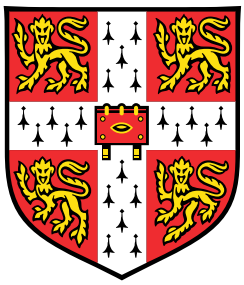
Secure Programming with Dispersed Compartments



Zahra Tarkhani

Department of Computer Science and Technology
University of Cambridge

This dissertation is submitted for the degree of
Doctor of Philosophy

St Edmund's College

February 2022

Declaration

This thesis is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the preface and specified in the text. I further state that no substantial part of my thesis has already been submitted, or, is being concurrently submitted for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. It does not exceed the prescribed word limit for the relevant Degree Committee.

Zahra Tarkhani
February 2022

Abstract

This dissertation proposes novel approaches and mechanisms for application compartmentalization and isolation to reduce their ever-growing attack surface. Our approach is motivated by the key observation that while hardware vendors compete to provide security features, notably memory safety and privilege separation, existing systems software like commodity OSs fail to utilize such features to improve application security and privacy properly. By proposing a novel principled approach to privilege separation and isolation, this work enables application security to be designed and enforced *within* and *across* different isolation boundaries yet remain flexible in the face of diverse threats and changing hardware requirements.

We begin by analyzing the effectiveness of existing systems in mitigating ever-increasing threats. We explore their efficacy where diverse compartments, such as processes, sandboxes, or trusted execution environments (TEEs)/enclaves, are involved. We call such computing environments *hetero-compartment*, which are becoming the future of modern applications. This thesis focuses on resolving three fundamental limitations of the state-of-the-art compartmentalization techniques. The most important one is the inability to scale, extend, and monitor compartments beyond a fixed security model, a single privilege layer (e.g., userspace), or address space boundaries in hetero-compartment environments. Second is the lack of flexible isolation and secure resource sharing. Finally, the third key limitation is ineffective hardware utilization, which leads to significant overhead and weak security, particularly in resource-constrained devices.

We propose *dispersed compartments* as a fundamentally new approach for building applications by encapsulating *arbitrary* isolation boundaries across privilege levels. Dispersed compartments provide a unified model for extensible and auditable compartmentalization. To enable such system-wide privilege separation, we introduce two key concepts; first, *dispersed monitoring* to check extensible security policies. Second, *dispersed enforcement* to enforce isolation and security policies across various privilege boundaries while reducing the trusted computing base (TCB) through deprivileging the host kernel on-demand. Furthermore, we present SIRIUS, our implementation of these security primitives on commodity hardware by focusing on ARM and x86-64 platforms. SIRIUS includes new security extensions and abstractions within the underlying OSs, firmware, and TEE stacks. Moreover, it provides a novel userspace API to reduce application modifications during compartmentalization. Finally, we show the significant security and performance benefits of SIRIUS through microbenchmarks, compartmentalizing real-world applications, and investigating major attack vectors.

Acknowledgements

I want to start by thanking my advisors, Anil Madhavapeddy and Jon Crowcroft, for their support, encouragement, and great technical discussions. I am also indebted to Geoffrey Brown, my Master's advisor at Indiana University Bloomington, who always believed in me, encouraged me to try new experiences, visit new places, and do my very best.

I am thankful to many current and former colleagues and friends at SRG and the Computer Lab. Thank you, Richard Mortier, for your constant support and all your advice in the past few years. My dear friend, Lorena Qendro, you have been so kind and such a supportive friend. Allison Randal, I am lucky that I had a shared office with you; thank you for your feedback on my thesis and for always being supportive whenever I need your advice. Also, thank you, Diana Andreea Popescu, Marno van der Maas, Alessandro Montanari, Heidi Howard, Marco Caballero, Jat Singh, Andrew Moore, Ian Lewis, Srinivasan Keshav, Matthew Danish, Smita Vijayakumar, Domagoj Stolfa, Lawrence Esswood, Martin Kleppmann, KC Sivaramakrishnan, Andrea Ferlini, Noa Zilberman, Cecilia Mascolo, Alastair Beresford, Zohreh Shams, Diana Vasile, Peter Pietzuch, Poonam Yadav, Liang Wang, Helena Andres Terre, Ilia Shumailov, Eiko Yoneki, Chelsea Edmonds, Krittika D'Silva, Cătălina Cangea, Anwaar Ali, and Mahwish Arif.

During my PhD, I also worked with many incredible researchers and engineers at Microsoft Research Redmond and Cambridge. That was an excellent opportunity to use my skills in real-world use cases. I have been privileged to work with Ed Nightingale, whom I learned so much from. I am deeply grateful to David Chisnall for his mentorship and insightful discussions. I am also thankful to Galen Hunt, Ryan Fairfax, Nathaniel Filardo, Reuben Olinsky, Istvan Haller, Robert Norton-Wright, Matthew Parkinson, Jewell Seay, and Akshitha Sriraman. Additionally, I had the great opportunity to work with Shweta Shinde at ETH's Secure and Trustworthy Systems (SECTRS) group, which had to be virtual due to the COVID19 pandemic. I also like to thank Hamed Haddadi, my external examiner, for his thorough feedback on the thesis.

Last but not least, I am indebted to my family for their continued love. My best friend and husband, Ali, for the happiness you brought to my life. My wonderful father, who unconditionally loved me, believed in me, and made it possible for me to follow my dreams. My mother and my brother, Ahmad, always supported me through difficult times. I can never thank my family enough.

Finally, I dedicate this thesis to the victims of flight PS752, Mahsa Amini, and courageous Iranians who have been fighting for freedom and sacrificing their lives for a better world.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

More than ever, we depend on highly connected computing systems in today's world. Computers control almost every essential aspect of our lives, including health care, finance, smart infrastructures, energy sectors, and government, to name a few. As of 2021, over 6.3 Billion people use smartphones, and 35.82 billion IoT (Internet of Things) devices are installed worldwide [11]. The rise of cloud-based services, mobile devices, IoT, and domain-specific hardware alongside open-source software/hardware fundamentally changed application designs. Modern applications run on multiple platforms and seek scalability, performance, rich functionalities, and security/privacy at the same time.

However, despite the fast adaption of computing technology, security has been significantly overlooked. The business demand to design low-cost and faster systems with security as an optional feature or "future work" led to ever-increasing extensive attacks with expensive and even life-threatening real-world consequences [12]. Our growing reliance on edge-cloud services in recent years has been constantly and increasingly threatened by a wide range of security and privacy breaches at scales never seen before[13–15].

There are many case-specific ways to attack a system, so it is impossible to suggest a single technique to solve all security issues. However, proper and secure compartmentalization is the greatest weapon against the ever-growing and non-trivial security threats. When done correctly, compartmentalization is proven to significantly improve the security of systems and applications and restricts the propagation of vulnerabilities. However, existing compartmentalization frameworks fail to sufficiently reduce the complex attack surface in modern applications without significant overhead or cost. In this dissertation, we argue the hypothesis that:

To significantly reduce the attack surface in modern applications, an effective compartmentalization mechanism must be extensible, auditable, and properly utilize hardware-assisted protection and control within and across complex isolation boundaries while requiring the slightest application modification, overhead, and TCB.

For the remainder of this chapter, we first justify the importance of this dissertation, starting with the state of ever-growing security threats (§1.1) and summarizing how existing compartmentalization mechanisms are insufficient. We then present our argument on key features of effective compartmentalization for modern applications and on hetero-compartment environments. We briefly introduce dispersed compartments principles to achieve effective compartmentalization, and SIRIUS our system to make dispersed compartments a reality on commodity hardware (§1.2). Finally we define our contributions (§1.3) and the dissertation structure (§1.4).

1.1 Security in ever-growing complexity

We are living in a world of highly-connected smart spaces that are able to capture and process tremendous amounts of our data (e.g., voice and videos) to provide us with interesting services and a more comfortable lifestyle [16–18]. Achieving such use cases requires complex software and hardware collaboration. The technology industry aggressively develops and combines numerous hardware and software products for powerful, high-accuracy, large-scale, and high-performance computing. As a result, the ever-increasing security threats are not surprising; developers, that already had been struggling with designing and building secure systems, now need to reason about much larger attack surfaces and non-trivial attack vectors [19].

The attack surface of modern applications includes a mixture of traditional attack vectors with new threats within/across various dependencies and system abstractions. For example, despite significant improvements in mitigating conventional attack vectors such as memory vulnerabilities (e.g., by the widespread use of memory-safe languages), memory safety issues are still the main class of security vulnerability in code written in C-like languages. Notably, most of our systems software, such as language runtimes, OSs, hypervisors, and firmware, suffers from memory safety violations. This includes about 70% of all security issues addressed in Microsoft products, over 75% in Google's Android, and $60-70\%$ vulnerabilities in iOS and macOS [20, 21]. Re-writing these vast and complex codebases (e.g., Debian Linux alone contains over half a billion lines) in memory-safe languages is not practical. Even compiling legacy code with more safe extensions of *C* (e.g., Checked C [22] or Cyclone [23]) that prevent buffer overruns and out-of-bounds memory accesses could not scale for complex and large codebases.

On the other hand, software written in memory-safe languages often relies on unsafe bindings to use the system's software services or legacy third-party libraries or suffers from various concurrency attacks (§2.1.1). Besides memory safety, there is also a wide range of attack vectors caused by inadequate privilege management and access control which are considered the most widespread reason for numerous attacks. For example, according to OWASP's 2021 report, the broken access control category moves up to first from the fifth position in 2017; where 94% of applications were tested for some form of broken access control that occurred more than any other category. Consequently, mitigating and detecting such a large attack surface is extremely

challenging. That is why it takes a long time, even years, to clean up known vulnerabilities. For instance, according to a recent study, 75% attacks in 2020 exploited vulnerabilities that were at least two years old, and 18% of attacks exploited vulnerabilities that were disclosed in 2013 or before, making them at least seven years old [24].

As a result, considerable effort has been made to provide hardware-assisted isolation and privilege separation to *(i)* improve compartmentalization, *(ii)* reduce TCB, and *(iii)* limit the propagation of vulnerabilities in each abstraction layer more efficiently and securely than software-only mechanisms [25]. Hence, hardware-based virtualization and Trusted Execution Environments (TEEs) were introduced over a decade ago to compartmentalize various parts of systems stack or isolate applications' sensitive partitions. More recently, various forms of TEEs/enclaves are introduced, on almost all hardware architectures, to enable protecting security-critical code/data against privileged system software such as the host OS or hypervisor [26–32].

Despite similarities, TEEs are highly architecture-dependent and various TEEs support different confidentiality, integrity, and freshness features (usually they do not guarantee availability). TEEs are also designed with specific threat and security models which could be different and even conflicting with other TEEs. For instance, Intel SGX user space enclaves [28] run only with user mode privilege and do not include the host privileged system stacks into their TCB. On the other hand, Intel TDX (Trust Domain Extensions) and AMD's SEV isolate an entire virtual machine as an enclave [27, 31]. At the same time, ARM and RISC-V TEE architectures and threat models are even more different than x86-64-based TEEs. ARM TrustZone TAs (trusted apps) could run in different privilege layers in the secure world (usually in secure-EL0), which have more privileges than the normal world [33, 34]. TAs' TCB also includes any codebase runs in secure kernel mode (secure-EL1) and monitor mode (EL3). ARM will soon enable more partitioning and privilege management via its recently announced Confidential Compute Architecture (CCA) as a key component of the Armv9 architecture [29]. Arm CCA enables more fine-grained and flexible isolated units, called realm, which allow privilege separation inside TrustZone secure world. On RISC-V we also have several research TEEs, such as Keystone [35], Sanctum [9], Timber-V [36], and CURE [37]; each offering different security, performance, and flexibility functionalities.

