In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication

Sungbae Yoo[†‡] Jinbum Park[†‡] Seolheui Kim[†] Yeji Kim[†] Taesoo Kim[†*]

[†] *Samsung Research*,
[*] *Georgia Institute of Technology*

Abstract

This paper presents an in-kernel, hardware-based control-flow integrity (CFI) protection, called PAL, that utilizes ARM's Pointer Authentication (PA). It provides three important benefits over commercial, state-of-the-art PA-based CFIs like iOS's: 1) enhancing CFI precision via automated refinement techniques, 2) addressing hindsight problems of PA for in-kernel uses such as preemptive hijacking and brute-forcing attacks, and 3) assuring the algorithmic or implementation correctness via post validation.

PAL achieves these goals in an OS-agnostic manner, so could be applied to commodity OSes like Linux and FreeBSD. The precision of the CFI protection can be adjusted for better performance or improved for better security with minimal engineering efforts. Our evaluation shows that PAL incurs negligible performance overhead: e.g., <1% overhead for Apache benchmark and 3–5% overhead for Linux perf benchmark on the latest Mac mini (M1). Our post-validation approach helps us ensure the security invariant required for the safe uses of PA inside the kernel, which also reveals new attack vectors on the iOS kernel. PAL as well as the CFI-protected kernels will be open sourced.

1 Introduction

Memory safety issues are the foremost security problems in today's operating systems—in 2020 alone, there were 149 memory safety CVEs assigned to potentially exploitable bugs in Linux [21]. To prevent latent bugs from exploitation, commodity OS vendors have been developing and deploying modern mitigation techniques such as KASLR, DEP, and SMA/EP (PXN). However, more powerful exploitation techniques, such as return- and jump-oriented programming [14, 60], have been developed and demonstrated that such migration schemes can be ultimately bypassed [35]. As a response, control-flow integrity (CFI) [7, 16], which enforces a program's control transition (e.g., an indirect call or a return) to strictly follow the known control graph curated at compilation time, has been considered as a promising, necessary direction to mitigate these emerging exploitation techniques. Accordingly, modern operating systems like Android, Windows, and iOS all implement some forms of CFI [8, 55, 67, 68].

During the last several years, there has been exhaustive research exploration of CFI's design space [16], which falls broadly into two categories: ① enhancing the precision of CFI (i.e., reducing the number of targets that an indirect call can take); and ② making CFI protection faster and practical (i.e., incurring minimum CPU and memory overheads). The community has improved CFI precision by providing better algorithmic advances to model control-flow transitions accurately [30, 45], or by utilizing exact run-time contexts [27, 31]. However, in practice, the performance overhead often determines the feasibility of actual deployment—it would be acceptable to prevent the most common cases with negligible overhead rather than fully preventing all of them with obtrusive overhead. One recent approach taken by Apple [8] and researchers [47, 72] is to speed up CFI by utilizing hardware-based protection, called ARM Pointer Authentication (PA).

In this paper, we propose yet another in-kernel CFI protection based on PA, called PAL, which aims to enhance CFI precision (see Table 2) in an automated manner while imposing negligible performance overhead (see §6.3).

More specifically, we design a context analyzer (see §4.3) that can capture common idioms and design patterns in commodity OSes (see §4.2) to enhance the CFI precision. For example, the context analyzer can refine the indirect call targets based on the invariants of a kernel object or based on a calling context of a kernel API. By using the context analyzer, an OS developer can instrument the kernel code with minimal manual involvement.

In addition, PAL introduces a static validator (see §4.5) that can recognize common pitfalls and attack vectors of PA on the *final* kernel image, unlike other PA-based schemes placing their compiler and underlying algorithms as TCB. Such separation of concerns helps us develop PA-based CFI and refinement rules in a higher-level, early-stage IR (i.e., GIMPLE in the GCC) without dealing with the subtleties of the back-end compiler optimizations. In other words, as long as the static validator assures that a certain image has no pitfall, we know that the final kernel image meets all the invariants necessary for the secure enforcement of PAL's CFI at run-time. For example, we commonly see that a later-stage

[‡] The authors contributed equally.

optimization puts out even a PA-protected pointer located in for-loop and stashes it into memory, which consequentially disarms the PA-based mitigation. (see §3.1 and Google's PoC on iOS [11]).

PAL's PA-based approach exhibits better CFI precision than any other commercial solutions: e.g., compared with Android's CFI [68], the number of indirect calls having >100 targets reduces from 7.0% to 0.08%. Most importantly, to the best of our knowledge, PAL is the first PA-based scheme that has successfully evaluated performance on real PA-supported hardware (Mac mini based on the M1 chip). Our evaluation shows that PAL imposes negligible performance overhead on user applications: <1% in the Apache benchmark, while incurring 6.8% kernel-space overhead on average in the LMbench's benchmarks that had measurable overhead. (i.e., benchmarks with over 1 μs overhead)

This paper makes the following contributions:

- **New attack vectors.** We provide a systematic categorization of attack vectors under a precise yet powerful definition of threat models when using ARM PA to protect the OS kernel.
- **Automated refinement techniques.** We implement a context analyzer to capture idioms and design patterns that enhance the precision of the CFI protection in automated yet OS-agnostic ways. We demonstrate our context analyzer on two commodity OSes: Linux and FreeBSD kernels.
- **Static validator.** We implement a static validator to automatically verify non-trivial security problems unintentionally introduced by complex compiler optimizations as well as implementation mistakes. It found new attack vectors in the latest iOS kernel as well as bugs in other PA-based schemes.
- **Open source.** We will make the end-to-end ecosystem open source including compiler plugin, static validator, the CFI-protected Linux and FreeBSD kernels, and context analyzer.[1]

2 Background

2.1 Control-Flow Integrity (CFI)

As modern defense schemes like Data Execution Protection (DEP) prevent *code injection* attacks, offensive techniques like return- or jump-oriented programming (ROP/JOP) [14, 60] have been proposed to reuse existing code snippets for exploitation, commonly known as *code-reuse attacks*. With ROP and JOP techniques, an attacker can perform arbitrary computation [60] by chaining existing code gadgets *after* hijacking a program's control flow (e.g., overwriting a function pointer or a return address). Since the code gadgets end with a return or a jump instruction that allows further chaining, they are called *return-* or *jump*-oriented programming.

[1] https://github.com/SamsungLabs/PALinux

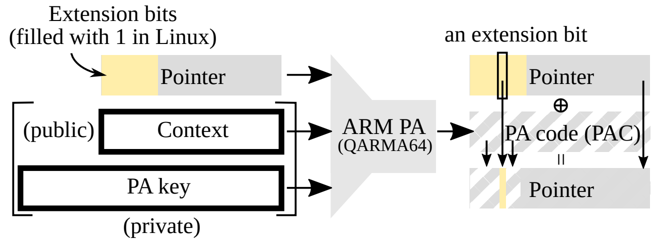


Figure 1: A PA code (PAC) is generated using QARMA64 [66] block cipher with three inputs: a pointer, a context (public), and a PA key (private). And the PA-signed pointer contains the PAC in the prefix of the pointer that indicates its authenticity [12]. The signed function pointer will be copied or moved around together with the PAC, and checked its authenticity right before uses.

One promising mitigation to prevent such code-reuse attacks is to ensure that the integrity of control-flow transfers remains intact (i.e., making an intended transition) at run-time, so-called control-flow integrity (CFI). The most common approach is to extract control-flow graphs (CFG) from the source code during compilation and validate that all transitions are legitimate at run-time (i.e., following an edge of the extracted CFG). As a standard CFG is overly conservative—an indirect jump instruction can lead to any location in a program's memory, CFI often utilizes high-level structures from programming languages. To limit indirect calls, there are different strategies for identifying what is the set of functions that can be involved at each call site: 1) each function's entry [55], 2) a group of functions with the same types in `C` [65, 67], 3) virtual functions in the class hierarchy in `C++` [15, 67]. As most transitions depend on the program's execution state, dynamic approaches that attempt to associate the execution state and transition are proposed, e.g., using inputs [57], shadow execution/traces [27], and collecting control-sensitive information [31]. The precision of these dynamic approaches, however, comes with non-trivial run-time overheads and hindering deployment in practice.

Common CFI schemes mostly concern *forward* transitions (i.e., an indirect call) while relying on *backward* transitions (i.e., a function return) to be protected by other orthogonal techniques such as a shadow stack [22, 37]. In PAL, we aim to design an end-to-end solution that protects *both* forward and backward edges.

2.2 Pointer Authentication

ARMv8.3-A introduced a new hardware-based security feature, called *Pointer Authentication* (PA), that can check the integrity of a pointer with minimal performance (i.e., using a faster block cipher) and storage overhead (i.e., using the extended bits of a pointer). Simply put, a pointer is signed (`PAC`) when generated and is authenticated (`AUT`) before its use, similar in concept to tagged memory [62].

❶ **`PAC`** *signs* a function pointer with a signing key and a context as a nonce (see Figure 1). Since the ARM 64-bit processor does not fully utilize the entire 64-bit address space (using 39-48 bits in Aarch64), the signed pointer can carry the authentication code (PAC) as part of the pointer (25-16 bits in the extension bits). This design decision simplifies the heavy bookkeeping required to propagate the PAC for pointers—now, a normal instruction like `mov` seamlessly propagates the PAC of a pointer in a register without additional costs.

PA provides five registers for encryption keys that can only be set in privilege mode: APIA and APIB for code pointers, APDA and APDB for data pointers, and APGA for general-purpose use, each of which is used to compute a cryptographic hash (i.e., QARMA64 [66]) to generate a Message Authentication Code (MAC).

❷ **`AUT`** *authenticates* the signed pointer and restores it to its original form (i.e., discarding the PAC and restoring the extension bits) given the original key and context. If the authentication succeeds, the restored function or data pointer is used as intended by the following instructions. However, when the authentication fails, `AUT` simply *flips an error bit* in the pointer to indicate the corrupted state, and any *later use* of the pointer raises an exception, as the restored function pointer does not conform to the canonical form of the virtual address space.

Using context. PA's context is a critical element to narrow down the protection domain because all function pointers signed by the same context can be used alternatively in an indirect call (note, all signed by the same PA key as long as PA key is not changed). There are two well-known choices of contexts: a stack pointer (`sp`) as a context to sign a return address (i.e., a backward CFI), and zero as a context to sign all function addresses, which does not need to identify all call-sites and their functions to be involved. More advanced uses, like using a type signature as a context [47], also not requiring to identify relations between all call-sites and functions, have also recently been proposed, but our focus in this paper is to refine the context by using *dynamic* information that can be captured from design idioms in commodity OS kernels.

