CAPACITY: Cryptographically-Enforced In-Process Capabilities for Modern ARM Architectures

Kha Dinh Duy
khadinh@skku.edu
Sungkyunkwan University

Kyuwon Cho
kyuwon.cho@skku.edu
Sungkyunkwan University

Taehyun Noh
dove0255@skku.edu
Sungkyunkwan University

Hojoon Lee*
hojoon.lee@skku.edu
Sungkyunkwan University

ABSTRACT

In-process compartmentalization and access control have been actively explored to provide in-place and efficient isolation of in-process security domains. Many works have proposed compartmentalization schemes that leverage hardware features. Newer ARM architectures introduce Pointer Authentication (PA) and Memory Tagging Extension (MTE), adapting the reference validation model for memory safety and runtime exploit mitigation. Despite their potential, these features are underexplored in the context of userspace program compartmentalization.

This paper presents CAPACITY, a novel hardware-assisted intra-process access control design that embraces capability-based security principles. CAPACITY coherently incorporates the new hardware security features on ARM, based on the insight that the features already exhibit inherent capability characteristics. It supports the life-cycle protection of the domain's sensitive objects – starting from their import from the file system to their place in memory. With intra-process domains authenticated with unique PA keys, CAPACITY transforms file descriptors and memory pointers into cryptographically-authenticated references and completely mediates reference usage with its program instrumentation framework and an efficient system call monitor. We evaluate our CAPACITY-enabled NGINX web server prototype and other common applications in which sensitive resources are isolated into different domains. Our evaluation shows that CAPACITY incurs a low-performance overhead of approximately 17% for the single-threaded and 13.54% for the multi-threaded webserver.

CCS CONCEPTS

• **Security and privacy → Operating systems security**; **Software security engineering**.

KEYWORDS

compartmentalization; capabilities; pointer authentication

*Corresponding author

CCS '23, November 26–30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0050-7/23/11…$15.00
https://doi.org/10.1145/3576915.3623079

ACM Reference Format:
Kha Dinh Duy, Kyuwon Cho, Taehyun Noh, and Hojoon Lee. 2023. CAPACITY: Cryptographically-Enforced In-Process Capabilities for Modern ARM Architectures. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3623079

1 INTRODUCTION

Modern software is often large and complex. As a result, it suffers from bugs, some of which are security vulnerabilities that concede program control to adversaries or leak sensitive program resources (e.g., cryptographic keys). Researchers and industry have therefore sought to isolate the monolithic program into multiple process-level compartments [6, 7, 10, 29, 43, 57], each ideally performing a specific task (*separation of privilege*) and is given only the essential privileges (*least privilege*). The process-level compartments of the program must now communicate via *Inter-Process Communication (IPC)* which accompanies inherent performance overhead.

Many works have proposed in-process and in-place compartmentalization methodologies [8, 25, 34, 58, 59, 63, 66, 68] that either re-purposes existing hardware features [8, 34, 68] or adapts of newly introduced hardware features, most notably using Intel's *Protection Keys for Userspace (PKU)* [26]. Recent PKU-based proposals [25, 30, 58, 63, 66] have demonstrated in-place isolation with limited performance overhead. Hardware-assisted isolation on ARM was previously explored [8] using *domains* memory protection feature that had an uncanny resemblance to PKU. However, the domains feature has now been deprecated in AArch64.

We argue that the in-process compartmentalization designs on the modern ARM architectures are currently underexplored. The ARM processor architecture's recent iterations introduced new hardware-assisted software security features. *Pointer Authentication (PA)* [2, 3], is a hardware feature in ARMv8 that employs cryptographic *Authentication Code (AC)* to protect pointers from corruption. *Memory Tagging Extension (MTE)* is another hardware feature included in the ARMv8.5-A architecture that implements a key-and-lock mechanism to enable tagging of pointers and the 16-byte *pointee* memory blocks. However, the principles and mechanisms of the ARM's direction in hardware-assisted security are vastly different from those of x86, i.e., PKU, and open a new design space for program compartmentalization.

In this paper, we propose CAPACITY, a novel OS access control model that revolves around capability security principles [13, 61]. A plethora of existing research has discussed capability as the ideal

scheme for achieving the *least privilege* compartments and eliminating *ambient authority* in OSes [12, 61, 69, 70, 70, 72]. We observe that the new ARM hardware extensions, PA and MTE, carry inherent characteristics of *capabilities*. Capacity's design choices fully leverage these features. It creates in-process *domains*, subprogram components that are given exclusive access to private resources. Each domain is identified and authenticated by their *domain authentication keys* (or domain keys), which is a PA cryptographic key that Capacity reserved for authenticating resources. Our work retrofits the OS-based access control with its consistent capability scheme that provides *life-cycle* protection of sensitive objects; it transforms not only memory references (i.e., pointers) but also file object references into non-forgeable tokens.

In-process capabilities for object life-cycles. Capacity brings a coherent in-process capability for compartments within the process (subjects) and abstract process resources (objects) whose state may alternate between a file or memory content throughout its life cycle. Capability-based *memory* access control has been explored by many previous works [12, 14, 33, 70, 72]. Notably, the CHERI architecture [12, 70, 72] applies memory capabilities to pointers, although the requirement of customized processor architecture limits its applicability. Process-level capabilities have also been studied from OS design perspectives [13, 18, 50, 61, 69]. These works focus on access control of OS resources bestowed on the process often represented in the form of *files*. However, securing today's large monolithic user programs calls for finer-granular access control on files, not to mention the necessity of consolidating a memory access control for in-process compartments.

Simple and efficient reference monitoring. Capacity's coherent capability scheme also seamlessly incorporates a simple and efficient reference monitor design for in-process access control. The necessity and design space of efficient *system call (syscall)* reference monitors for in-process domains have been discussed in many previous works [11, 58, 66, 67]. Since the OS kernel is unaware of the in-process compartments, a reference monitor must mediate accesses to process resources among the potentially mutually distrusting compartments in addition to syscalls filtering that may undermine the security of the compartments. We take a different route from previous works to achieve both goals coherently. Capacity's capability-engraved file descriptors carry information for authentication and authorization. This eliminates the need for separate data structures, i.e., an *Access Control List (ACL)*, to keep track of each compartment domain's ownership and access rights on file objects. Also, the validation process of the capability is fast, as it is essentially achieved through a single PA instruction.

Novel domain and PA context binding scheme. Our way of creating PA contexts for in-process compartments is unique and specifically devised for Capacity's life-cycle capabilities. Previous work has presented a framework for using PA and MTE that establishes kernel compartments with policy-defining PA *modifiers* [43]. However, Capacity chooses to assign a unique *PA key* for each domain and perform *key switches* during domain transitions and employs the PA modifier to isolate references between domain *instances*. Capacity then uses the per-domain key and instance modifier to compute the cryptographic AC for the resources and embeds them into the resource handles themselves. This seemingly small difference is a key design component for Capacity. With a single key switch, Capacity can efficiently switch the authentication context for both system resource handles and signed userspace pointers. This allows Capacity to establish arbitrary compartmentalization boundaries within programs and enables a unified and consistent PA-based authentication throughout the life cycles of program resources without a complex modifier management scheme.

Challenges. Design and implementation of Capacity must satisfy security requirements and be mature enough to be adopted to real-world applications. Capacity rigorously assesses and addresses the security requirements for the capability references, namely *non-forgeability* and *non-reusability*. This effort has been made for every sensitive operation carried out by Capacity, from domain transitions to signing and authentication of the capability tokens. We detail our security considerations as we elaborate on the design and provide a dedicated security analysis.

In addition, Capacity implements a robust instrumentation framework for complex user programs from scratch. Capacity's PA+MTE-accelerated pointer capability requires *complete mediation* of all pointer uses. Applying such a scheme to complex programs poses a daunting challenge since a single incorrect instrumentation would inadvertently crash the program. Similar PA-based complete mediation of pointer uses have been developed by previous works; however, none with the level of maturity of Capacity. For instance, PARTS [38] was only evaluated with a benchmark suite. Also, Capacity incorporates MTE to inter-domain memory isolation. HAKC [43] presented PA+MTE instrumentation for kernel module compartmentalization that only authenticates the cross-domain pointers only once before their first use, while Capacity must provide a domain-aware and completely mediated pointer load and store instrumentation. As we will show through our evaluation, Capacity's instrumentation framework is mature enough to compile programs such as NGINX-LibreSSL and OpenSSH SSH client. In summary, our contributions are as follows:

- We introduce a novel in-process compartmentalization design that adapts hardware-accelerated capabilities for the life-cycle protection of domain resources.
- We design a simple and efficient capability-based reference monitor to isolate in-process system resources.
- We establish PA key-identified domains to efficiently isolate kernel and userspace resource references.
- We assess and address the unique security challenges of capability-based isolation that requires complete mediation on reference uses, prevention of impersonation, and protection against forging and reusing of references.
- We develop an instrumentation framework that is robust enough to completely mediate complex user programs by addressing compatibility issues.
- We evaluate our implementation[1] on real-world applications and report low overheads on the approach.

[1] Available at: https://github.com/sslab-skku/capacity

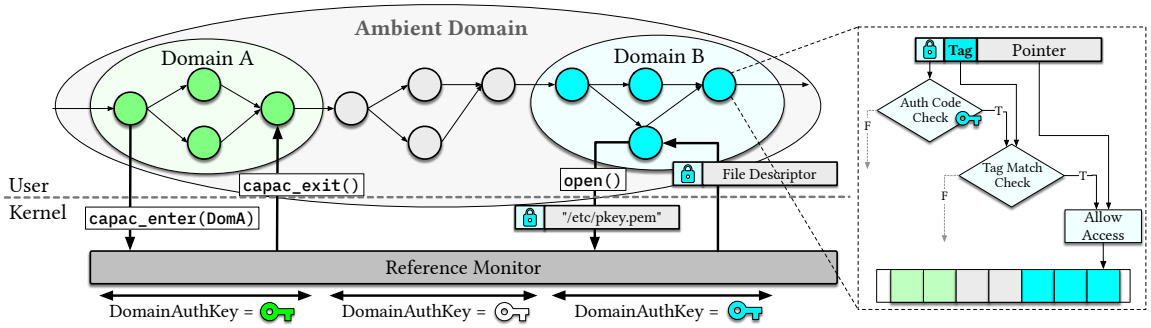


Figure 1: Overview of CAPACITY's in-process domains and life-cycle objects protection.