However, most hardware vendors considered a few limited use cases, such as DRM (digital rights management) or only small cryptography services, when designing TEEs. Hence, these hardware features are added without a comprehensive security model and mechanism for secure integration with existing system and privilege boundaries could instead increase the system complexity and open new attack vectors.

As a simple example, most compartmentalized applications such as data analytic require sharing of data/messages between compartments via host OS-controlled resources. Hence, the program may use untrusted memory to share results across multiple enclaves due to in-enclave memory limitations, untrusted storage to persist intermediate results, and an untrusted

network/IPC to receive requests and send results. An attacker can then compromise unsafe interfaces, take advantage of insecure shared memory, or exploit synchronisation vulnerabilities [38–41] for extracting cryptography keys, bypass attestation [42], or even attacking the host OS [43]. Vulnerabilities like Boomerang [43] and HPE (Horizontal Privilege Escalation) [44], show how the *semantic gap* between different compartments (in this case user space processes and TrustZone TAs) leads to severe privilege escalation threats. Boomerang exploits a confused deputy vulnerability inside a TA, via the shared memory between the two worlds, and takes advantage of secure world privilege to make the host kernel memory accessible. Similarly, HPE attacks demonstrate different ways that an adversary process could compromise another user space process through a shared TA or insecure privilege management between two TAs.

These direct attacks through untrusted interactions are only one of the security threats, which are also easier to detect. However, most attack vectors in such complex environments are much harder to investigate or mitigate. For instance, consider attack vectors that are related to one-way/fixed trust models, like when an enclave is fully trusted. Then a compromised or malicious enclave is allowed to collect sensitive data and leak them using OS facilities such as files and network [45, 46]. Moreover, attackers could compromise another application process via a misbehaving third-party TA/enclave [44]. As shown by previous works, such a large attack surface can not be mitigated with ad-hoc security patches. Therefore, as long as we do not have a principled approach to re-visit our system stack, naively relying on hardware security features could be a large-scale security mistake [47]. Effective hardware-assisted compartmentalization should reduce such attack vectors and help to understand and resolve the limitations of existing hardware security features and system software.

1.2 A taste of dispersed compartments

Security and performance improvements through hardware-assisted security have led to more practical application compartmentalization techniques [48–53, 35, 54, 55, 28, 4, 56]. These systems aim to reduce the attack surface by dividing the application into isolated components that communicate via well-defined channels. However, the isolation, communication, and execution resources could be provided through multiple software/hardware mechanisms. As a result, we are moving toward computing environments where application developers not only have the possibility of using traditional forms of compartments such as processes [48], but also TEE/enclave-assisted compartments and also intra-address space compartmentalization (e.g., intra-process [57–59, 7] or intra-enclave [60–62]).

We call such computing model a *hetero-compartment* environment. In a hetero-compartment environment, each compartment type could have its own threat/security model, designed by an independent vendor, and does not have enough knowledge about other compartment types. For instance, consider a single platform simultaneously running regular applications (isolated via processes), sandboxed applications (e.g., isolated via Wasm [63]), and TrustZone-based TAs.

Each of these three compartments has a different threat model, privileges, and isolation mechanism. Together they form a simple hetero-compartment environment, which becomes more complicated when enabling ARM CCA to allow realm-based compartments provided by different vendors or adding CHERI-based compartments into the system (as in ARM Morello [64]). Indeed in the near future, hetero-compartment environments will be more and more complex because of such hardware features and heterogeneous SoC architectures (e.g., the combination of RISC-V and ARM-based TEEs in i.MX8ulp [30]).

Hetero-compartment computing, in theory, gives developers great freedom to design secure software based on their specific performance, workloads, and security requirements. In practice, though, the current outcome is quite different. The development experience is extremely challenging, and it particularly is hard to reason about whole-system security. In fact, there are many vulnerabilities *inside* and *across* such heterogeneous isolation boundaries that cannot easily be observed, debugged, or mitigated by existing designs.

Current partitioning frameworks mainly focus on facilitating code refactoring and porting applications to isolated compartments. However, unsafe refactoring can just as easily lead to *new* attack vectors [43, 38, 42], over-privileged compartments with potential vulnerabilities, and a high cost in re-engineering and overhead [65–67]. For instance, previous studies show a wide range of attacks on most enclave/TEE frameworks due to vulnerabilities inside an over-privileged enclave [38, 42, 44] or insecure data/resource sharing between enclaves and processes (two distinct compartments) [43, 38, 41, 42].

Developers, therefore, have difficulty avoiding over-privileged compartments, especially in applications with large/complex codebases. As expected, these results caused a familiar line of research on intra-enclave/TEE privilege separation, following previous techniques for in-process isolation, either through software or hardware mechanisms [68, 60, 62, 55]. Unfortunately, though beneficial, nested compartment architectures do not have a proper security model for supporting strong cross-compartment security policies, which eventually causes these designs to fail. Our system foundations are not ready for hetero-compartment computing. Introducing additional types of compartments to a hetero-compartment environment by itself does not resolve these non-trivial issues.

For instance, figure 1.1 illustrates six compartment types (i.e., user space process, enclave, TA, in-process, in-enclave, and in-TA) and how vulnerabilities can easily be introduced and propagated in the cross-compartment boundaries (within the different or same address spaces). Detecting and protecting against such a large attack surface is one of the primary obstacles to securely adjusting to the ever-growing hetero-compartment environment. Here, ad-hoc approaches are neither practical nor scale well. Combining per-attack security patches is limited to a few attack vectors and it is not a principled approach to deal with future security threats. Particularly when migrating compartmentalized applications to another platform or compartment

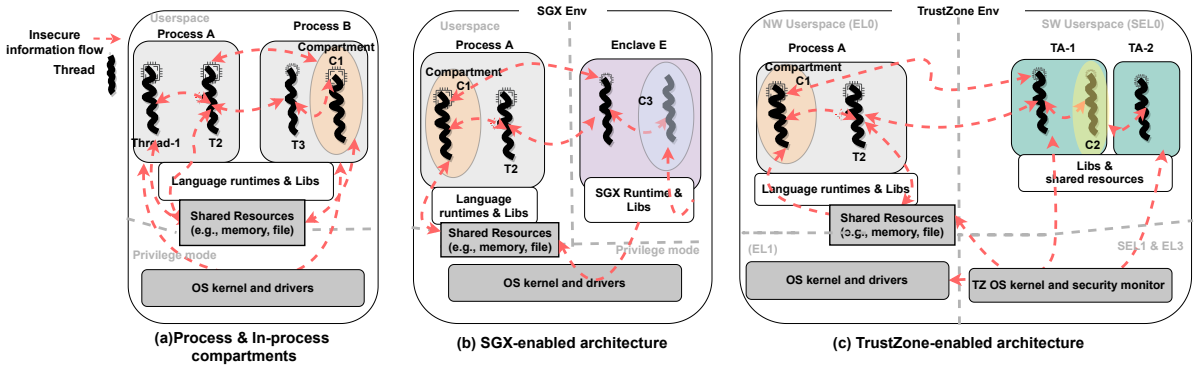


Fig. 1.1 Few simplified attack vectors in hetero-compartment environments.

type (e.g., from SGX enclaves to TrustZone TAs and realm). A naive compartment migration could widen or worsen the attack surface [43].

Therefore, we believe the traditional single-compartment and coarse-grained security model in existing system software, such as per-process protection in UNIX OSs, is no longer sufficient to protect modern applications and properly utilize hardware security features. It is time for a principled way to introduce the concept of hetero-compartment computing to the underlying OSs (and even to hypervisors and firmware) and extend them with new security primitives and compartmentalization abstractions to enable the following fundamental characteristics:

Extensible. An extensible privilege separation mechanism should enforce *arbitrary* security policies over system objects either within the *same* or *different* address spaces. Achieving this requires to *simultaneously* tackle three fundamental issues by providing: *(i)* flexible security policies based on mutual-distrust, *(ii)* fine-grained isolation, and *(iii)* cross-privilege boundaries monitoring and enforcement.

Auditable. One of the significant obstacles for developers to securely adjust to hetero-compartment computing is the lack of a principled way to investigate attack vectors, audit, and reason about various security threats within applications' compartments. Unlike information flow control (IFC)-based systems, permission-based or object capability-based compartmentalization focus on enforcing strong isolation but are not designed to systematically track and monitor sensitive information flows within applications compartments or their interactions with heterogeneous compartments alongside the host resources.

Hardware-assisted security & efficiency. Achieving extensibility and audibility without properly using modern hardware leads to significant performance overhead, weaker security, or a much larger TCB (trusted computing base), particularly in a hetero-compartment environment. Effective compartmentalization should not neglect the significant efforts to support more efficient hardware support for isolation, memory safety, and privilege separation [28, 56, 69, 5, 70, 64]; instead, it must extend the OS (and possibly other systems software) with new abstractions to sufficiently utilize these hardware features.

This dissertation presents the design and implementation of a full OS-based compartmentalization framework for achieving the above characteristics. To this end, we first introduce the concept of *dispersed compartments* as a new technique to enable applications to operate within and across privilege boundaries securely.

More specifically, dispersed compartments are arbitrary isolation units consisting of CPU and system resources (e.g., threads, address spaces, files, etc.) but with three notable differences from traditional isolation units such as user space processes, sandboxes, or enclaves. *(i)* each of these isolation units have an inadequate view of the system, so the protection and control over their resources and data can not scale beyond their trust boundary. This means there are weak or no security guarantees when sharing data or resources with other processes or enclaves; that is a major limitation for hardening complex applications. On the other hand, dispersed compartments could contain and protect resources from different address spaces so that they can operate as a fine-grained container for secure data sharing and interactions between mutually distrustful execution units. *(ii)* unlike a process or enclave that does not allow any further privilege separation inside their address space, dispersed compartments enable efficient intra-address space isolation by utilising hardware-based memory protection. *(iii)*, a dispersed compartment does not follow a fixed security model and set of policies, and it can dynamically change (shrink or extend) its trust boundaries at runtime.

As an example, consider a TrustZone-assisted ML framework that wants to protect sensitive data, model, or inference by porting sensitive functionalities inside a TA/enclave. As shown in code sample 1.1, that is a part of TrustZone-based TVM framework [71], developers may use shared memory and other shared resources, which are located in the untrusted world, for efficient data and model parameter transfer to the TA. Since encrypted channels and interactions are expensive and inconvenient, many use cases overlook the security threats such as leaking model parameters in this case.