EnhancedPAC2 and FPAC. ARM recently announced two new features, namely, EnhancedPAC2 and FPAC [53] to address problems found by Google [11]. EnhancedPAC2 changes PAC bits to be larger by XOR-ing with the upper bits of the pointer, helping PA to avoid brute-forcing attacks. FPAC makes an `aut` instruction raise an exception immediately upon the authentication failure. With FPAC, PAL can optimize the performance even further, but our current focus is to be compatible with ARM v8.3-A, available on the consumer market right now. Such optimization techniques based on these hardware features can also be used to improve the performance of PAL.

3 Overview

3.1 Threat Model

We assume an attacker has the capabilities of arbitrary reads and writes *at arbitrary moments*, similar to other attack models assumed in several studies like PaX/RAP [65]. We also assume that the victim has all modern defense mechanisms deployed, namely, secure boot, stack protection, DEP, KASLR, PXN, and page-table protections [9, 10] to protect the kernel's non-writable regions from page-table modification attacks such as KSMA [74]. It is worth noting that the capabilities of arbitrary reads and writes do not mean that it is possible to inject code so that existing code snippets should be reused for attack. Lastly, we assume KASLR can be bypassed either by inferring the layout of the kernel image [34, 38] or via common information leakage [71] otherwise an arbitrary write or read is prevented in the first place.

In the kernel, arbitrary read/write capabilities implicate more than just crafting memory; since an attacker can force all registers to be spilled to memory via preemption, an attacker can modify all values of the *registers* stored in memory (i.e., execution context). This model would be pessimistic, as real-world vulnerabilities typically allow limited capabilities like restricted overflow, but we believe this is the right way to reason about the strong security guarantees of the defense mechanisms in the kernel [65].

Control-flow hijacking. We assume the primary goal of an attack is to hijack the control-flow of the kernel and then, leverage it to launch post-exploitation payloads such as obtaining a root shell, exfiltrating information, installing a rootkit, etc. With arbitrary memory reads/writes, this is not the only approach (e.g., data-oriented attack [33, 63]), but still is the most prevalent, reliable and stealthy form of powerful attacks [16].

Out-of-scope attacks. We do not consider side-channel attacks such as micro-architectural [42, 50], timing [49, 73], or electromagnetic side-channels [61], because they are mostly limited to secrecy violation (i.e., information leak). Similarly, hardware attacks such as Rowhammer [40] caused by faulty DRAMs are not of our concern. All high-privilege components like the hypervisor, firmware and hardware are our TCB (Trusted Computing Base). We also do not consider any advanced forms of data-oriented attacks [33] but have a plan to extend our approach to mitigate prevalent forms of data-oriented attacks, similar to HDFI [63], by using PA.

Correctness assumption. We do not rely on the correctness of complex compilation tool-chains that have sophisticated optimization algorithms in their back-ends. Instead, we trust a static post validator that directly analyzes the *final* kernel image and ensures that the invariants required for the secure uses of PA are correctly respected in the binary (see §4.5).

3.2 Attack Vectors against ARM PA

We categorize the fundamental limitations of ARM PA into two classes, namely, pointer substitution attacks (❶, ❷, ❸) and improper uses of the PA protection (①, ②).

❶ **Replaying attack**. A leaked, signed function pointer can be legitimately reused in any indirect call with the same context. This means an adversary having arbitrary read capability can scan the entire kernel memory and collect outstanding function pointers for the replaying attack. This fundamental problem can be quantified by measuring the number of function pointers signed by the same context and the number of indirect call sites authenticating pointers with the same context (§6.1). To reduce the substitution targets, any PA-based protections should minimize the uses of the same contexts.

❷ **Forging pointers via signing gadgets**. Instead of passively scrubbing function pointers, an adversary can generate a signed pointer with the context of his/her own choice for the targeted indirect call. There are two prevalent situations: 1) an attacker hijacks a stored function pointer or the context during the signing process and 2) an attacker provides an arbitrary function pointer to a re-signing routine that ignores authentication failures.

❸ **Brute-forcing attack**. PA reserves a small number of bits (e.g., 15 bits in the 48-bit address space) to embed a MAC as a part of the pointer. Unfortunately, this is so small that an attacker can identify the correct MAC by enumerating all the input space if there exists a proper oracle. This problem is particularly difficult to mitigate because the production kernel cannot simply panic when an authentication failure happens, whether that is a malicious attempt or not [69]. All the existing PA-based protections suffer from this attack vector.

① **Key leakage and cross-EL attack**. ARM [51] and Qualcomm [66] specify PA's behaviors only in the context of the user space. To utilize PA for kernel protection, according to these references, the PA keys should be multiplexed (or virtualized) for both user and kernel spaces. However, under our threat model, an adversary can sign any function pointers in user space by using the same key and context as the kernel, a so-called cross-EL attack. Any preventive measures that keep track of PA keys in the kernel space should be cautious about *not storing* the key to memory—our adversary with the arbitrary read capability can obtain the PA key.

② **Time-of-check to time-of-use (TOCTOU)**. PA does not guarantee the atomicity of its check and use, meaning that there is a time window between two PA instructions. Two problems can occur during this time window: 1) a raw pointer is unintentionally spilled before a use mainly due to later-stage compiler optimizations or machine-code generation, and 2) an attacker enforces a preemption right before its use, causing a raw pointer stored in the register to be spilled to an attacker-controlled memory.

One naive solution is to disable an interrupt during this time window but we observed an important drawback: it increases the interrupt-disabled regions and will finally turn out unacceptable due to performance fall-off. (e.g., a virtual call in the VFS layer will disable the interrupt for the rest of executions, which drops the concurrency of each file operation). Any in-kernel PA defenses should prevent this attack vector without introducing complexity and performance overheads.

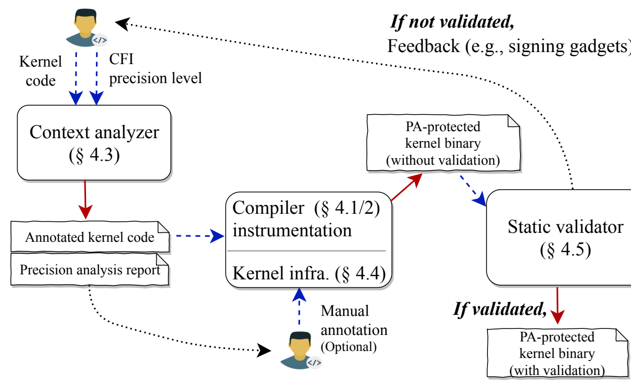


Figure 2: PAL's workflow.

4 Design

Each component of PAL works as illustrated in Figure 2. First, the context analyzer (§4.3) takes kernel code and the desired CFI precision level (i.e., the number of the allowed targets) as inputs, and returns 1) kernel code annotated with the best run-time contexts to meet the given precision, and 2) a summary of the precision analysis that describes the effectiveness of each PA context (see, Table 2) as outputs. Besides that automatic annotation, developers can manually add annotations based upon their needs. Second, the compiler tool-chain (§4.1, §4.2) performs instrumentation according to the given annotated code, and outputs a PAL-enabled but not validated kernel binary. Third, static validator (§4.5) validates that the kernel binary respects a set of security properties for the safe uses of ARM PA, and informs developers if any violation is detected. Developers are in charge of eliminating such a violation by modifying either source code or a compiler tool-chain. Lastly, PAL modifies the kernel infrastructure for secure management of the protection scheme (§4.4).

4.1 Compiler Instrumentation

To protect the integrity of function pointers, PAL intervenes at two life-cycle events of each function pointer: its generation (`GEN`) and use (`USE`). Simply put, PAL signs a function pointer at its generation and authenticates it right before use. Since neither the compiler nor the PA hardware can track life-cycle events(`GEN` and `USE`) of a pointer completely, the security claim of PA-based solutions is largely dependent on the proper selection of a PA context, which is bound to the pointer at `GEN` and used for authentication before `USE` (see §4.2).

In this section, we first describe our design decisions relevant to static analysis and instrumentation:

Life-cycle of function pointers. Function pointers in C are generated (`GEN`) from a function designator [44]—an expression that has a function signature such as a function declaration or casting. More specifically, PAL instruments three places to insert an authentication code: variable initialization, assign statements, and parameters for function calls. For each function designator, PAL handles two kinds of scenarios: 1) *constant* where the value is emitted as part of the instruction thus can be signed *in place*; and 2) *immaterial* where the value will be determined at run-time (e.g., to support ASLR) thus should be signed *at loadtime*. For type casting to a non-function type, PAL leaves them intact (i.e., no authentication and re-sign), as the dereferenced values are already signed at their generation and the integrity will be ultimately checked at use. Note that such behaviors can be abused as a signing gadget by an attacker and our static validator is designed to identify such behaviors (see §4.5).

An indirect call in C uses (`USE`) a signed function pointer so PAL enforces the authentication and restores it to the original form *right before* taking it into the target function. In addition, PAL handles two exceptional situations to support the practical use of function pointers in the kernel: function pointer comparison and type casting to an integer.

Protecting backward edges. `GEN` and `USE` of backward edges are semantically clear: `GEN` at a function call and `USE` when returning back. To handle this case, PA provides two dedicated instructions, namely, `paciasp` and `autiasp`, which protect a link register that stores the address to return after the function call by using the address of the local frame as the context.

Unfortunately, under PAL's strong threat model, the current scheme is vulnerable to a replay attack, meaning that an attacker can craft a signed return address by reusing the stack memory (i.e., repeatedly creating new tasks). To mitigate it, PAL simply combines a stack frame and a hash of the current function name as a context. This ensures that the signed return addresses cannot be reused across either different functions or different stack layouts.

Avoidance of user space pointers. PAL recognizes function pointers from user space and selectively opts out of signing and authentication. In theory, it is difficult to distinguish these two types of pointers from a compiler's perspective, but PAL can recognize them based on existing idioms (e.g., `__user` in Linux) at their type declarations and properly propagate them in our instrumentation.

4.2 Refined Context Generation

The granularity of protection provided by PA depends on the way the context parameter is generated (at `GEN`) and used (at `USE`). For example, in terms of substitution attacks (see §3.2), one leaked signed pointer can be used at any `USE` of the same context parameter—to be precise, the key should be equal as well, but all pointers in the same address space (i.e., kernel space) will be signed by the same key.

There are two *known* techniques for refining the context parameter further, one using its type signature (static) [8, 47] and another using the stack pointer (dynamic) [8, 66]. Although both approaches can be applied without changing the original source code, they are still far from ideal: 1) they are too coarse-grained (e.g., one function type is used 470 times in Linux) and 2) show high false positives (e.g., 6.5k authentication places using *zero* as a context in iOS 13).