2 BACKGROUND

2.1 Pointer Authentication (PA)

Pointer Authentication (PA), introduced on ARMv8.3-A [2], provides hardware acceleration and ISA extension for cryptographically authenticated pointers. The design of PA is inspired by previous works that protect pointers from corruption [32, 42]. PA allows attaching a *Pointer Authentication Code (PAC)* to a pointer in the unused bits [54:48] of a 64-bit address. A PAC is generated using one of the five keys stored in the newly added registers that can only be accessed or modified with privileged instructions. The five keys consist of two keys for signing code/instruction pointers (Instruction-{A,B}), another two for signing data (Data-{A,B}) pointers, and one general-purpose key (G). We denote these keys as $\mathbf{K}_{IA}$,$\mathbf{K}_{IB}$, $\mathbf{K}_{DA}$, $\mathbf{K}_{DB}$ and $\mathbf{K}_{G}$. PA also introduces `pac` and `aut` families of instructions to *sign* (i.e., calculate the PAC and attach it to the value in the operand register) and *authenticate* a 64-bit value with an *optional* modifier using the key indicated by the instruction name. If authentication on a pointer succeeds, the authentication code in the pointer is cleared, making the pointer usable. The PAC value is corrupted if the check fails, which triggers a segmentation fault when the pointer is dereferenced. We denote the operations of `pac` and `aut` instructions that use the key **K** to sign and authenticate a given data *D* as `PAC`(**K**, *D*, *mod*) and `AUT`(**K**, *D*, *mod*).

2.2 Memory Tagging Extension (MTE)

Memory Tagging Extension (MTE) [4] is another hardware-backed security feature to ARMv8.5-A [2]. MTE adapts the principles of tagged memory in the existing research proposals and implementations in other architectures [46, 71, 75]. It facilitates memory safety by enabling the 4-bit tagging of pointers and 16-byte aligned address ranges. When MTE is enabled, the processor raises an interrupt if the tags differ between the memory and the pointer, allowing vulnerabilities such as buffer overflow to be captured. We denote `TAG`(*P*, **T**, `size`) as the tagging operation that first assigns the 4-bit tag **T** to the bits [59:56] of a pointer *P*, then tag 16-byte-aligned memory region between *P* and *P*+size with **T**. In addition, PA and MTE can be *simultaneously* enabled. Hence, a pointer can be tagged *then* signed, and the resulting pointer would carry a tag in bits [60:56] and PAC in [54:48].

3 OVERVIEW

Figure 1 illustrates an overview of CAPACITY. CAPACITY's design demonstrates a comprehensive capability model for file and memory objects access control that can be applied to commodity ARM systems that support PA and MTE. CAPACITY's primary objective is to guarantee each domain's exclusive access rights to the sensitive domain-private objects. It associates each domain, called *a CAPACITY domain*, with a unique PA cryptographic key and *switches* the currently activated key using an in-kernel reference monitor before entering a domain.

Consider the common pattern of sensitive object use shown in Figure 1 (Domain B). First, a file-system *path reference* (`"/etc/key.pem"`) is used as an argument to the `open()` syscall. A *file descriptor* to the file object is then obtained. Finally, the object is loaded into a memory region (e.g., with the `read()` syscall) so the program can interact with it through *pointers* (`0x00004bfff`...). In CAPACITY, the creation and usage of three types of resource references are *completely mediated* with a coherent PA+MTE-based authentication, provided by (1) a lightweight in-kernel reference monitor that efficiently validates path and file descriptor (`FD`) references in syscall arguments, and (2) PA+MTE-assisted tagged memory allocation and pointer authentication instrumentation that isolate domain-private memory and their references.

This section provides a high-level overview of CAPACITY's capability model and its programming model. We will delve into the design and implementation of each of CAPACITY's components that enforce its security in §4, §5, and §6.

3.1 Threat model

We assume that the domains are mutually distrusting. The program's vulnerabilities can potentially grant adversaries with arbitrary memory manipulation and control-flow subversion primitives, compromising an CAPACITY domain or the *ambient* domain (the rest of the program code not inside CAPACITY domains). Even so, a compromised domain must not access other domains' private objects. We assume that the processor and the kernel are trusted and that modern OS security measures, such as the W⊕X policy and ASLR, are in place. Additionally, we deem the program initialization, including the initialization of CAPACITY, free of attacker influence. We exclude side-channel attacks and microarchitectural attacks from the scope of this paper.

Capacity also incorporates the existing backward-edge and forward-edge *Control-Flow Integrity (CFI)* in its implementation and is compatible with recent advances in PA-assisted CFI techniques. PA-based backward-edge CFI [37, 38, 51, 55] authenticates the return addresses on the stack with $\mathbf{K}_{IB}$, which render the traditional attacks (e.g., ROP attacks) that overwrite the return address on the stack infeasible. PA-based forward-edge CFI [38, 73] authenticates code pointers with $\mathbf{K}_{IA}$ with its LLVM *ElementType* ID as the modifier, which restricts indirect calls to (1) valid entry points of functions (2) functions of the matching type.

3.2 Security requirements

The capability principles hold only when the common security requirements for capability are met. In particular, we observe that Capacity must satisfy the following security requirements:

- (R1) *Non-impersonatable domains:* Capacity must be able to identify and authenticate intra-process domains and prevent impersonation.
- (R2) *Complete mediation:* all access to file and memory objects must be mediated by Capacity.
- (R3) *Non-reusable references:* Capacity domain-private references are only valid within the domain that owns the object.
- (R4) *Non-forgeable references:* Capacity references must not be forged by the adversary.

Throughout the rest of this paper, we use the above requirements to analyze and manifest the security guarantees of Capacity. Upholding the security requirements, therefore, is a challenge that must be addressed by Capacity's design and implementation.

3.3 Subjects and Objects

In Capacity's capability model, the subjects are subgraphs of program execution that interact with sensitive objects called *Capacity domains*. Program code that does not belong to a domain is called the *ambient* domain. Capacity identifies and authenticates each domain with a unique PA key, called the *domain key*. Each domain is also associated with an MTE tag for the tagging of the domain's private memory. We use $\mathbf{K}_{DB}$ among the kernel-managed PA keys as the domain key. $\mathbf{K}_{DA}$ is the *ambient key* used to sign and authenticate resources accessible by the entire program. $\mathbf{K}_{\{IA,IB\}}$ are reserved for proposed PA-based forward-edge and bardward-edge CFI defenses [1, 37, 38, 55].

Domain switching. A Capacity domain is encapsulated between the APIs `capac_enter` and `capac_exit`. Domain switching is performed with the help of a lightweight in-kernel reference monitor, as we will describe in §4. Upon entering a domain, the userspace program updates its current MTE tag and requests the reference monitor to switch the currently activating domain authentication key to that of the target domain. A *per-instance* PA modifier is also maintained in both kernel and userspace to identify *instances* of a domain invocation. `capac_exit()` restores the domain key, MTE tag and modifier to those of the ambient domain. We do not support the nesting of domains in the current prototype.

Object encapsulation. As summarized by Figure 2, Capacity facilities employ PA to *sign* private resource references with the domain key and the per-instance modifier (e.g., a unique session ID), engraving the reference ownership into its embedded AC. Before a

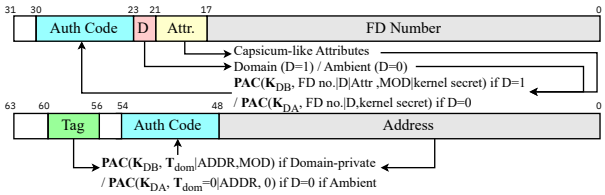


Figure 2: Capacity's reference encapsulation of pointers and file descriptors with cryptographic authentication code (AC).

reference is used to access some resource, Capacity *authenticates* its AC with the currently activating domain key and modifier before granting the access. Consequently, object references are turned into non-forgeable ((R4)) *capability tokens* that are valid only when the context is within their owner domain. By simply switching the authentication key, Capacity prevents cross-domain reusing of both kernel and userspace object references ((R3)).

We introduce the *ambient objects*, non-private and accessible from all domains, e.g., global variables and non-sensitive files, to avoid the programming model becoming too restrictive. All references to ambient objects are to be signed and authenticated with $\mathbf{K}_{DA}$. The ambient memory objects are given a $\mathbf{T}_{Dom}$ of 0, which makes all untagged memory pointers ambient memory references, and only the deliberately tagged pointers become references to domain-private memory. To facilitate multi-domain interactions, Capacity also supports the *delegation* of references. When a reference delegation request is received, Capacity first authenticates the reference to prevent delegation of non-owned resources ((R1)), then *re-signed* it with the target's domain key and instance modifier. Especially for memory reference delegation, Capacity compares the pointee memory's tag with the pointer's tag before recoloring both to the target domain's memory.

3.4 Programming model overview

A programmer interacts with Capacity through a set of well-defined APIs provided by the runtime library and program *annotations* that direct the program instrumentation, shown in Table 1. We demonstrate the programmer's perspective when retrofits intra-process capabilities into programs through a quintessential example of secret import and uses in Figure 3.

The programmer first conceptually defines in-program domains (e.g., `DOM_CRYPTO`) within the program with respect to the domain-private resources. The subprogram interval is enclosed by `capac_enter()` and `capac_exit()` (lines 4 and 7), and each interval is paired with a domain key. In the example, `ctx->id` contains the unique ID for the encryption context, which is set as the currently activated PA modifier for `DOM_CRYPTO`. A domain can span a few function calls, as shown in the example. The domain entrance cannot be nested; however, Capacity is designed to support the sharing of functions between domains.

The programmer then identifies domain-private objects and their references. In the case of file(-like) objects, a list of domain-to-file mappings is provided as an argument to `capac_init()` (line 1) to notify the reference monitor of the domain ownership of file

	API & Compiler Annotation	Description
API	`capac_init(configurations)`	Initializes CAPACITY facilities, assigns `PATH` to domains, enables syscall authentication
	`capac_enter(target_id, mod)/capac_exit()`	Enters the target domain/Exits to the ambient domain
	`capac_limit_fd(fd, cap_mask)`	Limit the capabilities of an `FD`
	`capac_delegate_fd(fd, target_id, mod, cap_mask)`	Limits the capabilities and delegate an `FD` to the target domain
	`capac_malloc(size)/capac_free(ptr)`	Allocates/frees memory tagged with the currently active domain
	`capac_delegate_ptr(ptr_loc, size, target_dom, mod)`	Delegates an in-memory pointer to `target_dom` and also re-colors the memory
Ann.	`DOM_PRIV_FUNC`	Tags function's stack frame with the currently executing domain's tag ($\mathbf{T}_{Dom}$)
	`DOM_PRIV`	Marks a *source* domain-private pointer for taint analysis.