Using dispersed compartments we can ensure that only enclave thread T_e owns a protected shared memory block M (`mem` in listing 1.1), which means only T_e can grant or revoke M associated privileges to another dispersed compartment thread. This way, even if the host application's unsafe parts (e.g., untrusted third-party libraries) can not compromise M or other sensitive system objects (e.g., files, sockets, etc). Listings 1.2 shows simple usage of our SIRIUS API for creating a dedicated dispersed compartment for secure interactions with T_e (L2-4).

```
1  void optee_cmd_LoadModelParamBlob(struct optee_ctx *ctx ,char* param_blob, size_t
       param_blob_size){
2    TEEC_Operation op;
3    TEEC_SharedMemory mem;
4    uint32_t origin;
5    TEEC_Result res;
6    memset(&op, 0, sizeof(op));
7    op.paramTypes = TEEC_PARAM_TYPES(TEEC_MEMREF_WHOLE,
8             TEEC_NONE,
9             TEEC_NONE,
10            TEEC_NONE);
```

```
11    memset(&mem, 0, sizeof(mem));
12    mem.buffer = (void*)param_blob;
13    mem.size   =  param_blob_size;
14    mem.flags  =  TEEC_MEM_INPUT;
15    TEEC_RegisterSharedMemory(&ctx->ctx, &mem);
16    op.params[0].memref.parent = &mem;
17    op.params[0].memref.size   = mem.size;
18    op.params[0].memref.offset = 0;
19
20   res = TEEC_InvokeCommand(&ctx->sess ,TA_ML_WITH_TEE_CMD_LoadModelParamBlob,
21           &op, &origin);
22    if (res != TEEC_SUCCESS)
23      errx( "TEEC_InvokeCommand(LoadModelParamBlob) failed 0x%x origin 0x%x",
24        res, origin);}
25
26  int main(int argc, char** argv) {
27    //more initialisation
28    struct optee_ctx ctx;
29    memset(&ctx, 0 , sizeof(optee_ctx));
30    optee_init_ta(&ctx);
31    optee_cmd_TVMModCreateFromCModule(&ctx);
32    // Load the model graph structure
33    // Load parameters of model matrixies to TA
34    size_t params_bytes;
35    char* parameter_blob = LoadFile(params_path, &params_bytes);
36    optee_cmd_LoadModelParamBlob(&ctx, parameter_blob, params_bytes);
37    // get the results and inference
38   }
```

Listing 1.1 An example of host process-TA interactions in TustZone-assisted ML inference.

Similarly listings 1.3 shows that with small modifications inside TA, we can define a dynamic security policy over shm_ae shared memory to ensure restricted access, layout/permission modifications, or memory allocation/deallocationnoun. This guarantees a secure per-thread interactions with the host T_a (L3-7). Particularly in multithreaded use cases, such thread-granularity capabilities could mitigate non-trivial attack vectors such time-of-check to time-of-use (TOCTOU) [72] or AsyncShock [41] where dispersed compartments could enforce safe dataflows between threads across privilege boundaries. The API details with more examples are explained in Chapter 5.

```
1  //user space process side
2  int main(int argc, char** argv) {
3    //create a dispersed compartment
4    c1 = s_create(S_LABEL);
5    //add the current thread object to c1
6    s_add(c1, self);
7    //pass c1 metadata to bind it with an enclave
8    e = s_create_enclave(c1, &ctx); //create an enclave
9    optee_cmd_TVMModCreateFromCModule(&ctx);
10   // Load the model graph structure and parameters to TA
11   optee_cmd_LoadModelParamBlob(&ctx, parameter_blob, params_bytes);
12   // get the results and inference}
```

Listing 1.2 TrustZone-assisted ML example: the host process side

```
TEE_Result TVMGraphExecutor_LoadParamsWrapper(uint32_t param_types, TEE_Param params[4]) {
    // initialisation
    c2 = s_create(S_LABEL);
    shm_id = s_vao_create(DEFAULT);// create a protected VA (virtual address space) object
    s_add(c2,vao,shm_id) // add the VAO to c2
    shm_ae = s_malloc(shm_id,1024*1024);
    s_grant(c1->t_a, shm_id,MEMDOM_READ | MEMDOM_WRITE);
    // the rest of loading parameter functionality
    return TEE_SUCCESS;
}
TEE_Result TA_InvokeCommandEntryPoint(void __maybe_unused *sess_ctx,
      uint32_t cmd_id,
      uint32_t param_types, TEE_Param params[4])
{
  (void)&sess_ctx; /* Unused parameter */
  switch (cmd_id) {
  case TA_ML_WITH_TEE_CMD_LoadModelParamBlob:
    return TVMGraphExecutor_LoadParamsWrapper(param_types, params);
    // the rest of TA ecalls
  default:
    return TEE_ERROR_BAD_PARAMETERS;
  }
}
```

Listing 1.3 TrustZone-assisted ML example: TA/enclave side

Hence, dispersed compartments offer strong security principles toward achieving effective compartmentalization. However, it is challenging to enable and audit them on underlying system and commodity hardware without causing significant overhead. Therefore, we propose a novel unified set of OS abstractions and user space framework, called SIRIUS, to efficiently enable dispersed compartments on commodity hardware. SIRIUS enforces user-specified confidentiality and integrity policies across heterogeneous isolation boundaries and traces dispersed compartments alongside their arbitrary system objects (e.g., address space, threads, files, IPC/RPC, etc.). SIRIUS provides developers with low-level building blocks and security primitives for fine-grained protection, tracking compartments, and combining/migrating application compartments while requiring small code changes and overhead.

SIRIUS's OS abstractions and hardware-based security primitives could be utilised to design and build *domain-specific* solutions for enhancing the security and privacy of modern edge-cloud use cases. As an example, we describe the design and implementation of a SIRIUS-assisted TEE framework. Unlike existing TEE systems, it provides strong defense-in-depth against vulnerability propagation and non-trivial attacks within interaction layers [38, 42–44, 41, 73]. As another example, we utilise SIRIUS for reducing major attack vectors on sensitive IoT applications such as wearable brain computing interfaces (BCIs).

1.3 Contributions

We propose the possibility of significantly reducing the attack surface of modern applications through a novel compartmentalization mechanism. We first briefly introduce the following new primitives which shape the foundation of our compartmentalization solution:

- **Dispersed compartments.** These novel forms of compartments have an elastic characteristic to extend or shrink their view of the underlying system and support dynamic trust models based on mutual distrust. Each dispersed compartment guarantees protection and control over an arbitrary set of fine-grained system objects within the same or different address spaces.

- **Dispersed monitoring.** We need a mechanism to maintain ownership and control over system objects across different privilege boundaries. Dispersed monitoring enables dispersed compartments to achieve this goal by mapping high-level principals to underlying system objects and monitoring flows of information within them. This approach enables each dispersed compartment to control and monitor its resources across multiple trust boundaries without fully trusting a single central entity such as OS kernel.

- **Dispersed enforcement.** Enforcing dispersed compartment policies on commodity hardware in a practical and secure way needs a *decentralised* enforcement mechanism. Our decentralised mechanism ensures different privilege levels on the device could protect their mutually-distrustful resources. It helps separating policy assignment and monitoring from achieving strong enforcement guarantees. This enables a more flexible and modular solution; which is particularly useful for resource constrained use cases. We introduce dispersed enforcement as a new hardware-assisted approach for achieving dispersed compartments fine-grained and extensible features with configurable TCB and relatively small overhead.

We evaluate these new primitives and their role in delivering effective compartmentalization by first implementing SIRIUS, a novel framework for secure, efficient, and high-performant implementation of dispersed compartments, monitoring, and enforcement. To resolve security and performance challenges of achieving SIRIUS goals, we have implemented and evaluated a set of new hardware abstractions, OS extensions, and user space libraries. Then we utilise SIRIUS to significantly reduce attack surfaces in various sensitive use cases and to build security-sensitive systems such as a TEE framework on different platforms. The following summarises this thesis contribution:

- A comprehensive investigation of attack surfaces in diverse range of modern applications on the edge-cloud environment alongside the limitations of existing compartmentalization and isolation mechanisms. Our analysis has led to introducing new design principles and

security primitives to enable dispersed compartments for contemporary applications on modern hardware and system software.

- Introducing three new hardware-assisted abstractions for enabling intra-address space isolation, dispersed monitoring, and dispersed enforcement efficiently and securely. These abstractions allow proper use of underlying modern hardware such as TEEs, MMU features, virtual memory tagging, memory protection keys (or memory domains), TLB tagging. We implement and evaluate these abstractions on AArch32-64 and x86-64 Linux. We show that these abstractions are lightweight and practical, even for mobile/IoT devices with resource constraints. These abstractions are essential building blocks of our system-wide compartmentalization framework.

- Design and implementation of SIRIUS, a novel compartmentalization system based on principles of dispersed compartments, monitoring, and enforcement. SIRIUS enables dispersed compartments on commodity hardware, allowing developers to systematically reason about non-trivial threats on hetero-compartment environments (e.g., TEE-enabled systems), detect them, and guard against them via multi-layer defense-in-depth *within* and *across* privilege boundaries. SIRIUS's hardware-assisted implementation and optimization of dispersed monitoring and enforcement allows dispersed compartments to achieve extensibility and audibility while encountering reasonable TCB, performance overhead, and memory footprint. It is the first system to efficiently achieve this enforcement level without requiring custom hardware or specific language features (e.g., Java classes).

- An extension to the Linux kernel to achieve SIRIUS design principles. We modified the Linux kernel, with new security architecture based on mutual distrust and a hetero-compartment model suitable for efficient implementation of dispersed compartments. We further added new abstractions for SIRIUS's hardware-assisted isolation and privilege management. Our implementation of SIRIUS required modifying several parts of the kernel (e.g., memory management, file system, networking, task management) to efficiently add our new security primitives while handling conflicts with the existing model to maintain compatibility for legacy applications.

- A prototype of SIRIUS-assisted TEE frameworks to evaluate the effectiveness of dispersed compartments on reducing the attack surface of existing TEE-assisted systems as a primarily example of hetero-compartment environment. Our TEE framework provides strong defense-in-depth against non-trivial threats on sensitive host-enclave data and control flow (e.g., Van Bulck et al. [38], COIN [42], Boomerang [43], HPE [44], Iago [73]). In this thesis, our evaluation mainly focuses on TrustZone- and SGX-based (that are currently the two primarily enclave models) hardware and TEE systems.

- An extensive evaluation of SIRIUS by compartmentalizing and attack investigation in real-world use cases on the edge-cloud environment. We targeted a diverse range of use cases for proper evaluation and optimization, including web services, databases, cryptography libraries, data distribution services, wearables, and machine learning frameworks. We show that SIRIUS user space API activates fine-grained and flexible policies for these applications with relatively small code changes.

1.4 Thesis Outline

Chapter 2 describes the current state of ever-growing complexity on modern software and hardware stacks and its affect on security threats. It presents background material on compartmentalization techniques, their limitations, and challenges of improving them from different perspectives, which are crucial to understanding SIRIUS. Chapter 3 gives a bottom-up view of SIRIUS and dispersed compartments. It first describes details of modern hardware-based isolation and privilege separation techniques and their limitations. These architectural details help to understand dispersed compartments and our hardware-assisted abstractions. We then provide a formal definition of dispersed compartments and explain their security principles before presenting our three hardware-assisted abstractions. Chapter 4 presents the system view of SIRIUS and shows how to build system software suitable for dispersed compartments principals and based on mutual distrust for supporting various compartment types. This chapter describes details of SIRIUS OSs and TEE designs and implementation. Chapter 5 presents our user space stack and API. It presents details of utilising SIRIUS to reduce attack surfaces of different use cases and explains how SIRIUS can be integrated to widely-used systems. Together our system and user space stacks make dispersed compartments security principles a practical reality. Finally, Chapter 6 contains concluding thoughts and discussion about our approach and compartmentalization technique.

Chapter 2

The ever-growing attack surface

The number of security vulnerabilities and incidents is alarmingly increasing every year, partly because of better awareness and tooling for finding vulnerabilities, but more importantly, due to the ever-growing unhandled complexity of modern software and hardware stacks. Today's applications not only rely on numerous third-party libraries, but they also rely on complex hardware, language runtimes, and systems software for various performance, compatibility, and security reasons. However, as this complexity increases over time, it would be more and more difficult to reason about attack vectors and fundamentally mitigate them within a short time.

This chapter elaborates on the concept of complexity and its consequences on security (§2.1). It argues why compartmentalization is one of the most effective techniques for reducing complexity and improving security and discusses existing primarily compartmentalization solutions (§2.2). Then we thoroughly investigate the limitations of existing privilege separation techniques by exploring the attack surface of multiple widely used real-world applications in different settings(§ 2.3), particularly hetero-compartment circumstances. Our analysis focuses on applications threat model, security architecture, dependencies, existing vulnerabilities, and security patches. Importantly, we will show that naive sandboxing or privilege separation techniques may extend an application attack surface instead of reducing it.