PAL provides two static (❶, ❷) and two dynamic (①, ②) refinement techniques for context generation:

```
1  struct irqaction {
2    irq_handler_t handler;
3    const char *name;
4    ...
5  /* This is an auto-generated annotation by PAL */
6  } __attribute__((objbind("name", "handler")));
7
8  int request_percpu_irq(unsigned int irq,
9                         irq_handler_t handler,
10                        const char *devname, ..) {
11   irqaction *action;
12   action->name = devname;
13
14   /* [GEN (typesig)]:
15    *   action->handler = pac(handler, hash(typeof(handler)))
16    * [GEN (objtype)]:
17    *   action->handler = pac(handler, hash(typeof(handler)) \
18    *                      || hash(typeof(*action)))
19    * [GEN (objbind)]:
20    *   action->handler = pac(handler,                        \
21    *                      pac(hash(typeof(handler))          \
22    *                           || hash(typeof(*action)),     \
23    *                         (u64)(action->name)))
24    * [USE]:
25    *   handler = aut(handler, hash(typeof(handler)))        */
26   action->handler = handler;
27   ...
28 }
```

Figure 3: Example of contexts based on `typesig`, `objtype` and `objbind` refinement techniques for a `IRQ` handler. All `GEN` and `USE` will be automatically instrumented given the annotation in line 5, automatically generated by the context analyzer.

❶ **Build-time context: `typesig`.** Our *baseline context* for a function pointer is its type signature—a hash of its type declaration, similar to PaX RAP [65]. With `typesig` as a context in `GEN` and `USE`, it effectively implements a type-based CFI (see Figure 3), or the 1-layer confinement of MLTA [54].

❷ **Build-time context: `objtype`.** A type signature can be further refined with a corresponding owner's type when a function pointer is owned by a kernel object (e.g., `irq_handler` in `irqaction` in Figure 3). For example, one common function signature (`void (*) (void *)`) is used in 170 different indirect calls (`USE`) and introduced from 200 different function designators (`GEN`) in Linux. With this refinement—effectively the 2-layer confinement of MLTA [54]—the signatures can be refined to 35 different contexts during compilation.

PAL can further refine the context generation to the granularity of each function pointer instance *at run-time*—each instance of the kernel objects is assigned with a unique context for protection. The key idea is to take advantage of the idiomatic design patterns used in the OS kernels, which are commonly enforced at code reviews or as part of the maintenance cycle [48]. And the idea has been fully proved by PAL, especially the context analyzer (see, §6.4). In particular, PAL provides this scheme as two annotations to capture common invariants of a relationship between a function pointer to an object (`objbind`) and an invocation context (`retbind`). The context analyzer can automatically generate these annotations so that developers do not need to worry about how they supplement these annotations properly.

① **Run-time context: `objbind`.**

Annotation: `objbind({&}?field, {*|fptr}+)`

This specifies which *field* or its address (&) should be bound to which function pointers (one or more, or all with `*`) in an object's declaration.

As an embedded function pointer is often invariant over the lifetime of its owner object, an `objbind` annotation indicates the compiler has to bind the authenticity of the function pointer to various properties of `struct`. Once a `struct` is annotated at its declaration, all objects of the `struct` will be instrumented to have a dynamic context, thus uniquely *binding* the function pointer to the created object (`GEN` of the member function). For example, `irq_handler()` can be bound to the device's `name` (i.e., a pointer to a static string) as in Figure 3, limiting the target places for the leaked `irq_handler()` to the one it was originally in. Accordingly, it can be viewed as the 3-layer confinement of MLTA [54].

PAL implements a generic technique to composite multiple contexts together by *chaining* the result of a previous `pac` as a context argument of the subsequent `pac` (see Figure 3). This technique, unlike simple `xor` of multiple contexts, provides better security, especially when an attacker chooses an arbitrary value as one of the context (e.g., a device name).

At a glance, one would imagine binding all the embedded function pointers in its object's base pointer, but this results in too many false negatives for automation—for example, when an object is `memcpy()`-ed, all signed function pointers should be properly resigned for the new context, namely, the new object pointer as well as its types. This not only is fragile but incurs high performance overhead for `memcpy()`, which is commonplace in Linux (see Table 4).

② **Run-time context: `retbind`.**

Annotation: `retbind({params}+)`

This specifies its calling context is bound to which function arguments (*params*) at the function's declaration.

A `retbind` annotation indicates the compiler has to bind a function pointer to its calling contexts[2], which is effective in protecting a function pointer not embedded in its owner object. One such design pattern is reference counting in Linux—`kref` where its `release()` function is not stored as part of the object but should be provided together with the `kref_put()` function for reclamation (see Figure 4). Note that this pattern saves a lot of memory uses, as `kref` instances outnumber their invocation sites.

As `kref` is frequently used in Linux (e.g., over 110 release functions and 127 call sites), an attacker would substitute any counterfeited function pointer (e.g., via signing oracle (§3.2) or leaked pointer) to any one of such candidates. With `retbind`, a function pointer becomes unique per calling context—the leaked `release` pointer can be used only at the legitimate calling context.

Note that PAL currently is able to handle `GEN` that occurs

[2] This means the context where a function is being called from, not PA's

```
1  struct kref { refcount_t refcount; };
2
3    /* [GEN]
4     *  release = pac(release, hash(typeof(release))
5     *                          ^ each call site's address) */
6    kref_put(&rbdc->kref, rbd_client_release);
7    kref_put(&device->kref, drbd_destroy_device);
8
9  int
10 /* This is an auto-generated annotation by PAL */
11 __attribute(retbind(release))
12 kref_put(struct kref *kref, void (*release)(struct kref *kref)) {
13   ...
14   /* [USE]
15    *  release = aut(release, hash(typeof(release))
16    *                          ^ __builtin_return_address(0)) */
17   release(kref);
18 }
```

Figure 4: Example of `retbind` in protecting `kref` with PAL. All `GEN` and `USE` will be automatically instrumented given the annotation in line 10, automatically generated by the context analyzer.

only in a depth-0 calling context (e.g., `kref_put()`). But, it can happen in more depth, in other words, there might exist several layers of wrapper calls to invoke a specified function. For example, `inode_insert5()` in Linux contains three call sites in its own calling context (depth-0) and 33 call sites in the callee's context (depth-1). We plan to refine such calls as future work.

Overhead comparison of contexts. `typesig` has the smallest overhead, just one `mov` and one `pac`/`aut` operation. Compared to `typesig`, `objtype` takes one more bitwise operation to concatenate. `objbind` needs one extra memory access to a chosen field. In many cases, the access utilizes memory locality enough because the structure that the field belongs to would be used. Finally `retbind` does not require any additional overhead in signing but, in authentication, one more memory access to the stack frame.

4.3 Context Analyzer

PAL provides a context analyzer that spots adequate places for adopting run-time contexts with the given kernel code and the desired precision level. Through an inter-procedural analysis on the IR level, the context analyzer automatically annotates places having lower precision even with build-time contexts.

For `objbind`, the context analyzer is based on the sound assumption that the structure with the larger allowed targets likely has the higher *diversity score* that represents each fields' compile-time diversity (see Table 6 for their correlation). It first estimates each structure's score by simply counting the number of assignments of a new constant or a stack address or the address of heap objects. The rationale behind choosing these as criteria is twofold– 1) a new constant intuitively means more diversity in a field value, and 2) stack and heap address would have sufficient randomness to be used as runtime contexts if they are newly allocated (i.e., current stack frame address or address that comes from heap allocator).

As a running example, Figure 5 shows how to estimate the `s1.p`'s diversity score. First, the context analyzer collects

```
1  struct s1 { char *p; void (*fp)(void); }
2  void init_s1(struct s1 *s, char *p, void (*fp)(void)) {
3      s->p = p;
4      s->fp = fp;
5  }
6  void f1(char *p) {
7      struct s1 a, b, c, d;
8      init_s1(&a, "a", h1); // score += 1 (new constant)
9      init_s1(&b, "b", h2); // score += 1 (new constant)
10     init_s1(&c, "b", h2); // ; not new constant
11     init_s1(&d, p, h3);   // ; cannot be determined
12 }
13 void f2(struct s1 *s, struct o1 *o) {
14     s->p = o->p;    // score += o1's score
15     s->fp = o->fp;
16 }
17 void f3(struct s1 *s) {
18     char name[64] = "c";
19     init_s1(s, name, h3); // score += 1 (stack address)
20 }
```

Figure 5: A running example to estimate diversity score for objbind

all assignments related to `s1` structure (line 3, 14) and starts data-flow analysis at each of them. Specifically, at line 3, a data-flow analysis starts on `p` across function boundaries as `p` cannot be resolved within `init_s1()`, so it continues using a worklist algorithm until the value of `p` is statically determined. As a result, the diversity score will increment at line 8, 9, 19, but not at line 10, 11. In case a score depends on other structure's (line 14–`o1` object), it first estimates `o1.p`'s score and accumulates to the score. Lastly, the context analyzer starts to annotate `objbind` to structures with the highest diversity score first until they meet the given precision level. (see Appendix A for the more detailed algorithm)

For `retbind`, the context analyzer identifies functions that take a code pointer as input and consumes it in place, then estimates the number of call sites (e.g., for `kref` (Figure 4), the function is `kref_put()`, and its number of call sites is 127.) Lastly, the context analyzer starts to annotate `retbind` to functions with all of the related call sites.

Besides automatic annotation, the context analyzer provides a CFI precision report of the kernel (see, Table 2). By analyzing the report, developers can identify unresolved low-precision contexts and refine them via manual annotations.

4.4 Kernel Infrastructure

Under our threat model where an attacker can launch arbitrary memory reads and writes at arbitrary moments, the kernel should take various design decisions into special consideration: 1) making sure that plain function pointers are neither stored in memory nor unintentionally spilled from register files via preemption, 2) effectively mitigating the brute-force attacks, 3) managing the keys' life-cycle without ever storing them in memory.

Preventing preemption hijacking. To prevent TOCTOU (§3.2), caused by spilled function pointer in preemption, PAL should sign/authenticate the preemption context while ensuring that it never acts as a *signing oracle* (§3.2) nor remains vulnerable against a *replay attack* (§3.2). We achieve that by introducing two new techniques as follows:

1) Secure signing via key-chaining technique. A simple approach to sign the entire preemption context is signing each register *individually*. Unfortunately, it is vulnerable against a replay attack because an attacker can *selectively* substitute registers in the preemption context for control-flow hijackings. To overcome this problem, PAL uses the key-chaining technique that signs each register with the previously signed code as a context in the chain (see, Figure 6).

