Information Flow Tracking for Heterogeneous Compartmentalized Software

Zahra Tarkhani
Microsoft
Cambridge, UK
ztarkhani@microsoft.com

Anil Madhavapeddy
University of Cambridge
Cambridge, UK
avsm2@cam.ac.uk

Abstract

We are now seeing increased hardware support for improving the security and performance of privilege separation and compartmentalization techniques. Today, developers can benefit from multiple compartmentalization mechanisms such as process-based sandboxes, trusted execution environments (TEEs)/enclaves, and even intra-address space compartments (i.e., intra-process or intra-enclave). We dub such a computing model a "hetero-compartment" environment and observe that existing system stacks still assume single-compartment models (i.e., user space processes), leading to limitations in using, integrating, and monitoring heterogeneous compartments from a security and performance perspective.

We introduce Deluminator, a set of OS abstractions and a userspace framework to enable extensible and fine-grained information flow tracking in hetero-compartment environments. Deluminator allows developers to securely use and combine compartments, define security policies over shared system resources, and audit policy violations and perform digital forensics across heterogeneous compartments. We implemented Deluminator on Linux-based ARM and x86-64 platforms, which supports diverse compartment types ranging from processes, SGX enclaves, TrustZone Trusted Apps (TAs), and intra-address space compartments. Our evaluation shows that our kernel and hardware-assisted approach results in a reasonable overhead (on average 7-29%) that makes it suitable for real-world applications.

CCS Concepts

• **Security and privacy → Information flow control**; **Operating systems security**; **Trusted computing**.

Keywords

Information Flow Tracking, Compartmentalization, TEEs, enclaves

ACM Reference Format:
Zahra Tarkhani and Anil Madhavapeddy. 2023. Information Flow Tracking for Heterogeneous Compartmentalized Software. In *The 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '23), October 16–18, 2023, Hong Kong, China.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3607199.3607235

RAID '23, October 16–18, 2023, Hong Kong, China

ACM ISBN 979-8-4007-0765-0/23/10…$15.00
https://doi.org/10.1145/3607199.3607235

1 Introduction

Modern applications face diverse attack vectors from threats within and across dependencies and system abstractions. Compartmentalization is a powerful protection technique that splits an application into isolated components (or trust domains), each with well-defined communication channels [4, 13, 46, 65, 103]. Many compartmentalization mechanisms have emerged offering different levels of isolation and enforcement for different usecases, such as isolating unsafe libraries, subprocesses, or code blocks [14, 36, 48, 55, 64, 69, 83, 98, 102]. These mechanisms show that compartmentalization significantly reduces the attack surface and impact of unknown vulnerabilities, and is often the best defense against the ever growing and complex security threats in modern computing environments.

That is why to enhance the security and performance of compartmentalization techniques, many recent efforts have focused on providing hardware-assisted isolation and privilege separation mechanisms that can *(i)* reduce TCB, *(ii)* provide stronger or finer-grained isolation, or *(iii)* achieve better performance than software-only approaches [9]. Such progress has led to significant security and performance improvements and more practical compartmentalization techniques [4, 6, 13, 24, 33, 38, 39, 46, 53, 65, 93, 97, 103]. Various forms of Trusted Execution Environments (TEEs)/enclaves have been introduced on almost all modern hardware architectures to protect security-critical coded and data against privileged system software such as the host OS or hypervisor [2, 5, 28, 38, 71, 78, 86].

This allows application developers to simultaneously use compartments with different characteristics, such as traditional process-based sandboxes [13], TEE/enclave-assisted compartments [2, 5, 28, 38], and intra-address space sandboxing (intra-process [18, 57, 98, 104] or within an enclave [62, 73, 87]). Each compartment type has a different threat model, privileges, and isolation and sharing mechanisms. We dub this a *hetero-compartment environment* which runs multiple compartment types simultaneously; each with a different threat model, privileges, and isolation and sharing mechanism. Hetero-compartments allow developers to combine different compartmentalization mechanisms and design secure software that meets application-specific performance, efficiency, or compatibility requirements. However, they also introduce challenges in practice since there are many vulnerabilities *within* and *across* these heterogeneous isolation boundaries that are difficult to detect, debug, and prevent. It is especially hard to reason about whole-system security with existing tools and multiple heterogeneous compartments.

Consider existing TEE/enclave-assisted partitioning frameworks which enable developers to refactor and port code to isolated compartments. Refactoring code without rigorous security analysis can introduce *new* attack vectors [45, 55, 58, 99], overly privileged compartments with potential vulnerabilities, and a high cost in

reengineering [7, 82, 96]. Previous studies have demonstrated attacks on enclave/TEE applications due to vulnerabilities within an overprivileged enclave [45, 91, 99], insecure resource sharing between enclaves and processes (just two distinct compartments) [45, 58, 99, 106], or compartment interface vulnerabilities [55]. It is extremely challenging for developers to securely migrate a compartmentalized application to another platform; e.g. from SGX enclaves to TrustZone TAs. Despite surface similarities, TEEs/enclaves are highly architecture dependent and TEEs like SGX or TrustZone have different interfaces and confidentiality, integrity, and freshness guarantees (they do not usually guarantee availability). Hence migrating enclave-assisted applications will likely widen the attack surface [58], especially without a security analysis tool to enforce the same security policies across heterogeneous compartments.

A key blocker for developers to secure hetero-compartment environments and leverage hardware-assisted compartmentalization is the lack of tools to reason about security threats within and across these compartments. We present Deluminator– a unified set of OS abstractions designed for a hetero-compartment computing model that tracks sensitive data and control flows across arbitrary compartments. Deluminator offers an extensible userspace framework to trace hetero-compartment's data and system objects (address space, threads, files, IPC/RPC, etc.) based on desired confidentiality and integrity policies across different trust boundaries. The goal is to provide developers with building blocks to audit and analyse applications in hetero-compartment environments. Although Deluminator can be useful in mitigating different classes of attacks, it is not intended as an isolation or enforcement mechanism (which requires a stronger threat model to fit the TEE/enclave's threat model [92]). It instead finds and analyzes information flow violations across compartments. As an OS-assisted framework, Deluminator does not force developers to use a specific programming language and can therefore migrate existing applications. To demonstrate its extensibility, we use Deluminator on three compartment types: process-based sandboxes, SGX enclaves, and TrustZone.

Achieving our goal requires overcoming several challenges. Heterogenous compartments have different granularity and security models, and so preventing attacks within and across them requires a principled specification of *mutually distrustful* security policies. Developers need to efficiently trace dataflows on fine-grained objects between compartments located in the same or separate address spaces, since some compartments are managed by a different kernel than the host OS (such as in TrustZone). The framework needs to be easy to integrate, deploy, and extend to more compartment types.

Currently developers need to manually investigate potential security threats and combine fuzzing or analysis tools – an error-prone and unscalable approach. We therefore contribute Deluminator as a set of OS abstractions and security primitives for tracking sensitive data and control flows in hetero-compartment environments. Deluminator provides the first extensible and fine-grained tracing capability for hetero-compartment environments, and supports an essential set of system objects (including address spaces and threads) and a POSIX-friendly API. We demonstrate an efficient implementation on commodity TrustZone- and SGX-enabled hardware, with reasonable modifications to the Linux kernel and existing TEE/enclave stacks, and conduct a thorough security and performance evaluation.

2 Motivation & Assumptions

We next discuss the attack vectors in hetero-compartment environments that Deluminator can detect, examine, and audit (§2.1), followed by our threat model (§2.2) and assumptions (§2.3).

2.1 Attacks on Compartmentalized Software

Unsafe sharing and interactions. Most compartmentalized applications follow a one-way trust model in which some compartments are fully trusted, but still require the sharing of data and resources with untrusted compartments or the untrusted host. An enclave-assisted application may use untrusted memory to share results across multiple enclaves, or use untrusted storage to persist intermediate results, or an untrusted network/IPC to for IO. There are a wide range of compartment interface vulnerabilities [45, 55] that can be exploited to extract secrets, take control of compartments, or attack the host/TEE [12, 54, 58]. For example, the Boomerang vulnerability [58] is a confused deputy attack on TEE-assisted applications whereby a userspace compartment can leverage its TA via untrusted shared memory to manipulate memory regions that it does not own, including from the host Linux kernel.

The attack surface is harder to define in hetero-compartment environments, since each compartment type can have its own security model and view of system resources – a much bigger semantic difference between compartments. Figure 1 illustrates six compartment types (i.e., user space process, SGX enclave, TrustZone TA, in-process, in-enclave, and in-TA) and shows how vulnerabilities can be introduced and propagated in the cross-compartment boundaries within different or the same address spaces. Detecting and analysing such a large attack surface is one of the primary obstacles to securing the ever-growing hetero-compartment environment. Combining per-attack security patches only addresses a few attack vectors and is not a principled approach to dealing with future security threats and unknown vulnerabilities.

Malicious TA/enclave A compromised or malicious enclave can collect sensitive data [59, 84] and leak them using files, network sockets, or pipes. Attackers can launch horizontal privilege escalation (HPE) attacks [91] to compromise another application process via a shared misbehaving TA/enclave (e.g., a compromised third-party in-enclave cryptography service). When the host OS/-TEE framework supports Deluminator, developers can investigate and monitor these attack vectors during development or at runtime, something that is not possible with existing tools. However, side channel attacks [85, 107] are out of scope (§2.2).

Intra-address space threats Intra-address space threats are an increasingly exploited attack vector in compartmentalized applications. Malicious third-party libraries in the same process (e.g., CVE-2021-3711, CVE-2019-15295, CVE-2016-6309) often lead to information leakage. New attacks have been shown recently within an enclave address space, due to large in-enclave TCBs or within process-to-enclave shared memory [34, 62, 97]. Despite diverse in-enclave isolation techniques such as language runtimes (e.g., Civet Java extension [97]) or via Memory Protection Keys (MPK)-based sandboxing [62], fully exploring cross-compartment attacks requires enabling information flow tracking within same-address-space objects. In-enclave isolation is not always possible due to the platform, language, or performance limitations. Conffuzz [55] and

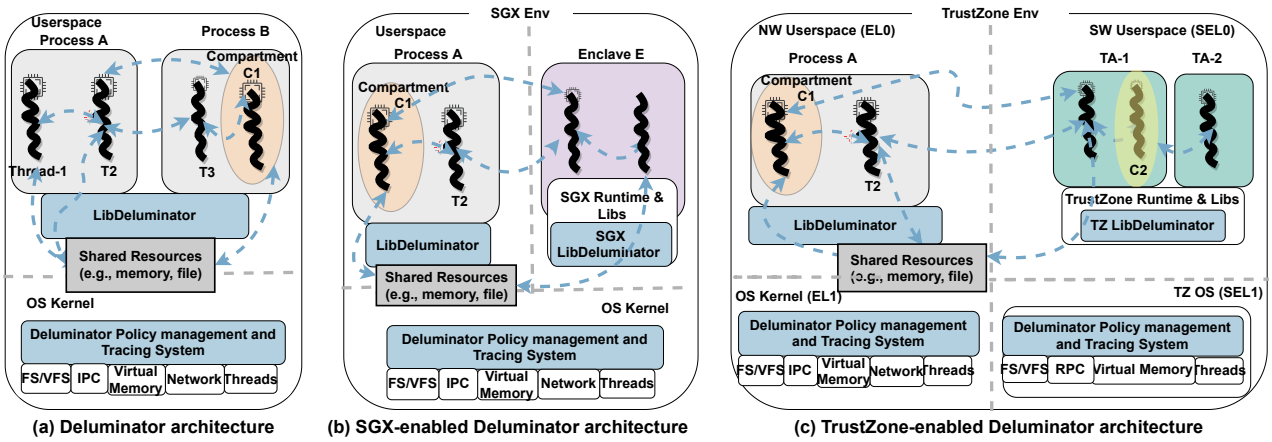

Figure 1: Fine-grained and extensible information flow tracking within heterogeneous compartments by using Deluminator.