2.1 Complexity's role in security

Historically, systems and security experts argued that complexity leads to security problems. For example, MITRE's CWE (Common Weakness Enumeration) considers excessive attack surface weakness (CWE-1125) as a member of complexity issues (CWE-1226) and caused by excessive code complexity (CWE-1120).

Hence, various metrics are proposed to measure software complexity, including lines of code (LOC), McCabe's Cyclomatic Complexity (MCC) [74], Halstead's metrics [75], function points [76], and architecture coupling measures [77]. Each of these static code attributes tries to measure complexity from different perspectives. For example, LoC is a size-based complexity metric that counts each physical source line of code (including all third-party dependencies) in a

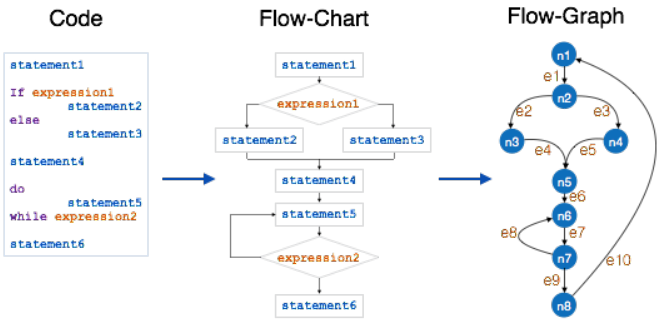
program. On the other hand, Halstead considers lower-level factors for measuring complexity by using operands and operators, and based on four scalar numbers derived directly from a program's source code: $n1$ (the number of distinct operators), $n2$ (the number of distinct operands), $N1$ (the total number of operators), and $N2$ (the total number of operands) as shown in Figure 2.1.(a). Excessive Halstead complexity (CWE-1122) makes it more difficult to understand and/or reason about the code security and vulnerability propagation. It also makes it easier to introduce vulnerabilities while much harder or time-consuming to find and/or fix vulnerabilities.

McCabe, in 1976, proposed cyclomatic complexity measure to quantify complexity of a given software based on decision-making constructs that change the flow of the program such as `if-else`, `do-while`, `switch-case` and `goto` statements. This metric argues that between two programs of the same size, the one with more decision-making statements will be more complex as the program's control jumps frequently. MCC requires to break the program into smaller blocks, delimited by decision-making constructs, and construct its flow graph as Figure 2.1.(b) demonstrates. Then to calculate MCC, we use $V(G) = e - n + 2$ where e is the total number of edges, and n is the total number of nodes. Excessive MCC issue (CWE-1121) means the code contains MCC that exceeds a desirable maximum (often above 10).

These metrics are already used in a variety of use cases for measuring the quality of software. For instance, previous work shows the Linux kernel (version 2.6.37) includes more than 800 functions with MCC values above 50, and 369 functions with MCC of 100 or more [78]. They discuss that 76% of functions with highest MCC come from the drivers and others coming from `arch` (8 functions), `fs` (9 functions), `sound` (3 functions),`net` (3 functions), and `crypto` (1 function) parts of the kernel. Similarly these measures are used for predicting more vulnerable parts of large codebases [79] or comparing security of programming languages [80].

Classification	Variable	Description
Variable	n_1	Number of Operators
	n_2	Number of Operands
	N_1	Total Operators
	N_2	Total Operands
	n	$n_1 + n_2$
	N	$N_1 + N_2$
Complexity Formula	$V = N \times log_2 n$	Program Volume
	$D = \frac{n_1}{2} \times \frac{N_2}{n_2}$	Program Difficulty
	$E = D \times V$	Effort
	$B = \frac{E^{\frac{2}{3}}}{3000}$	Estimated Bugs

(a)



(b)

Fig. 2.1 (a) Halstead complexity measures. (b) McCabe's Cyclomatic Complexity measure based on program's flow graph [1] (image credit: [2]).

However, such metrics are still far from ideal for measuring complexity, and the *exact* correlation with security issues is not straightforward. For instance, another research's initial

results show that these complexity measures have a weak correlation with security problems for Mozilla JavaScript Engine [79]. They distinguish between faulty code, which is an accidental condition that causes a functional unit to fail to perform its required function and a vulnerability that is an instance of a fault such that its execution can violate an implicit or explicit security policy [81]. Using these metrics, they show that vulnerable functions are more complex than faulty functions. This fact indicates that the fault prediction models that use such complexity metrics also could be useful for vulnerability prediction in general. Despite being useful, these metrics can not capture more fundamental aspects of complexity, including architectural complexity. For instance, previous work proposes limited architecture coupling metrics (direct, indirect, and cyclic coupling) on the Google Chromium project [77]. Their findings show a strong relationship between existing Chromium vulnerabilities and architecture coupling metrics. Yet, they conclude that the effects of different types of coupling are hard to distinguish.

Therefore, these metrics are unable to capture the actual architectural, semantic, and logical complexity of modern applications (though they still can somewhat show the affects of complexity on security). Over time, application designs and architectures are fundamentally changed due to the rise of cloud-based services, mobile devices, IoT, and domain-specific hardware alongside open-source software. Contemporary applications rely on a wide range of techniques and dependencies to improve portability (e.g., unikernels [82], containers, lightVMs [83]), performance (e.g., microservices, kernel-bypass mechanisms [84]), or security (e.g., sandboxing, compartmentalization, trusted hardware) that significantly widen their attack surface with non-trivial security threats.

2.1.1 The end of the eternal war?

As hardware and software stacks change over time, many attack vectors are fully mitigated or significantly reduced, while new security threats are introduced in various layers of abstractions. For instance, though memory corruption attacks [85] are still one of the major security issues, their types and scale are radically changed. The widespread use of memory-safe languages and various sets of defenses significantly reduced different classes of such attacks. For instance, as Figure 2.2 illustrates, since 2017, the number of traditional memory corruption vulnerabilities has reduced by 45.2%, such as buffer overflows (reduced by 40%), while code execution based vulnerabilities increased by 15.3% that can also happen in memory-safe languages due to inadequate access control or authorization issues (e.g., CVE-2021-27198, CVE-2020-24639, CVE-2018-1000622, or CVE-2019-19810). This does not imply that memory vulnerabilities are not important anymore but only shows that attackers exploit and combine them for more powerful and non-trivial forms of attacks with greater speed than before.

The current results prove that as long as we rely on legacy code written in memory unsafe languages such as most system software, including language runtimes, OSs, or hypervisors, it is not possible to completely win the eternal war against memory attacks [85]. For example, as Figure 2.3 shows, most memory corruption issues are still among the most exploited vulnerabili-

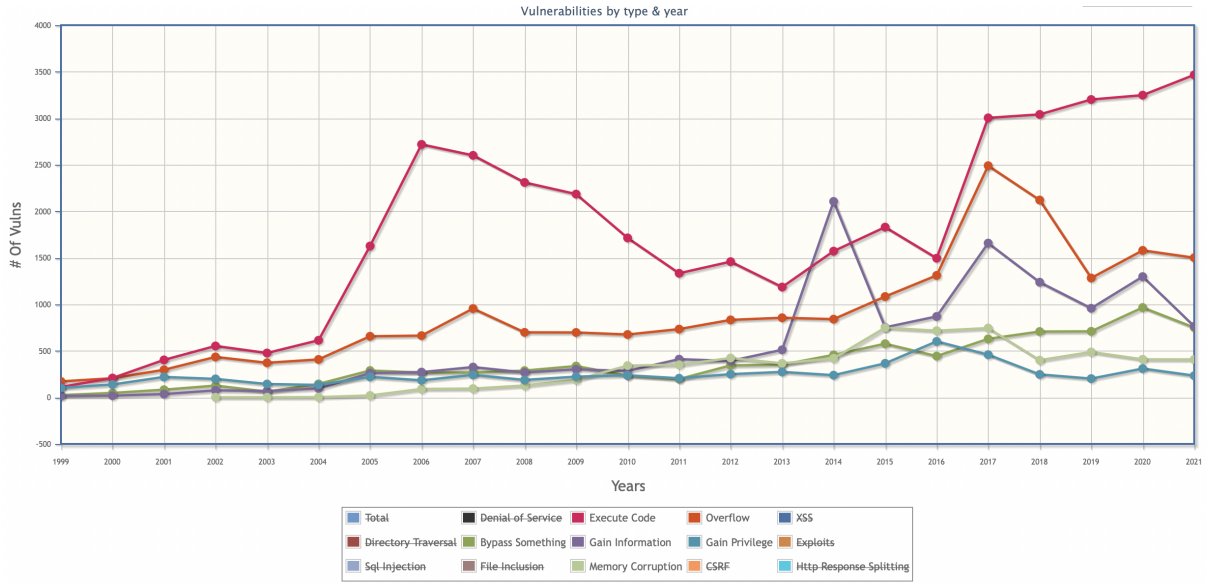


Fig. 2.2 Memory vulnerability classes over time (image credit: [3]).

ties such as out-of-bounds read/write (CWE-125/CWE-787). Any unsafe binding or dependency to these attack vectors undermine the security of our system even if the majority of it is written in safe languages (see Table 2.1).

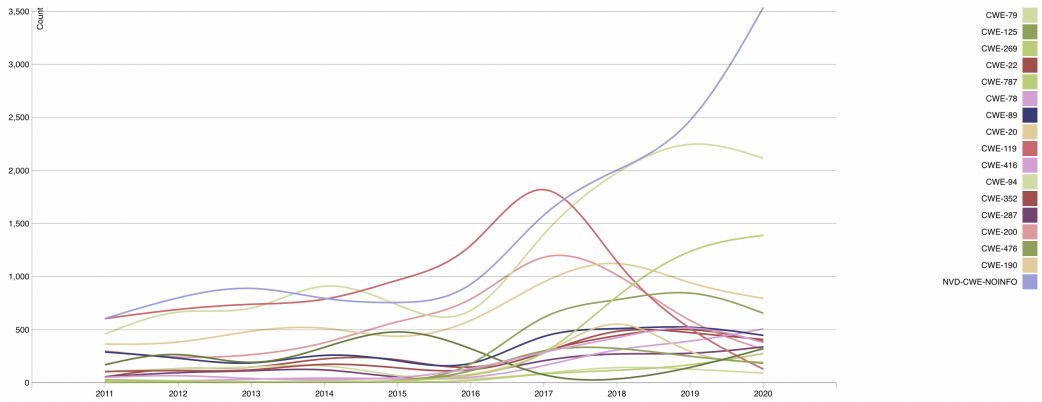


Fig. 2.3 Mostly exploited vulnerability classes over the past decade (image credit: [3]).

Therefore, propagation of such vulnerabilities in abstraction levels and their combinations with other attack vectors, particularly without improper privilege management (CWE-269), are causing an alarming increasing rate in non-trivial attack surfaces (e.g., CWE-NOINFO in Figure 2.3). In fact, after about two decades, access control and privilege separation issues are once again considered the number one security problem by leading security organizations (e.g., MITRE and OWASP). This highlights the need for more flexible, extensible, and pervasive compartmentalization techniques that fully consider the complex nature of the modern application.