2) Timebind: using a timestamp as a PA context. The above scheme prevents attackers from modifying individual fields of the execution context, but one can replace the *whole* preemption context, similar in concept to a replaying attack. There are three potential defenses: 1) Similar to `typesig`, we can use a certain signature presenting the preempt context as a PA context. However, it leaves a large number of substitution targets [12] since all preeempt context are signed using the same PA context; 2) Similar to `objbind`, we can use the base address of the preemption context as a PA context. However, this approach still leaves a *universal* signing oracle because an attacker would control the preemption moment to generate a signed register value that can be used to substitute the same register field on the target preemption context allocated at the same memory region (see Appendix B for more detail); 3) Another solution to avoid the signing oracle is to simply dedicate another PA key (i.e., `pacga`) for signing and authenticating the preemption context, leaving no other keys for userspace or the hypervisor.

To avoid the signing oracle without an additional PA-key, PAL introduces a notion of `timebind` that uses an unmodifiable one-time value, *timestamp* and the base address of the preemption context, to generate a unique context parameter for signing (Figure 6). To get a timestamp that is resilient against the forgery, PAL uses the Physical Timer Count on Aarch64 that monotonically increases once the system boots but cannot be changed by system software. This scheme prevents the replay attack because the context will be different at every time and it requires two additional fields (`pac` and `time_pac`) in the preemption context.

Note that if an attack gets to know the two additional fields, the security of this scheme will not be negatively affected because the attacker still does not know the PA key.

Mitigating brute-force attacks. As the number of bits allocated for PAC is physically limited (15 bits), an attacker can launch a brute-force attack against the PA-protected indirect calls. Since it is not feasible for the kernel to simply halt the entire system upon authentication failure, an attacker would just enumerate 2^{15} possible PACs, which takes a few minutes (e.g., 15-min in Google's PoC [12]). To mitigate such a scenario, PAL senses the forgery attempts and backs off the execution with a delay increasing exponentially at every trial based on the context; this means an attacker-chosen context with a randomly chosen PAC would delay the testing oracle exponentially. An attacker might change the context to bypass the back-off delay for incorrect authentication, but it does not

```
struct preemption_context {
  unsigned long regs[MAX_REGS]; /* both general-purpose and
                                 * special-purpose regs */
  unsigned long time_pac; /* pac(base-addr, time) */
  unsigned long pac;      /* pac for the entire object */
};

void aut_pctx(struct preemption_context *pctx) {
  unsigned long ptr;
  unsigned long pac = pctx->time_pac;

  for (int i = MAX_REGS - 1; i >= 0; i--) {
    ptr = pctx->regs[i];
    pac = pacib(ptr, pac);
  }
  if (pac != pctx->pac) { /*  authentication failure */ }
}

void pac_pctx(struct preemption_context *pctx) {
  unsigned long time = read_sysreg(cntpct_el0);
  unsigned long ptr = (unsigned long)pctx;

  unsigned long time_pac = pacib(ptr, time); /* timebind */
  unsigned long pac = time_pac;

  for (int i = MAX_REGS - 1; i> -1; i--) {
    ptr = pctx->regs[i];
    pac = pacib(ptr, pac); /* chaining */
  }
  pctx->time_pac = time_pac;
  pctx->pac = pac;
}
```

Figure 6: Simplified pseudo code to explain the use of key-chaining and timebind techniques to prevent preemption hijacking.

increase the likelihood of selecting the correct PAC for the new target address.

The back-off history and strategy need special care to securely mitigate the outstanding attacks—for example, an attacker can try to manipulate the hashmap that records the number of authentication failures to render the back-off ineffective. To address this situation, PAL manages the number of authentication failures per context on a read-only memory region and temporarily makes it writable to update for a short window of time while halting the entire machine. This strategy not only prevents a concurrent attack from forging the faulting history (as halted), but also does not exhibit the overhead to the normal execution (such an event would not happen except for under attack).

Key management. The security of PA relies on the secrecy of its keys. Given a leaked key, an attacker can counterfeit a function pointer via cross-EL attacks because its signing algorithm is publicly known. In our threat model, an attacker in user space can forge a code pointer to jump to an arbitrary location, say an ROP chain, in the kernel. Finally, the attacker can steal data that they want via arbitrary read.

Therefore, once the PA keys for kernel are generated at boot time, PAL guarantees that they are never stored into memory during execution to protect the key from an attacker capable of reading an arbitrary memory region. For key generation, PAL leverages the randomness provided by the bootloader via a device tree and utilizes the HW-based random generator if available.

To avoid storing the PA keys for the kernel, another important design decision is that user spaces do not share the same PA-keys as kernel space, meaning the kernel and user space have a dedicated set of PA-keys—B (APIB and APDB) for the kernel and A (APIA, APDA, and APGA) for the user spaces. Moreover, at context switching, PAL inverts all bits of the key in place (see Figure 7)—only the kernel can access the key and user programs can sign pointers with the *inverse key* but cannot infer the original key. In conclusion, the PA keys for kernel do not need to be stored for context switching.

```
.macro kernel_entry, el
  mrs_s x0, SYS_APIBKEYLO_EL1
  mvn x0, x0
  msr_s SYS_APIBKEYLO_EL1, x0
  mov x0, 0
```

```
.macro kernel_exit, el
  mrs_s x20, SYS_APIBKEYLO_EL1
  mvn x20, x20
  msr_s SYS_APIBKEYLO_EL1, x20
  mov x20, 0
```

Figure 7: PAL inverse all bits of the PA key in place to prevent cross-EL attack (§3.2) at context switching where preemption is disabled.

PAL dedicates each key to the kernel and user space, which restricts the number of available PA-based protection domains to only one at a time. To avoid this problem, CPU designers would consider either using independent sets of keys per execution domain or adding per-domain nonces in the key assignment of each execution level.

Loadable kernel module support. PAL also implemented the support of module loading and unloading while providing the PA-based protection. Similar to what PAL generally treats global variables (§4.1), it signs the function pointers in the module object (e.g., `init()` and `exit()`) as well as variables used in the module before the loading process starts. The modules are required to use PAL in their builds to prevent them from exploiting.

4.5 Static Validator

The correctness of existing PA-based solutions is largely dependent on the correctness of the compiler's back-end logic like optimizations and machine-code generation. Due to the complexity of whole compiler code, it is an error-prone task for a compiler writer to guarantee that PA-related concerns written at the higher layer are preserved, even after many stages, at the lowest layers like produced binary. For example, Google Project Zero recently discovered a security hole in iOS [11] where an address of a jump table for a switch statement was hoisted out of a for-loop and stored in memory due to the large number of registers used in the for-loop. In our threat model, an attacker can hijack the control-flow by crafting the stored address of the jump table at the moment.

PAL's security, however, relies on the correctness of the static validator, which independently certifies that the *produced binary* respects a set of security-critical invariants and assumptions taken during the compilation. This design separation greatly simplifies the implementation of PAL, using a higher IR layer (i.e., GIMPLE in the GCC) without being concerned about the potential interference from the back-end optimizations or machine code generation. In addition, our

```
1  # violation-1: bgmac_chip_reset(...) from PARTS
2  mov x19, x0    # x0: first arg of this function
3  ldr x21, [x19, x8]
4  blr x21        # unprotected
5
6  # violation-2: sort(void (*swap_func)()) from PAL
7  autib x2, x0            # x2: swap_func
8  stp x1, x2, [x29, 144] # spilled
9  ....
10 ldr x2, [x29, 144]      # reused
11 blr x2
12
13 # violation-3: from iPhone's CFI
14 mov x19, x2          # 3rd arg of function
15 ldr x21, [x19, 240] # load an attacker-chosen pointer
16 pacia x21, x8        # sign
17
18 # violation-3: from iPhone's CFI
19 mov x19, x0       # 1st arg of function
20 pacia x19, x20    # sign
21
22 # violation-3: usb_stor_CB_transport() from PARTS
23 adrp x22, 0xffff0000108cf000  # x22: callee-saved register
24 bl 0xffff0000108cf320   # call usb_stor_msg_common()
25 pacia x22, x23          # sign
26 ....
27 #0xffff0000108cf320:    usb_stor_msg_common()
28 stp x22, x21, [sp, 48]  # spilled
29 ....
30 ldp x22, x21, [sp, 48]  # load an attacker-chosen pointer
31 ret
```

Figure 8: ❶, ❷, ❸ violations from PARTS [47] and iOS and PAL. An attacker can break PA with an attacker-chosen pointer by controlling either function arguments or spilled stack memory.

static validator can be used to evaluate other PA-based solutions, such as Apple's and PARTS [47].

Our static validator checks if four principles that PAL assumes during the compilation are still preserved after the back-end optimization:

❶ **Complete protection.** All indirect branches are authenticated and the result is checked prior to use (line 1 in Figure 8).

❷ **No time-of-check-time-of-use.** Raw pointers after authentication (`aut`) or clearance (`xpac`) are never stored back in memory (line 6 in Figure 8).

❸ **No signing oracle.** There should be no gadget that signs attacker-chosen pointers (line 13, 18 in Figure 8).

❹ **No unchecked control-flow change.** All direct modifications of program counter register must be validated. The validator correctly guides us to handle special cases such as scheduling, signal handling, and preemption (see §4.4).

Algorithm. It performs a simple *intra-procedural* analysis with CFG recovery and loop detection given a binary image. It first scans all instructions within a function and runs VALIDATE_BB (Figure 9) on PA instructions.

With Figure 8 as a running example, we first describe the cases in which the analyzer can detect violations within a basic block. To validate ❶ and ❸, it invokes VALIDATE_BB on `blr` and `pacia`, respectively (line 4, 16 in Figure 8) with `x21` as a symbolic register *sym* in Figure 9. Then, VALIDATE_BB attempts to find the origin of `x21` in a backward recursive way by exploring all possible paths. Conservatively, due to `ldr` (line 3, 15 in Figure 8) in a previous path, it reports a violation (line 4 in Figure 9). If *sym* is originated from the function parameters (line 19 in Figure 8), VALIDATE_BB would conclude as a potential violation as it cannot be resolved even after exploring the whole function. To validate ❷, it starts from `autib` (line 7 in Figure 8) with `x2` as *sym* and attempts to find the uses of `x2` in a forward recursive way, and then reports a violation because of `stp` (line 8 in Figure 8). As a special case, to detect the violation at line 22 in Figure 8, it checks if a call instruction places between address calculation (line 23) and PA instruction (line 25). This trick enables detecting such a violation without inter-procedural analysis but entails false positives because a register containing PA-relevant values might not be stored in the memory.