SGXRacer [16] enable fuzzing cross-compartment vulnerabilities with a focus on memory and threading issues, which is aligned with our goal. However, we argue it is time for commodity OSs to provide a principled approach for also discovering non-trivial policy and logic issues in hetero-compartment environments, and not just focus on cross-compartment API sensitization. Such OS abstractions can be used by fuzzing techniques as well.

Concurrency threats Attackers can also launch malicious threads from inside the host application to exploit synchronisation vulnerabilities such as TOCTTOU [105, 106] or concurrency attacks [45] on shared resources between compartments. This greatly limits the secure compartmentalisation within a multithreaded application. Deluminator allows labeling threads and per-thread resources which facilitates tracking and monitoring such attacks.

2.2 Threat Model

We model system resources (memory, threads, RPCs) as *objects* to define a unified view of resources across different compartment boundaries. We assume that each compartment C contains a private set of execution threads T_C and other arbitrary system objects O_C such that $C = \{T_C \cup O_C\}$. Each compartment has its own security policy, an enforcement mechanism, and interfaces to the untrusted world or other compartments. We assume that the compartmentalization framework has its own TCB and threat model and that compartmentalization frameworks are responsible for enforcing the security model and the isolation over compartment resources, as well as cross-compartment sharing or data exchange mechanisms.

Since Deluminator is not an enforcement mechanism, its information flow tracking mechanism does not interfere with the execution and isolation of compartments. It allows developers to define mutually distrustful policies to track but does not enforce those mutually distrustful policies. Deluminator considers commodity systems running software from numerous independent vendors. Deluminator also requires modifications to commodity systems stack like the Linux kernel which is outside the TCB for TEE/enclave stacks, as well as modifications to TEE frameworks such as the TrustZone kernel which slightly increases their TCB size. In future re-writing Deluminator systems components in safe languages like Rust would be ideal, particularly considering the increasing progress in Rust-based Linux kernel extensions [41].

Deluminator assumes application developers correctly specify their information flow policies through the Deluminator userspace library and API. Similar to most security analysis tools, Deluminator works best when application developers have basic knowledge of potential attack vectors and security-sensitive parts of their application. Additionally, Deluminator is not designed to target side channel attacks [85, 107] which is more in the scope of hardware-based information flow tracking [27].

2.3 System Assumptions

A key goal of Deluminator is to support commodity hardware, OSes and existing compartmentalization frameworks. The Deluminator implementation depends on the Linux kernel (versions later than 4.19) for non-TEE-based compartmentalization, and on the TrustZone OPTEE-OS (on ARM Cortex-A) or SGX SDK (on x86-64) for TEE-assisted compartmentalization. Supporting Deluminator on other architectures such as RISC-V requires adding support to a RISC-V TEE framework such as Keystone [53]. Deluminator system components are written in C, since we extended existing system software. However, developers can write their own userspace tools or language extensions in any programming language as it does not assume any specific language dependency. Deluminator is compatible with existing Linux kernel security and auditing subsystems, which makes it useful with sandboxing tools such as seccomp, LandLock [79] or other LSM-based compartmentalization frameworks.

The current implementation introduces new system calls to the Linux kernel (§3) to make prototyping easier. For eventual upstreaming, most of these syscalls can be merged with existing ones; e.g. `d_mmap/d_munmap`, `d_clone`, `d_wait` and `d_execv` can all be merged with their corresponding syscalls. Deluminator labelling options lie `open`, `socket`, and `pipe` already extend existing syscalls. The Linux kernel auditing subsystem [1] design can also Deluminator's remaining syscalls further; for instance all of `d_add`, `d_create`, `d_remove` and `d_cleanup` can be merged with the userspace kernel tools similarly to the `kauditd` approach.

[1] https://github.com/linux-audit

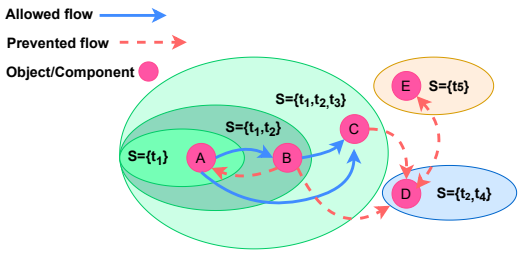

Figure 2: Secrecy information flows example

3 Deluminator Overview

Deluminator contains a tracing system and policy management module running inside the OS kernel or TEE/enclave system stack, depending on the target execution platform (Figure 1). It also includes a userspace library (LibDeluminator) that exposes the Deluminator API via new syscalls to developers and analyzers for specifying tracing policies on compartments and system objects. Deluminator policy management captures userspace specifications from LibDeluminator and converts them to associated policies based on its underlying tracing model. The Deluminator tracing system (DTS) is designed to efficiently trace the execution of a target compartment without any manual interaction. The tracing searches for policy violations and unsafe information flows on a rich set of system objects including tagged address spaces, threads, files, sockets, pipes, and IPC/RPCs. The tracing mechanism is extensible and customizable to add or remove different objects, and is customized to support tracing within TEE kernel objects alongside the Linux kernel (Figure 1.c). We next explain each of the Deluminator components in detail.

3.1 Tracing & Labeling Model

To systematically model the attacks in our threat model (§2.2), we need a tracing mechanism that *simultaneously* solves two issues in existing solutions: coarse granularity and inextensibility. The Deluminator model supports tracing both intra- and inter-address-space compartments by proposing a customizable tracing precision over fine-grained system objects. It is designed for hetero-compartment environments, which requires supporting mutually distrustful policies and security models between the compartments.

We model each system resource (threads, address spaces, RPCs, files, pipes, and sockets) as an *object* to create a unified view of resources across trust boundaries. We selected these objects after analyzing real-world applications (§5.3), existing vulnerabilities, and their interfaces to find the most frequently used objects. We then map each compartment as a set of these system objects and at least one execution thread object. The Deluminator API enables developers and analyzers to define per-compartment policies over objects, covering integrity, confidentiality, and sharing. Deluminator then traces and reports any violations given such a policy.

Traditionally, many variants of information flow control (IFC) are used to model or monitor vulnerabilities and enforce system security. At their core, IFC models are a set of simple rules over selective entities or objects. IFC systems check dataflows based on a partial ordering of security contexts (usually a lattice) that defines where data can flow legitimately. This is enforced by assigning unique tags and labels (a set of tags) to the monitored entities. In such systems each security principal (i.e., execution threads in our system) can define per-object security policies by assigning secrecy and integrity labels. All that is needed to define a security policy in this model is to describe the desired information flows via labels. Under the simplest model (Figure 2), object B can receive information from A since $A \subseteq B$, while B (and also C, D, E) cannot transmit to or access A since $B \not\subseteq A$. Similar rules are applied for integrity flows. Information flow policies can be controlled either explicitly or implicitly. With explicit flows, the security principal makes data flow from one level to another by requesting and then assigning labels. In implicit flows, the labelling changes happen in the background as the data flows through the system. For instance, when an untainted process accesses a tainted file the process becomes tainted automatically. Implicit labeling can thus silently lead to label state explosions.

There are diverse IFC and labeling models depending on the programming model, enforcement granularity, needed policies, performance, and scalability requirements. The implementation of complex security policies in hetero-compartment environments via IFC in a practical and expressive way is challenging. A well-known example is dynamic information flow tracking (DIFT), a technique that analyzes the information flowing within the execution of a program [20, 26]. Since we target kernel system objects and require tracing mutually-distrustful compartment policies, we choose a tracing model based on core decentralized IFC (DIFC) concepts [66]. This model is known to be well suited for monitoring large systems and complicated dynamic mutually-distrustful policies [19, 42, 51, 67, 68, 77], and is efficient if provided with the right support from the OS kernel [109]. Previous work has demonstrated integration of DIFC into the Linux kernel [77] and Android [68].[2]

There are several labeling models to implement DIFC systems. DC (Disjunction Category) labels [90] – inspired by capability-based systems – provides a simpler and more expressive labeling compared to the earlier Myers and Liskov decentralized label model (DLM) [67]. This is suitable for userspace tools or languages – as used in Haskell programming language[101] and Hails [29] – to enable DIFC for monitoring privacy issues in web applications. The Deluminator userspace API hides details of its labeling model and so can support tracing for any labeling model that supports DIFC. In our implementation, we followed the Flume [51] labeling model due to its simplicity for OS abstractions, but there is no barrier with replacing it with DC labels.

In Deluminator, every principal thread can enable or restrict a flow by adding or removing tags from labels if they have the *capability*—a right to perform an operation—to do so. It defines two capabilities per tag, t^+ and t^-, that enable adding or removing t to a label respectively. Capabilities are stored in capability list of each principal $C = C^+ \cup C^-$ ($t^+ \in C^+$ and $t^- \in C^-$). So if $C_p = \{t^+\}$, principal p has the capability to add t to its secrecy label for accessing or reading the object that is tagged with t, but cannot remove it from its label since $t^- \notin C_p$. A principal that owns both capabilities for t and can completely control how t appears in its labels. The set $D_p = \{t | t^+ \in C_p \wedge t^- \in C_p\}$ represents all tags for

[2] Details and proofs for decentralized-IFC models are in previous work [19, 42, 51, 67, 77]

which p has both t^+ and t^- capabilities (full ownership). Adding a tag to a secrecy label S and removing a tag from an integrity label I are safe operations since the principal only tightens the constraints. However, declassification (removing a tag from a secrecy label) and endorsement (adding a tag to an integrity label) are unsafe operations. For example, when $t \in S_p \wedge t^- \notin C_p$, declassification of tagged data with t is not allowed, and so it must not be exported to untrusted sources.

3.2 LibDeluminator

To detect and analyze unsafe dataflows, Deluminator provides new APIs for defining per-compartment security policies. Each policy describes a compartment—adding the objects that need to be monitored—as well as assigning per-object secrecy/integrity labels. The policy can be defined per thread of execution inside a compartment, which can be userspace processes, SGX enclaves, or TrustZone TAs, without dealing with the details of the underlying tracing system and DIFC concepts (Table 1). As with language-based tracing systems which require code annotations, Deluminator requires minimal code modifications as the framework is programming language agnostic. This enables extra annotations to existing applications and not via modifications to third-party libraries.

Our userspace library enables seamless tracing of compartment objects with few code changes. After switching to the tracing mode (by calling `d_enable`), any execution thread inside a compartment can assign/remove secrecy or integrity policies over single or multiple objects (by calling `d_add, d_remove` or via a config file), which leads to automatically initializing and labeling the objects. LibDeluminator is a standalone library which we ported to TrustZone TAs and SGX enclaves for enabling cross-compartment tracing. Developers only need to specify if the tracing is looking for confidentiality or integrity violations by passing the secrecy label `SLABEL` or the integrity label `ILABEL` flags to the API.