Vulnerability	Language	Description	Vulnerability type
CVE-2021-29922	Rust	access control bypass when handling IP address string	Bypass
CVE-2021-28879	Rust	Overflow in the standard library in Rust	Info leak
CVE-2021-31162	Rust	Memory corruption in the standard Library in Rust	Mem corr, info leak
CVE-2021-28875	Rust	Overflow from unsafe context due to not validating the return values	Overflow, Mem corr
CVE-2020-2604	Java	Vulnerability in the Java SE serialization component.	Priv esc, bypass, code exec
CVE-2021-0481	Java	Possible access of unauthorized files due to an unexpected URI handler	Priv esc, bypass,
CVE-2021-2302	Java	Vulnerability in Java security platform handling network access	Info leak, bypass,
CVE-2018-9838	OCaml	Overflow in byterun/bigarray.c in the standard library in OCaml	Mem corr, Exec Code, Overflow
CVE-2017-9772	Ocaml	Privilege escalation due to compiler insufficient sanitisation	Code Exec
CVE-2020-13388	Python	A vulnerability in the configuration-loading functionality of the jw.util	Code Exec
CVE-2021-3177	Python	Buffer overflow in PyCArg_repr in _ctypes/callproc.c	Code Exec, Mem corr. overflow
CVE-2020-5311	Python	Buffer overflow in libImaging/SgiRleDecode.c	Mem corr. overflow
CVE-2021-28363	Python	Improper SSL certificate validation in the urllib3 library	Info leak, bypass

Table 2.1 A few selective vulnerabilities in memory safe languages and their dependencies.

2.1.2 Compartmentalization for security

compartmentalization and privilege separation are techniques to separate monolithic software or hardware into isolated modules while giving each module only the necessary privileges. These techniques are historically considered the primary weapon to reduce complexity and improve security and reliability, mainly by limiting vulnerability propagation from one code piece to another. compartmentalization aims to ensure that a security compromise or failure in one untrusted part of the system can not lead to the whole system compromise. There are various software- and hardware-based methods of compartmentalization, but they all have the following key requirements:

Resource isolation. Each compartment should have its own share of hardware resources, such as CPU and memory for running programs. Depending on designs and implementations, each compartment's resources could be managed by the compartment or the host systems software such as OS or hypervisor. Hence, in most designs, the host system could access and modify the compartments' resources, which is much restricted in TEEs. However, it is necessary to isolate the compartment's sensitive resources from another compartment in all forms of compartmentalization. Usually, isolation refers to memory protection from confidentiality and integrity perspectives. These refer to whether a compartment is allowed to read or write another compartment's data or code. Therefore, isolation rules and directions vary depending on the trust model between compartments. For example, most sandboxing techniques follow a one-way trust relationship where only the in-sandbox code is untrusted; thus, it is restricted from interacting and accessing outside resources. This dissertation considers more comprehensive aspects of compartmentalization with a focus on mutually distrustful compartments. However, our approach and results could be applied to diverse compartmentalization forms.

Privilege separation. In addition to isolating compartment resources, there should be an access control mechanism to assign each compartment with the least necessary privileges. This includes defining security policies to control which system resources a compartment can access, share, or revoke. Depending on design goals, security policies could cover various system

services such as file system, networking, and IPC and ideally should follow the principle of least privilege for protecting each resource and interaction.

Data/resource sharing. Often compartments need a way to share and communicate with each other or any other execution units (e.g., processes) outside so that they can work together to finish a computing task. Usually, this is the only efficient way to obtain all the required data and resources. However, context switching from one compartment to another execution unit requires a controlled mechanism to maintain the essential computing states and ensure to follow the security policies. For example, the compartment crossing should not grant unnecessary and unauthorized permissions to another domain when switching execution states. Typically, a trusted component such as OS kernel is responsible for securely saving the state of one compartment (e.g., core registers) and restoring another during a controlled invocation (e.g., via a system call). Hence, usually, compartmentalization systems provide restricted per-compartment call gates for communicating with the outside world. IPC mechanisms over sockets, pipes, or shared memory are a well-known example of such communication means where userspace processes are the unit of isolation.

Enforcing security policies. Enforcing isolation and security policies can be done via either software- or hardware-based techniques. However, a combination of hardware and system software is often involved for proper enforcement. For example, userspace processes are isolated via hardware-based virtual memory while the OS kernel enforces access control policies over other resources such as file systems, networking, and IPC. Furthermore, security policies can be enforced either *statically*, which is fixed after initialization, or *dynamically*, which is modifiable during runtime. Ideally, a compartmentalization technique should ensure that attackers can not bypass the enforcing mechanism directly or indirectly (e.g., through data races during communications or cover or side-channel attacks).

2.2 Existing compartmentalization solutions

2.2.1 Hardware privilege rings/levels

Multiple privilege layers or rings of protection were among the primary security concepts introduced by the Multics OS [86], which is inherited by all commodity OSs including today's Unix family. Software running in each execution mode has specific privileges for accessing resources and handling events (e.g., exceptions or interrupts) and can only communicate with another layer through dedicated mechanisms. For instance, historically, applications run in userspace mode separated from the OS that runs in supervisor (or kernel) mode. This allows the OS to manage hardware resources and handle userspace requests while limiting userspace applications to abuse the entire system resources or compromise other processes. Hardware-based execution modes are one of the oldest and coarse-grained compartmentalization techniques supported by most modern CPU architectures.

ARM privilege levels The ARM architecture supports several privilege levels (PLs) or exception levels (ELs), where EL0 (called unprivileged execution) has the lowest software execution privilege. EL1 provides supervisor mode for running OS kernel, EL2 provides support for processor virtualization, and EL3 provides support for two security states (Figure 2.4). Execution can only move between exception levels on taking an exception or returning from an exception. When taking an exception, the exception level either increases or remains the same (it cannot decrease). But, when returning from an exception, the exception level either decreases or remains the same (it cannot increase). Each privilege mode has its own set of banked registers. For instance, in EL*x* (where *x* is the exception level), the stack pointer (SP) maps to the SP_EL*x* stack pointer register.

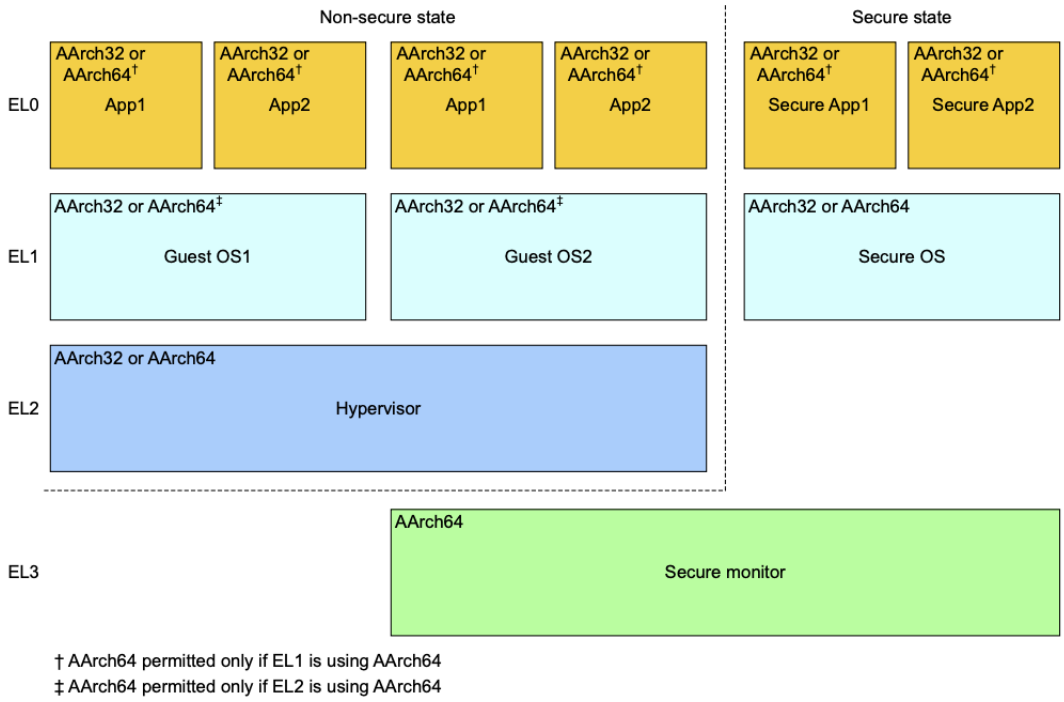


Fig. 2.4 Privilege levels in ARM Cortex-A aarch32/64 (image credit: [4]).

x86-64 rings Briefly, the x86 architecture provides four privilege levels, numbered from 0 to 3, where a greater number means less privilege. The highest privilege level 0 is used for segments that contain the most critical code modules in the system, usually the OS kernel. The outer rings (with progressively lower privileges) are used for segments that contain code modules for less critical software. Code modules in lower privilege can only access modules operating at higher privilege via tightly controlled call gates. Attempts to access higher privilege without going through a call gate and without having sufficient access rights causes a general-protection exception (#GP) to be generated and handled by privileged code.

The call gate descriptor provides access rights information, the segment selector for the code segment of the called procedure, and an offset into the code segment that is the instruction

pointer for the called procedure. Each privilege level has its own stack and core registers. The segment selector and stack pointer for the privilege level 3 stack are stored in the SS and ESP registers, respectively, and are automatically saved when a call to a more privileged level occurs. The segment selectors and stack pointers for the privilege level 2, 1, and 0 stacks are stored in a system segment called the task state segment (TSS). From the calling procedure perspective, the use of a call gate and the TSS during a stack switch are transparent, except when a general-protection exception is raised. There are other modes of operations in modern x64-64 such as SMM (System Management Mode) and ME (Management Engine) that we do not explain here and their complete details can be find in the IA-32/64 architecture manual [87].

RISC-V privilege modes RISC-V by default supports three privilege levels that are encoded as a mode in one or more CSRs (control and status registers) as shown in Table 2.2. The machine-level has the highest privileges and is the only mandatory privilege level for a RISC-V hardware platform. Code run in machine-mode (M-mode) is usually inherently trusted, as it has low-level access to the machine implementation. M-mode is often used to manage secure/trusted execution environments on RISC-V. User-mode (U-mode) and supervisor-mode (S-mode) are intended for conventional application and operating system usage, respectively. Each privilege level has a core set of privileged ISA extensions with optional extensions and variants. For example, machine-mode supports an optional standard extension for memory protection via physical-memory protection (PMP) scheme. Readers can find details of each privilege mode in the RISC-V architecture manual [70].

Privilege level	Name	Abbreviation	Encode
0	Userspace	U	00
1	Supervisor	S	01
2	Reserved	-	10
3	Machine	M	11

Table 2.2 Privilege levels in RISC-V architecture

2.2.2 Memory isolation

The need for memory protection in the multi-programmed computer system (MCS) led to various mechanisms for virtualizing and isolating memory regions. Here we focus particularly on solutions at the hardware and compiler levels.

Segmentation & virtual memory

Segmentation is one of the oldest and primary methods of memory virtualization and processes isolation [88]. Each segment is an ordered set of the smallest unit of stored information (e.g., word). The early implementation of segmentation technology by x86 processors (beginning with the Intel 8086) did not provide any protection. Any program running on those processors could access any segment with no restrictions. In those architectures, a segment is only identified

by its starting location, and there was no length checking. However, in modern implementation, a segment is defined with its base address, definite length (which may vary with time), and permissions. Each memory access is made with respect to a segment at a given offset. When accessed, the MMU adds the segment's base to the offset to form a physical address. MMU then compares the offset to the segment length and checks permission bits. Memory faults occur by lack of permissions, accessing out-of-bounds, or because the segment is not located in memory. Due to performance issues or limited segment support, segments are often used for major program regions, such as binary text, data, stack, etc., instead of fine-grained objects.