The validator, of course, can work across basic blocks in the following cases– 1) if *sym* cannot be resolved within a basic block (i.e., reaching line 18 in Figure 9), the validator recursively invokes VALIDATE_BB on all predecessors of the current *bb* (line 21 in Figure 9), and 2) If we encounter any of branch instructions jumping to somewhere in the current function before *sym* is resolved, it invokes VALIDATE_BB on the target basic block. (line 17 in Figure 9)

Results. We applied static validator to PARTS, iOS kernel, and PAL itself, as a result, confirmed 15/5/0 violations respectively. We found 7 violations during PAL development, 1/1/5 for ❶/❷/❸ respectively, and fixed all by modifying either our compiler pass or kernel code.

False positives. Since it is infeasible to implement perfect binary analysis on the kernel, the validator reported about 100 false positives, and their root causes were mainly due to– 1) too complicated control flows in the kernel, meaning that too many basic blocks are involved in a PA instruction, or 2) the absence of inter-procedural analysis that is needed to detect the violation at line 22 in Figure 8, or 3) uncertainty of the type of memory to be loaded (e.g., if `x19` at line 15 in Figure 8 points to read-only memory, that is not the violation).

The task to eliminate those took two days by a person who is knowledgeable with binary analysis. We plan to improve our static validator to reduce such false alarms as future work.

Potential to extend coverage. At present, our validator aims to only detect misuses of function pointers. Adversaries however can turn their focus to compromising contexts, another threat to break PA-based CFIs. Since the techniques used in the validator would be applicable to detect that threat, we plan to extend the validator as future work.

5 Implementation

We implemented PAL's instrumentation by adding a new GIMPLE pass (after the CFG pass) on the GCC 7.4.0 as a plugin. Also, we put very small modifications to the GCC, in order to interpose on function prologues and epilogues for the backward-edge CFI. We chose the GCC as a first target, to actually deploy the CFI protection to our commercial products,

```
VALIDATE_BB (bb, si, sym)
  Input   : bb: a basic block to be inspected
            si: a first instruction to be inspected in bb
            sym: a symbolic register containing PA-relevant value
  Output  : true if no violations, otherwise false
  Symbol  : A: arithmetic/bitwise instructions  L: load instructions
            C: address calculation instructions  P: predecessors of bb
            i_*op: source/destination register operand of i
  for i ← si; i ≠ bb.first(); i ← i.prev(); do
      if i ∈ A & i_destop = sym then
          sym ← i_srcop
      else if i ∈ L & i_destop = sym then
          return false;
      else if i ∈ C & i_destop = sym then
          return true;
      else if i = "auti*" & i_destop = sym then
          return true;
      else if i = "xpaci*" & i_destop = sym then
          return false;
      // call instruction
      else if i = "bl" then
          return false;
      // jump or conditional branch instruction
      else if i = "b" then
          target ← i.target
          return VALIDATE_BB(target, target.end(), sym);
  if P = ∅ then
      return false;
  foreach bb ∈ P do
      if not VALIDATE_BB(bb, bb.end(), sym) then
          return false;
  return true;
```

Figure 9: The core algorithm to verify ❶, ❸ and ❹. The algorithm for ❷ is explained in §4.5.

Components	Lines of code
the GCC plugin	3,632 LoC (for forward-edge CFI)
the GCC	127 LoC changes (for backward-edge CFI)
Linux	491 LoC changes
`objbind`	87 structs
`retbind`	79 functions
FreeBSD	258 LoC changes
`objbind`	25 structs
`retbind`	3 functions
Static validator	848 LoC (python)
Context analyzer	1943 LoC (c++)

Table 1: The complexity of PAL's components (in LoC).

such as appliances, IoT devices, and smartphones, that rely only on the GCC. We also implemented instrumentation in LLVM, but it supports only a subset of PAL features. (e.g., `typesig`).

Kernel modifications. We made minimal changes to Linux (491 LoC) and FreeBSD (258 LoC). We manually fixed incorrect declarations of function types (e.g., dummy console and filler) similar to Android's patches to support CFI [39, 67]. The context analyzer automatically adds 166 annotations to Linux and 28 annotations to FreeBSD, for contexts having more than 100 targets even when objtype is used (see Table 2).

Preemption hijacking protection. We prevent preemption hijacking in two places in Linux: 1) `el1_irq()` called when an IRQ occurs at the kernel mode, and 2) `el0_irq()` called when an IRQ occurs at the user mode. In 1), we sign and authenticate not only all general-purpose registers (i.e., `x0`–`x30`) but also some special-purpose registers (e.g., `elr_el1`, `spsr_el1`) as in Figure 6. In 2), we simply perform sanity checks on special-purpose registers to prevent the hijacking to kernel space instead of returning back to user space. This protection cannot be exploited as a signing oracle (§3.2) because arm64 guarantees that all registers are preserved when an interrupt is raised and Linux does not allow nested interrupts on both IRQ handlers mentioned above.

Backward-edge protection. In PAL, creating a context for backward edge protection requires an operation with a constant and stack pointer register(`sp`) that is not allowed direct uses as an operand in Aarch64.

For this reason, function prologues (left-side) and epilogues (right-side) use two additional registers—a register as an operand of combine instruction (`bfi`) and a register as a context for PA—as follows.

```
mov x9, sp
mov x10, hash(FUNC_NAME)
bfi x9, x10, 32, 32
pacib lr, x9
```

```
mov x9, sp
mov x10, hash(FUNC_NAME)
bfi x9, x10, 32, 32
autib lr, x9
ret
```

Note that those registers should be caller-saved registers (e.g., x9, x10) to protect register spilling causing performance overhead.

Supporting Linux. We found developer guides that motivate to devise `objbind`. Linux kernel has provided design patterns for inside components (e.g., device driver [48]), strongly recommending developers to use special functions and structures. As a result, most code consists of some patterns, which helps the context analyzer refine contexts easier to reduce allowed targets in Linux.

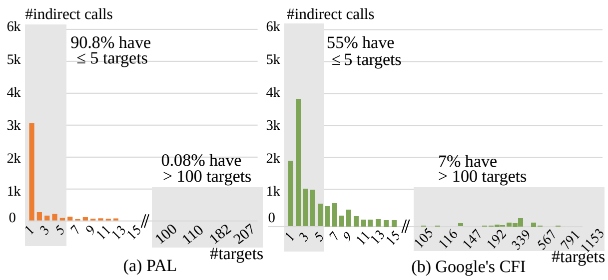
Supporting FreeBSD. We found two interesting function types—`kobjop_t` and `sy_call_t`—used for better software abstraction. In other words, function pointers are stored as different types from the actual type of pointed function. We also found 125 and 342 function types, stored after type-converted as `kobjop_t` and `sy_call_t` respectively, which means that FreeBSD allows many allowed targets. In PAL, the context analyzer automatically applied `objbind` to refine these function types.

Context Analyzer. The context analyzer, written in C++, first takes kernel codes and builds the kernel. Afterward, it extracts an LLVM bitcode file for the fully linked kernel binary (e.g., vmlinux for Linux) to enable inter-procedural analysis for a whole, and starts the static analysis.

6 Evaluation

In this section, we evaluate PAL's approach in four key areas:

Q1. How does our approach compare with known PA-based CFI solutions? (§6.1)



#Tgts	Google's	BTI	Linux w/ PAL typesig	+objtype	+objbind +retbind
≤ 5	55.0%	0.0%	84.9%	88.6%	90.8%
> 100	7.0%	100.0%	2.8%	1.6%	0.08%
Max	1,153	59,300	35,264	30,622	207

Table 2: The precision of PAL in terms of the number of *allowed* indirect call targets. We show ≤ 5 and > 100 in comparison with reported Google's CFI applied to Android 4.14 [68]. We also added ARMv8.5 BTI scheme for comparison. It shows PAL can effectively enhance the precision of the in-kernel CFI.

Q2. How do we validate its security guarantee and the correct functionality of PAL-enabled kernels? (§6.2)
Q3. How much performance overhead does PAL impose on user applications and the kernel? (§6.3)
Q4. How do we check the soundness and effectiveness of our context analyzer? (§6.4)

Experimental setup. We selected two target devices, Mac mini (M1) and Raspberry Pi 3 (Pi3), to represent a high-end and a low-end ARM device respectively. We applied PAL to Linux (Asahi Linux [1] customized for the M1 chip-based on Linux 5.12.0-rc1 [2], and Linux 4.19.49 for Tizen 5.5) and FreeBSD (FreeBSD 11.0-CURRENT), and evaluated them on two real devices and one virtual platform: Mac mini for Asahi Linux and the Pi3 for Tizen 5.5 and QEMU for FreeBSD. We reported the real performance on the M1 (using actual PA instructions). But for Pi3, because of a lack of PA support, we estimated the PA's performance by measuring real cycles taken to execute each PA instruction on both Apple A12 and the M1 on userspace (Table 5). To provide a realistic kernel configuration, we adopted the union of Asahi's and `arm64`'s default config for Linux 5.12.0-rc1. Also, we used the default configuration for Tizen 5.5 and FreeBSD.

6.1 Comparing with Other Approaches

We first compare the precision of PAL's protection with two other state-of-the-art CFI schemes that have been deployed on Android. Then, we compare ours with two other PA-based protection schemes, namely, PARTS [47], and iOS's CFI (since iPhone XS). Last, we compare PA-based solutions in terms of PA context changes.

Allowed targets for indirect calls. The precision of forward-edge CFI can be estimated by counting the number of allowed targets for each indirect call. In PA, an indirect call can be taken to any location if the function pointer is signed by the same context used in the call site. Since our static validator checked that there is no signing gadget embedded in the final binary, the precision can be measured by simply counting the number of pointers signed by the same context. Table 2 shows the total number of allowed targets by each indirect call—if there are two call sites (`AUT`) and five different calls (`PAC`) using the same context, we conservatively estimated its allowed set to 10. We ran the context analyzer with 100 allowed targets as the precision level and used the automatically annotated kernel code that the analyzer produced.

Compared to the state-of-the-art CFI protection applied to Android [68], PAL improves the precision of CFI significantly: the number of indirect calls with fewer than five targets increases by 5.9% (to 90.8%) and the ones with more than 100 targets decreases from 2.8% to 0.08%. Most important, only 3 contexts are included in > 100 after applying both objbind and retbind.

We reference estimations from Google's public report on the precision of the deployed CFI on recent Android [68]. The differences in Google's type-based CFI and our `typesig` are due to the version differences of each kernel– 4.14 in Google's and 5.12.0-rc1 in PAL– as well as PAL's large kernel configuration. As a comparison, we also added the estimation of ARM's hardware-based CFI, BTI (Branch Target Identification) introduced in ARMv8.5 [52], which only limits all indirect transitions to the function entry.

Table 2 also shows the effectiveness of refined context generation used in PAL. Our static context (i.e., `objtype`) reduces the overall number of target sets while the run-time contexts (i.e., `objbind` and `retbind`) effectively refine the most common call targets (from 30622 to 207).