Table 1: CAPACITY APIs and program annotations

```
1  capac_init(...);
2  ...
3  // K_DB = K_Dom_crypt , Dom_curr = Dom_crypt , Mod_curr = ctx->id
4  capac_enter(DOM_CRYPTO, ctx->id);
5  load_secret(ctx, ''secret.key'');
6  encrypt(ctx, req);
7  capac_exit();
8  ...
9  void load_secret(DOM_PRIV crypto_ctx_t* ctx,
10                  const char * key_path){
11 // Import the secret key from the filesystem
12 int fd                                              FD-Sign
13     = open(key_path, O_RDONLY);                      PATH-Auth
14 ctx->secret_key = capac_malloc(KEY_LEN + 1);        PTR-Sign
15 read(fd, ctx->secret_key, KEY_LEN);                 FD-Auth / PTR-Auth
16 ...
17 }
18 DOM_PRIV_FUNC // Isolate function stack frame
19 void encrypt(DOM_PRIV crypto_ctx_t* ctx, req_t* req) {
20 // Use the secret key to encrypt a plaintext string
21 for (int i = 0; i < KEY_LEN; i++)
22     req->ciphertext[i]                               PTR-Auth
23         = ctx->secret_key[i]                         PTR-Auth
24           ^ req->plaintext[i];                       PTR-Auth
25 }
```

: Domain-private references : Ambient references
Action : CAPACITY actions ANNOT: CAPACITY annotations

Figure 3: Example of CAPACITY's programming model, annotated with CAPACITY references and their enforcement.

resources. Afterward, the reference monitor automatically authenticates any path and FD references in the syscall arguments (lines 11, 12, 14) and issues FDs signed with their owner domain when FDs are created. For memory objects, the programmer must mark the domain-private region containing the object and its domain-private pointer references. To allocate domain-private memory, the programmer either changes their allocation site to use `capac_malloc` (line 13), which allocates memory tagged with the domain's tag and returns a tagged pointer, or uses `DOM_PRIV_FUNC` (line 17) to direct the instrumentation to isolate the function's stack frame. On `DOM_PRIV_FUNC`-annotated functions, the function is instrumented such that the stack frame is *tagged* with the currently executing domain's tag number and reverted upon function return. Then, the programmer applies `DOM_PRIV` (line 18) to the *source* domain-private pointer. From then on, the pointer is *taint-tracked* to mark all derived references to the object within the function. CAPACITY then instruments PA-based signing/authentication with the domain key of tracked pointers, including the member variables (e.g., `ctx->secret_key`) (lines 13, 14, 21).

4 DOMAIN SWITCHING AND AUTHENTICATION

CAPACITY's API design allows the introduction of flexible compartmentalization boundaries to existing software through in-place annotations. However, such a design must be robust against *domain impersonation* (R1). In CAPACITY's threat model, domain impersonation happens when a compromised domain diverts the control flow into CAPACITY APIs, e.g., calling the domain entry gates with arbitrary arguments, or corrupts CAPACITY's critical values, e.g., MTE tag ID stored in memory to gain illegal access to domain-private resources. CAPACITY prevents such domain impersonation using two techniques. First, CAPACITY *authenticates* the domain entry sites by requiring them to construct *entry tokens*, which the reference monitor can verify before granting the domain switch. Second, it introduces an in-userspace *authentication* protocol that securely fetches the currently executing domain's MTE tag. In both cases, after successful authentication, CAPACITY securely places its critical values (the domain's memory tag and the per-instance modifier) in compiler-reserved registers and fetches them when needed to prevent illegal modifications.

Secure domain switching. The program enters a domain through the API `capac_enter` (Table 1) that internally invokes an `ioctl` syscall to the reference monitor. The reference monitor then switches the currently activating $\mathbf{K}_{DB}$ to that of the target domain by writing into the dedicated PA key registers with its kernel privilege. Hence, an attacker who can invoke `ioctl` calls with arbitrary values could achieve *impersonation* and access domain-private references. CAPACITY protects the domain switches from impersonation by authenticating domain entry points with $\mathbf{K}_G$. Before a domain switching request, the call gate signs the domain ID argument in `capac_enter(domID, mod)` with a `pacga` instruction, then passes the signed argument to CAPACITY reference monitor. Since there is no dedicated authentication instruction that uses $\mathbf{K}_G$, the reference monitor uses `pacga` to generate the valid argument with the domain ID and compare it with the argument from the call gate. Domain entry is granted if the two values match. After the entry is granted, the call gate loads the per-instance modifier into a reserved floating-point register (`ModReg`), which is to be used by

our instrumentation for pointer authentication demonstrated in §6. Finally, the call gate clears the token-containing register to prevent the leaking of the entry token. `capac_enter` is implemented as the following call gate, which is to be inlined to the domain switching sites:

```
PACGA(token, target_id, 0)
if(!ioctl(capac_fd, CAPAC_ENTER, token, modifier))
    exit();
// Load per-instance modifier into ModReg register
asm volatile("fmov ModReg, %[mod]" ::[mod] "r"(modifier) :);
// Clear entry token from register
```

Capacity endows authenticity to the entry points by preventing attackers from diverting the control flow into arbitrary instructions with CFI and ensuring that the instruction `pacga` is absent within the process but the domain entry sites. $\mathbf{K}_G$ is currently unused in both deployed and proposed PA-based defenses. Therefore, it is unlikely that it would naturally appear under normal circumstances. Also, since ARM is a RISC architecture, we can generally rule out the issue of unaligned, unintended occurrences of the instruction, an issue that makes preventing illegal instruction occurrences challenging on x86 architectures [63].

Authenticated domain ID retrieval. Capacity enables a trustworthy runtime domain ID fetching procedure. This avoids hard-coding MTE tags into functions that require memory tagging since these functions might be used in confused deputy attacks to tag arbitrary memory regions. Moreover, fetching the domain ID at runtime allows a Capacity-protected function to be called from multiple domains. The procedure is also imperative in runtime domain memory and pointer tagging in Capacity's instrumentation, as we will explain in §6. Capacity's private heap memory allocator, `capac_malloc()`, also uses the procedure to authenticate the callee's domain ID and tag the allocated memory.

To prepare for runtime domain ID retrieval, Capacity first constructs a *Domain Signature Table (DST)* that is filled with *domain signatures* of each declared domain (`DST[i] = PAC`($\mathbf{K}_{DBi}$, 0, 0)). After initialization, the table remains read-only throughout program execution to prevent tampering. Additionally, a global variable that stores the domain ID of the currently active domain, `int curr_dom`, is updated on each `capac_enter(domID)` and `capac_exit()` invocation.

The following snippet is inlined to verify the current domain ID, such that it can be used as a MTE tag for the pending operation:

```
domain_signature_t dom_sig = DST[curr_dom];
AUTDB(dom_sig);
assert(dom_sig == curr_dom);
// Form a tag mask shifting left by 56 bits
asm volatile("lsl %[tag], %[tag], 56" : [tag] "=r"(target_id)::);
// Load tag mask into TagReg register
asm volatile("fmov TagReg, %[tag]" ::[tag] "r"(target_id) :);
// T_Dom is known from this point onward
```

The above procedure first fetches the current domain's signature from the DST, then authenticates it with the domain key. If authentication succeeds, a *tag mask* is constructed, e.g., `0x00ff00..00`, which can be applied to an untagged pointer with a bitwise `OR`. The tag mask is stored in a reserved floating-point register (`TagReg`), so the memory tagging operations can efficiently retrieve it.

5 FILE SYSTEM OBJECT ISOLATION

Capacity includes a reference monitor that verifies the *arguments* of syscalls to enforce the complete mediation ((R2)) of system resource accesses. This capability-inspired authentication mechanism drastically differs from previous works implementing ACL-based reference monitors for in-process syscall filtering and file access control [11, 25, 58, 63, 66]. In those systems, enforcing access control on domain-private sensitive file objects requires a separate logic for the monitor detached from the program semantics. Moreover, ACL-based access control requires keeping track of the isolated objects and the domains that can access them at runtime [58]. In contrast, the `FD`s in Capacity are cryptographically secured; the proof of ownership and object attributes are attached to the reference itself, regardless of the number of subjects sharing a resource.

We implement the reference monitor in about 1000 LoC. It provides domain key switching and syscall interception through syscall table hooking using a similar approach to a previous efficient reference monitor [58]. To avoid the system monitor affecting all processes within the system, we set an unused flag in `struct thread_info->flags` to mark Capacity-enabled processes during initialization and check for the flag before performing syscall authentication.

5.1 Enforcing domain-private file-system paths

Capacity supports file path isolation to be compatible with existing programs without significant rewriting effort. Capacity's file path authentication utilizes the pre-existing methods for attaching policies to file system objects used by kernel security subsystems (e.g., SELinux [56]).

File paths protection is kickstarted by the `capac_init` API call that sends a list of domain-private files and their owners to the reference monitor. The reference monitor first resolves each path to obtain the corresponding `inode` structure. It creates a per-process table in the file's `inode->f_security`. For each owner domain of a file path, it stores the *domain signature*, `PAC`($\mathbf{K}_{DB}$, `NULL`, 0), in the table. If the file is not domain-private, the table is left as `NULL`. After, the reference monitor transitions the process into the *protected mode*, where the reference monitor authenticates every path and `FD` arguments used in syscalls.

Before authenticating a file object import, the requested file path is first resolved to obtain the underlying `inode` structure. This is to avoid confusion when encountering relative paths and symbolic links. If the table for the current process in `inode->f_security` is `NULL`, the reference monitor recognizes that it is an *ambient* object and can be accessed without further authentication. If domain signatures are present, the reference monitor authenticates them using the corresponding domain key until a valid signature is found. The reference monitor verifies that the current user context is in the object's owner domain if any valid signature is found while rejecting the file access otherwise. Finally, the returned `FD` for the requested file is signed with $\mathbf{K}_{DA}$ for ambient objects or the domain key for domain-private objects.

5.2 Enforcing domain-private file descriptors

Capacity's reference monitor intercepts syscalls that generate `FD`s and create signed `FD`s before they are sent to the userspace. It also

intercepts FD-accepting syscalls (e.g., `read`) for FD authentication. Hence, all FDs issued to the process carry ACs that are authenticated on use. Additionally, we use a *secret modifier* only known to the kernel as the PA signing modifier, to prevent forging by signing a plain integer with `pac` instructions in the userspace (R4). CAPACITY's security model also requires that the integer for FDs are not reused (R3), even after they are closed. We achieve this by changing the hook for the `close` to reserve the closed FD in the thread's FD table, preventing the OS from reusing it.