Page-based virtual addressing (VA) is widely used in isolating processes and privilege abstractions. When enabled, it splits both virtual and physical memory into equally-sized pages (often 4KB). This enables page granularity VA spaces and enforces per-page access control. Virtual memory relies on the processor MMU that works with the cache memory system to control access to and from external memory. The MMU also controls the translation of VAs to physical addresses. As an MMU structure, a pagetable maps virtual pages to physical ones. To access each page, the MMU translates a part of the address (e.g., the page number) from virtual to physical by looking up in a pagetable. In addition to address translation, the MMU controls memory access permissions, memory ordering, and cache policies for each region of memory. Each pagetable entry (PTEs) also includes permission bits and the mode a processor has to be running in to use them. Virtual memory enables processes to have no knowledge of the physical memory map of the system or about other programs that might be running simultaneously, as shown in Figure 2.5. This enables the same virtual memory address space to be used for each program. It also works with a contiguous virtual memory map, even if the physical memory is fragmented. This VA space is separate from the actual physical map of memory in the system. Applications are written, compiled, and linked to run in the virtual memory space.

Since page-table walks per memory access are slow, modern architectures support TLBs as associative caches to accelerate access. Depending on the design, a TLB may be software- or hardware-managed. Each architecture provides different levels of address translation depending on the instruction sets and memory layout. There are few mechanisms to prevent requiring a TLB invalidation on a context switches such as using Application Space IDentifier (ASID), memory domains, or memory protection keys that we explain more in Chapter 3.

Modern architectures allow virtual memory enabling in various privilege layers. For example, in ARM, it is possible to enable parallel VA spaces in EL0, EL1, EL2, and EL3 using different TTBR registers as shown in Figure 2.6. The same mechanism enables hypervisors to isolate multiple OSs from each other. For example, Armv8-A virtualization introduces a second stage of address translation. Hence, the hypervisor must perform some extra translation steps to share the physical memory system between the different guest operating systems in a two-stage process. In the first stage, the VA is translated to an Intermediate Physical Address (IPA). This is usually

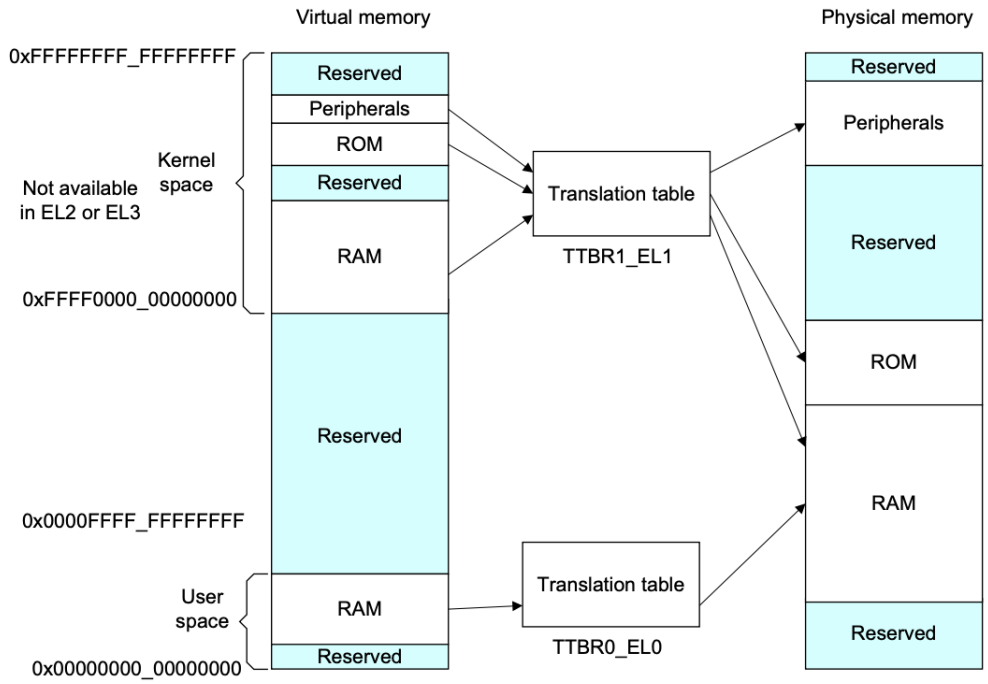


Fig. 2.5 Separate pagetables for userspace and kernel in ARMv8-A (image credit: [5]).

under OS control. A second stage, which is controlled by the hypervisor, translates the IPA to the final physical address (PA).

IBM POWER9 platform has different addressing mechanim [89], where threads use 64-bit effective addresses, which are formed of an effective segment identifier and offset to access different storage objects. The address is valid if it includes the effective logical partition identifier that uniquely identifies the processing thread. Conversion of a 64-bit effective address to a virtual address is done by searching the Segment Lookaside Buffer (SLB). The segment table is managed by the operating system running in a logical partition (LPAR) and resides in virtual memory. The SLB specifies the mapping between Effective Segment IDs (ESIDs) and Virtual Segment IDs (VSIDs). The number of SLB entries is implementation-dependent, except that all implementations provide at least 32 entries. Lastly, the page-based translation mechanism converts the virtual address to a 56-bit real address.

To summarise, VA-based isolation is one of the fundamental techniques that is used by many widespread compartmentalization systems (e.g., process-based sandboxes). Despite having performance or isolation granularity issues (page granularity as the minimal unit of protection) in such systems, VA-based privilege separation significantly improved the security of our systems.

Hardware-based bounds checking and guarded pointers

There are diverse hardware features for memory bounds checking and limiting the corruption of pointers [90–95]. To name a few, HardBound [90] proposes a hardware bounded pointer

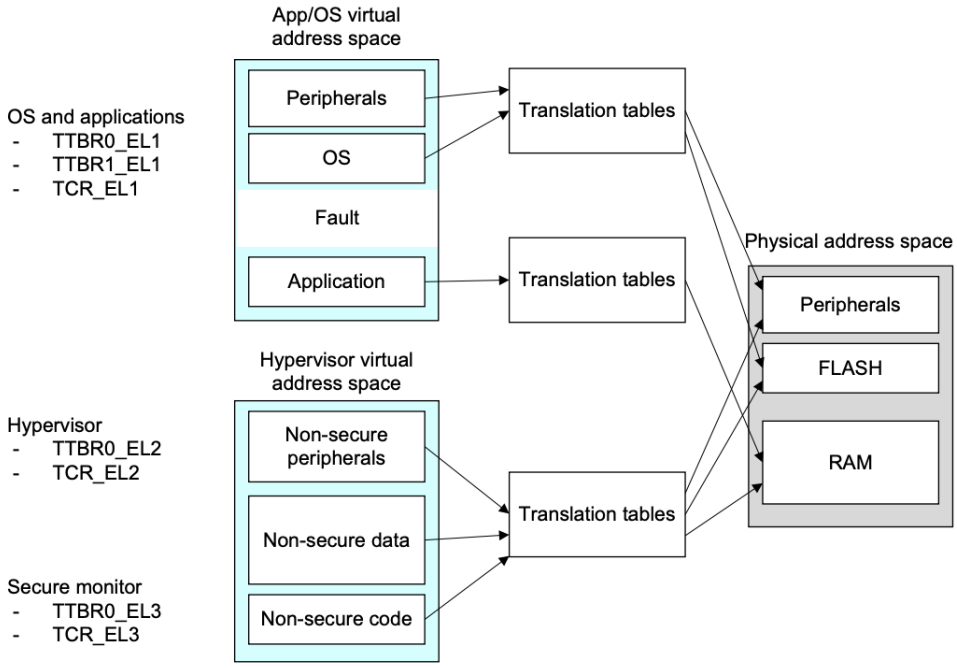


Fig. 2.6 Simplified virtual memory address translation in ARMv8-A(image credit: [5]).

architectural primitive and an x86-based hardware-software codesign for providing spatial memory safety for C programs. Similarly, In 2015, Intel introduced Memory Protection Extensions (MPX) [93] as a set of new ISA extensions as part of the Skylake microarchitecture[1]. At its core, MPX provides seven new instructions and a set of 128-bit bounds registers (see Figure 2.7). The architecture provides four registers named bnd0-bnd3, each with a lower bound in bits 0-63 and an upper bound in bits 64-127. In general, MPX requires a cross-layer design involving *(i)* the hardware instructions and registers to operate on pointer bounds, *(ii)* OS abstraction to support memory management and exception handling, and *(iii)* compiler and runtime layer that adds instrumentation passes and programming wrappers, and *(iv)* MPX-specific changes in the application layer. Figure(2.7.b) shows a simple usage of MPX, how the bounds for the array a[] are created (L2), or how two MPX bounds checks are inserted to detect if a[i] overflows (L5-6).

However, Intel MPX instructions cause up to 4x slowdown in the worst case and about 50% on average. MPX does not offer protection against temporal memory safety errors and does not support multithreading inherently, leading to unsafe data races in legacy multithreaded programs. Moreover, it does not support some C/C++ programming idioms due to restrictions on the allowed memory layout, causing non-trivial manual fixes. Due to these issues, alongside conflicting with some other ISA extensions (resulting in even worse performance issues), the feature has now been withdrawn from GCC and the Linux kernel [6].

[1]Intel MPX is now removed from the Linux kernel (the only OS that supported it) and GCC compiler due to the lack of sufficient adaptation despite the maintenance cost.

Instruction	Description
`bndmk b,m`	create pointer bounds
`bndcl b,m`	check mem-operand against lower bound
`bndcl b,r`	check reg-operand against lower bound
`bndcu b,m`	check mem-operand against upper bound
`bndcu b,r`	check reg-operand against upper bound
`bndmov b,m`	move pointer bounds from memory
`bndmov b,b`	move pointer bounds to other register
`bndmov m,b`	move pointer bounds to memory
`bndldx b,m`	load pointer bounds from BT
`bndstx m,b`	store pointer bounds in BT

(a)

(a) Original code

```
struct obj { char buf[100]; int len }
1 obj* a[10]                                   ;; Array of pointers to objs
2 for (i=0; i<M; i++):                         ;; M may be greater than 10
3   ai = a + i                                 ;; Pointer arithmetic on a
4   objptr = load ai                           ;; Pointer to obj at a[i]
5   lenptr = objptr + 100                      ;; Pointer to obj.len
6   len = load lenptr
```

(b) Intel MPX

```
1  obj* a[10]
2  a_b = bndmk a, a+79                         ;; Make bounds [a, a+79]
3  for (i=0; i<M; i++):
4    ai = a + i
5    bndcl a_b, ai                             ;; Lower-bound check of a[i]
6    bndcu a_b, ai+7                           ;; Upper-bound check of a[i]
7    objptr = load ai
8    objptr_b = bndldx ai                      ;; Bounds for pointer at a[i]
9    lenptr = objptr + 100
10   bndcl objptr_b, lenptr                    ;; Checks of obj.len ⎤
11   bndcu objptr_b, lenptr+3                  ⎦
12   len = load lenptr
```

(b)

Fig. 2.7 Summary of Intel MPX instructions and simple usage [6].

As another example, Capability Hardware Enhanced RISC Instructions (CHERI) extend existing ISAs with hardware-defined capabilities. CHERI-MIPS is currently the most mature implementation of CHERI though both CHERI-RISCV and CHERI-ARM (a.k.a. Morello) are under development. CHERI capabilities are fixed-length register-sized values allowed to reside anywhere in memory. The validity and integrity of capabilities are protected via memory tagging. A capability has an address, bounds, permission set, type, and tag. Although the tag can be read from a register, it is not visible in memory to be written (sub-capability writes could clear the tag). Figure 2.8 shows the current CHERI MIPS format (different versions of CHERI have different capability formats) is 64-bit (without the tag), with a 32-bit address space. All memory accesses must use a capability with bounds covering the selected location and the correct access permissions.