Context diversity on indirect calls. To compare with other PA-based solutions, we measured the CFI precision by counting the number of indirect calls sharing the same context (shown Table 3). For a fair comparison, we applied PARTS to Linux 5.12.0-rc1 and performed binary analysis on the latest iOS firmware image. Compared to PARTS, PAL improves the proportion of small set (≤ 5 contexts) from 20.7% to 94.9% and reduces the proportion of large set (> 100) from 18.4% to near zero. Compared to iOS[3], PAL also effectively refines the attack targets: it reduces the proportion of the large set from 21.2% to near none while eliminating using the zero context (i.e., 6513 indirect calls using the zero context in iOS). Note that iOS's proportion of the large set (21.2%) is due to not only the zero context but also the context containing offsets for jump tables.

Context changes. PA-based solutions are often required to change the context used to sign a pointer: e.g., type-casting on a function pointer requires authentication with a previous context and re-signing with a new context. For PA solutions relying on *static* contexts, this task is straightforward, but for

[3] We treated *dynamic* contexts as an *unique* context, so counted in ≤ 5.

#Contexts	PARTS	iOS kernel	PAL
≤ 5	20.7%	62.2%	94.9%
> 100	18.4%	21.2%	0.0%
Max	353	6,513	70

Table 3: The diversity of context used in PAL, iOS and PARTS in terms of the number of indirect calls that share equal context.

Context property	PARTS	iOS kernel C	iOS kernel C++	PATTER	PAL
Build-time	typesig	zero	namesig		typesig, objtype
compatibility	×	●	●		●
security	×	×	●		●
Run-time			address	address	field
compatibility			●	●	●
security			●	×	●

Table 4: Comparing PA-based forward-edge protections in terms of context changes. *address* means the storage address of a function pointer to be signed/authenticated; *namesig* means a hash of mangled function name used in iOS for C++ V-Table entries [43]; PAL's approach is not only compatible with memcpy() but also provides finer protection by utilizing both static and dynamic contexts.

ones using *dynamic* contexts, this conversion is often implicit and non-trivial to handle (see Table 4). For example, when an object is copied with `memcpy()` or `memmove()`, the member functions are no longer considered properly signed with the context (e.g., the base address of the object).

For this reason, iOS uses *zero* context for all C function pointers as well as C++ V-Table pointers (not entries inside the table) [43], which can be vulnerable to replay attack as demonstrated by Google project zero. [13] Meanwhile, PATTER [72] interposes these memory-related functions, checks each byte of the source to identify a signed pointer, and re-signs with a new address, which can be leveraged for a signing gadget (§3.2) because it ignores authentication failures. In contrast, PAL's approach capturing the kernel's design patterns (see §4.2) performs in a robust manner.

Backward-edge protection. Unlike Apple's primitive backward-edge protection [43], PAL enhances its precision by combining the hash of a function name and a stack pointer, similar to PARTS [47]. To quantify the improvement, we estimated the maximum number of allowed targets for backward edges while running LMbench on Linux. Our evaluation shows that it effectively reduces the allowed targets from 203 (Apple's) to 14.

Other finer-grained solutions like PACStack [46] or Camouflage [26] impose undesirable performance overheads: PACStack requires excessive memory accesses to trace every call stack [46] and Camouflage needs to reserve a register to retain the function address until the function epilogue [26].

6.2 Security and Functional Validation

Correctness testing. We tested the correctness of the PAL-protected Linux by applying micro- and macro-benchmarks: LMbench, perf bench, Apache bench, leveldb, Blogbench and Linux Test Project (LTP). We also confirmed that the original

Instruction	iPhone (A12)	Mac mini (M1)
`paciza` / `pacizb`	2.8172 / 2.8162 ns	– / 2.1501 ns
`pacia` / `pacib`	2.8103 / 2.8170 ns	– / 2.1503 ns
`autiza` / `autizb`	2.8165 / 2.8170 ns	– / 2.1502 ns
`autia` / `xpaci`	2.8177 / 2.8189 ns	– / 2.1504 ns
eor/orr	0.4032 / 0.4030 ns	0.3077 / 0.3074 ns

Table 5: Performance of PA instructions measured on iPhone (A12) and Mac mini (M1).

kernel exhibits the same behaviors in all benchmarks.

CVE studies. We tested three known CVEs (CVE-2017-7308 and CVE-2018-9568 for Linux, CVE-2019-5602 for FreeBSD) and corresponding exploits against both original and protected kernels. We confirmed that all exploits are successfully prevented—original exploits were simply prevented by `typesig`. However, a stronger adversary could easily launch a replaying attack (§3.2), e.g., CVE-2019-5602 that abuses `struct sysent.sy_call` having 547 allowed targets, which `objbind` in PAL could effectively prevent.

Run-time validation. To check if there are any *overlooked* function pointers not sanitized by our analysis, we took a series of memory snapshots of the running kernel while executing LMbench tests. With this run-time validation, we found several, non-trivial bugs during the development of PAL: e.g., a raw function pointer made in `kernel_thread`, saved in the `x19` field of `task` structure, eventually loaded and called in *assembly code* without `aut` instruction. We manually added PA instructions to protect the function pointers.

6.3 Performance Overhead

We measured the performance overhead imposed by PAL in terms of computation throughput and latency (see Appendix C for detailed numbers).

Micro-benchmark. We used *two* micro-benchmarks: LMbench and perf on Mac mini and the Pi3 (emulated).

① **LMbench**. We ran LMbench v3 to measure the potential impact of system call latency increased by PAL. Compared to stock Linux, PAL increases the latency by 0-3 μs (median. 7%) depending on system calls, on both Mac mini and Pi 3. Due to the additional `pac`/`aut` instructions used for signing and validating the process context, the latency of `fork()` is impacted the most: 28.0 μs (3.7%) on Pi3, 6.1 μs (6.1%) on Mac mini. Also, we measured the only impact of backward-edge protection, PAL increases the latency by 0-1 μs on average.

② **perf**. To measure the performance overhead associated with context switching (i.e., the chained signing operations with the time stamp), we ran the perf benchmark (5.12.0-rc1) [25]. Our experiments show that it degrades the latency of messaging and pipe by 3-5% on both Mac mini and Pi3.

Macro-benchmark. We ran Apache benchmark (v2.3) and leveldb (v1.22) and Blogbench (v1.1) to estimate the performance impact of network and database and file server applications on PAL-enabled Linux, respectively.

① **Apache benchmark**. We used two Pi3 boards—the client

OS	TAT	TDS at 10	TDS at 20	TDS at 30	TDS at 40
Linux	89	35/39.3%	50/56.1%	67/75.2%	80/89.8%
FreeBSD	48	18/37.5%	30/62.5%	38/79.1%	41/85.4%

Table 6: The correlation result between allowed targets and diversity scores; TAT: the number of structs that rank top 10% in allowed targets, TDS at *k*: the number of structs that rank top *k*% in diversity scores; A higher TDS indicates better results.

sends GET requests with varying sizes through ethernet to the PAL-enabled and stock Linux (server) for comparison. PAL degrades the performance by about 1% for 1 KB files but has negligible overhead ($< 0.08\%$) for requests over 100 KB.

On Mac mini, we set up both client and server in one mac mini because, at the time of this writing, Asahi Linux lacks ethernet support. PAL degrades the performance by 0.75% for 1 kb files. Note that for larger requests we could not correctly measure it because client-side overheads affect its result a lot.

② **leveldb**. We ran the default benchmark included in leveldb [4] (NoSQL-style DB), on Mac mini. PAL degrades the performance by 1-3% and 0.3% for write and read, respectively.

③ **Blogbench**. We ran Blogbench [3] on ext4 filesystem with the default configuration in order to reproduce the load of a real-world busy file server. PAL reported negligible overhead (0.2%) for both write and read, on Mac mini.

6.4 Context Analyzer

We evaluated four characteristics of the context analyzer, described as follows.

Soundness. The context analyzer is based on the sound assumption that the structures with the larger allowed targets likely have the higher diversity scores. To validate this assumption in practice, we measured the correlation between allowed targets and diversity scores: the majority of TAT ranks top 20% in diversity score as shown in Table 6.

Security. To see the security enhancement, we measured how many contexts out of what rank top 10% in allowed targets could be successfully refined via `objbind`/`retbind`. As a result, 272 out of 312 (87.1%) and 350 out of 376 (93.0%) could be refined for Linux and FreeBSD, respectively.

Failure cases. We found several cases in which the context analyzer could not refine. Some function pointers do not have any relation with any structure so they are not able to use `objbind`. For example, two common cases are: 1) local function pointers (e.g., `fptr_t fp = func`) and 2) type casting (e.g., `fptr_t fp = obj->fp`).

Engineering efforts. Despite the automation capability of the analyzer, PAL still requires some, yet minimal, engineering efforts to enhance the CFI precision, especially when its diversity score is low (e.g., zero or one) and the number of allowed targets is high. Since the analyzer chooses by default the address binding for such cases, it would cause a kernel panic if not properly handled (e.g., in a memory copy function). In our experiences (Table 2), it takes about one-person day to resolve these issues.

7 Discussion

Assurance over assembly code. All assembly code is checked by the static validator—all inserted PA instructions respect their security invariant (see §4.5). However, it is still possible in theory that PAL misses the protection of function pointers generated and used in inline assembly. Fortunately, if such an assembly code ever just runs in PAL, 1) the system crashes immediately (`aut` failure) in most cases, so our exhaustive benchmarks help us address this issue, and 2) if not crashed, our run-time validator helps us identify the problem by scanning the entire memory for raw pointers. We observed only one case in `kernel_thread` explained in §6.2.

Implication of reserving one PA key. PAL reserved one PA key for the kernel protection and another key for the user space. However, this does not mean that all user space applications share the same key—each application has its own dedicated key that is multiplexed by the kernel.

Applicability to other kernels. We intensively tested PAL with other available kernels including Minix [5] and Zircon [6], The context analyzer reported many sites where objbind is effective, but a small number of sites for retbind. However, since their codebases are too small compared with Linux and FreeBSD, it was trivial to apply PAL to their projects.

Potential to improve performance. Developers can opt to prioritize performance over security. The context analyzer given a lower CFI precision level applies a smaller number of `objbind`/`retbind` and resultantly builds a faster kernel. Note that even with no use of runtime contexts, PAL's security is still better than that of conventional type-based CFIs (e.g., RAP [65]) thanks to the use of `objtype`.

Impacts of speculative executions. We exclude side-channel attacks from our threat model, but they would be practical if a chip enables speculative executions on PA instructions. For example, if attackers could control and monitor what happens in PA instructions in the context of speculative executions, they would be able to make a signing gadget without being affected by the brute-force attack mitigation, in a similar way to BlindSide [29]. So it would be interesting to evaluate PA-enabled SoCs in the view of speculation.

