Using the GNU Compiler Collection (GCC), Version 12.1*. 2022. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-12.1.0/gcc/

[14] "The LLVM project clang 14.0.0 documentation." 2022. [Online]. Available: https://releases.llvm.org/14.0.0/tools/clang/docs/

[15] J. Corbet. "The rest of the 6.6 merge window." Sep. 2023. [Online]. Available: https://lwn.net/Articles/943245/

[16] (Microsoft, Redmond, WA, USA). *MWC 2022: The Next Microsoft Pluton Device + PAC Technology*. 2022. [Online]. Available: https://blogs.windows.com/windowsexperience/2022/02/28/mwc-2022-the-next-microsoft-pluton-device-pac-technology/

[17] G. Serra, P. Fara, G. Cicero, F. Restuccia, and A. Biondi, "PAC-PL: Enabling control-flow integrity with pointer authentication in FPGA SoC platforms," in *Proc. 28th IEEE Real-Time Embed. Technol. Appl. Symp*, 2022, pp. 241–253, doi: 10.1109/RTAS54340.2022.00027.

[18] R. Avanzi et al., "The tweakable block cipher family QARMAv2," Cryptol. ePrint Arch., IACR, Bellevue, WA, USA, Rep. 2023/929, 2023. [Online]. Available: https://eprint.iacr.org/2023/929

[19] "Linux 5.0 changelog." 2018. [Online]. Available: https://cdn.kernel.org/pub/linux/kernel/v5.x/ChangeLog-5.0

[20] "ARMv8.3 pointer authentication in xnu." 2021. [Online]. Available: https://opensource.apple.com/source/xnu/xnu-7195.50.7.100.1/doc/pac.md

[21] "Linux 5.7 changelog." 2020. [Online]. Available: https://cdn.kernel.org/pub/linux/kernel/v5.x/ChangeLog-5.7

[22] "Apple pointer authentication guidelines." 2023. [Online]. Available: https://developer.apple.com/documentation/security/preparing_your_app_to_work_with_pointer_authentication

[23] S. Thomas et al., "CortexSuite: A synthetic brain benchmark suite," in *Proc. IEEE Intl. Symp. Workload Characterization*, 2014, pp. 76–79, doi: 10.1109/IISWC.2014.6983043.

[24] 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 *Proc. 28th USENIX Conf. Security Symp.*, 2019, pp. 177–194.

[25] Y. Wang, J. Wu, T. Yue, Z. Ning, and F. Zhang, "RetTag: Hardware-assisted return address integrity on RISC-V," in *Proc. 15th Eur. Work. Syst. Security*, New York, NY, USA, 2022, pp. 50–56, doi: 10.1145/3517208.3523758.

[26] J.-P. Aumasson and D. J. Bernstein, "SipHash: A fast short-input PRF," in *Proc. Int. Conf. Cryptol.*, 2012, pp. 489–508.

[27] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter, "The performance of μ-kernel-based systems," in *Proc. 16th ACM Symp. Oper. Syst. Princ.*, New York, NY, USA, 1997, pp. 66–77, doi: 10.1145/269005.266660.

[28] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): The case for a scalable operating system for multicores," *ACM SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 76–85, Apr. 2009.doi: 10.1145/1531793.1531805.