For example, consider a TOCTOU attack, in which there is a window of opportunity between when a program checks a file (e.g., `fname` in the following listing) and when it operates on that file. In that window, an attacker can launch a thread to replace `fname` with, for example, a symlink to `/etc/passwd`, and the operation meant to perform on `fname` happens to an important system file instead. But when the owner compartment creates a tagged `fname` using LibDeluminator, no other thread can do any operations on `fname` without being detected by Deluminator. So Deluminator detects it when the attacker thread accesses `fname` or replaces it via symlink. The following listing shows how the compartment thread can add `fname` to its secrecy label (L4) to enable monitoring over it.

```
1  d_enable();//enable tracing in this compartment
2  //add fname to start tracing secrecy violations
3  //use ILABEL for integrity violations
4  open(fname,O_CREAT|O_SLABEL); //create a labeled file
5  // or use d_add(fname,FILE,SLABEL) to label an existing one
6  if(!access(fname,W_OK)) {
7  //When an attacker thread launches TOCTOU e.g., by using
       symlink to redirect fname, Deluminator can detect it
8  f = fopen(fname,"w+");
9  operate(f);
10 ...}
11 else{...}
```

Listing 1: Simple use of Deluminator for detecting symlink-based TOCTOU vulnerability on a file.

API	Description
New system calls	
d_enable()->c	enable tracing for current c
d_add(obj, type, policy)->ret	start tracing object from c
d_remove(obj, type, policy)->ret	stop tracing obj from c
d_cleanup(c)->ret	cleanup the tracing
d_create(hw_mode)> mobj	create a memory object
d_clone(&fn, policy, ...)->tid	create a new labeled thread
d_execv(mobj, bin...)	trace binary execution
d_wait(&policy)	wait for labeled threads
Address space tracing calls	
d_malloc(mobj, size)	allocation from mobj
d_free(mobj, size)	deallocation from mobj
d_mprotect(mobj, ...)	change permissions of mobj
d_mmap/munmap(mobj, ...)	change layout of mobj
memcpy, memcmp, memset, etc	other memory operations
Modified system calls	
open(*pathname, SLABEL|ILABEL,...)	create labeled file
socket(domain, SLABEL|ILABEL,...)	create labeled socket
pipe(pipefd, SLABEL|ILABEL,...)	create labeled pipe

Table 1: Simplified LibDeluminator interface, where c represents a compartment.

Address space objects. To analyze memory vulnerabilities, Deluminator supports tracing virtual memory objects in the form of tagged blocks of contiguous address space objects. In Section 4.2, we describe our modifications to the host OS and TEE kernel/runtime for efficiently tagging virtual memory objects and tracing information flows over them. Our API allows developers to label a memory object for tracing any unauthorized access or changes in its layout or permissions to another thread. For instance, in the code below, both the host process and enclave compartments can label their sensitive memory objects to monitor heap leakage when interacting with each other (L29 and L7). The enclave thread explicitly assigns policy (e.g., RW only for the enclave thread) over its memory objects. LibDeluminator also provides `malloc` style memory management API over the labeled memory objects.

```
1  //-------------Enclave compartment side--------------
2  void ecall_heap_leak(struct eData* data){
3  //....initialization
4   d_enable();//enable tracing
5   int mobj=d_create(MMU_MODE, MEMDOM_READ|MEMDOM_WRITE);
6  //change malloc -> d_malloc
7   data->msg = (char*) d_malloc(mobj,strlen(temp));
8   d_memcpy(mobj, data->msg, temp, strlen(temp));
9   data->len = strlen(temp);
10  data->left = strlen(temp);
11  while(data->left > 0){
12  buf = data->msg + data->len - data->left;
13  ocall_write_out(&ret, buf, data->left);
14  if(ret <= 0) return;}}
15 //-------------App/process compartment side--------------
16 void ocall_write_out(char *buf, int left)
17 { printf("%c\n", buf[0]);
18   printf("%d\n", left-1);}
19 int main()
20 {//init enclave ...../
21  d_enable();//enable tracing
22  int a_mobj=d_create(MMU_MODE,MEMDOM_READ|MEMDOM_WRITE);
23  struct eData* data = (struct eData*) d_malloc(a_mobj,sizeof(
        struct eData));
24  ret = ecall_heap_leak(eid, data)
25   //cleanup enclave......//}
```

Listing 2: Pseudocode of using Deluminator to detect enclave heap leakage

3.3 Policy Management & Tracing System (PMTS)

The PMTS implements the Deluminator tracing model—including all the information flow tracking rules—and enforces the userspace specification from LibDeluminator into the required system objects. Although the OS kernel is the best place to check dataflows within fine-grained system objects efficiently, it is also a huge codebase that exposes over 300 syscalls. Tracing and monitoring security policies over arbitrary objects within such a complex stack is challenging.

A naive solution leads to label explosion, excessive modification to the kernel, and huge performance overhead. This is difficult due to Deluminator supporting intra-address space policies that existing OSs typically do not support abstractions for. To resolve this, Deluminator PMTS introduces a unified virtual address space-based object extension for the Linux kernel and TrustZone OS over their virtual memory abstraction, as we describe later in Section 4.2.

4 Implementation

We next explain the details of the Deluminator implementation, beginning with a technical background (§4.1) and then a description of the changes to the Linux kernel (§4.2), OPTEE-OS (§4.3), and the SGX system stack (§4.4).

4.1 Technical Background

4.1.1 ARM VMSA & Security Extensions The ARM virtual memory system architecture (VMSA) is tightly integrated with the security extensions, the multiprocessing extensions, the Large Physical Address Extension (LPAE), and the virtualization extensions. VMSA implements MMUs that control address translation, access permissions, and memory attribute determination and checking for memory accesses. The extended VMSAv7/v8 provides multiple stages of memory system control for operation in both the secure state (e.g. EL1&0 stage 1 MMU) and in the non-secure state (e.g., EL2 stage 1 MMU, EL1&0 stage 1 MMU, and EL1&0 stage 2 MMU). VMSAv8.5 adds more MMUs for additional isolation in the secure world. Each MMU uses a set of address translations and associated memory properties held in TLBs. If an implementation does not include the security extensions, it has only a single security state, with a single MMU with controls equivalent to the secure state MMU controls. A similar argument holds for when an implementation does not include the virtualization extensions. In ARM Cortex processors, the security extensions or TrustZone are implemented by splitting each physical core into two virtual CPUs. Depending on the value of the Non-Secure (NS) bit, hardware resources (e.g. DRAM or peripherals) may run either in the secure world (SW) or the normal world (NW), where each run a separate software stack. TrustZone's one-way security model isolates SW by restricting NW to only its own resources; however, code running in the SW can access memory and I/O assigned for both worlds.

Each world has its own user-mode (EL0/SEL0) and kernel-mode (EL1/SEL1). The control transition between the two worlds happens through a Secure Monitor Call (SMC) instruction that invokes the secure monitor code, which runs at the highest privilege level (EL3). Although the TZ APIs are not uniform across different devices, the popular implementations (e.g., OPTEE, Kinibi, Teegris or QSEE) follow the GlobalPlatform [30] TEE specification. It requires TAs (trusted applications) to run in the secure world userspace SEL0 and communicate with TZ OS kernel, which runs in SEL1, via SVC supervisor calls (like processes). There are also privileged TAs like Trusted Drivers (TDs) in Kinibi [11] or Pseudo TA (PTAs) in OPTEE [3, 95] that have access to a richer set of operations and SVCs to map physical memory, setting peripherals, manage threads, and making SMC calls directly. OPTEE runs these privileged TAs directly as a TZ kernel driver in EL1. RPC requests between the two worlds consist of the TA identifier (a UUID), a command ID that dictates which function to run, and a shared buffer for arguments or data transfer. The TEE kernel driver in NW allocates shared memory from the host application heap and only checks buffer sizes and direction flags as a basic security mechanism. Inadequate authorization allows unsafe communication between any set of applications and TAs (§2).

4.1.2 Intel SGX An SGX enclave is part of the host userspace application, sharing the same virtual address space. The enclave runs in encrypted and integrity-protected memory after loading, although they can access all of the host application memory. Only the enclave memory is trusted whereas all other memory regions are untrusted. Intel SGX encompasses two collections of instruction extensions for enabling userspace enclaves, referred to as SGX1 and SGX2. The SGX2 extensions allow additional flexibility in runtime management of enclave resources (e.g. adding memory to an enclave after the enclave is built and running) and thread execution within an enclave. The enclave instructions available with SGX are organized as leaf functions under three instruction mnemonics: `ENCLS` (ring 0), `ENCLU` (ring 3), and `ENCLV` (VT root mode). Each leaf function uses `EAX` to specify the leaf function index and may require additional implicit input registers as parameters. Enclave memory management is divided into two parts: address space allocation and memory commitment. Address space allocation is the specification of the range of linear addresses that the enclave may use, called the `ELRANGE` where no actual resources are committed to this region. Memory commitment is the assignment of actual memory resources (as pages) within the allocated address space. During enclave creation, code and data are loaded from a non-enclave memory. Enclave code, data, and metadata reside in a protected region of physical memory called the enclave page cache (EPC). The EPC is divided into 4KB chunks called EPC pages that can contain either an enclave page or an enclave control structure, SECS. The host system software map the enclave virtual address space to a valid EPC page and does not allow dynamic extensibility of an enclave memory (will be supported in SGX2). Enclave memory is protected by two main mechanisms, CPU access controls and a dedicated memory encryption engine (MEE). CPU memory protection mechanisms physically block access to Processor Reserved Memory (PRM) and EPC from all unauthorized access. MEE is a hardware unit that encrypts and integrity protects EPC cache lines written to and fetched from the main memory (DRAM).

SGX adds several instructions extensions for enclave handling. Most of these instructions can only be accessed in privileged mode. The host OS declare protected memory for the enclave (ECREATE), allocates and load secrets into the enclave memory (EADD), initialize an enclave and measure its memory (EEXTEND, EINIT), and

cleanup enclave pages after the application have completed (EREMOVE). Userspace applications can enter (EENTER), exit (EEXIT), or resume (ERESUME) into their enclaves. An enclave may also exit asynchronously due to interrupts or exceptions (AEX). In the case of asynchronous exit, the enclave secrets will be protected. An enclave can also request to get a signed measurement (EGETKEY, EREPORT) for remote or local attestation. The Intel SGX SDK [23] trusted runtime system (tRTS) is statically compiled into the enclave code, while the untrusted runtime system (uRTS) is dynamically loaded when the host executes. It provides an Enclave Description Language (EDL) to defines the entry points into the enclave in the form of ECALLs (enclave calls) and OCALLs (outside calls) to control the transition from enclave to outside. When the application is compiled, the Edger8r tool parses the EDL file and generates the appropriate ecall/ocall RPCs in the form of wrappers for the functions mentioned in the EDL. Each interface function is assigned with two wrappers, one in the untrusted half and one in the trusted half. These interfaces marshal data inside and outside of the enclave and maintain return status.

4.2 Enabling Deluminator in the Linux kernel

Our work on enabling Deluminator in the Linux kernel is distinct from previous DIFC systems. Deluminator is the first system that follows a distributed model to monitor dataflows locally as well as across the host-TEE privilege boundary. Our primary design goal is to achieve fine-grained information flow tracing while providing good performance. Deluminator supports not only intra-address space tracing but also enables efficient cross-compartment monitoring. The Linux kernel is not currently designed to facilitate these features efficiently, and so previous work is limited to process-level monitoring [51], or had to build a non-POSIX kernel to label intra-address space objects efficiently [109] or to support multithreading. Any control switch and data exchange to and from TEEs and enclaves are expensive. We next present techniques that reduce this cost, particularly for embedded use cases, and that also integrate smoothly with the Linux kernel and are compatible with existing security or auditing features. We achieved this with an $8K$ LoC kernel module and fairly small changes elsewhere in the kernel.