CHERI capabilities demonstrate great potential for mitigating most memory safety issues. Particularly, CHERI's recent progress on improving its performance and supporting many legacy applications with $0.1\% - 0.5\%$ code changes are very promising [94]. However, it still requires significant modifications in all the layers of stacks: hardware, OS (only supports FreeBSD),

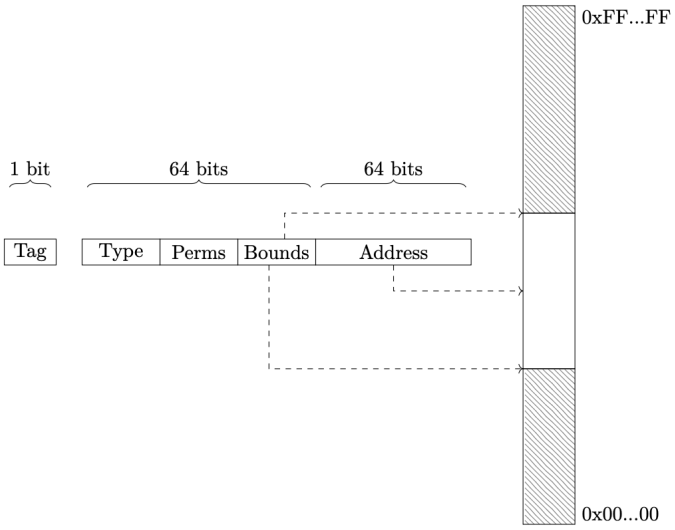


Fig. 2.8 A CHERI capability [7]

compiler, language runtimes, and applications. Hence, it is not feasible to utilize it in many current use cases.

Memory safe languages

Memory safety at the language level is a high-level construct to safeguard access to in-memory objects, which often requires the correct language and compiler design, alongside the assistance of hardware, OSs, and runtimes for security and efficiency matters.

Some languages or extensions support both spatial and temporal memory safety, while some focus on improving one of them. Hence, safe languages such as OCaml, Java, Rust, or even safe dialects of C such as CCured [96], Cyclone [23], or CheckedC [22] can eliminate many memory vulnerabilities using strong type systems and runtime checks.

Fortunately, safe languages are increasingly popular at the application level. However, they still suffer from memory-safety issues in practice due to unsafe compilers, dependencies, and runtimes (as summarised in Table 2.1). More importantly, for legacy applications, it is required to rewrite programs entirely in a safe language or port legacy code to safer C extensions to take advantage of these languages. This requires a huge engineering cost for complex software, which in the best case, leads to protecting some parts while trusting large safety-critical dependencies written in unsafe languages. Particularly, replacing commodity OSs like Linux or macOS with safer alternatives remains open to research. For instance, the singularity [97] effort to provide a libraryOS written entirely in C#, and Mirage OS [82] in OCaml, needed years of engineering. Despite the cost, these systems are mostly used in a few domain-specific use cases, such as web services, and are not widely adopted.

Besides, compared to C, these languages add much more performance overhead and are often less flexible for low-level programming and systems use cases. Recently, Rust showed great potentials as a safe while high-performant alternative for C. As a result, there have been

several proposals for integrating Rust into the Linux kernel for secure implementation of kernel drivers and modules [98, 99]. However, these are still at early stages to elucidate performance and security benefits.

2.2.3 Access control & sandboxing

Access control and sandboxing mechanisms are foundations of computer security that can be deployed at a number of levels. We will focus on the primary solutions at the OS and application levels.

Historically, OS-level access controls, though limited, are widely used to implement and model protection mechanisms. The simplest form of access controls typically authenticate security principles (e.g., users) and then restrict their access to system objects such as files, networking ports, and other system resources by looking at the access permissions inside access control matrices with columns for objects and rows for users. Such matrices (whether in single or multiple dimensions) do not scale well since they quickly need handling substantial entries that are incredibly inefficient and error-prone in modern systems. Hence, there are mechanisms to compress the principles and the permissions by using groups or roles to manage the privileges of large sets. Other alternatives are access control lists (store the access control matrix by columns) and capability/ticket-based systems (store the access control matrix by rows) [100–102].

Access control list or ACL stores the access control matrix a column at a time, along with the resource to which the column refers. ACLs are the basic access control mechanism in Unix-based systems such as GNU/Linux, Apple's OS/X, and Windows, though now all have become more complex over time. ACLs are suited to environments where access control policy is set centrally and requires limited data-oriented protection. They are not suitable where the number of principals is large, subjects/objects are constantly changing, or where users require more complex policies (e.g., revoking permissions or delegations for a period of time). Though ACLs are simple to implement, they are not efficient for security checking at runtime since the OS must either check the ACL at each file access or keep track of the active access permissions in some other way [50].

Another way to manage the access control matrix is to store it by rows that form capability models. More specifically, a capability is an unforgeable token of authority along with an associated set of access rights to objects. Unlike ACLs, runtime security checking here is more efficient as well as delegating a privilege (by just passing a capability). However, managing a large number of capabilities in a complex system is a non-trivial task from a security and performance perspective. Once a capability is passed, the owner could easily lose control over the object; either privilege revocation or controlling further delegations are difficult and costly [102].

Conventional OSs, however, mostly rely on a form of Mandatory Access Control (MAC) designed to enforce a broad class of access control policies. MAC implementation schemes could include multi-level security (MLS), type enforcement (TE), or dynamic role-based access

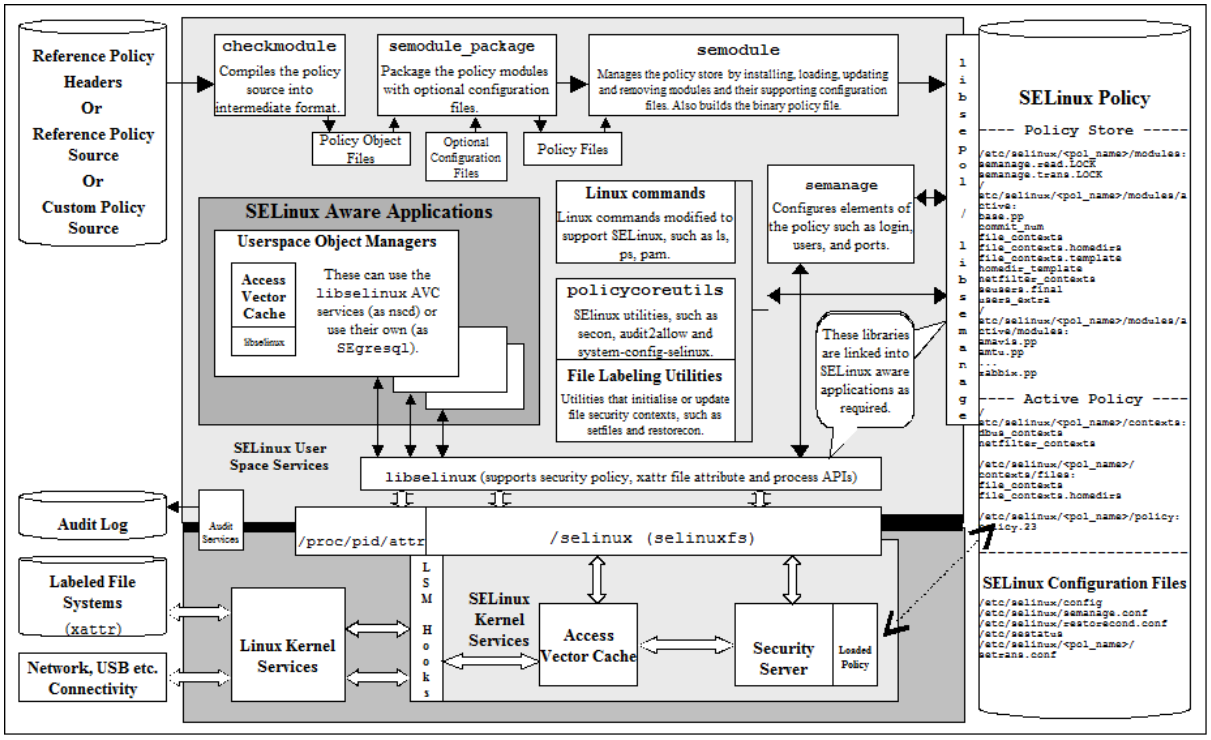


Fig. 2.9 SELinux architecture (picture credit: [8]).

control (RBAC). In 2001, SELinux was introduced as a MAC implementation for the Linux Kernel [103] and then also adapted for Android [104]. Despite significant progress in providing userspace tooling [105], SELinux policies are not widely adopted, mostly due to inflexibility and difficulty of proper usage. This leads to disabling SELinux or only using its default security policies that are not suitable for many use cases. Additionally, as Figure 2.9 shows, over time, SELinux has been adding notable complexity to various layers of the kernel and userspace. Therefore, to support variants of MACs, several frameworks were introduced, such as LSM (Linux kernel module)-based MACs and FLASK [106] architecture in 2002. LSM provides a set of kernel extension hooks to facilitate the integration of systems such as SELinux without committing the Linux operating system to a particular model (see Figure 2.10). LSM now supports several MAC implementations such as AppArmor, Smack, Landlock, or Tomoyo that could be used for different use cases. In 2003, TrustedBSD [107] presented a similar architecture to FreeBSD. However, these traditional MACs provide limited and coarse-grained protection [8]. As a result, more fine-grained isolation techniques have been proposed alongside MACs to support more comprehensive protection.

For instance, to provide a more fine-grained and extensible solution, Capsicum [108] framework introduced a capability-based sandboxing mechanism to FreeBSD. Before Capsicum, most capability systems remained as research work and were never widely adapted to commodity OSs [102, 109]. Capsicum extends standard UNIX file descriptors to assign and check capabilities. Processes which enter capability mode are denied direct access to global namespaces

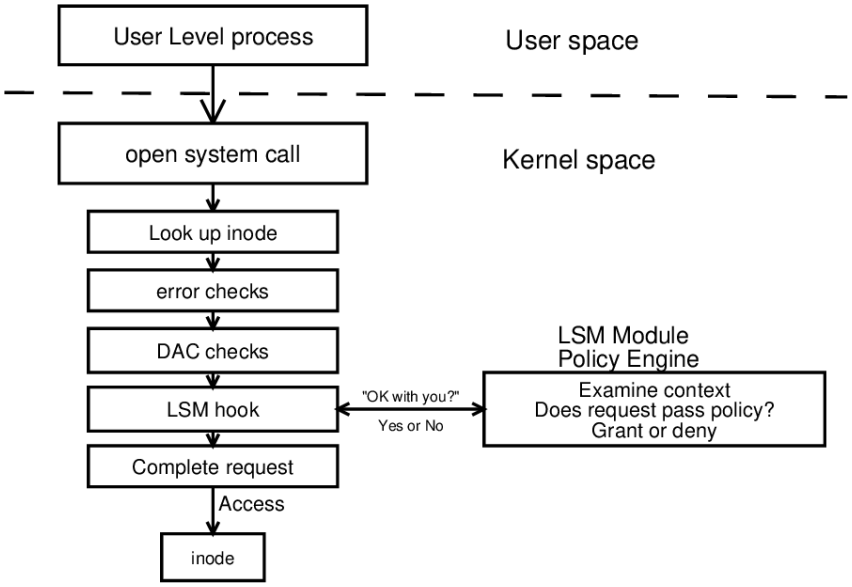


Fig. 2.10 LSM simplified architecture.

such as the file system and process IDs. Instead, they may only access resources through their own capabilities. Capsicum allows capabilities to be inherited from the parent process or passed between sandboxes via IPC channels such as sockets. The framework userspace library, libcapscium, helps to develop and run sandboxed code by providing support for loading sandboxes and a runtime environment that emulates much of the traditional UNIX API inside the sandbox via RPC proxy to some operations (such as DNS lookups) and via a more privileged process. Though the overhead of the extra capability checks in system calls was reasonable, the overall cost of sandboxing complex applications was high, a problem in common with other sandbox modes relying on process isolation such as chromium [110] or wedge [48]. That is why the CHERI project expanded on the ideas of the Capsicum, pushing its capability model down into an RISC-based hardware architecture to achieve finer-grained security more efficiently [7].