8 Related Work

Ever since a CFI-based approach was introduced to mitigate code-reuse attacks [7], a number of research ideas have been proposed to improve its protection precision and runtime performance [16]. Since precision and performance are fundamental trade-offs in CFI, the finest target estimation comes with non-negligible performance overhead, rendering them unattractive for practical adoption. In contrast, the coarse-grained CFI solutions, like Microsoft's Control-flow Guards [55], Google's Indirect Function-Call Checks [67], PaX's Reuse Attack Protector (RAP) [65], and Apple's PA [8], have been successfully deployed to protect web browsers and operating systems.

Hardware-based CFI. Silicon-level features can significantly alleviate the performance overhead of CFI. For example, commodity technologies have been used to design lightweight CFI schemes: Intel PT [36] to trace control-flow changes [27, 32], Intel LBR [41] to get the history of branch changes [18, 59], and Intel MPX [58] to quickly enforce target boundaries [56]. Since these hardware features are not intended for security, retrofitting them for CFI leaves a lot of weaknesses in security like PT packet losses [27, 32] or overflowing branch history [59].

Recently, more hardware primitives [17, 19, 23, 24, 37, 53, 64] are designed specifically to assist CFI—we use the term, "primitives," as they are dependent on the software counterpart that utilizes the primitive for the full protection. ARM's PA [66] is one of the most promising primitives that Apple first utilized to enforce CFI in iOS and M1-based macOS [8]. In academia, PARTS [47] and PATTER [72] also proposed type-based signing by using PA, but hardly beyond the intended design of PA [47]. Apple's CFI implemented much advanced type analysis to address unique challenges to its own kernel—mixed uses of objective-C and native components [8, 43]. Unfortunately, Apple's approach is not universally applicable to other monolithic commodity OSes like Linux and FreeBSD in providing finer-grained target enforcement for CFI (§6.1).

In-kernel CFI. Commercial solutions such as PA-based CFI for iOS [8], LLVM's CFI for Android [68], and PaX RAP for Linux [65] use the type-based approach to refine the precision of CFI without breaking code-level compatibility. Academic approaches have explored various directions to further enhance its precision, by utilizing mapping tables derived from finer-grained CFGs, for system software [20, 28, 70]. Our approach, while providing a commercial solution, aims to achieve the finest precision with minimal performance overheads on commodity hardware supporting PA.

9 Conclusion

This paper presents PAL, an in-kernel, ARM PA-based protection that enhances the precision of CFI with minimal performance overhead. We define new attack vectors for PA when used to protect the kernel and found erroneous cases in the state-of-the-art PA-based protections such as iOS and PARTS. PAL provides two techniques: automated refinement techniques to capture idioms and design patterns for better CFI, and a static validator to check error-prone usage patterns of PA in the final OS images. PAL has been ported to Linux and FreeBSD and our evaluation shows negligible performance overhead. We will make PAL publicly available upon acceptance and for artifact evaluation.

References

[1] *Asahi Linux*. https://asahilinux.org/.

[2] *Asahi Linux Kernel*. https://github.com/AsahiLinux/linux/commits/7d2d16ccf15d8eb84accfaf44a0b324f36e39588.

[3] *Blogbench*. https://github.com/jedisct1/Blogbench.

[4] *LevelDB*. https://github.com/google/leveldb.

[5] *Minix*. https://www.minix3.org/.

[6] *Zircon kernel of fuchsia OS*. https://fuchsia.googlesource.com/fuchsia/+/refs/heads/main/zircon/.

[7] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, 2005.

[8] Inc. Apple. *Pointer Authentication*, 10 2019. https://github.com/apple/llvm-project/blob/a63a81bd9911f87a0b5dcd5bdd7ccdda7124af87/clang/docs/PointerAuthentication.rst.

[9] A. Azab, P. Ning, J. Shah, Quan Chen, Rohan Bhutkar, G. Ganesh, Jia Ma, and Wenbo Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *CCS '14: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[10] A. Azab, Kirk Swidowski, Rohan Bhutkar, Jia Ma, Wenbo Shen, Ruowen Wang, and P. Ning. SKEE: A lightweight Secure Kernel-level Execution Environment for ARM. In *NDSS*, 2016.

[11] Brandon Azad. *Examining Pointer Authentication on the iPhone XS*, 2 2019. https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html.

[12] Brandon Azad. *A Study in PAC*, 2019. https://github.com/bazad/presentations/blob/master/MOSEC-2019-A-study-in-PAC.pdf.

[13] Ian Beer. *An iOS zero-click radio proximity exploit odyssey*, 12 2020. https://googleprojectzero.blogspot.com/2020/12/an-ios-zero-click-radio-proximity.html.

[14] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40, 2011.

[15] Dimitar Bounov, Rami Gökhan Kici, and Sorin Lerner. Protecting C++ Dynamic Dispatch Through VTable Interleaving. In *NDSS*, 2016.

[16] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Comput. Surv.*, 50(1), April 2017.

[17] Tony Chen and David Chisnall. Pointer Tagging for Memory Safety. Technical Report MSR-TR-2019-17, Microsoft, July 2019.

[18] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Huijie Robert Deng. Ropecker: A generic and practical approach for defending against ROP attacks. In *In Symposium on Network and Distributed System Security (NDSS)*, 2014.

[19] Nick Christoulakis, George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. HCFI: Hardware-Enforced Control-Flow Integrity. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, pages 38–49, 2016.

[20] John Criswell, Nathan Dautenhahn, and Vikram Adve. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 292–307, USA, 2014. IEEE Computer Society.

[21] Linux Kernel CVEs. *All Indexed CVEs*, 2020. https://www.linuxkernelcves.com/cves.

[22] Thurston HY Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 555–566, 2015.

[23] L. Davi, M. Hanreich, D. Paul, A. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin. Hafix: Hardware-assisted flow integrity extension. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015.

[24] L. Davi, P. Koeberl, and A. Sadeghi. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2014.

[25] Arnaldo Carvalho De Melo. The new linux perf tools. In *Slides from Linux Kongress*, volume 18, 2010.

[26] R´emi Denis-Courmont, Hans Liljestrand, Carlos Chinea, and Jan-Erik Ekberg. Hardware-assisted cfi for the arm linux kernel, 2019. https://arxiv.org/pdf/1912.04145.pdf.

[27] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. Efficient Protection of Path-Sensitive Control Security. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, August 2017.

[28] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Ray Jaeger. Fine-grained control-flow integrity for kernel software. In *Proceedings - 2016 IEEE European Symposium on Security and Privacy, EURO S and P 2016*, Proceedings - 2016 IEEE European Symposium on Security and Privacy, EURO S and P 2016, pages 179–194. Institute of Electrical and Electronics Engineers Inc., 5 2016.

[29] Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative probing: Hacking blind in the spectre era. *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[30] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *International Symposium on Code Generation and Optimization (CGO 2011)*, pages 289–298. IEEE, 2011.

[31] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing Unique Code Target Property for Control-Flow Integrity. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, October 2018.

[32] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R Harris, Taesoo Kim, and Wenke Lee. Enforcing unique code target property for control-flow integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1470–1486. ACM, 2018.

[33] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[34] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks Against Kernel Space ASLR. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, pages 191–205, San Francisco, CA, May 2013.

[35] Zero Day Initiative. *Pwn2Own 2020*, 05 2020. https://www.thezdi.com/blog/2020/3/17/welcome-to-pwn2own-2020-the-schedule-and-live-results.

[36] Intel. *Processor Tracing*, 09 2013. https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing.

[37] Intel. *Control-flow Enforcement Technology Specification (rev 3.0)*, 05 2019. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.

[38] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, October 2016.

[39] Sami Tolvanen Kees Cook. *console: Expand dummy functions for CFI*, 2 2018. https://patchwork.kernel.org/patch/10243971/.

[40] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors. 42(3):361–372, June 2014.

[41] Andi Kleen. *An introduction to last branch records*, 03 2016. https://lwn.net/Articles/680985/.

[42] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy*. IEEE, 2019.

[43] Ivan Krstić. Behind the Scenes of iOS and Mac Security. In *Blackhat USA*, 2019. https://i.blackhat.com/USA-19/Thursday/us-19-Krstic-Behind-The-Scenes-Of-IOS-And-Mas-Security.pdf.

[44] ISO/IEC JTC 1/SC 22 Programming languages. *6.3.2.1 Lvalues, arrays, and function designators*. ISO, 12 2010. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1548.pdf.

[45] O Lhoták and Laurie Hendren. Context-sensitive points-to analysis: is it worth it? In *Compiler Construction (2006)*, pages 47–64, 2006.

[46] Hans Liljestrand, Thomas Nyman, Lachlan J Gunn, Jan-Erik Ekberg, and N Asokan. PACStack: an Authenticated Call Stack. 2019. https://arxiv.org/pdf/1905.10242.pdf.

[47] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N Asokan. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 177–194, 2019.

[48] Greg Kroah-Hartman Linus Walleij. *Device Driver Design Patterns*, 12 2013. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/driver-api/driver-model/design-patterns.rst.

[49] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clementine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, 2016.

[50] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, 2018.

[51] ARM LTD. *ARMv8 architecture reference manual, for ARMv8-A architecture profile (ARM DDI 0487C.a).*, 12 2017. https://static.docs.arm.com/ddi0487/ca/DDI0487C_a_armv8_arm.pdf.

[52] ARM LTD. *Code reuse attacks: the compiler story*, 5 2019. https://community.arm.com/developer/tools-software/tools/b/tools-software-ides-blog/posts/code-reuse-attacks-the-compiler-story.

[53] ARM LTD. *Developments in the Arm A-Profile Architecture: Armv8.6-A*, 9 2019. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/arm-architecture-developments-armv8-6-a.

[54] Kangjie Lu and Hong Hu. Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, November 2019.

[55] Microsoft. *Control Flow Guard*, 12 2019. https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard.

[56] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W Hamlen, and Michael Franz. Opaque Control-Flow Integrity. In *Symposium on Network and Distributed System Security (NDSS'15)*, 2015.

[57] Ben Niu and Gang Tan. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 914–926, 2015.

[58] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX explained: A cross-layer analysis of the intel MPX system stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(2):28:1–28:30, 6 2018.

[59] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, pages 447–462, 2013.

[60] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):1–34, 2012.

[61] Asanka P. Sayakkara, Nhien-An Le-Khac, and Mark Scanlon. A survey of electromagnetic side-channel attacks and discussion on their case-progressing potential for digital forensics. *CoRR*, abs/1903.07703, 2019.

[62] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. *Memory Tagging and how it improves C/C++ memory safety*, 2018. https://arxiv.org/pdf/1802.09517.pdf.