PMTS & security hooks. Deluminator's PMTS dynamically redirects every syscall on a tagged object to a separate path for monitoring any policy violation. The PMTS is implemented from scratch as an extension to the Linux Security Module (LSM) and enabled by configuring the kernel with `CONFIG_EXTENDED_LSM_PMTS`. It is implemented at a lower-level than LSM, and any existing LSMs can be enabled on top of it. PMTS thus avoids conflicts with any existing Linux kernel security features like DAC, LSM, and seccomp filtering. PMTS also implements the DIFC principles and adds new syscalls and security hooks for managing label operations, capability lists, and checking dataflows. PMTS extends LSM with 29 new security hooks that are placed inside various kernel subsystems to track and control dataflows within tagged objects. Some of the hooks are object-specifics; e.g. the `check_tasks_labels_allowed` hook checks whether the dataflow between two Linux `tasks` are allowed. PMTS provides security hooks for labeling and tracing other kernel objects such as `inode`,`file`, `VA`, `socket`, and `pipe`.

Tags and their corresponding capabilities (`capability_t`) are represented by unsigned integers. Tags are generated from a monotonically increasing counter and randomized to prevent an attacker inferring the counter value. Each capability is represented by two bits for the `PLUS_CAPABILITY` or `MINUS_CAPABILITY` types. When labeling any object or execution entity, PMTS initialises and adds their tags to the `object_security_struct ->label_struct` and capabilities to the `cap_segment` list. There are two new helper syscalls (`alloc_label` and `set_task_label`) for facilitating the process of converting userspace policies and for labeling threads and system objects. Each object struct is extended with with a pointer to the Deluminator metadata which holds its label (e.g. `inode->s_obj_label`) and corresponding capabilities. PMTS assigns the metadata before creating a labeled object.

Tagged threads. The `task_struct->cred` structure is extended with a `task_security_struct` to tag kernel threads. There are new hooks for labeling threads including `difc_cred_alloc_blank`, `difc_cred_free`, `difc_cred_prepare` and `difc_cred_transfer` as replacements for the existing LSM hooks `cred_alloc_blank`, `cred_free`, `cred_prepare` and `cred_transfer` so that both tagged and unmodified threads could be supported. There is no support for inheriting credentials and capabilities by default in the style of `fork`, and `copy_creds` and `copy_process` disallow `cred` inheritance by allocating an empty `cred` per labeled thread.

Tagging virtual memory objects. There is a new kernel virtual memory abstraction to enable a mutually distrustful model for each labeled thread to tag and monitor its own VAOs (virtual address space objects). Each VAO is a *labeled* and *contiguous* range of virtual memory that maintains a label, a virtual segment descriptor, and a private virtual page table (`pgd_t`). Linux tasks in a single process share the same `mm_struct` that describes the process address space. Having separate `mm_struct` for tagged threads would significantly impact system performance, as all the memory operations related to page tables must maintain strict consistency. We instead extend `mm_struct` to embed VAO metadata as lightweight protected regions in the same address space (Listing 3). This stores a per-thread `pgd_t` and metadata for memory management, fault handling, and synchronization.

```
1  struct mm_struct {
2  ...
3  #ifdef CONFIG_EXTENDED_LSM_PMTS
4      struct vao_struct *vao_metadata[VAO_MAX];
5      atomic_t num_vao; /* number of vaos */
6      /* vao page tables per thread */
7      pgd_t *vao_pgd_list[VAO_MAX];
8      int curr_using_vao;
9      spinlock_t sl_vao[VAO_MAX];
10     struct mutex vao_metadata_mut;
11     DECLARE_BITMAP(VAO_InUse, VAO_MAX);
12 #endif //CONFIG_EXTENDED_LSM_PMTS//
13 ... };
```

Listing 3: Extended mm_struct with VAO data structures.

The standard Linux kernel avoids reloading page tables during a context switch if two tasks belong to the same process, and so `check_and_switch_context` is extended to reload VAO-based page tables and flush related TLB entries if one of the switching threads owns a VAO. These private page tables are tagged via the ASID/PCID or mapped to hardware memory domains (if available) to reduce the number of TLB flushes. During this lightweight switch the virtual page tables are loaded into the TTBR register (on AArch)

and CR3 (on x86) when a labeled thread needs to do memory operations inside a VAO.

The Linux memory management module is extended with VAO-based paging operations for allocation, deallocation, mapping, unmapping, tracking, and permission management and exposed to userspace via 12 new syscalls. `vao_ops` and `vao_mem_ops` are implemented as multi-purpose syscalls for either initialisation or cleaning up a process. All these operations are only allowed after successfully passing the Deluminator security hooks for monitoring information flows. To keep track of VAO-mapped memory ranges there are `vao_mmap/mumap` syscalls to manage the memory layout of each VAO, and `mmap.c` is extended to check the virtual memory partitions of non-Deluminator processes from memory dedicated to VAOs. The `do_mmap` call now checks if the caller thread is labeled and has the right capabilities to change the memory layout of an address space and also checks for any overlapping virtual memory ranges by keeping lists of dedicated virtual memory ranges to Deluminator VAOs. The `madvise` syscall and `mm/memory.c` operations are also now aware of VAO ranges and check for unauthorised operations by labeled threads.

Files, sockets, & pipes. The VFS layer now monitors each thread's security policies for all operations on `inode`, `file`, and VFS address space objects; these kernel abstractions are used to perform operations on unopened files and file handles. The PMTS checks all operations on tagged files to ensure no thread can operate on them without having the right label and capabilities, including any operation that changes file contents or other file attributes such as its existence, location, size, or linkage type. The label of an `inode` is stored in the `inode-> i_security` structure. The PMTS extends the LSM with new file-specific security hooks for verifying safe information flows. `difc_inode_set_security` is used internally for labeling file objects and a new `inode_permission` security hook checks DIFC rules before any file operation. Most `inode` operations (e.g. `create`, `link`, `mknod`, `mkdir`, `permission`) require a lookup to find related `inodes` and `dcaches` and so `namei` now has PMTS security hooks to monitor unauthorized information flow earlier in the lookup stages. The `open` syscall now has two new flags (`SLABEL` and `ILABEL`) that a thread can use to create a labelled file (`O_CREAT | SLABEL`) and new `file/inode_permission` security hooks trace file operations such as `open/close`, `read/write`, `stat`, `seek`, `link`, and so on.

The kernel networking subsystem is also extended to support DIFC in operations like `create`, `listen`, `connect`, `sendmsg`, and `recvmsg` by placing PMTS security hooks in those functions and at the end of the lookup process (e.g. in `sockfd_lookup_light`). Deluminator also monitors communications through pipes, which—as with files—are associated with `inodes`. Deluminator pipe labeling is similar to files and checks the security context of a pipe against the security context of the thread that is reading from or writing to the pipe. The `pipe` syscall is extended with `SLABEL` and `ILABEL` flags for facilitating the creation of labeled pipes.

4.3 TrustZone kernel

The Deluminator TrustZone OS is built on top of the OP-TEE (V3.14) core kernel, which is a popular open-source TrustZone kernel. Compared to the Linux kernel, OPTEE has a much less complex design that only provides about 50 syscalls and supports fewer system objects. We mainly modified it for threading, memory management, shared memory, and RPC functionality. We added a simplified version of PMTS as an OPTEE driver (i.e., PTA) to check DIFC rules on the TA's system objects (e.g., VAOs, threads, RPCs).

Deluminator configures the `TTBRC` registers to support several L1 translation tables, one large spanning 4 GiB and two or more small tables spanning 32 MiB. The large translation table handles secure kernel mode mapping and matches all addresses not covered by the small translation tables. The small translation tables are assigned per thread and cover the mapping of the virtual memory space for one TA context. This design facilitates mapping VAOs virtual segments descriptors to small private page tables. TA-specific page tables are managed with the page table cache. In OPTEE, a memory object (MOBJ) describes a piece of memory. There are different kinds of MOBJs describing physically contiguous memory, virtual contiguous memory, and shared memory. To enable our VAO abstraction inside TAs, Deluminator labels virtual and shared memory MOBJs and uses OPTEE-PMTS security hooks (similarly to the Linux-PMTS) to check and control information flows over them. It also adds DIFC checking hooks on all sensitive MOBJs interfaces.

```
1  enum thread_state {
2    THREAD_STATE_FREE,
3    THREAD_STATE_SUSPENDED,
4    THREAD_STATE_ACTIVE,
5  #ifdef CONFIG_EXTENDED_PMTS
6      THREAD_STATE_LABELED
7  #endif //CONFIG_EXTENDED_PMTS//
8  };
9  struct thread_ctx {
10   struct thread_ctx_regs regs;
11   enum thread_state state;
12   vaddr_t stack_va_end;
13   uint32_t hyp_clnt_id;
14   uint32_t flags;
15   struct core_mmu_user_map user_map;
16   bool have_user_map;
17 #ifdef CONFIG_EXTENDED_PMTS
18     struct label_struct label; // secrecy and integrity labels
19     struct mutex m; // for thread-safe label operations
20     struct vao_struct *vao_list[VAO_MAX]; // list of VAOs
21 #endif //CONFIG_EXTENDED_PMTS//
22   void *rpc_arg;
23   struct mobj *rpc_mobj;
24   struct thread_specific_data tsd;
25   };
```

Listing 4: Thread labeling data structures in modified OPTEE (when enabling CONFIG_EXTENDED_PMTS kernel config)

Deluminator allows a static and configurable number of threads to support running jobs in parallel within a TA. Each thread context can be labeled and has a list of attached VAOs (Listing 4). RPC services are built on top of the ARM `SMC` calling convention and Deluminator ensures that labeled RPC objects are tracked based on DIFC principles. An RPC exit occurs when the kernel needs some service from the normal world. RPC can currently only be performed with a thread that is in a running state, and is initiated with a call to `thread_rpc()` that uses the OPTEE-PMTS interface to label the RPC if required by user policies. This saves the state such that when the thread is restored it will continue at the next instruction as if the function did a normal return.

Source/Project	Attacker	Victim	Description	Compartments Objects	Policy Violation	LoC
COIN [45]	host app	enclave	heap info leak	$C1\{t_e, mo, rpc\}\ C2\{t_a, rpc\}$	$t_e \not\longrightarrow ocall(mo) \not\longrightarrow t_a$	5
COIN [45]	host app	enclave	malicious calls	$C1\{t_e, mo_e, rpc\}\ C2\{t_a, mo_a, rpc\}$	$t_a^{\otimes} \not\longrightarrow ecall(mo_a) \quad t_e \not\longrightarrow ocall(mo_e)$	9
COIN [45]	host app	enclave	heap overflow	$C1\{t_e, mo, rpc\}\ C2\{t_a, rpc\}$	$t_e \not\longrightarrow ocall(mo) \not\longrightarrow t_a$	6
SGX-tls [110]	host app	enclave	stack info leak	$C1\{t_e^1, mo^1\}\ C2\{t_e^2, mo^2\}$	$t_a^1 \not\longrightarrow mo^2(memcpy(mo^2, mo^1, size))$	21
SGX-SQLite [60]	host app	enclave	malicious calls	$C1\{t_e^1, f_e\}\ C2\{t_a\}$	$t_a^{\otimes} \not\longrightarrow f_e$	8
SGX-Tor [47]	enclave	host app	export secret	$C1\{t_e, mo\}\ C2\{t_u, f_u, s_u, p_u\}$	$t_e \otimes \not\longrightarrow \{t_u, f_u, s_u, p_u\}$	15
SGX-Tor [47]	host app	enclave	concurrent calls	$C1\{t_e\}\ C2\{t_a\}$	$t_a^{\otimes} \not\longrightarrow \{t_e\}$	4
Boomerang [58]	host app	TA/kernel	break TZ-OS	$C1\{t_e, mo_e, rpc_e\}\ C2\{t_a, mo_e, rpc_e\}$	$t_a^{\otimes} \not\longrightarrow mo_e$	23
OPTEE [3]	host app	TA	TA mem corr	$C1\{t_e^1, mo_e\}\ C2\{t_e^2\}$	$t_e^2 \not\longrightarrow mo_e$	8
OPTEE [3]	TA	TA	memory leak	$C1\{t_e\}\ C2\{t_a, mo\}$	$t_e \not\longrightarrow mo$	11
OPTEE [3]	host app	TA	TA mem leak	$C1\{t_e, mo_e, rpc_e\}\ C2\{t_a\}$	$t_a \not\longrightarrow rpc_e(mo_e)$	9
tale [99]	host app	enclave	enclave mem corr	$C1\{t_e, mo_e^1, rpc_e\}\ C2\{t_a, rpc_e\}$	$t_e^{\otimes} \not\longrightarrow mo_e^1 \quad t_a^{\otimes} \not\longrightarrow t_e$	12
async [106]	host app	enclave	concurrency	$C1\{t_e, mo_e, rpc_e\}\ C2\{t_a, mo_a, rpc_e\}$	$t_a^{\otimes} \not\longrightarrow rpc_e \quad t_a^{\otimes} \not\longrightarrow mo_e \wedge mo_a$	18
HPE [91]	host app	other app	break shared enclave	$C1\{t_e, rpc_e^1, rpc_e^2\}\ C2\{t_a, rpc_e\}$	$t_a^{\otimes} \not\longrightarrow rpc_e^2$	9

Table 2: Utilizing Deluminator for investigating vulnerabilities across heterogeneous compartments. We use subscripts e for TA/enclave objects, a for application process, u for untrusted host source, $\otimes$ for attacker, t for thread, mo for memory object, rpc for RPC object that means ocalls/ecalls, f for file/db, s for socket, p for pipe, and $\not\longrightarrow$ for dataflow policy violations.