In-process FD capabilities. CAPACITY's FD access control scheme resembles that of Capsicum's capability-enabled FDs [69] but is much more lightweight and is enforced at the in-process granularity. CAPACITY's FD capabilities scheme requires no additional kernel metadata for the permissions of FDs since the information necessary to authorize an FD use is engraved in the FD itself. Moreover, thanks to integrity protection provided by cryptographic authentication, we can securely embed fine-grained access control attributes into the file descriptors themself (`FD->Attr.` in Figure 2) without extensive bookkeeping. A combination of PA-assisted authentication on the FD and a per-syscall check based on static policies is sufficient to grant or reject file descriptor access.

Our prototype supports four capabilities, represented as a bitmask of enabled capabilities. The `CAP_WRITE` and `CAP_READ` capabilities allow read-related syscalls (e.g., `read`, `recvfrom`) and write-related syscalls (e.g., `write`) to use the FD. `CAP_SOCKET` enables network-related syscalls, such as `accept` and `listen`. `CAP_DELEGATE` allows the FD to be delegated to other domains. The reference monitor uses a capability bitmask for each syscall to validate the FD's attributes. On the other hand, `cap_limit_fd()` instructs the reference monitor to remove the selected capabilities from an FD and re-sign it.

Signing and authentication of 32-bit FDs. Figure 2 demonstrates the signing strategy of file descriptors. Since most file descriptors are 32-bit signed integers, we devised a 32-bit signing and authentication scheme. To sign an FD, the reference monitor first signs its 64-bit zero-extended value like a normal pointer. Then, it attaches the PAC to bits[30:23] of the FD. The monitor uses bit[22] (`FD->D`) bit to distinguish between domain-private and ambient FD, such that domain-private FDs are authenticated using the domain key. Bit[17:21] stores the bitmask of the previously described FD capabilities. bit[31] is unused to prevent the descriptor from being interpreted as an error code (e.g., `if (fd < 0)`). FD authentication on 32-bit FDs happens likewise.

Handling special FD values. During our implementation, we found that special FD values, such as `STDIN`, `STDOUT`, or `AT_FDCW`, need to be handled differently since they are often hard-coded into the application as integer values (e.g., 1 and 2). We currently make our reference monitor omit their authentication for compatibility, given that they are always ambient objects.

6 DOMAIN MEMORY ISOLATION

This section describes how CAPACITY establishes its capability-inspired memory isolation model. CAPACITY enforces domain memory isolation through MTE-assisted segregation of domain-private memory regions and a *complete mediation* (R2) of memory access through pointers that are made *non-reusable* (R3) and *non-forgeable* (R4) capability tokens.

CAPACITY first introduces facilities to allocate domain-private memory, including a tagged heap memory allocator and compiler-instrumented stack tagging. CAPACITY also present an instrumentation that mediates pointer uses with PA-based *on-load* pointer authentication. It instruments the program to sign pointers before they are stored in memory and authenticate pointers as they are loaded from memory into registers. Different from PARTS [38], a previous work that also introduced an on-load pointer authentication scheme, CAPACITY uses $\mathbf{K}_{DA}$ to sign/authenticate ambient pointers, and *selectively* uses $\mathbf{K}_{DB}$ for domain-private pointers identified using a static taint analysis, making its instrumentation *domain-aware*. CAPACITY also uses a user-defined, instance-specific modifier for domain-private pointers to prevent their reuses across domain instances.

CAPACITY's instrumentation framework seeks to be compatible with complex user programs, as we will demonstrate through our evaluation (§7). It must enforce *complete mediation* of program pointer uses for whole-program pointers. We initially used the existing implementation of PARTS [38] but quickly found that it is inapplicable to large programs since even one incorrect handling would result in a crash. Toward this end, our instrumentation scheme is developed from scratch to support large userspace programs reliably. Our instrumentation handles *stack spill* with a new *pointer liveness tracking* algorithm. We also introduce solutions to compatibility issues we observed with whole-program PA instrumentation. The instrumentation framework is implemented on LLVM 14.0.0 [52], which includes built-in intrinsics for PA and MTE features. As of the time of writing, no available hardware supports MTE. For this reason, we used a QEMU virtual machine (`-march=armv8.5-a+pauth+memtag`) that supports emulating MTE instructions as a testing target during implementation.

6.1 Domain-private memory tagging

CAPACITY establishes domain-private memory regions by tagging each domain's private memory with the domain's integer domain ID ($\mathbf{T}_{Dom}$) obtained with domain authentication (§4).

Private heap memory allocation. `libcapacity` manages a domain-private heap memory with its `dlmalloc`-based allocator that maintains a statically allocated, MTE-enabled memory pool. Upon invocation, `capac_malloc(size)` first performs DST-based domain authentication so that it can determine the tag number of the currently executing domain as we explained in §4. Afterward, `capac_malloc()` allocates memory from its memory pool, tags it with the domain ID, and returns a pointer tagged with the domain ID. The returned pointer is not signed as it is delivered in the return value register, adhering to CAPACITY instrumentation's on-store sign and on-load authentication policy.

Private stack tagging. CAPACITY provides private *stack* memory to domains through in-place tagging and untagging of the stack objects with the current domain ID (i.e., no explicit stack switching). The `DOM_PRIV_STACK` annotation directive on a function notifies the instrumentation framework to instrument the function prologue and epilogue to prepare a domain-private stack before the function execution and to untag and zero out the tag before return.

```
.prologue:
 // Sign return addr
 pacib  x30, sp
 ...
 // x9: DST[curr_dom]
 // Auth current domain get T_Dom
 autdzb   x9
 cmp    x9, x8
 b.ne  .auth_failed
 // Build tag mask and save to ModReg
 lsl    x8, x8, #56
 fmov   TagReg, x8
```

(a) Prologue: Domain ID auth.

```
...
// Tag pointer
// x8: T_Dom tag mask
fmov x8, TagReg
mov  x24, sp
orr  x25, x24, x8
// Tag stack frame
stg  x25, [x25]
...
```

(b) Private stack tagging

```
// x0: domain-priv. ptr
fmov    x8, ModReg
pacdb   x0, x8          PTR-Sign
str     x0, [mem]
...
// x13: domain-priv. ptr
ldr     x13, [mem]
fmov    x8, ModReg
autdb   x13, x8         PTR-Auth
...
// x12: ambient ptr
autdza  x12             PTR-Auth
```

(c) Domain-aware pointer auth.

```
.epilogue:
 mov    w0, #1
 // Untag/zero out stack
 stzg   x24, [x24]
 // Clear tag mask register
 fmov TagReg, xzr
 ...
 // Auth return addr
 retab
```

(d) Epilogue: stack clean up

Figure 4: Capacity's instrumentation of an annotated function.

Figure 4 (a), (b), and (d) show the stack tagging instrumentation on an annotated function. The instrumented prologue authenticates and obtains the currently executing domain's ID by consulting DST, then loads the tagging mask into the reserved register. Then, it retrieves the tagging mask from the reserved register and tags the function's stack frame when a stack variable is allocated. The epilogue returns the stack frame to the ambient domain by zeroing out the stack contents and setting the tag on the memory region back to 0. The instrumentation uses `stzg` instruction introduced in MTE that performs such operation efficiently in the hardware.

6.2 Domain-aware pointer authentication

The instrumentation now inserts `pac` and `aut` instructions for pointer store and load sites and uses $\mathbf{K}_{DB}$ *selectively* for domain-private pointers. Compared to *on-use* pointer authentication schemes, such as the one used in PA-based CFI [38, 73], on-load authentication provides better efficiency and compatibility when adapted to whole-program data pointers [38].

The previous PA on-load pointer instrumentation scheme [38] instruments the program in LLVM *Machine IR (MIR)* to handle cases where pointers are *spilled* into memory by the compiler. However, we found that it does not reliably detect and instrument such cases due to the use of ad-hoc heuristics. To handle this issue, we also implement our instrumentation in MIR after most optimizations are already performed but introduce a vastly improved instrumentation algorithm. Our instrumentation relies on embedded metadata from our IR-level taint analysis and built-in type metadata to inform the instrumentation decisions without modifying the instruction selection pipeline. Moreover, instead of using heuristics, we introduce a liveness analysis that keeps track of sensitive pointers on the stack frame and inside the registers. Those improvements allow us to reliably enforce the on-load authentication scheme while also avoiding intrusive changes to instruction selection and register allocation stages.

Identifying domain-private pointer load/store. Capacity employs an intra-procedural data-flow analysis that aims to find all IR pointer-containing variables, where an annotated domain-private variable, or its members, may flow to. The analysis takes two sources: variables that are explicitly annotated with `DOM_PRIV` and pointers returned from `capac_malloc()`. We use a worklist algorithm starting from the taint source to visit all variables and instructions recursively. It follows the def-use chains of LLVM IR instructions and adds all users of the visited instruction to the worklist. When an instruction that addresses a pointer variable (e.g., `alloca`, `getelementptr`) is encountered along the def-use chain, the analysis attaches metadata to the LLVM `value` that the later instrumentation can retrieve. Additionally, the analysis performs backward tracking whenever it encounters a store instruction since def-use analysis cannot track such data-flow; it adds the instruction's *destination* operand to the worklist in such cases.

MIR instrumentation. Our MIR instrumentation is based on *liveness analysis* in compiler designs [49]. The instrumentation keeps track of registers and stack frame locations that potentially contain pointers/sensitive pointers, called the *live pointer set*. At compile time, the pass scans and visits MIR instructions. On load or store instructions, it tries to retrieve LLVM type and taint information to extract whether the instruction accesses a pointer/sensitive pointer. If such information is unavailable, the instrumentation consults the live pointer set. For a store instruction, the instrumentation checks whether the *source* operand (e.g., "`x1`" in `str x1, [sp, #8]`) is within the live pointer set, and whether the operand is sensitive. For a load instruction, the same check is performed on the instruction's *destination* operand (e.g., "`[sp, #8]`" in `ldr x8, [sp, #8]`).