[29] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller, "Remote core locking: Migrating critical-section execution to improve the performance of Multithreaded applications," in *Proc. USENIX Annu. Tech. Conf*, 2012, p. 6.
[30] "Memtier benchmark on Github." RedisLabs. 2024. [Online]. Available: https://github.com/RedisLabs/memtier_benchmark
[31] (StatCounter, Dublin, Ireland). *Browser Market Share Worldwide*. 2023. [Online]. Available: https://gs.statcounter.com/browser-market-share
[32] "The chromium project." 2023. [Online]. Available: https://www.chromium.org/Home/chromium-security/memory-safety/
[33] "WebKit's browserbenchmarks." 2023. [Online]. Available: https://browserbench.org
[34] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "CCFI: Cryptographically enforced control flow integrity," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Security*, New York, NY, USA, 2015, pp. 941–951, doi: 10.1145/2810103.2813676.
[35] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointGuard^{TM}: Protecting pointers from buffer overflow vulnerabilities," in *Proc. 12th USENIX Security Symp.*, 2003, pp. 90–104.
[36] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *Proc. 11th USENIX Symp. Oper. Syst. Design Implement.*, Broomfield, CO, USA, 2014, pp. 147–163.
[37] I. Evans et al., "Missing the point(er): On the effectiveness of code pointer integrity," in *Proc. IEEE Symp. Security Privacy*, 2015, pp. 781–796, doi: 10.1109/SP.2015.53.
[38] H. Liljestrand, Z. Gauhar, T. Nyman, J.-E. Ekberg, and N. Asokan, "Protecting the stack with PACed canaries," in *Proc. 4th Work. System Softw. Trusted Execution*, New York, NY, USA, 2019, pp. 1–6, doi: 10.1145/3342559.3365336.
[39] H. Liljestrand, T. Nyman, L. J. Gunn, J.-E. Ekberg, and N. Asokan, "PACStack: An authenticated call stack," in *Proc. 30th USENIX Security Symp*, 2021, pp. 357–374.
[40] R. M. Farkhani, M. Ahmadi, and L. Lu, "PTAuth: Temporal memory safety via robust points-to authentication," in *Proc. 30th USENIX Security Symp*, 2021, pp. 1037–1054.
[41] Y. Li et al., "PACMem: Enforcing spatial and temporal memory safety via ARM pointer authentication," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2022, pp. 1901–1915, doi: 10.1145/3548606.3560598.
[42] K. Hohentanner, P. Zieris, and J. Horsch, "CryptSan: Leveraging ARM pointer authentication for memory safety in C/C++," in *Proc. 38th ACM/SIGAPP Symp. Appl. Comput.*, 2023, pp. 1530–1539, doi: 10.1145/3555776.3577635.
[43] R. Schilling, P. Nasahl, and S. Mangard, "FIPAC: Thwarting fault-and software-induced control-flow attacks with ARM pointer authentication," in *Proc. 13th Int. Work. Constr. Side Channel Anal. Secure Design*, 2022, pp. 100–124, doi: 10.1007/978-3-030-99766-3_5.
[44] P. Nasahl, R. Schilling, and S. Mangard, "Protecting indirect branches against fault attacks using ARM pointer authentication," in *Proc. IEEE Int. Symp. Hardw. Oriented Security Trust*, 2021, pp. 68–79, doi: 10.1109/HOST49136.2021.9702268.
[45] A. Fanti, C. C. Perez, R. Denis-Courmont, G. Roascio, and J. Ekberg, "Toward register spilling security using LLVM and ARM pointer authentication," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 41, no. 11, pp. 3757–3766, Nov. 2022, doi: 10.1109/TCAD.2022.3197511.
[46] M. Rangarajan, A. Bohra, K. Banerjee, E. V. Carrera, R. Bianchini, and L. Iftode, "TCP servers: Offloading TCP processing in Internet servers. Design, implementation, and performance," Dept. Arts Sci. Comput. Sci., Rutgers Univ., New Brunswick, NJ, USA, Rep. DCS-TR-481, 2002.
[47] T. Brecht, G. J. Janakiraman, B. Lynn, V. A. Saletore, and Y. Turner, "Evaluating network processing efficiency with processor partitioning and asynchronous I/O," in *Proc. EuroSys Conf.*, 2006, pp. 265–278, doi: 10.1145/1217935.1217961.
[48] L. Shalev, J. Satran, E. Borovik, and M. Ben-Yehuda, "IsoStack–highly efficient network processing on dedicated cores," in *Proc. USENIX Annu. Tech. Conf.*, 2010, pp. 1–14, doi: 10.5555/1855840.1855845.
[49] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads," in *Proc. 16th USENIX Symp. Netw. Syst. Design Implement.*, 2019, pp. 361–378.
[50] J. Liu and B. Abali, "Virtualization Polling Engine (VPE): Using dedicated CPU cores to accelerate I/O virtualization," in *Proc. 23rd Int. Conf. Supercomput.*, 2009, pp. 225–234, doi: 10.1145/1542275.1542309.
[51] A. Landau, M. Ben-Yehuda, and A. Gordon, "SplitX: Split guest/Hypervisor execution on multi-core," in *Proc. 3rd Work I/O Virtualization*, 2011, pp. 1–7.