4.4 SGX runtime

Porting the Deluminator Linux kernel required only minor changes to the VAO abstractions and MMU registers. For the Deluminator-SGX implementation, we focused more on implementing a minimal proof-of-concept to evaluate Deluminator usability and performance for monitoring information flows in the host-enclave boundary. We therefore built the Deluminator-SGX stack over the Intel SGX driver and SDK.[3] The driver handles enclave initialisation and resource management and provides `ioctl` calls for the SDK (e.g. to manage memory via the `ENCLAVE_ADD_PAGE` or `ENCLAVE_PAGE_REMOVE` ioctls) and is where we integrated the PMTS functionality and VAO-based memory management for monitoring labeled shared memory.

We modified the SGX Platform SoftWare (PSW), which communicates with the SGX driver and initializes and loads an enclave memory image, handles enclave exceptions, attestation services, and executes ecalls/ocalls. We also ported the PMTS DIFC module as a standalone library to the SDK to provide proxy calls for monitoring shared resources, as with the TrustZone implementation. Table 3 shows that our approach requires less than $2K$ LoC changes for porting to a new TEE system.

	TEE system	Linux kernel	Userspace
AArch	$5.2K$	$8K$	$3K$
x86-64	$3K$	$8K$	$3K$

Table 3: Deluminator lines of code for each component.

5 Evaluation

Goals. We evaluate Deluminator to answer the following questions: *(1)* Is Deluminator practical for security analysis, auditing, and attack investigation? *(2)* How it can help with vulnerability analysis and auditing in real-world compartmentalized applications, and how much overhead does it add? *(3)* How does Deluminator help to detect and model fine-grained attack vectors?

[3]https://github.com/intel/linux-sgx and https://github.com/intel/linux-sgx-driver

Setup. We use NXP i.MX7Dual boards with two Cortex-A7 CPUs and Intel Core(TM) i7-6820HQ CPU for our evaluation. We use two sets of microbenchmarks, LMbench 3.0 [61] and a custom benchmark for evaluating Deluminator overhead.

5.1 Attack investigation

We integrated Deluminator into some existing security benchmarks including attack prototypes on already compartmentalized TrustZone-and SGX-based applications. Note that in these security benchmarks, all compromised applications are already compartmentalized and we do not change their structure. Our goal is to evaluate the practicality of Deluminator in detecting and analyzing these attack vectors through Deluminator-provided information flow graphs. We picked 14 sets of vulnerabilities from the literature (Table 2) with attack vectors ranging from privilege management issues (e.g., HPE [91] vulnerabilities) to memory vulnerabilities (e.g., COIN [45]) and unsafe interfaces (e.g., Asyncshock [106]).

We used two approaches for Deluminator integration into these security benchmarks. In cases where we knew the scope of attacks, we used Deluminator to monitor all key objects involved in the process-enclave interactions, many of which are described in the SGX SDK `.edl` files (or similarly in OPTEE RPC services). For example, in COIN attacks on an SGX interface to leak information from an enclave heap[4] the SGX SDK provides a `malloc`-style function to allocate heap memory in an enclave, and any data stored there can be processed only explicitly via ECALLs. But a common unsafe coding practice is storing public and secret data in the same data structure and providing an OCALL interface for transferring this public data. If an attacker modifies a pointer from the public data to the secrets, such OCALLs can leak the sensitive data. We used the Deluminator address space labeling APIs (Table 1) like `d_malloc` and `d_free` on all buffers used for process-enclave interactions as one the main targeted objects in that attack vector. As shown in code 5, we used Deluminator to label `ecall_heap_leak` and `ocall_write_out` with the `SLABEL` flag for enclave compartments (with `ecomp_id`). Table 2 summarizes our changes as simplified policies. Deluminator detects when an information leak happens from

[4]https://github.com/mustakimur/COIN-Attacks/tree/master/PoCs

Compartment types	d_enable	d_add	d_remove	d_cleanup
x86-64 process	30.1	32.3	32.82	35.8
SGX enclave	31.4	32.9	33.2	34.9
ARM process	37.2	39.4	41.5	41.9
TrustZone TA	37.8	40.2	42.1	40.8

Table 4: In average (5000 run) latency (µs) of Deluminator API for compartment management.

the enclave compartment ($C1\{t_e, mo, rpc\}$ that contains thread t_e, memory object mo, and labeled ecalls/ocalls as rpc), to the process compartment ($C2\{t_a, rpc\}$); which the violation showed as unsafe dataflow $t_e \not\longrightarrow ocall(mo) \not\longrightarrow t_a$.

In other cases, when the attack vector is not clear, the simplest way to integrate Deluminator is by identifying security-sensitive objects (e.g., cryptography keys) and labelling objects that are directly or indirectly associated with the application's secrets. We will explain this approach in our application case studies in Section 5.3. Note that like any other security analysis tool, Deluminator also works best when developers relatively know the security-sensitive parts of their application or have high-level knowledge of possible attack vectors, which is usually a reasonable assumption since our targets are already compartmentalized applications.

```
1  // in-enclave code
2  void ecall_heap_leak(struct eData* data){
3    char temp[] = "kioasdinkadssasdkjhsdaklj";
4    char *buf;
5    int ret,ecomp_id,mobj;
6    ecomp_id=d_enable();
7    d_add(self, THRD);
8   int mobj=d_create(MMU_MODE, MEMDOM_READ|MEMDOM_WRITE);
9  //..... other initialization
10   data->msg = (char*) d_malloc(mobj,strlen(temp));
11   d_memcpy(mobj,data->msg, temp, strlen(temp));
12 //....other computations
13   while(data->left > 0){
14     buf = data->msg + data->len - data->left;
15     d_ocall_write_out(cid,&ret, buf, data->left);
16     d_cleanup(ecomp_id);
17 //....other computations
18 }
19 // Modified Enclave.edl
20 enclave {
21     from "sgx_tstdc.edl" import *;
22     include "../eType.h"
23     trusted {
24         public void ecall_heap_leak([slabel,in] struct eData *
               data);//add slabel flag
25     };
26     untrusted {
27         int ocall_write_out([slabel, in, size=left] char* buf,
               int left);////add slabel flag
28     };
29 };
```

Listing 5: Changes to enclave side of COIN heap_leak PoC

5.2 Microbenchmarks

Effects on non-compartmentalized execution. We empirically show that Deluminator checks does not add a large overhead to the entire system at runtime. We use a system stress test benchmark (LMBench) to measure the worst case overheads per subsystem (Figure 3). Deluminator adds ≈ 16% latency overhead for the filesystem, ≈ 0.6% for networking, and ≈ 0.2% for IPC benchmarks. Deluminator modifies `fork` to ensure a child thread does not inherit its parents labels and capabilities by default (for mutually-distrustful multithreading policies), which adds 0.1% overhead.

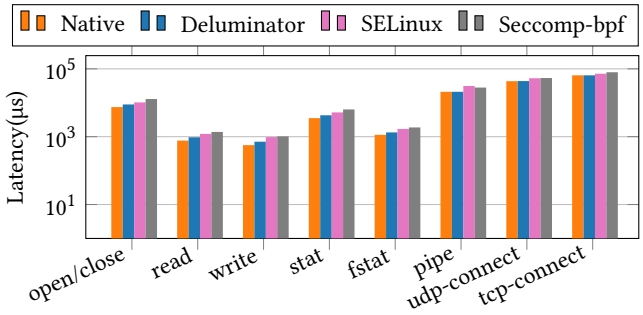

Figure 3: Running Lmbench: impact of Deluminator on overall performance of the host system.

Costs per operation in compartmentalized execution. Table 4 shows the average cost of $30\mu s - 95\mu s$ for Deluminator's basic tracing API within an application process and its enclave compartments. In-enclave compartment operations are 2.1% slower than in-app ones due to differences (e.g., memory management) in the kernel and SGX runtime. Our implementation changes the memory management performance and creating or deleting a tagged memory object (`d_malloc/free`) is about 1.12x slower. Increasing the size of compartment labels only linearly affects the performance of operations on that labeled object, without any change for other objects or operations.

5.3 Case-studies

We utilize LibDeluminator to audit and detect security violations in existing compartmentalized real-world applications to confirm that Deluminator is practical for integrating into TA/enclave-assisted applications with minimal code changes.

Auditing compartmentalized web services. Despite a large amount of work on securing web services [13, 29, 43, 49, 52, 98], their attack surface remains vast due to the lack of secure compartmentalization, fine-grained isolation, and suitable access control mechanisms. Various TA/enclave-assisted web services are available such as TaLoS [8] and SGX-OpenSSL [1] that allow existing applications with an OpenSSL interface to securely terminate their TLS connection inside an enclave. TaLoS places security-sensitive code and data of the TLS library inside an Intel SGX enclave while the rest of the application remains outside. Such building blocks are used in a wide range of security-critical applications for which the integrity and/or confidentiality of their connections must be guaranteed.