[63] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. HDFI: Hardware-Assisted Data-Fow Isolation. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[64] Dean Sullivan, Orlando Arias, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, and Yier Jin. Strategy without tactics: policy-agnostic hardware-enhanced control-flow integrity. In *53rd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 06 2016.

[65] PaX Team. *RAP: RIP ROP*, 10 2015. https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf.

[66] Qualcomm Technologies. *Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions*, 1 2017.

[67] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, 2014.

[68] Sami Tolvanen. *Control Flow Integrity in the Android kernel*, 10 2018. https://android-developers.googleblog.com/2018/10/control-flow-integrity-in-android-kernel.html.

[69] Linus Torvalds. *LKML: Re: [GIT PULL] usercopy whitelisting*, 2017. https://lkml.org/lkml/2017/11/17/767.

[70] Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *2010 IEEE Symposium on Security and Privacy*, pages 380–395, May 2010.

[71] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[72] Yutian Yang, Songbo Zhu, Wenbo Shen, Yajin Zhou, Jiadong Sun, and Kui Ren. Arm pointer authentication based forward-edge and backward-edge control flow integrity for kernels, 2019. https://arxiv.org/pdf/1912.10666.pdf.

[73] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, 2014.

[74] Wang Yong. KSMA: Breaking Android kernel isolation and Rooting with ARM MMU features. In *Blackhat Asia*, 2018.

Appendix

A Context Analyzer Algorithm

Figure 11 introduces ESTIMATE_DIVERSITY_SCORE, a core algorithm to estimate diversity score (shortly, DS) for the context analyzer annotating `objbind`.

The algorithm takes inputs – *funcs* (all functions in the kernel), *struct* and *fidx* (the desired target structure and its field index for DS), and breaks down to three phases :

1) It collects all assignments (i.e., store instructions) to the given *struct*.*fidx* (line 6) and performs the Andersen's pointer analysis to the value operand of each assignment (line 7) , which retrieves points-to-set (*pts*) of the value operand (line 8). The analysis is a flow-intensive and path-insensitive intra-procedural analysis.

2) It attempts to resolve all *pts* retrieved from the previous phase by running *check*(*pts*) function that checks if all value in *pts* meets any of the conditions to increment diversity score (see, §4.3) and returns true if that is the case. (line 12) If at least one *pts* fails in *check*(*pts*), it moves off to the third phase where an iterative inter-procedural analysis plays.

3) If the else branch at line 16 is taken, it starts to run an iterative worklist algorithm (line 17 ∼ line 38). This algorithm finds and adds functions that need to be further investigated (i.e., calling contexts) into *wk* in an iterative way, unless there is nothing in *wk* or the number of rounds goes over the threshold we set (five) to avoid being unterminated caused by the large code size of the kernel. To find such functions, it checks if a value in *pts* is used as an argument in a function call to *f* in *wk* (line 24), and if true it attempts to increment diversity score if possible (line 27) otherwise adds the current context into *upd* (line 30) to repeat this algorithm.

After all of the above phases are complete, it finally returns DS on *struct*.*fidx* (line 42).

B Abusing Preemption as Signing Oracle

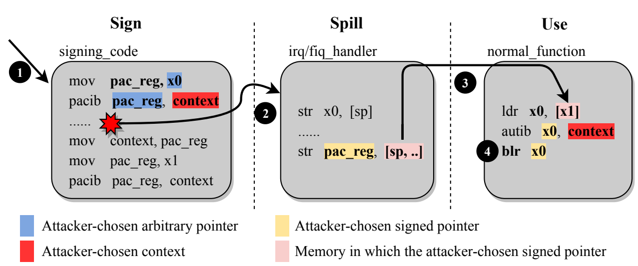


Figure 10: Exploiting the code for signing the preemption context as a signing oracle. ❶ enter the signing routine via IRQ or creating thread state (e.g., `arm_saved_state_t` in iOS) and sign an attacker-chosen pointer in the first register `x0` with an attacker-chosen context. ❷ preempt the signing via IRQ/FIQ and spill the signed pointer onto the stack memory (FIQ is the high-priority interrupt, which can preempt IRQ in arm64). ❸ read the pointer from the spilled stack memory and substitute the pointer for an indirect call that uses the attacker-chosen context. ❹ Consequently an attacker is able to jump to the attacker-chosen place. (i.e, `x0` in ❶)

```
ESTIMATE_DIVERSITY_SCORE (funcs, struct, fidx)
    Input  : funcs: all functions in the kernel
             struct: a target structure
             fidx: a target field index
    Output : diversity score
    Symbol : f.insns: all instructions in f
             si: store instruction
             ds: diversity score
             pts: points-to set
             wk: current worklist (map<f,pts>)
             upd: worklist for next round
             first_wk: worklist for first round
             apts(f, p): andersen pointer analysis on p within f
             check(pts): check if pts meets ds conditions
1   ds ← 0
2   wk ← ∅  first_wk ← ∅
3   foreach f ∈ funcs; do
4       // 1. set up the first worklist
5       foreach i ∈ f.insns; do
6           if i = si & i.pointer_op = struct.fidx then
7               pts ← apts(f, i.value_op)
8               first_wk.add(f, pts)
9           end
10      end
11      foreach f, pts ∈ first_wk; do
12          if check(pts) = true then
13              // 2. increment ds if possible
14              ds ← ds + 1
15          end
16          else
17              // 3. start iterative worklist algorithm
18              depth ← 0
19              wk ← first_wk
20              while wk.size() > 0 & depth < 5 do
21                  foreach f, pts ∈ wk; do
22                      // iterate instructions in funcs
23                      foreach i ∈ funcs.insns; do
24                          if i is a call to f & i.arg ∈ pts then
25                              pts ← apts(f, i.arg)
26                              if check(pts) = true then
27                                  ds ← ds + 1
28                              end
29                              else
30                                  upd.add(i.func, i.arg)
31                              end
32                          end
33                      end
34                  end
35                  wk ← upd
36                  upd ← ∅
37                  depth ← depth + 1
38              end
39          end
40      end
41  end
42  return ds
```

Figure 11: The core algorithm to estimate a diversity score for `objbind`.

C Evaluation Supplement Data

		4.19.49 on Rpi3			5.12.0-rc1 on Mac mini(M1)		
		Stock	**w/ PAL**	**Overhead**	**Stock**	**w/ PAL**	**Overhead**
SPEC 2006 (sec)	400.perlbench	1.631	1.634	0.003 / 0.18%		-	
	401.bzip2	4.616	4.615	-0.001 / -0.02%		-	
	403.gcc	12.701	12.68	-0.021 / -0.17%		-	
	429.mcf	19.082	19.085	0.003 / 0.02%		-	
	435.gromacs	9.769	9.775	0.006 / 0.06%		-	
	436.cactusADM	30.885	31.07	0.185 / 0.60%		-	
	444.namd	118.309	118.394	0.085 / 0.07%		-	
	445.gobmk	1.329	1.33	0.001 / 0.08%		-	
	447.dealII	168.277	168.773	0.496 / 0.29%		-	
	456.hmmer	3.275	3.268	-0.007 / -0.21%		-	
	458.sjeng	23.691	23.655	-0.036 / -0.15%		-	
	462.libquantum	0.324	0.324	0.000 / 0.00%		-	
	464.h264ref	151.026	150.628	-0.398 / -0.26%		-	
	470.lbm	42.153	42.143	-0.010 / -0.02%		-	
	471.omnetpp	3.751	3.75	-0.001 / -0.03%		-	
	473.astar	63.975	63.839	-0.139 / -0.20%		-	
	483.xalancbmk	0.937	0.936	-0.001 / -0.11%		-	
	999.specrand	0.113	0.113	0.000 / 0.00%		-	
LMbench (μs)	null	2.38	2.64	0.26 /10.9%	0.1489	0.1971	0.0482 / 32.4%
	fstat	3.70	3.97	0.27 / 7.3%	0.8170	0.2837	-0.5342 / -65.3%
	open_close	31.46	34.24	2.78 / 8.8%	1.0315	1.1890	0.1575 / 15.2%
	select_200	33.65	31.70	-1.95 / -5.8%	19.0521	2.5541	-16.4980 / -86.6%
	sig_install	4.92	5.44	0.52 /10.6%	0.7392	0.1989	0.2460 / 23.7%
	sig_catch	29.36	31.70	2.34 / 8.0%	7.2327	1.0529	-6.1798 / -85.4%
	protection_fault	0.30	0.60	0.30 / 100%	0.8306	0.2222	-0.6084 / -73.2%
	pipe	69.05	73.47	4.42 / 6.4%	17.3443	19.3347	3.2874 / 11.5%
	unix_sock	84.28	89.27	4.99 / 5.9%	18.0461	18.2759	0.2298 / 1.27%
	fork_exit	719.78	746.14	26.36 / 3.7%	85.6061	91.7627	6.1566 / 7.19%
	fork_exec	774.40	802.43	28.03 / 3.6%	101.3725	103.0943	1.7218 / 1.70%
Linux perf (sec)	`messaging`	2.977	3.089	0.112 / 3.76%	0.164	0.169	0.005 / 3.0%
	`pipe`	69.603	73.212	3.609 / 5.19%	18.087	18.941	0.854 / 4.7%
apache (ms)	1 KB	3.13	3.16	0.03 / 1.06%	0.132	0.133	0.001 / 0.75%
	10 KB	4.10	4.12	0.02 / 0.46%		-	
	100 KB	12.00	12.00	0.00 / 0.02%		-	
	200 KB		-			-	
	1 MB	92.57	92.64	0.08 / 0.08%		-	
	10 MB	895.87	895.78	0.10 / 0.01%		-	
leveldb (ms)	fillseq		-		1.692	1.745	0.053 / 3.10%
	fillsync		-		6.861	6.990	0.129 / 1.80%
	fillrandom		-		4.892	5.013	0.121 / 2.40%
	overwrite		-		4.869	4.665	-0.204 / -4.10%
	readrandom		-		9.332	9.368	0.036 / 0.38%
	readseq		-		0.573	0.575	0.002 / 0.34%
	readreverse		-		1.075	1.069	-0.006 / -0.55%
blogbench (ms)	write		-		382	381	1 / 0.2%
	read		-		416368	415550	818 / 0.2%

	5.12.0-rc1/Mac mini	4.19.49/Rpi3	FreeBSD/Qemu
Stock	123.5 MB	19.9 MB	5.9 MB
w/ PAL	130.7 MB	23.0 MB	6.4 MB
Overhead	7.2 / 5.8%	3.1 / 15.6%	0.5 / 8.5%

Table C1: Detailed data about performance overheads, and image sizes increased by PAL in Linux and FreeBSD kernels.