Instrumentation is performed on the load and store instructions as shown in Figure 4c. On non-sensitive pointer operands, it instruments with $\mathbf{K}_{DA}$ signing/authentication. Before signing an ambient pointer, the instrumentation inserts a masking instruction that zeros out the tag to prevent pointer forging through ambient code signing (R4). When instrumenting sensitive pointer operands, $\mathbf{K}_{DB}$ is used. An instruction that loads the modifier from `ModReg` into a spare register (`fmov`) is also inserted. The modifier is then used for signing/authentication of the pointer. Finally, the instrumentation updates the live pointer set after visiting each instruction. For instance, if an instruction *kills* a register or overwrites a stack frame location with non-pointer data, the register/stack frame index is removed from the live pointer set.

6.3 Handling compatibility issues

We developed supporting transformations that can be optionally enabled to solve PA-based instrumentation compatibility issues.

$$\text{visit}(id, t) = \begin{cases} \{\text{sign}(id)\} \cup \text{visit}(*id, t_1) & \text{if } t = t_1* \\ \bigcup_{i=1}^{n} \text{visit}(id_i, t_i) & \text{if t = struct } \{id_1 : t_1, ..., id_n : t_n\} \\ \varnothing & \text{if } t = \text{int} \end{cases}$$

Figure 5: Type-based global constructor generation. sign(id) signs an in-memory pointer indexed by LLVM variable id.

Type-unsafe object initialization. The following example demonstrates the type-unsafe object initialization pattern found in C applications, including NGINX and LibreSSL:

```
1 | obj_t* obj = calloc(1, sizeof(obj_t));
2 | //...
3 | if (obj->ptr == NULL){ /* Perform initialization */ }
```

In this example, when the program loads an uninitialized pointer from memory at line 3, the on-load authentication would fail since the pointer is not signed, making the branch condition evaluate incorrectly. To solve this, we introduce a pass that transforms the program's NULL checks to cover the AUT failed NULL value (e.g., `ptr != NULL && ptr != 0x20000000`). We also explored inferring the type of the zero-initialized objects, then recursively signing all of its containing pointers, using type inference methods proposed in [64], but found that the approach is infeasible on complex programs without heavy source code processing.

Global pointers in complex structures. The prototype of PARTS [38] only scans and signs pointers in the global structures at the top-most level, which misses many cases where pointers are stored in the nested structs. We tackle this problem more comprehensively with a type-assisted approach that recursively visits global variables, as shown in Figure 5. The algorithm uses the same type syntax and notations as PtrSplit [39], where int represents an integer type, t_1* represents a pointer type, and struct type contains a list of types for each member. visit(id, t) is invoked on every global variable id with type t to generate initialization functions. They are then inserted before the program logic to sign the global pointers.

Issues with MIR instrumentation. We also resolved numerous compatibility issues with the MIR-based instrumentation. First, we found that the MIR-based instrumentation must be aware of register liveness. When a pointer inside a register is signed and then stored in memory, the signed in-register pointer may still be *live*. This side effect of pointer signing can cause following legitimate memory access with the in-register pointer to cause a fault. As a solution, our instrumentation inserts a `xpac` instruction to remove the PAC of the in-register value, immediately following the `pac` instruction if (1) the `pac`-ed register is used again in the function, or (2) the `pac`-ed register is passed to another function as an argument. Another issue that we found was the cases of PA instrumentation failing on load and store instructions that have the same register in source and destination operands (e.g., `ldr, x8, [x8]`). To resolve this, we transform these instructions to use distinct operands (e.g., `ldr, x8, [x9]`) instead.

	Syscall & API	Base. (ns)	Cap. (ns)	Ovh. (%)	Syscall & API	Base. (ns)	Cap. (ns)	Ovh. (%)
Syscall	socket	718.5	722.5	0.56	setsockopt	235.7	242.2	2.75
	bind	400.0	408.0	1.98	listen	351.2	361.5	2.91
	accept4	1236	1248	0.96	recvfrom	192.6	199.4	3.56
	openat	471.5	486.9	3.26	read	247.1	254.2	2.88
	pwrite64	320.6	328.2	2.35	pread64	231.8	239.0	3.06
	fstat	185.9	191.5	3.03	fcntl	148.1	155.8	5.20
	dup3	132.5	138.0	4.22	lseek	149.7	154.8	3.38
	mmap	130.0	130.0	0.01	**getpid**	118.6	–	–
API	capac_malloc	23.46	50.28	114.6	capac_free	14.7	37.4	154.4
	capac_enter	–	167.7	–	capac_exit	–	161.2	–
	delegate_ptr*	–	327.7	–	delegate_fd*	–	182.9	–
	limit_fd*	–	142.4	–				

* capac_ prefix is omitted

Table 2: The latency of CAPACITY system calls intervened by reference monitor and `libcapacity` APIs (Cap.), in comparision with the baseline (Base.). Latency of `getpid` is also measured for comparison.

7 EVALUATION

In this section, we first conduct a set of microbenchmarks on the reference monitor and `libcapacity` to illustrate the overhead induced by individual operations of CAPACITY (§7.1). We describe the adaptation of CAPACITY in protecting sensitive resources of three real-world applications in §7.2. Finally, we evaluate the performance of CAPACITY-adapted applications in §7.3 to show CAPACITY's overall impact on performance.

Evaluation method. We ensured the functional correctness of our implementation by using a QEMU [54] ARM virtual machine that supports both PA and MTE and conducted performance evaluations using the Apple Mac Mini with an M1 processor running Asahi Linux [41]. The M1 processor includes the PA extension; however, to our knowledge, there is no publicly available hardware with ARM MTE. The QEMU's MTE emulation is mature to a point where it is used for developing a MTE-based security feature for the Linux kernel to be used in the near future [16]. We follow the method from a previous work [43] that emulated the *worst-case* performance impact of MTE by using *MTE analogs*. We insert the MTE analogs to replace the tagging instructions in our tagged heap memory allocator and also modify CAPACITY's instrumentation to automatically insert the MTE analogs in places of LLVM's MTE intrinsics.

7.1 Microbenchmarks

We perform a set of isolated microbenchmarks on the syscalls intervened by the reference monitor, `libcapacity` API functions, and the instrumentation. The results are listed in Table 2.

System call latency with reference monitor. We measured the average latency of CAPACITY-protected syscall used in our evaluated applications that authenticate file paths and FDs, and compared them against un-protected syscalls. The overhead induced by CAPACITY includes the latency from syscall hooking and CAPACITY's reference authentication. Our results reported an average of 2.65% across the measured syscalls. This shows the efficiency of CAPACITY reference monitor design that fully utilizes hardware acceleration provided by PA, not to mention the simplicity of capability where

	Domain	Isolated Objects {Ref. types}	Domain Switches	Auth. Syscalls	$\mathcal{A}$	Executed PACs		Executed AUTs	
						$\texttt{PAC}_{\texttt{K}_{DB}}$	$\texttt{PAC}_{\texttt{K}_{DA}}$	$\texttt{AUT}_{\texttt{K}_{DB}}$	$\texttt{AUT}_{\texttt{K}_{DA}}$
NGINX + LibreSSL	Connection	Server socket {FD } Client connection socket {FD }	2	16	–	–	104	–	590
	Handshake	Server priv. key {PATH, FD, PTR } TLS session key {PTR } Client connection socket {FD ‡}	3	18	4	9	10.1K	13	134K
	Session	TLS session key {PTR ‡} Client connection socket {FD ‡}	7/8/10/13 /22/37/70†	14/16/20/28 /44/76/140†	3	43/49/61/85 /133/229/421†	818/852/917/1K /1.3K/1.8K/2.9K†	3.4K/6.5K/12K/25K /50K/99K/199K†	7.7K/10K/14K/24K /42K/79K/154K†
	Ambient	–	–	94/99/100/109 /121/145/193†	–	–	33.6K	–	164K
SSH	PrivKey	Client private key {PATH,FD,PTR }	1	72	55	18	6K	24	33.5K
	Ambient	–	–	37	–	–	14.7K	–	114K
wget	FileDownload	Downloaded file {PATH,FD,PTR }	3	325	1	129	3K	385	8.6K
	Ambient	–	–	131	–	–	703	–	1.6K

† Measurements from {16K,32K,64K,128K,256K,512K,1024K} HTTPS file size configurations ‡ Delegated references

$\mathcal{A}$ = Number of domain-private memory allocations

Table 3: Capacity domains and protected assets in evaluated applications. Columns 3-9 show runtime measurements of Capacity operations executed during a single iteration of {HTTPS file transfer (NGINX) / SSH handshake / wget file download}.

the reference alone is enough to make access control decisions without complex bookkeeping on domains and resources. Also, this overhead is imposed only on the Capacity-enabled processes.

`libcapacity` APIs latencies. The overhead of `capac_malloc()` and `capac_free()` originates from domain authentication and MTE-based memory tagging. Measuring the latency of domain private memory allocation and free on blocks of 1KB, `capac_malloc()` and `capac_free()` reported approximately 114.6% and 154.4% . `capac_enter` and `capac_exit`, `capac_delegate_ptr`, `capac_delegate_fd` and `capac_limit_fd` are simple `ioctl` calls to the in-kernel reference monitor, where only `capac_limit_fd` does not requires the key switching. Their latencies are on par with general syscalls. Besides a roundtrip to the kernel mode, we suspect their main source of overhead is from writing to the PA key registers.

nbench-byte benchmark. We measure the isolated overhead of Capacity's instrumentation on the *nbench-byte* [62] benchmark, which is also used by several previous works on PA-based defenses [27, 38]. nbench-byte consists of 11 CPU and memory subsystem benchmarks that allow us to measure the overhead of Capacity in general computations. Figure 6 shows the results of nbench-byte compiled with Capacity's instrumentation that enforces PA on on-save signing and on-load authentication of all in-program pointers. On average, Capacity's instrumentation incurs 11.37% performance degradation to the uninstrumented version.

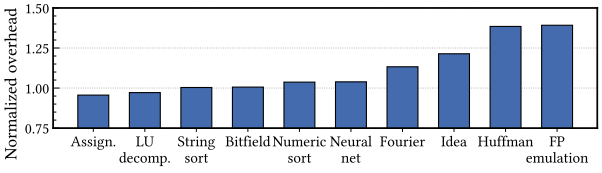


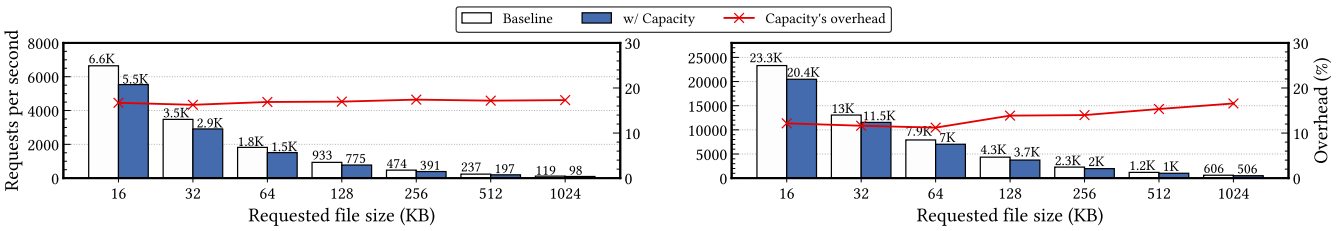
Figure 6: Overhead of instrumented nbench-byte, normalized to uninstrumented baseline and sorted by overhead.