OpenSSL is a widely used open-source library implementing cryptography operations and the transport layer security (TLS) protocol. It handles sensitive content such as private keys and encrypted data and hence benefits from isolating its sensitive content in separate compartments to mitigate information leakage attacks [25]. Many web servers, including Apache httpd, use OpenSSL for various operations like the TLS protocol. If compromised, the attacker can leak all private keys and certificates. There are multiple implementations of compartmentalized OpenSSL, with different architectures and levels of isolation enforcement, making it a good case study for compartmentalization tools. We used Deluminator to

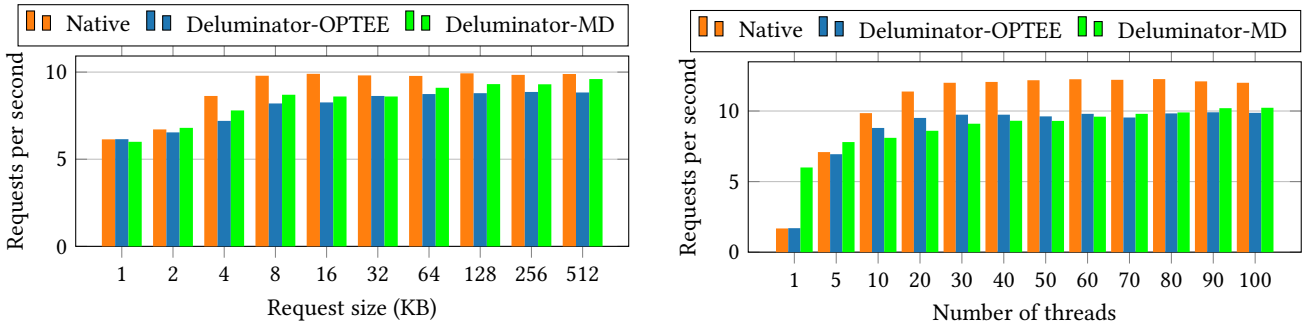

Figure 4: Overhead of tracking dataflows in compartmentalized httpd-OpenSSL with OPTEE and ARM MD-based compartments using Deluminator. For five minutes ApacheBench run per request size, with the TLS1.2 `DHE-RSA-AES256-GCM-SHA384` algorithm cipher suite.

explore three different forms of compartmentalized Apache httpd-OpenSSL stack via *(1)* SGX-assisted compartments, *(2)* TrustZone-assisted compartments, and *(3)* intra-process-based compartments.

SGX-OpenSSL partitions OpenSSL into three libraries; two trusted libraries inside an enclave compartment (`libsgx_tsgxssl.lib` and `libsgx_tsgxssl_crypto.lib`), and one library inside an untrusted compartment (`libsgx_usgxssl.a`) that provides an implementation of missing system APIs outside of an enclave. Since we integrated Deluminator into SGX-SDK, labeling in-enclave systems objects such as the SGX `protected_fs_file`, SGX `pthread`, and `socket` (which has interactions with untrusted system) is straightforward for in-enclave compartments.

We modified the `libsgx_tsgxssl.lib` API, consisting of calls to `tmem_mgmt`, `tpthread`, `tsocket` etc., to use Deluminator-enabled SGX SDK services for labeling associated objects when Deluminator is enabled. When Deluminator is not available, the build system is modified such that all our modified wrappers are replaced by the corresponding native ones, making it viable to support for Deluminator in existing open source projects.

Listing 6 shows some `ecalls/ocalls` from the httpd SGX-SSL interface that are labeled by the Deluminator `slabel` or `ilabel` flags for tracking information flow. Note that we labeled all buffers, files, sockets, and other enclave objects flagged as `user_check`. The SGX SDK provides `user_check` tags for developers to clarify that there are no security guarantees from the SDK over those objects interacting with an untrusted process compartment.

We wrote custom attack PoCs to audit when a malicious httpd worker thread attempts to compromise the enclave, by crafting different `ecall` requests, modifying their shared memory (e.g., when `make_asynchronous_ecall` is enabled on memory management operations), or by accessing enclave files used to storing sensitive configurations. One example of such a sensitive file is via the OpenSSL function `ecall_SSL_CTX_use_PrivateKey_file` used to load private keys into an untrusted buffer. Although Deluminator can detect unsafe dataflows in our custom tests, we need more security benchmarks and attack PoCs with wider coverage for a more through evaluation. The SGX-SSL architecture and implementation is different from TrustZone-based, process-based, or intra-process OpenSSL compartmentalization [74, 93, 94, 98].

```
enclave {
  from "sgx_tstdc.edl" import *;
  include "openssl/ossl_typ.h"
  include "openssl_types.h"
trusted {
  /* Other Apache ecalls*/
  public int ecall_OBJ_create([slabel, ilabel, user_check]
       const char *oid, [slabel, ilabel, user_check] const
       char *sn, [slabel, ilabel, user_check] const char *ln);
       //ilabel for most const objects

  public X509_STORE *ecall_SSL_CTX_get_cert_store([slabel,
       ilabel, user_check] const SSL_CTX *c);

  public int ecall_SSL_CTX_use_certificate_chain_file([slabel,
       slabel, user_check] SSL_CTX *ctx, [slabel, ilabel,
       user_check] const char *file);

  public int ecall_SSL_CTX_check_private_key([slabel, ilabel,
       user_check] const SSL_CTX *ctx);

  public void ecall_SSL_set_connect_state([slabel, user_check]
       SSL *s);

  public X509* ecall_SSL_get_certificate([slabel, ilabel,
       user_check] const SSL *ssl);

  public X509 *ecall_SSL_get_peer_certificate([slabel, ilabel,
       user_check] const SSL *s);

  public SSL_CTX *ecall_SSL_get_SSL_CTX([slabel, ilabel,
       user_check] const SSL *ssl);
  // ...remaining ecalls
}

untrusted {
  void* ocall_mmap([slabel, user_check] void *addr, size_t
       length, int prot, int flags, int fd, off_t offset);

  void* ocall_realloc([slabel, user_check] void* ptr, size_t
       size);

  void ocall_free([slabel, user_check] void* ptr);

  void* ocall_fopen([slabel, ilabel, user_check] const char *
       path, [slabel, ilabel, user_check] const char *mode);

  /* 2 ocalls to fwrite, depending on if ptr is allocated
       inside the enclave (fwrite_copy) or outside (fwrite) */
  size_t ocall_fwrite_copy([in, size=size, count=nmemb] const
       void *ptr, size_t size, size_t nmemb, [slabel,
       user_check] void *stream);
  size_t ocall_fwrite([slabel, ilabel, user_check] const void *
       ptr, size_t size, size_t nmemb, [slabel, user_check]
       void *stream);
  // ...remaining ocalls
}
```

Listing 6: labeling Apache user_check-flagged ecall/ocall objects with Deluminator

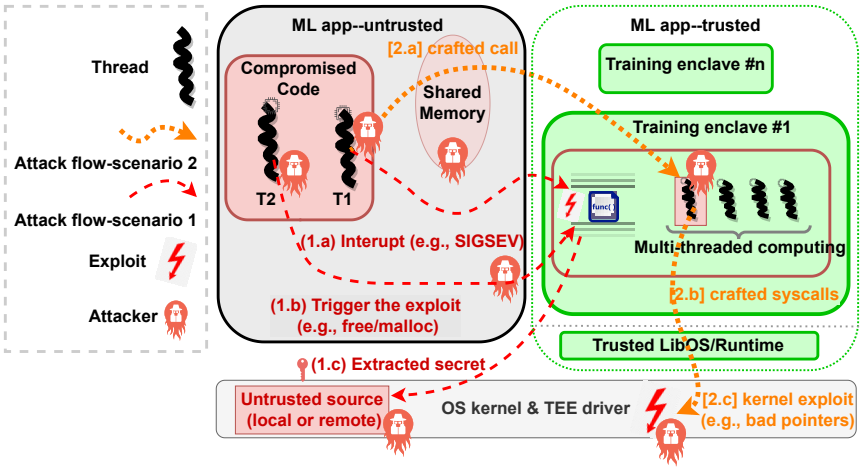

Figure 5: Deluminator can be used in auditing or detecting threats in TEE-assisted and compartmentalized ML.

On OPTEE-enabled devices—due to more limited TEE memory—it is impractical to run the entire OpenSSL stack (more than 45MB RSS and 750K LoC) as a single TA compartment [3]. One approach in an OPTEE compartmentalization framework is to provide an in-TA PKCS11 OpenSSL engine[5] to perform operations on cryptographic objects such as private keys within a TA compartment without requiring access to the objects themselves. For the OPTEE httpd-OpenSSL use case, we again use two compartments. We labeled httpd compartment interfaces to in-TA private keys, session keys, and certificates to track operations on them from any unauthorized worker threads. To also track possible malicious behaviours from the enclave thread (e.g., transferring secrets through uncontrolled channels to another enclave, or via untrusted memory, file or networking sockets) we labeled the enclave thread an all memory and storage objects containing sensitive keys to track such unsafe behaviours.

To evaluate the same use case with more than two compartments, we also integrated Deluminator into an intra-process sandboxing framework using ARM-MDs (memory domains) called uTiles [94]. This is similar to MPK-based isolation domains; to the best of our knowledge there is no hardware available with both SGX and MPK support simultaneously. uTiles provide per-thread lightweight compartments with private isolated domains which are used to compartmentalize OpenSSL private keys, session keys, and certificates into three compartments. All the data structures that store private keys (`EVP_PKEY`) are mapped to specific domains with per-domain memory operations which can be replaced with the Deluminator API (e.g., `utile_malloc/free` replaced by `d_malloc/free` as the backend of `CRYPTO_malloc/free`). Deluminator can then monitor the secrets that are being processed in these protected memory regions. We also labeled the main httpd thread to be the only thread with access to the EVP compartment and that can interact with the FS to store encrypted content, keys, and certificates (e.g., in `OPENSSLDIR`). We also labeled OpenSSL storage to detect unsafe dataflows from other httpd worker threads to monitor unauthorised or accidental information leaks.

Figure 4 shows the overhead of ApacheBench applied against the original OpenSSL library on a baseline kernel and the Deluminator-assisted httpd. ApacheBench ran with a timeline of 5 minutes for each request size, with the TLS1.2 `DHE-RSA-AES256-GCM-SHA384` algorithm cipher suite. The results show that Deluminator adds $\approx$ 8% overhead to SGX-SSL compartments, $\approx$ 10.8% overhead over OPTEE-based compartments, and adds 1.1x slowdown to ARM MD-based compartment, with average added code of 180 LoC. This is a reasonable overhead for compartmentalized applications that now gain strong information flow tracking capabilities across heterogeneous compartments.

Auditing TEE-assisted ML frameworks. There have been various TEE-based solutions for enabling more trustworthy Machine learning (ML), usually by running sensitive parts of ML service in an enclave so that the remote platform can verify that the computation is protected on the untrusted host [37, 63, 64, 80, 100].

Developers can use Deluminator to specify security policies that they want to audit, test, or if they need to explore diverse attack scenarios on such TEE-assisted ML solutions. We integrated LibDeluminator into the DarkneTZ [63] (TrustZone-based) and SGX-Darknet [44] frameworks. These two frameworks have slightly different security goals. SGX-Darknet ports sensitive deep-learning algorithms and layers, such as the connected layer, convolutional layer, softmax layer, cost layer, and the maxpool layer, into an SGX enclave. DarkneTZ focuses on membership inference attacks (MIA) [88] and aims to only port the most sensitive layers since OPTEE TAs usually have less memory available than SGX ones.

These two frameworks have originally added or modified around 350 LoC for DarkneTZ and 290 LoC for SGX-Darknet (from a 32K LoC codebase) for compartmentalizing the framework. Most of these changes are in the interface and system abstraction layer, which we also modified to integrate Deluminator. Since Darknet is heavily multithreaded, we modified its classifier (`classifier.c`) to be able to audit synchronization and concurrency threats by

[5]See https://github.com/OP-TEE/optee_os/tree/5d6b6c795b8f/ta/pkcs11

enabling Deluminator labeling on process's compartment threads, we call it compartment $C1$, for communicating with in-enclave ML compartment $C2$ and use regular threads for the rest of the data loading logic. We further add RPC and shared memory objects to $C1$ for tracing concurrency and interface attacks [81, 99, 106]. We also enabled tracing on all sensitive resources located in the host OS such as (`/cfg`), (`/models`), and (`/data`) to detect any unauthorized access to them. Figure 5 simplifies some of these unsafe information flows in these two attack vectors. We implemented multiple attack scenarios to confirm that Deluminator can detect policy violations.

We evaluated the performance using AlexNet, a benchmark used by prior works [63]. For evaluating Deluminator-enabled DarkneTZ, we train a model with four layers outside and one layer inside an enclave using standard CIFAR-100 dataset [50]. Compared to native, Deluminator adds 4.3% overhead to train ML layers inside an enclave, 7.8% to load a pre-trained model inside the enclave, and 9.2% for in-enclave inference. Similarly, a Deluminator-enabled SGX-Darknet, adds about 9% overhead.

6 Related Work

Compartmentalization techniques are significantly strengthened via hardware-assisted security features that reduce the attack surface of applications. We are now moving toward hetero-compartment computing to take advantage of these, and there are numerous TEE programming frameworks for ergonomic code development for a single platform [3, 23, 56] or cross platform [24, 33] development. The goal of these frameworks is to abstract the underlying TEE details from the enclave programmers. We now see deployment frameworks such as Enarx that support running the same binary within enclaves in different hardware platforms. These systems are inspired by Haven [10] which ports Drawbridge [75] (a Windows library OS) inside an SGX enclave. Since a large portion of applications system support is provided by an in-enclave library OS, it exposes only a small external interface. Following the in-enclave LibOS approach for complex applications results in a huge TCB through porting all dependencies inside the enclave. Graphene-SGX [96] ports the Linux-based Graphene library OS in an enclave. Scone [7] reduces the TCB by porting musl libc and a portion of the Linux Kernel Library (LKL) [76]. TrustShadow [35] also protects unmodified applications from the host OS following the overshadow system [17]. All these systems can benefit from Deluminator since it is the first fine-grained and extensible OS-assisted tracing tool for security analysis tasks such as auditing, forensics, and vulnerability investigation across heterogeneous compartments.

There have been several efforts towards enforcing information flow control policies for enclaves via language-based mechanisms [31, 32, 40, 72, 89, 97]. Although these systems can enforce fine-grained policies, they are either language-specific or are hard to use. More importantly, they are still limited to one compartment type and a one-way trust model (i.e., fully-trusted enclave) which does not allow for monitoring mutually-distrustful security policies.

Variants of dynamic IFC can be enforced at either the software or hardware level, and *granularity* of policy enforcement and the *complexity* of the target system are two key factors determining the practicality of any IFC mechanisms. At the programming language level, these techniques assign explicit security policies (or labels) to every variable and within operations between them [15, 66, 67, 77]. At the OS level, dataflows are enforced within OS kernel objects such as processes and files [19, 51, 108]. They can also be enforced within hardware components [27]. No single IFC mechanism provides the most efficient or practical solutions for these diverse use cases, and Deluminator is the first tracing system over fine-grained system objects that is designed specifically for these hetero-compartment environments.

Some of the attack vectors targeted by Deluminator can also be explored by dedicated tracing, debugging, fuzzing, and dynamic analysis tools [20–22, 70]. No previous work targets the attack vectors described earlier for hetero-compartment environments (e.g., heterogeneous TEE/enclave-assisted applications). Ninja [70] provides relatively expensive (4x to 154x slowdown) debugging and tracing subsystems for transparent malware analysis on ARM that also supports TrustZone-assisted usecases. TEEREX [21] automatically analyzes SGX enclave binary code for memory vulnerabilities introduced at the host-to-enclave boundary via symbolic execution. Some TEE/enclave frameworks may provide debuggers and fuzzers with limited functionalities inside a TA/enclave [22] but none are designed to systematicaly tackle the challenges we mentioned for security development across heterogeneous isolation boundaries(§ 2). To the best of our knowledge, Deluminator is the first system that provides a principled and extensible mechanism for monitoring dataflows and tracing fine-grained system objects across completely different types of compartments including processes, SGX enclaves, and TrustZone TAs. Deluminator also provides intra-address space tracing which can be used by in-TA/enclave compartmentalization frameworks, such as Occlum [87], to monitor finer-grained security violations or attack investigation. Additionally, Deluminator's novel kernel abstractions cause a relatively small performance overhead (on average 7-29%) that is significantly less than related systems.

7 Conclusion

We have proposed Deluminator, a set of OS abstractions and a userspace framework to enable extensible and fine-grained information flow tracking in hetero-compartment environments. Deluminator allows developers specify mutually-distrustful security policies over shared system resources and cross-compartment interactions. It provides building blocks to detect unsafe dataflows which can be used for composing security analyses across heterogeneous compartments. We implemented Deluminator on Linux-based ARM and x86-64 platforms, with support for hardware-assisted compartment types including processes, SGX enclaves, TrustZone Trusted Apps (TAs), and intra-address space compartments. Our evaluation shows that our kernel and hardware integration results in a reasonable overhead (on average 7-29%) that makes it suitable for real-world applications.

Our code is available at https://github.com/Deluminator-System under liberal licenses, and we welcome feedback or patches to improve it further.

Acknowledgments

We thank the anonymous shepherd and reviewers for their constructive feedback, as well as comments on earlier iterations of this work from Jon Crowcroft, Hamed Haddadi and Richard Mortier. We gratefully acknowledge the VMWare Research award that funded this research at the University of Cambridge.

References

[1] 2019. SGX-OpenSSL. https://github.com/sparkly9399/SGX-OpenSSL.

[2] 2020. Intel Trust Domain Extensions (Intel TDX). https://software.intel.com/content/www/us/en/develop/articles/intel-trust-domain-extensions.html.

[3] 2020. OP-TEE. https://github.com/OP-TEE. Access Date : 2020-03-28.

[4] Ross Anderson. 2008. *Security engineering*. John Wiley & Sons.

[5] ARM. 2009. Security technology building a secure system using TrustZone technology (white paper). *ARM Limited* (2009).

[6] ARM. 2012. Architecture Reference Manual; ARMv7-A and ARMv7-R edition. https://static.docs.arm.com/ddi0406/c/DDI0406C_C_arm_architecture_reference_manual.pdf. Access Date : 2020-5-26.

[7] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark Stillwell, et al. 2016. SCONE: Secure Linux Containers with Intel SGX.. In *OSDI*, Vol. 16. 689–703.

[8] Pierre-Louis Aublin, Florian Kelbert, Dan O'keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch. 2017. TaLoS: Secure and transparent TLS termination inside SGX enclaves. *Imperial College London, Tech. Rep* 5, 2017 (01 2017). https://doi.org/10.25561/94936 See https://github.com/lsds/TaLoS.

[9] Andrew Baumann. 2017. Hardware is the new software. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. 132–137.

[10] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding applications from an untrusted cloud with Haven. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 8.

[11] David Berard. 2018. Kinibi TEE: Trusted Application Exploitation.

[12] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. 2018. The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX. In *27th USENIX Security Symposium (USENIX Security 18)*. 1213–1227.

[13] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting applications into reduced-privilege compartments. In *USENIX Association*.

[14] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. 2022. Provably-Safe Multilingual Software Sandboxing using WebAssembly. In *31st USENIX Security Symposium (USENIX Security 22)*. 1975–1992.

[15] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. 2015. HLIO: Mixing static and dynamic typing for information-flow control in Haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. 289–301.

[16] Sanchuan Chen, Zhiqiang Lin, and Yinqian Zhang. 2023. Controlled Data Races in Enclaves: Attacks and Detection. In *32nd USENIX Security Symposium (USENIX Security 22)*.

[17] Xiaoxin Chen, Tal Garfinkel, E Christopher Lewis, Pratap Subrahmanyam, Carl A Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan RK Ports. 2008. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. *ACM SIGOPS Operating Systems Review* 42, 2 (2008), 2–13.

[18] 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)*. IEEE, 56–71.

[19] Winnie Cheng, Dan RK Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. 2012. Abstractions for usable information flow control in Aeolus. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 139–151.

[20] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*. 196–206.

[21] Tobias Cloosters, Michael Rodler, and Lucas Davi. 2020. TeeRex: discovery and exploitation of memory corruption vulnerabilities in SGX enclaves. In *Proceedings of the 29th USENIX Conference on Security Symposium*. 841–858.

[22] Tobias Cloosters, Johannes Willbold, Thorsten Holz, and Lucas Davi. 2022. {SGXFuzz}: Efficiently Synthesizing Nested Structures for {SGX} Enclave Fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. 3147–3164.

[23] Intel Corporation. 2019. Intel Software Guard Extensions for Linux OS. https://github.com/intel/linux-sgx. Access Date :2019-03-01.

[24] Microsoft Corporation. 2019. Open Enclave SDK. https://github.com/openenclave/openenclave. Access Date :2019-08-12.

[25] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. 2014. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*. ACM, 475–488.

[26] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 1–29.

[27] Andrew Ferraiuolo, Mark Zhao, Andrew C Myers, and G Edward Suh. 2018. HyperFlow: A processor architecture for nonmalleable, timing-safe information flow security. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1583–1600.

[28] Charles García-Tobin. 2021. Unlocking the power of data with ARM CCA. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/unlocking-the-power-of-data-with-arm-cca?_ga=2.220985304.13311694.1639690475-1159947857.1639439044.

[29] Daniel B Giffin, Amit Levy, Deian Stefan, David Terei, David Mazieres, John C Mitchell, and Alejandro Russo. 2012. Hails: Protecting data privacy in untrusted web applications. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 47–60.

[30] GlobalPlatform. 2018. GlobalPlatform Security Task ForceRoot of Trust Definitions and Requirements. Available at: https://globalplatform.org/wp-content/uploads/2018/06/GP_RoT_Definitions_and_Requirements_v1.0.1_PublicRelease_CC.pdf.

[31] Anitha Gollamudi and Stephen Chong. 2016. Automatic Enforcement of Expressive Security Policies Using Enclaves. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 494–513. https://doi.org/10.1145/2983990.2984002

[32] Anitha Gollamudi, Stephen Chong, and Owen Arden. 2019. Information Flow Control for Distributed Trusted Execution Environments. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. 304–30414. https://doi.org/10.1109/CSF.2019.00028

[33] Google. 2018. Asylo: An open and flexible framework for enclave applications. http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm.

[34] Jinyu Gu, Bojun Zhu, Mingyu Li, Wentai Li, Yubin Xia, and Haibo Chen. 2022. A {Hardware-Software} Co-design for Efficient {Intra-Enclave} Isolation. In *31st USENIX Security Symposium (USENIX Security 22)*. 3129–3145.

[35] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. 2017. TrustShadow: Secure execution of unmodified applications with ARM TrustZone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 488–501.

[36] Khilan Gudka, Robert NM Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G Neumann, and Alex Richardson. 2015. Clean application compartmentalization with soaap. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1016–1031.

[37] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. 2018. Chiron: Privacy-preserving Machine Learning as a Service. *arXiv preprint arXiv:1803.05961* (2018).

[38] Intel. 2016. Overview of Intel Software Guard Extensions Instructions and Data Structures. https://software.intel.com/en-us/blogs/2016/06/10/overview-of-intel-software-guard-extensions-instructions-and-data-structures.

[39] Intel. 2019. Intel® 64 and IA-32 Architectures Software Developer's Manual. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf

[40] Jin Soo Jang, Sunjune Kong, Minsu Kim, Daegyeong Kim, and Brent Byunghoon Kang. 2015. SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment.. In *NDSS*.

[41] Jinghao Jia, Raj Sahu, Adam Oswald, Dan Williams, Michael V Le, and Tianyin Xu. 2023. Kernel extension verification is untenable. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*. 150–157.

[42] Limin Jia, Jassim Aljuraidan, Elli Fragkaki, Lujo Bauer, Michael Stroucken, Kazuhide Fukushima, Shinsaku Kiyomoto, and Yutaka Miyake. 2013. Run-time enforcement of information-flow properties on Android. In *European Symposium on Research in Computer Security*. Springer, 775–792.