7.2 Adapting Capacity to real-world programs

We exemplify Capacity's capability domains by adapting it to several open-source applications studied by previous isolation frameworks [8, 39, 63]. These applications include OpenSSH's `ssh` utility [22], `wget` file download utility [21], and NGINX webserver [17] compiled along the TLS library LibreSSL [53]. Representative code examples of our modifications can be found in the appendix of the extended version of this paper [15]. The applications are also compiled with the Capacity-instrumented *musl* libc [20] adapted to handle signed pointers in system call arguments. Table 3 summarizes the Capacity domains in protected applications and their protected assets.

OpenSSH's ssh (v9.3p1). We demonstrate Capacity's program secrets protection throughout their life-cycles with `ssh`. We compile OpenSSH to use its built-in crypto library and use Capacity to isolate the private key of `ssh`. We introduce a new domain, PrivKey, with exclusive access to the private key file and its in-memory buffer. This domain spans from when the private key is loaded from the file system until after the login operation succeeds. We then assign the static private key to the domain, allocate the private key buffer (`struct sshkey`) in private memory, annotate its pointers with `DOM_PRIV`, and finally wrap functions that access the private key within domain entry gates. These changes require about 50 LoC changes over 13 functions across 5 files.

`wget` (v1.21.2). We use Capacity to isolate the file received from the internet in `wget` to demonstrate least-privilege compartmentalization. We create a domain called FileDownload with exclusive access to the downloaded buffer and the output file path/FD. We



(a) Single-threaded **(b) Multi-threaded (8 Threads)**

Figure 7: Baseline webserver vs. Capacity-enabled webserver throughput benchmark performed with local ab client.

then modify the initialization of `wget` to assign the output file access to FileDownload, and use `cap_limit_fd` to revoke all other attributes from the file's `FD` except `CAP_WRITE`. This ensures that the domain has write-only access to the output `FD`[2]. The domain encapsulates the function `retrieve_url`, which fetches the requested file from a URL into a `capac_malloc`-allocated buffer before writing into the output file's `FD`. We also annotate pointers variables that point to the downloaded file in memory with `CAPAC_VAR`. The total modifications to `wget` is about 30 LoC.

NGINX (v1.23.0) + LibreSSL (v3.5.3). We use the NGINX webserver and its crypto library, LibreSSL, to demonstrate the multi-domain interaction of Capacity. We create three domains in the webserver, namely Handshake, Session, and Connection to isolate the server private key, session keys, and connection sockets with the least privilege principle in mind. Table 3 displays a summary of the webserver domains and their isolated resources. We modified 223 LoC in LibreSSL and 114 LoC in NGINX.

Figure 8 illustrates the main operations and their interaction with domain-private objects in domain Handshake and Session. Handshake, which consists of 5 NGINX and 2 LibreSSL functions endowed with exclusive rights to the server private key.

Session protects session keys by hosting 17 functions, mostly located inside LibreSSL's AES-GCM implementation. We generate a unique instance ID and use it as the modifier for each session since a session's resources are mostly temporary and can be isolated. As Handshake execution finishes, it produces a TLS session key, which is to be *delegated* to the corresponding Session instance using the API `capac_delegate_ptr` (Table 1). The domain-private stack is a noteworthy feature of Capacity used in Session. When a function performs complex cryptographic operations, we use `DOM_PRIV_STACK` instead of tracking the data flow of sensitive data bouncing around in the function's local variables.

[2]While `wget` already make the file handle write-only, Capacity enables more fine-grained in-process access control that supports read and write domains.

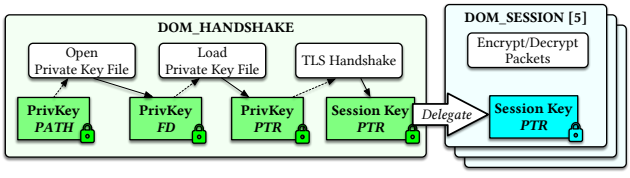


Figure 8: Capacity enables life-cycle private key protection and per-instance isolation of session keys in the NGINX webserver.

Program	Baseline (ms)	w/ Capacity (ms)	Overhead (%)
`ssh`	347.35	348.62	0.37
`wget`	17.12	18.14	5.95

Table 4: Capacity overheads on `ssh` and `wget`.

Finally, Connection manages the webserver's sockets. We wrap the functions that handle sockets inside its domain switches of Connection, such that when the server requests its *server socket* `FD` using `sys_socket`, the returned `FD` is domain-private. On client connection requests, Connection invokes `sys_accept4` with the server socket as the input and retrieves a *client* socket `FD`. Connection then delegate the returned *client* socket `FD` to Handshake, using `capac_delegate_fd`. After TLS handshaking, Handshake again delegates the socket `FD` to Session. Notably, `CAP_SOCKET` attribute is enabled by default when socket `FD`s are created, segregating socket and non-socket `FD`s in syscalls without the user's intervention.

7.3 Application performance benchmark

We perform performance benchmarks on the adapted applications. These applications run with all design components of Capacity explained thus far, including compile-time instrumentation, compatibility fixes, `libcapacity`, the kernel-level reference monitor, and backward and forward-edge CFI. To accurately profile the source of Capacity's performance overhead, we inserted probes into the reference monitor, `libcapacity`, and the instrumented program code to collect *runtime* execution count of Capacity operations during a single iteration of execution, as shown in Table 3.

`ssh` and `wget`. For `ssh`, we measure the average overhead of connecting to a local server 10,000 times and compare it against the unmodified version. We evaluate the protected `wget` by timing the average overhead in latency when requesting a 1MB file from a local http webserver 10,000 times compared to the baseline. Table 4 displays the Capacity-incured performance overheads in `ssh` and `wget`, which shows only minimal slowdown for both applications.

NGINX + LibreSSL. We evaluate the throughput of the Capacity-enabled NGINX webserver against its unmodified counterpart, which we shall call Baseline. We use the Apache benchmarking tool, `ab` [23] locally (i.e., no network latency) to measure

the number of requests processed per second, with varying file sizes, {16K, 32K ... 1M}, on also single-threaded and multi-threaded (8 threads) settings. 100,000 requests were performed, and the keep-alive option was used to avoid re-initializing the connections. The cipher suite negotiated between the client and the server was `ECDHE-RSA-AES256-GCM-SHA384`.

The webserver benchmark results are displayed in Figure 7, which shows the average performance degrades about 17% for the single-threaded server and 13.54% for the multi-threaded settings across all file size configurations. The latency measurements for Capacity operations from our microbenchmark (Table 2) allow us to reason that the performance overheads from the reference monitor and `libcapacity` would be very limited. The varied file sizes allow us to capture the characteristics of Capacity's performance overhead, which increases as the requested file sizes increase. Along with the runtime statistic collected in Table 3, it can be observed that Capacity's overhead protecting pointers and memory is proportional to the amount of computation. We can conclude that the performance overhead of Capacity would be predominantly from the increased number of instructions due to instrumentation.

8 SECURITY ANALYSIS & DISCUSSION

In this section, we use the security requirements outlined in §3.2 to analyze the security of Capacity. We show that these security requirements are achieved throughout the life cycles of program resources. We also discuss the possible mitigation for brute-forcing attacks at the end of the section.

Non-impersonable domains (R1). The domain entry explained in §4 renders invoking domain switches into arbitrary domains in unintended locations impossible. Also, our trusted domain ID fetching and in-register tag protection prevent the attacker from impersonating a domain by corrupting the currently activating domain ID spilled onto the stack or accessible globally.

Complete mediation (R2). Capacity's reference monitor ensures the use of `FD` and `PATH` in syscalls is always authenticated, and all userspace `FD`s are signed. Its whole-program instrumentation guarantees the signing and authentication of all ambient and domain-private pointers. Bypassing PA checks by jumping into non-entry locations of a function is infeasible with the presence of CFI. Hence, assuming no uninstrumented code, Capacity achieves complete mediation of all resource usage.

Non-reusable references (R3). Under Capacity's protection, a non-owner domain cannot access a protected file since the inode signature check will fail during file import. A domain-private `FD` leaked to another domain remains unusable thanks to our `FD` authentication scheme using the domain key. The reference monitor also prevents the OS from reusing `FD` numbers after one is deallocated. In addition, a domain compromised by the adversary cannot reuse in-memory pointers pointing to other domains' private memory. Due to domain key mismatch, the authentication would fail at `pacdb` sites. A pointer signed with $\mathbf{K}_{DB}$ would also fail to authenticate at pointer load sites where $\mathbf{K}_{DA}$ is used.

Non-forgable references (R4). It is impossible to forge an illegal path reference to a protected file object since path string references are resolved to their in-kernel data structures. Forging an `FD` in the userspace is also prevented using a secret (to userspace) modifier, which allows the kernel to remain the sole issuer of unforgeable and protected file descriptors.

With two methods, the adversary may forge a tagged and signed pointer to access domain-private memory. First, the adversary may overwrite an existing in-memory pointer through memory corruption. Such attacks are thwarted by the PA-based pointer authentication, whose primary purpose is to detect corruptions of in-memory pointers. Second, through code reuse attacks, the adversary may sign arbitrarily manipulated data (e.g., 64-bit data with the victim's domain tag and target sensitive memory address) using the available `pacda` and `pacdb` gadgets. For `pacdb` gadgets, Capacity PA contexts created by key switching force the attacker to use them within the target domain. We consider the isolated domains small enough so that it can be verified that the domain code is free of signing gadgets. However, this is a substantial and realistic threat when completely safe TCB is not guaranteed [35, 73], and a scanner to remove such gadgets could be utilized [73]. Given that no such convenient gadgets exist in Capacity domains, the adversary may still launch an extremely sophisticated attack by finding gadgets that (1) leak the sensitive memory address, (2) create a pointer and add the appropriate tag to it, (3) use `pacda` to create an ambient pointer that points to domain-private area, and (4) pass the signed pointer to an `autda` site. Capacity's instrumentation currently zeros out the tag of ambient pointers before signing them, rendering such code reuse attacks infeasible.