[43] David Kaloper-Mersinjak, Hannes Mehnert, Anil Madhavapeddy, and Peter Sewell. 2015. Not-Quite-So-Broken TLS: Lessons in Re-Engineering a Security Protocol Specification and Implementation.. In *USENIX Security Symposium*. 223–238.

[44] Ryan Karl, Jonathan Takeshita, and Taeho Jung. 2020. Using Intel SGX to Improve Private Neural Network Training and Inference. In *Proceedings of the 7th Symposium on Hot Topics in the Science of Security* (Lawrence, Kansas) *(HotSoS '20)*. Association for Computing Machinery, New York, NY, USA, Article 31, 2 pages. https://doi.org/10.1145/3384217.3386399

[45] Mustakimur Rahman Khandaker, Yueqiang Cheng, Zhi Wang, and Tao Wei. 2020. COIN attacks: On insecurity of enclave untrusted interfaces in SGX. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 971–985.

[46] Douglas Kilpatrick. 2003. Privman: A Library for Partitioning Applications.. In *USENIX Annual Technical Conference, FREENIX Track*. 273–284.

[47] Seongmin Kim, Juhyeng Han, Jaehyeong Ha, Taesoo Kim, and Dongsu Han. 2018. SGX-Tor: A Secure and Practical Tor Anonymity Network With SGX Enclaves. *IEEE/ACM Transactions on Networking* 26, 5 (2018), 2174–2187.

[48] 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*. 132–148.

[49] Akshay Krishnamurthy, Adrian Mettler, and David Wagner. 2010. Fine-grained privilege separation for web applications. In *Proceedings of the 19th international conference on World wide web*. 551–560.

[50] Alex Krizhevsky. 2009. The CIFAR-100 dataset. https://www.cs.toronto.edu/~kriz/cifar.html. Access Date : 2020-5-26.

[51] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M Frans Kaashoek, Eddie Kohler, and Robert Morris. 2007. Information flow control for standard OS abstractions. In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 321–334.

[52] Maxwell N Krohn. 2004. Building Secure High-Performance Web Services with OKWS.. In *USENIX Annual Technical Conference, General Track*. 185–198.

[53] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. 2020. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*.

[54] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. 2017. Hacking in darkness: Return-oriented programming against secure enclaves. In *26th USENIX Security Symposium (USENIX Security 17)*. 523–539.

[55] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Yi Chien, Felipe Huici, Nathan Dautenhahn, and Pierre Olivier. 2022. Assessing the Impact of Interface Vulnerabilities in Compartmentalized Software. *arXiv preprint arXiv:2212.12904* (2022).

[56] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, P Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. 2017. Glamdring: Automatic application partitioning for Intel SGX. In *USENIX*.

[57] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 49–64.

[58] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. 2017. BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments.. In *NDSS*.

[59] Marion Marschalek. 2018. The Wolf In SGX Clothing. *Bluehat IL (Jan 2018)* (2018).

[60] Yerzhan Mazhkenov. 2019. SGX-SQLite. https://github.com/yerzhan7/SGX_SQLite.git.

[61] Larry W McVoy, Carl Staelin, et al. 1996. lmbench: Portable tools for performance analysis.. In *USENIX annual technical conference*. San Diego, CA, USA, 279–294.

[62] Marcela S Melara, Michael J Freedman, and Mic Bowman. 2019. EnclaveDom: Privilege separation for large-TCB applications in trusted execution environments. *arXiv preprint arXiv:1907.13245* (2019).

[63] Fan Mo, Ali Shahin Shamsabadi, Kleomenis Katevas, Soteris Demetriou, Ilias Leontiadis, Andrea Cavallaro, and Hamed Haddadi. 2020. DarkneTZ: Towards Model Privacy at the Edge Using Trusted Execution Environments. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services* (Toronto, Ontario, Canada) *(MobiSys '20)*. Association for Computing Machinery, New York, NY, USA, 161–174. https://doi.org/10.1145/3386901.3388946

[64] Fan Mo, Zahra Tarkhani, and Hamed Haddadi. 2022. SoK: Machine Learning with Confidential Computing. *arXiv preprint arXiv:2208.10134* (2022).

[65] James Morris, Stephen Smalley, and Greg Kroah-Hartman. 2002. Linux security modules: General security support for the linux kernel. In *USENIX Security Symposium*. ACM Berkeley, CA, 17–31.

[66] Andrew C Myers and Barbara Liskov. 1997. A decentralized model for information flow control. In *SOSP*, Vol. 97. Citeseer, 129–142.

[67] Andrew C Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. 2001. Jif: Java information flow. *Software release. Located at http://www. cs. cornell. edu/jif* 2005 (2001).

[68] Adwait Nadkarni, Benjamin Andow, William Enck, and Somesh Jha. 2016. Practical {DIFC} Enforcement on Android. In *25th USENIX Security Symposium (USENIX Security 16)*. 1119–1136.

[69] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Retrofitting fine grain isolation in the Firefox renderer. In *Proceedings of the 29th USENIX Conference on Security Symposium*. 699–716.

[70] Zhenyu Ning and Fengwei Zhang. 2017. Ninja: Towards Transparent Tracing and Debugging on ARM.. In *USENIX Security symposium*. 33–49.

[71] NXP. 2022. ASUG-i.MX Android Security User's Guide. https://www.nxp.com/docs/en/user-guide/IMX_ANDROID_SECURITY_USERS_GUIDE.pdf.

[72] Aditya Oak, Amir M Ahmadian, Musard Balliu, and Guido Salvaneschi. 2021. Language Support for Secure Software Development with Enclaves. In *IEEE Computer Security Foundations Symposium (CSF 2021)*.

[73] Joongun Park, Naegyeong Kang, Taehoon Kim, Youngjin Kwon, and Jaehyuk Huh. 2020. Nested enclave: supporting fine-grained hierarchical isolation with SGX. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 776–789.

[74] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2018. libmpk: Software Abstraction for Intel Memory Protection Keys. *arXiv preprint arXiv:1811.07276* (2018).

[75] Donald E Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C Hunt. 2011. Rethinking the library OS from the top down. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 291–304.

[76] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A. Sartakov, and Peter Pietzuch. 2020. SGX-LKL: Securing the Host OS Interface for Trusted Execution. arXiv:1908.11143 [cs.OS]

[77] Indrajit Roy, Donald E Porter, Michael D Bond, Kathryn S McKinley, and Emmett Witchel. 2009. *Laminar: Practical fine-grained decentralized information flow control*. Vol. 44. ACM.

[78] Mark Russinovich, Manuel Costa, Cédric Fournet, David Chisnall, Antoine Delignat-Lavaud, Sylvan Clebsch, Kapil Vaswani, and Vikas Bhatia. 2021. Toward confidential cloud computing. *Commun. ACM* 64, 6 (2021), 54–61.

[79] Mickaël Salaün. 2017. Landlock LSM: toward unprivileged sandboxing. *Linux Security Summit* (2017).

[80] Ahmed Salem, Yang Zhang, Mathias Humbert, Pascal Berrang, Mario Fritz, and Michael Backes. 2018. Ml-leaks: Model and data independent membership inference attacks and defenses on machine learning models. *arXiv preprint arXiv:1806.01246* (2018).

[81] Jose Rodrigo Sanchez Vicarte, Benjamin Schreiber, Riccardo Paccagnella, and Christopher W Fletcher. 2020. Game of threads: Enabling asynchronous poisoning attacks. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 35–52.

[82] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. 2014. Using ARM TrustZone to build a trusted language runtime for mobile applications. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 67–80.

[83] 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)*. 1677–1694.

[84] Michael Schwarz, Samuel Weiser, and Daniel Gruss. 2019. Practical enclave malware with Intel SGX. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 177–196.

[85] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2020. Malware Guard Extension: abusing Intel SGX to conceal cache attacks. *Cybersecurity* 3 (2020), 1–20.

[86] AMD SEV-SNP. 2020. Strengthening VM isolation with integrity protection and more. *White Paper, January* (2020).

[87] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and efficient multitasking inside a single enclave of Intel SGX. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 955–970.

[88] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership inference attacks against machine learning models. In *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 3–18.

[89] Rohit Sinha, Manuel Costa, Akash Lal, Nuno P. Lopes, Sriram Rajamani, Sanjit A. Seshia, and Kapil Vaswani. 2016. A Design and Verification Methodology for Secure Isolated Regions. In *Proceedings of the 37th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*. 665–681.

[90] Deian Stefan, Alejandro Russo, David Mazières, and John C Mitchell. 2012. Disjunction category labels. In *Information Security Technology for Applications: 16th Nordic Conference on Secure IT Systems, NordSec 2011, Tallinn, Estonia, October 26-28, 2011, Revised Selected Papers 16*. Springer, 223–239.

[91] Darius Suciu, Stephen McLaughlin, Laurent Simon, and Radu Sion. 2020. Horizontal Privilege Escalation in Trusted Applications. In *29th USENIX Security Symposium (USENIX Security 20)*.

[92] Zahra Tarkhani. 2022. *Secure Programming with Dispersed Compartments*. Ph. D. Dissertation. University of Cambridge.

[93] Zahra Tarkhani and Anil Madhavapeddy. 2020. Enclave-aware compartmentalization and secure sharing with sirius. *arXiv preprint arXiv:2009.01869* (2020).

[94] Zahra Tarkhani and Anil Madhavapeddy. 2020. uTiles: Efficient Intra-Process Privilege Enforcement of Memory Regions. *arXiv preprint arXiv:2004.04846* (2020).

[95] Zahra Tarkhani, Anil Madhavapeddy, and Richard Mortier. 2019. Snape: The dark art of handling heterogeneous enclaves. In *Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking*. 48–53.

[96] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 645–658.

[97] Chia-Che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca Ada Popa, and Donald E Porter. 2020. Civet: An efficient java partitioning framework for hardware enclaves. In *29th USENIX Security Symposium (USENIX Security 20)*. 505–522.

[98] 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.

[99] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D Garcia, and Frank Piessens. 2019. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1741–1758.

[100] Peter M VanNostrand, Ioannis Kyriazis, Michelle Cheng, Tian Guo, and Robert J Walls. 2019. Confidential Deep Learning: Executing Proprietary Models on Untrusted Devices. *arXiv preprint arXiv:1908.10730* (2019).

[101] Marco Vassena, Alejandro Russo, Deepak Garg, Vineet Rajani, and Deian Stefan. 2019. From fine-to coarse-grained dynamic information flow control and back. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–31.

[102] Nicholas C Wanninger, Joshua J Bowden, Kirtankumar Shetty, Ayush Garg, and Kyle C Hale. 2022. Isolating functions at the hardware limit with virtines. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 644–662.

[103] Robert NM Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. 2012. A taste of Capsicum: practical capabilities for UNIX. *Commun. ACM* 55, 3 (2012), 97–104.

[104] Robert NM Watson, Ben Laurie, Steven J Murdoch, Robert Norton, Michael Roe, Stacey Son, Munraj Vadera, Jonathan Woodruff, Peter G Neumann, Simon W Moore, et al. 2015. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy (SP)*. IEEE, 20–37.

[105] Jinpeng Wei and Calton Pu. 2005. TOCTTOU Vulnerabilities in UNIX-Style File Systems: An Anatomical Study.. In *FAST*, Vol. 5. 12–12.

[106] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. 2016. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In *European Symposium on Research in Computer Security*. Springer, 440–457.

[107] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 640–656.

[108] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. 2006. Making information flow explicit in HiStar. In *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 263–278.

[109] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazieres. 2008. Securing Distributed Systems with Information Flow Control.. In *NSDI*, Vol. 8. 293–308.

[110] Fan Zhang. 2019. SGX-mbedtls. https://github.com/bl4ck5un/mbedtls-SGX.