Brute-forcing atttacks. An attacker can exploit certain ideal situations, such as infinite thread spawning [5], to try different AC values until authentication succeeds. This is an inherent weakness with PA-based security. Such attacks can be overcome by placing thresholds on the number of crashed threads due to PA traps and systems call authentication failures [73].

9 RELATED WORK

We explain the previous works that are closely related to our work in terms of security principle (capability), use of hardware primitives (ARM PA and MTE), and objective (compartmentalization).

Capability-based resource access control. Capability-based access control has long been sought after due to its advantages in achieving the principle of least privilege [9, 13, 18, 31, 44, 45, 61, 65, 69, 70, 72]. Previous works explored capabilities in OS access control and memory safety. In the OS sphere, Capsicum [69] introduced *process-level* `FD` capabilities to UNIX and has been adapted by FreeBSD [50]. The SeL4 microkernel provides secure and fine-grained access to system resources through capability tokens [31]. Toward applying capabilities principles for memory safety, CHERI [72] introduced a CPU architecture with built-in support for capability-based memory access control. Subsequent works introduced extensions to the CHERI architecture with additional security features such as domain memory isolation [9, 12, 70]. Recently, Capstone [74] revised CHERI's base design to support revocable delegation and an extensible privilege hierarchy.

Unlike previous capability systems, Capacity's capabilities can be consistently applied to file and memory object references. Compared to Capsicum, Capacity does not require isolating program components into processes; its capability model seeks to achieve

least-privilege domains *within* a process. Moreover, Capacity's enforcement is much more lightweight thanks to hardware-assisted cryptographic authentication, Compared to capability architectures like CHERI, Capacity's design must address the security requirements of capability on existing hardware features, e.g., complete mediation of pointer uses since the processor does not enforce them automatically. Also, without hardware support, Capacity lacks fine-grained permissions that CHERI provides over pointers, e.g., per-object read/write/execute capabilities.

PA and MTE-based software defenses. Many PA-based runtime defense mechanisms have been proposed [1, 19, 27, 36–38, 55, 73] and deployed in commodity systems [1, 55]. MTE-supported bug detecting tools have been introduced [24, 40, 60] with improved performance compared to software-only approaches. Notably, PARTS [38] introduces a *Data Pointer Integrity (DPI)* policy that protects program-wide data and code pointers, but its instrumentation framework is not evaluated on real-world applications. PacTight [27] enforces *non-forgability*, *non-copyability*, *non-dangling* properties on only *sensitive pointers* with an intricate signing and authentication scheme. PTauth [19] introduces a PA-based use-after-free by authenticating pointers with their pointed-to object.

Capacity introduces a robust framework for whole-program instrumentation of pointer authentication. Moreover, previous PA-based systems rely on modifiers to establish different authentication contexts. On the other hand, Capacity establishes authentication contexts through its key switching mechanism and uses the modifier to isolate references between domain instances. Hence, different types of references can share a consistent authentication method in kernel and userspace without a complicated modifier assignment scheme. However, Capacity's use of $\mathbf{K}_{DB}$ for the domain key allows it to incorporate previous works on PA-based CFI into its design [1, 37, 38, 73].

Intra-process compartmentalization. Several works have leveraged x86 architecture's PKU to protect program domains and showed that PKU-based isolation significantly lower overhead compared to process-based and *Software Fault Isolation (SFI)*-based isolation [25, 30, 58, 63, 66]. Notably, ERIM [63] and Hodor [25] laid the groundwork for utilizing PKU for intra-process isolation through a meticulously designed call gate between domains. Capacity is motivated by the underexplored design space for ARM-based in-process compartmentalization and access control. Shred [8] proposed isolating the memory of in-process execution units with AArch32's Domain memory protection, but the feature has been removed in AArch64.

HAKC [43] is a framework for Linux device driver compartmentalization, also leveraging PA and MTE. Different from HAKC, Capacity's use of PA+MTE is specialized for memory and system resources isolation in the userspace. In particular, with pre-defined ACLs as PA modifiers, HAKC's pointer-use sites are tied to an access control policy, regardless of the calling context. On the other hand, Capacity's pointer authentication scheme checks if the reference is issued for the currently active domain, depending on the context (the effective $\mathbf{K}_{DB}$). This key difference brings implications that are pivotal in Capacity's design. First, it allows PA contexts to span across the user and kernel, enabling Capacity's authentication scheme without a complex user-kernel modifier sharing scheme. Second, functions can be called from multiple domains; the `AUT` authentication checks in the exact code location and uses different keys and therefore authenticates differently for each domain context.

Recently, there have been proposals for reference monitors better suited for in-process isolation [48, 58, 59, 66]. Jenny [58] proposed a secure and efficient syscall reference monitor that delegates access control to userspace. μSwitch [48] leverages implicit context switches to delay the kernel resource context switching until a syscall is performed to achieve better performance. While previous works on reference monitors require developers to write filtering rules to be applied to the monitor, Capacity's cryptographically-secured `FD` references are coherently built into the program's semantic, and their PA-assisted authentication is both transparent and efficient.

Isolation boundaries. Capacity currently only provides intra-procedural sensitivity annotation propagation. Automated whole-program and inter-procedural tracking of secret propagation is a field of its own, and previous works have proposed methods towards the objective [10, 28, 39, 47]. We expect that Capacity can incorporate such methods in the future, although it is currently out of the scope of this paper, which focuses on introducing a new isolation mechanism. On another note, a recent work [35] discussed the perils of artificially drawn in-process boundaries and their interfaces. Such attack surface must be eliminated or minimized through validation and interface narrowing when porting existing programs to use Capacity. If a program is redesigned or written from scratch with Capacity, then a conscious effort can be made to have a secure interface by design. The evaluated prototypes focus on showing the feasibility of Capacity as an isolation mechanism, and we regard the validation process to be an orthogonal issue.

10 CONCLUSION

This paper proposed a novel design called Capacity that enables the compartmentalization of in-process domains by employing hardware-accelerated and cryptographically-authenticated capabilities. We presented complete mediation and authentication schemes that satisfy the security requirements of capability systems throughout the life cycle of sensitive domain objects and addressed compatibility issues when adapting PA+MTE-based instrumentation for large programs. We evaluated Capacity through microbenchmarks and real-world applications, including an NGINX webserver prototype in which the important resources are protected in secure domains. The results show the efficacy of Capacity the in terms of performance with an average webserver throughput overhead of 17% for single-threaded and 13.54% for multi-threaded experiments.

ACKNOWLEDGMENTS

We deeply appreciate the anonymous reviewers for their constructive comments and feedback. This work was supported by grants funded by the Korean government: the National Research Foundation of Korea (NRF) grant (NRF-2022R1C1C1010494), Institute of Information & Communications Technology Planning & Evaluation (IITP) grants (No. 2022-0-00688, No. 2022-0-01199), and Korea Internet & Security Agency (KISA) grant (1781000009).

REFERENCES

[1] Apple. 2021. Apple Platform Security. https://manuals.info.apple.com/MANUALS/1000/MA1902/en_US/apple-platform-security-guide.pdf. Last accessed May 05 , 2021,.

[2] ARM Ltd. 2021. Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile. https://developer.arm.com/documentation/ddi0487/ga. Last accessed Nov 18 , 2021,.

[3] ARM Ltd. 2022. Armv8-M Architecture Reference Manual. https://developer.arm.com/documentation/ddi0553/bs. Last accessed May 15 , 2022,.

[4] ARM Ltd. 2023. ARMv8.5-A Memory Tagging Extension. https://developer.arm.com/-/media/ArmDeveloperCommunity/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf. Last accessed March 10 , 2022,.

[5] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking Blind. In *2014 IEEE Symposium on Security and Privacy*. 227–242. https://doi.org/10.1109/SP.2014.22

[6] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (San Francisco, California) *(NSDI'08)*. USENIX Association, Berkeley, CA, USA, 309–322.

[7] David Brumley and Dawn Song. 2004. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13* (San Diego, CA) *(SSYM'04)*. USENIX Association, Berkeley, CA, USA, 5–5.

[8] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. 2016. Shreds: Fine-Grained Execution Units with Private Memory. In *2016 IEEE Symposium on Security and Privacy (SP)*. 56–71.

[9] David Chisnall, Brooks Davis, Khilan Gudka, David Brazdil, Alexandre Joannou, Jonathan Woodruff, A. Theodore Markettos, J. Edward Maste, Robert Norton, Stacey Son, Michael Roe, Simon W. Moore, Peter G. Neumann, Ben Laurie, and Robert N.M. Watson. 2017. CHERI JNI: Sinking the Java Security Model into the C. *SIGARCH Comput. Archit. News* 45, 1 (apr 2017), 569–583. https://doi.org/10.1145/3093337.3037725

[10] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. 2018. ACES: Automatic Compartments for Embedded Systems. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 65–82.

[11] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. 2020. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 1409–1426.

[12] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. 2019. CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-Time Environment. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 379–393. https://doi.org/10.1145/3297858.3304042

[13] Jack B. Dennis and Earl C. Van Horn. 1966. Programming Semantics for Multiprogrammed Computations. *Commun. ACM* 9, 3 (March 1966), 143–155.

[14] Gregory J. Duck and Roland H. C. Yap. 2016. Heap Bounds Protection with Low Fat Pointers *(CC 2016)*. Association for Computing Machinery, New York, NY, USA, 132–142. https://doi.org/10.1145/2892208.2892212

[15] Kha Dinh Duy, Kyuwon Cho, Taehyun Noh, and Hojoon Lee. 2023. Capacity: Cryptographically-Enforced In-Process Capabilities for Modern ARM Architectures (Extended Version). https://doi.org/10.48550/arXiv.2309.11151

[16] Eklektix. 2022. kasan: add hardware tag-based mode for arm64. https://lwn.net/Articles/831624/. Last accessed Jan 14 , 2022,.

[17] Inc. F5 Networks. 2023. Advanced Load Balancer, Web Server, & Reverse Proxy. https://www.nginx.com. Last accessed Jan 14 , 2022,.

[18] R. S. Fabry. 1974. Capability-Based Addressing. *Commun. ACM* 17, 7 (jul 1974), 403–412. https://doi.org/10.1145/361011.361070

[19] Reza Mirzazade farkhani, Mansour Ahmadi, and Long Lu. 2021. PTAuth: Temporal Memory Safety via Robust Points-to Authentication. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association.

[20] Rich Felker. 2022. musl libc. https://musl.libc.org.

[21] Free Software Foundation. 2023. GNU Wget. https://www.gnu.org/software/wget. Last accessed Jan 14 , 2022,.

[22] OpenBSD Foundation. 2023. OpenSSH. https://www.openssh.com. Last accessed Jan 14 , 2022,.

[23] The Apache Software Foundation. 2022. ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html. Last accessed Jan 14 , 2022,.

[24] Vincenzo Frascino. 2020. Memory Tagging Extension (MTE) in AArch64 Linux. https://www.kernel.org/doc/html/latest/arm64/memory-tagging-extension.html. Last accessed March 10 , 2022,.

[25] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-process isolation for high-throughput data plane libraries. In *2019 USENIX Annual Technical Conference (USENIXATC 19)*. 489–504.

[26] Intel Corporation. 2021. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 325462-075US.

[27] Mohannad Ismail, Andrew Quach, Christopher Jelesnianski, Yeongjin Jang, and Changwoo Min. 2022. Tightly Seal Your Sensitive Pointers with PACTight. https://doi.org/10.48550/ARXIV.2203.15121

[28] X. Jin, X. Xiao, S. Jia, W. Gao, H. Zhang, D. Gu, S. Ma, Z. Qian, and J. Li. 2022. Annotating, Tracking, and Protecting Cryptographic Secrets with CryptoMPK. In *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 473–488. https://doi.org/10.1109/SP46214.2022.00028

[29] Douglas Kilpatrick. 2003. Privman: A Library for Partitioning Applications.. In *USENIX Annual Technical Conference, FREENIX Track* (2003-09-03). USENIX, 273–284.

[30] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2022. PKRU-Safe: Automatically Locking down the Heap between Safe and Unsafe Languages. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 132–148. https://doi.org/10.1145/3492321.3519582

[31] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. SeL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[32] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 147–163.

[33] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight, and Andre DeHon. 2013. Low-Fat Pointers: Compact Encoding and Efficient Gate-Level Implementation of Fat Pointers for Spatial Safety and Capability-Based Security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (Berlin, Germany) *(CCS '13)*. Association for Computing Machinery, New York, NY, USA, 721–732. https://doi.org/10.1145/2508859.2516713

[34] Hojoon Lee, Chihyun Song, and Brent Byunghoon Kang. 2018. Lord of the X86 Rings: A Portable User Mode Privilege Separation Architecture on X86. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1441–1454.

[35] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Yi Chen, Felipe Huici, Nathan Dautenhahn, and Pierre Olivier. 2023. Assessing the Impact of Interface Vulnerabilities in Compartmentalized Software. In *Proceedings 2023 Network and Distributed System Security Symposium. NDSS*.

[36] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. 2022. PACSan: Enforcing Memory Safety Based on ARM PA. https://doi.org/10.48550/ARXIV.2202.03950

[37] Hans Liljestrand, Thomas Nyman, Lachlan J. Gunn, Jan-Erik Ekberg, and N. Asokan. 2021. PACStack: an Authenticated Call Stack. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association.

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

[39] Shen Liu, Gang Tan, and Trent Jaeger. 2017. PtrSplit: Supporting General Pointers in Automatic Program Partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2359–2371. https://doi.org/10.1145/3133956.3134066

[40] Arm Ltd. 2022. -mmemtag-stack, -mno-memtag-stack. https://developer.arm.com/documentation/100067/0612/armclang-Command-line-Options/-mmemtag-stack---mno-memtag-stack. Last accessed Jan 14 , 2022,.

[41] Hector Martin, Alyssa Rosenzweig, Asahi Lina, Dougall Johnson, Sven Peter, Mark Kettenis, Martin Povišer, and Janne Grunau. 2022. Asahi Linux. https://asahilinux.org. Last accessed March 08 , 2022,.

[42] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. 2015. CCFI: Cryptographically Enforced Control Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 941–951.

[43] Derrick McKee, Yianni Giannaris, Carolina Ortega Perez, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. 2022. Preventing Kernel Hacks with HAKC. In *Proceedings 2022 Network and Distributed System Security Symposium. NDSS*, Vol. 22. 1–17.

[44] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. 2008. Caja: Safe active content in sanitized JavaScript. (June 7 2008).

[45] Myoung Jin Nam, Periklis Akritidis, and David J. Greaves. 2019. FRAMER: a tagged-pointer capability system with memory safety applications. In *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC 2019, San Juan, PR, USA, December 09-13, 2019*, David Balenson (Ed.). ACM, 612–626. https://doi.org/10.1145/3359789.3359799

[46] Oracle. 2022. Using Application Data Integrity (ADI). https://docs.oracle.com/cd/E37838_01/html/E61059/gqajs.html. Last accessed March 02 , 2022,.

[47] Tapti Palit, Jarin Firose Moon, Fabian Monrose, and Michalis Polychronakis. 2021. DynPTA: Combining Static and Dynamic Analysis for Practical Selective Data Protection. In *2021 IEEE Symposium on Security and Privacy (SP)*. 1919–1937. https://doi.org/10.1109/SP40001.2021.00082

[48] D. Peng, C. Liu, T. Palit, P. Fonseca, A. Vahldiek-Oberwagner, and M. Vij. 2023. μSwitch: Fast Kernel Context Isolation with Implicit Context Switches. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 2956–2973. https://doi.org/10.1109/SP46215.2023.10179284

[49] Massimiliano Poletto and Vivek Sarkar. 1999. Linear Scan Register Allocation. *ACM Trans. Program. Lang. Syst.* 21, 5 (sep 1999), 895–913. https://doi.org/10.1145/330249.330250

[50] FreeBSD Project. 2023. FreeBSD Manual Pages. https://www.freebsd.org/cgi/man.cgi?capsicum(4).

[51] LLVM Project. 2022. [AArch64] - return address signing. https://reviews.llvm.org/D49793. Last accessed May 05 , 2022,.

[52] LLVM Project. 2022. The LLVM Compiler Infrastructure. https://llvm.org. Last accessed Jan 14 , 2022,.

[53] OpenBSD Project. 2022. LibreSSL. https://www.libressl.org. Last accessed Jan 14 , 2022,.

[54] QEMU. 2022. QEMU: A generic and open source machine emulator and virtualizer. https://www.qemu.org. Last accessed March 08 , 2022,.

[55] QUALCOMM TECHNOLOGIES, INC. 2017. Pointer authentication on ARMv8.3. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf. Last accessed Nov 15 , 2021,.

[56] Red Hat. 2021. What is SELinux. https://www.redhat.com/en/topics/linux/what-is-selinux. Last accessed Apr 28 , 2021,.

[57] Nick Roessler, Lucas Atayde, Imani Palmer, Derrick McKee, Jai Pandey, Vasileios P Kemerlis, Mathias Payer, Adam Bates, Jonathan M Smith, Andre DeHon, et al. 2021. μSCOPE: A Methodology for Analyzing Least-Privilege Compartmentalization in Large Software Artifacts. In *24th International Symposium on Research in Attacks, Intrusions and Defenses*. 296–311.

[58] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. 2022. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 936–952. https://www.usenix.org/conference/usenixsecurity22/presentation/schrammel

[59] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. 2020. Donky: Domain Keys – Efficient In-Process Isolation for RISC-V and x86. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1677–1694.

[60] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. 2018. Memory Tagging and how it improves C/C++ memory safety. https://doi.org/10.48550/ARXIV.1802.09517

[61] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. 1999. EROS: A Fast Capability System. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles* (Charleston, South Carolina, USA) *(SOSP '99)*. Association for Computing Machinery, New York, NY, USA, 170–185.

[62] Uwe F. Mayer. 2017. Linux/Unix nbench. https://www.math.utah.edu/~mayer/linux/bmark.html. Last accessed March 08 , 2022,.

[63] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*. 1221–1238.

[64] Erik van der Kouwe, Taddeus Kroes, Chris Ouwehand, Herbert Bos, and Cristiano Giuffrida. 2018. Type-After-Type: Practical and Complete Type-Safe Memory Reuse. In *Proceedings of the 34th Annual Computer Security Applications Conference* (San Juan, PR, USA) *(ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 17–27. https://doi.org/10.1145/3274694.3274705

[65] Thorsten von Eicken, Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, Deyu Hu, and Dan Spoonhower. 1999. *J-Kernel: A Capability-Based Operating System for Java*. Springer Berlin Heidelberg, Berlin, Heidelberg, 369–393. https://doi.org/10.1007/3-540-48749-2_17

[66] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. 2022. You Shall Not (by)Pass! Practical, Secure, and Fast PKU-Based Sandboxing. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 266–282. https://doi.org/10.1145/3492321.3519560

[67] Xiaoguang Wang, SengMing Yeoh, Pierre Olivier, and Binoy Ravindran. 2020. Secure and Efficient In-Process Monitor (and Library) Protection with Intel MPK. In *Proceedings of the 13th European Workshop on Systems Security* (Heraklion, Greece) *(EuroSec '20)*. Association for Computing Machinery, New York, NY, USA, 7–12. https://doi.org/10.1145/3380786.3391398

[68] Zhe Wang, Chenggang Wu, Mengyao Xie, Yinqian Zhang, Kangjie Lu, Xiaofeng Zhang, Yuanming Lai, Yan Kang, and Min Yang. 2020. Seimi: Efficient and secure smap-enabled intra-process memory isolation. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 592–607.

[69] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. 2010. Capsicum: Practical Capabilities for UNIX. In *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*. USENIX Association, 29–46.

[70] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav H. Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert M. Norton, Michael Roe, Stacey D. Son, and Munraj Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 20–37.

[71] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. 2019. TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V. In *Proceedings 2019 - Network and Distributed System Security Symposium (NDSS)*. Internet Society. https://doi.org/10.14722/ndss.2019.23068

[72] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert M. Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. IEEE Computer Society, 457–468.

[73] Sungbae Yoo, Jinbum Park, Seolheui Kim, Yeji Kim, and Taesoo Kim. 2021. In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication. https://doi.org/10.48550/ARXIV.2112.07213

[74] Jason Zhijingcheng Yu, Conrad Watt, Aditya Badole, Trevor E. Carlson, and Prateek Saxena. 2023. Capstone: A Capability-based Foundation for Trustless Secure Memory Access. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 787–804. https://www.usenix.org/conference/usenixsecurity23/presentation/yu-jason

[75] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. 2008. Hardware Enforcement of Application Security Policies Using Tagged Memory. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, 225–240.