Several forms of process-based sandboxing are used in various use cases. Besides their weak security guarantees and large overhead, such mechanisms require redesigning an application from scratch using a multiprocess architecture (e.g., Chrome) that is impractical for most multithreaded applications such as web servers [111–113]. Though systems such as Privtrans [114] and Wedge [48] provide more automatic sandboxing, they have inflexible security models and add a huge overhead (up to 40x slowdown in some cases). Such systems' impracticality is partially inherited from the conventional process abstractions such as `fork` that already suffer from various efficiency and security issues [115]. Even `fork` alternatives such as `clone`, are not flexible enough for fine-grained data sharing between processes for security-critical resources.

Process namespaces are also another mechanism to isolate and control OS resources. These techniques are based on the old concepts in resource virtualisation (e.g., Plan 9 [116] namespaces introduced in 1992). In Linux, kernel namespaces form an isolation layer by creating different

userland views. Namespaces split the traditional kernel global resource identifier tables and other structures into their own instances. This partitions processes, users, file systems, IPCs, network stacks and other components into separate analogous pieces to provide processes a unique view. Then different namespaces can be bundled together in any frequency or group to create a filter across resources for how a set of processes view the system as a whole. As a similar but less comprehensive method, FreeBSD Jails [117] provides isolated filesystem namespaces (using `chroot`) and isolated processes and network resources in such a way that a process might be granted root privileges inside the jail but blocked from performing operations that would affect anything outside the jail. Similarly, Solaris Zones proposed isolating processes into groups that could only observe or signal other processes in the same group, associates each zone with an isolated file system namespace, and defines restrictions for shared resources [118].

Containers such as LXC[2] and Docker[3] also provide coarse-grained sandbox environments utilizing a combination of OS features, including namespaces, cgroups, MACs, and seccomp filtering. However, this technology is designed for convenient application management and packaging (e.g., deploying microservices to the cloud) rather than proper isolation or privilege separation. Currently containers are widely used as lightweight alternatives of hardware-based virtual machines and suffer from weak security guarantees [119–121].

2.2.4 Hypervisor-based privilege separation

Many systems proposed software- or hardware-based isolation techniques to reduce the TCB from large OSs to much smaller hypervisors. However, this assumption is not entirely valid anymore since existing hypervisors are also complex and have large codebases due to ever-increasing functionalities [122].

Terra [123] was the first system that introduced the concept of close-box and open-box VMs, and proposed a mechanism to protect close-box VMs against the compromised host OS. It provides the enclaved VMs with services such as remote attestation and sealed storage through a trusted VM manager (similar to xen dom0). Terra's hypervisor and VM manager handle all access controls, enforce security policies, and isolates VMs from each other. Terra's slow boot and high overhead are partly because of its design as well as the dependency to legacy VMware GSX. Proxos [124] also protects applications from the host OS by allowing them to specify their sensitive system calls and then handling them via private OS isolated from the commodity OS. Hence the host OS cannot access the application secrets through system calls. The Private OS runs as an isolated VM from the commodity OS. Proxos provides a simple routing language for developers to define private system calls. It incurs a relatively high overhead, especially for IO intensive workloads. Bitvisor [125] also follows a similar approach, though focuses only on applications IO security on the untrusted OS. It applies shadow DMA descriptors for capturing IO data transferred by DMA and directly passes it to the host hypervisor.

[2] https://linuxcontainers.org

[3] https://www.docker.com

As another example, Overshadow [126]'s hypervisor presents an encrypted view of applications' memory pages to the host OS or another application; hence, it allows the OS to handle all resource management without compromising secrecy and integrity. Applications communicate with the hypervisor through a user-level shim via a hypercall interface, for example, to save or resume the execution state. To provide multiview pages, Overshadow needs to know the semantics of guest OS. Similarly, SP3 [127] introduce a secure domain that guarantees the secure execution of applications by encrypting their memory pages contents. Applications specify the pages' access permissions, and according to these permissions, the hypervisor determines which image of the page should be encrypted or decrypted.

Flicker [128]'s hypervisor protects the integrity of applications' sensitive modules using TPM hardware, though the high cost of frequent TPM switches limits its usability for arbitrary applications. TrustVisor [129] improves Flicker performance by including the hypervisor within the TCB and minimizing TPM operations (mostly for sealed storage). Similarly, InkTag [130] uses hardware virtualization and nested paging for isolating applications' secure pages. It uses a technique similar to paravirtualization called "paraverification" to detect OS is not compromised, then allows only paraverified operations on secure pages. Paraverfication requires the OS to inform hypervisor about security critical changes such as page table updates. Security of this approach is weak since verifying the behavior of a complex commodity OS is proven to be hard.

These systems had significant progress toward protecting VMs against compromised host OS. However, they require a significant overhead, particularly for supporting fine-grained compartmentalization. They also fully trust the host hypervisor or use complicated nested virtualization techniques to limit this trust. However, hypervisors suffer from large attack surfaces and the existing research on building more secure hypervisors [131] or reducing their attack surface [132, 133, 83, 134] is still work-in-progress.

2.2.5 Trusted execution environments

Trusted execution environments (TEEs) are one of the primary examples of the current trend in hardware-assisted privilege separation and isolation. TEEs aim to protect applications' sensitive compartments even in the presence of malicious OS or hypervisor.

In 2009, ARM TrustZone [26] or security extensions was introduced with ARMv6 and currently is enabled in all ARM Cortex-A processors. More recently, it is also supported in Cortex-M processors though the implementation is different. The TrustZone architecture allows the partitioning of all hardware resources, including the processor, memory, and peripherals, into two execution environments (as shown in Figure 2.11). The resource partitioning to secure and non-secure (S/NS) can be done using TrustZone AXI components. For example, TrustZone Address Space Controller (TZASC) partitions DRAM, and TrustZone Memory Adapter (TZMA) is used for partitioning SRAM. TrustZone Protection Controller (TZPC) configures peripherals. Both worlds have their own user space and kernel space, and only the normal world has hyp mode. The Generic Interrupt Controller (GIC) also supports both worlds. Secure Monitor

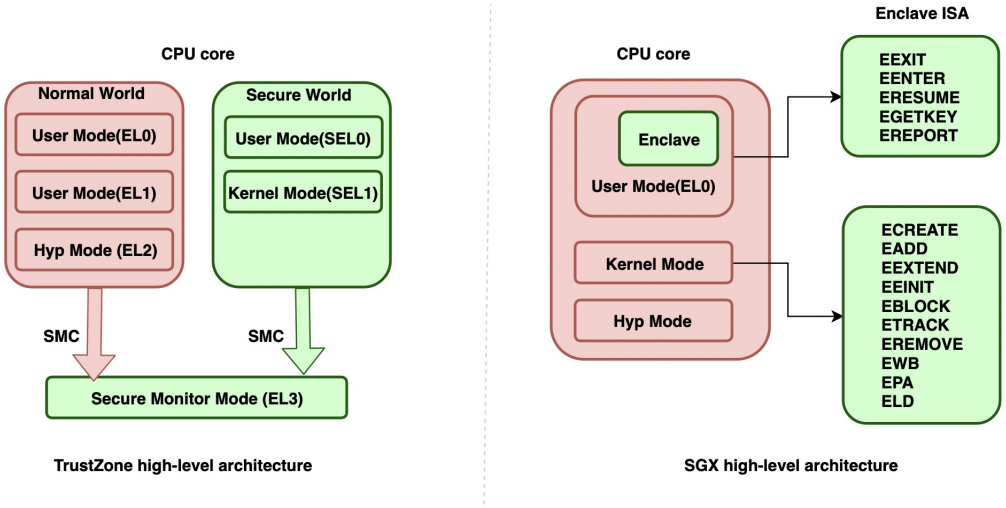


Fig. 2.11 TrustZone & SGX high-level Architecture

running in monitor mode (EL3) is responsible for communication between two worlds via a special instruction called "Secure Monitor Call" (SMC). Though TrustZone has only one secure world, we can create unlimited isolated regions using secure world MMU or virtualizing secure world [68, 135].

Intel then proposed software guard extensions (SGX) [136] in 2013[4]. SGX enclaves only run in userspace, and the host OS handles enclave resource management, creation, initialization, and cleanup. 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).

As Figure 2.11 shows, SGX adds several instructions extensions for enclave handling. Most of these instructions can only be accessed in privileged mode. For example, 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) their enclaves explicitly. An enclave may also be exited asynchronously due to interrupts or exceptions (AEX). In the

[4]In 2021, Intel announced the deprecation of SGX from the 11th and 12th generation Intel Core Processors, but development continues on Intel Xeon for cloud and enterprise use [137]. Since researchers found many side-channel vulnerabilities in SGX, Intel had more to gain from its deprecation from the security perspective.

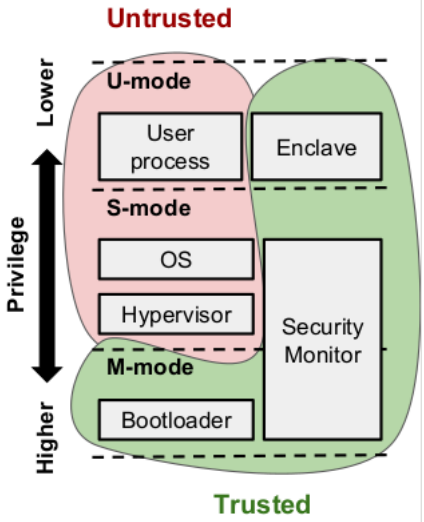


Fig. 2.12 Sanctum (Keystone) high-level Architecture (picture credit: [9])

case of asynchronous exits, the enclave's secrets will be protected. An enclave can also request to get its signed measurement (EGETKEY, EREPORT) for remote or local attestation. SGX hardware is vulnerable to different side channel attacks [138–141].

Sanctum [9] mainly tries to provide all SGX features while defending against cache-based side-channel attacks. Sanctum Security Monitor manages enclaves resource, maintains their state, and enforces the security policies such as access control. For example, the monitor loads enclaves page tables so the OS can not infer enclaves memory access patterns. Since the security of Sanctum enclaves depends on the monitor and the measurement root, these modules are running in RISCV machine mode, the highest privilege-level (similar to TrustZone secure monitor mode), as shown in Figure 2.12. Sanctum minimally extends RISC-V Rocket chip to isolate physical addresses by dividing DRAM into regions and allocating each to an enclave exclusively (See Figure 2.13). In Sanctum architecture, unlike SGX, cache channel attacks are prevented since each enclave has private cache set and exclusive control over its DRAM regions. Similarly, Keystone [35] proposes a RISC-V TEE design that is inspired by TrustZone and Sanctum architectures with a focus on hardware customizability and formal verification [142].

Iso-X [143] also offers the SGX security guarantees by small ISA extension to the OpenRISC platform. Confidentiality of Iso-X compartment's pages is protected through encryption, while their integrity is ensured by storing each page measurement in the internal Iso-X data structures. Unlike SGX and similar to Sanctum, it supports per-enclave page tables with a dedicated page walker. Iso-X also allows the dynamic addition of new pages to the compartment that is supported only in SGX2.

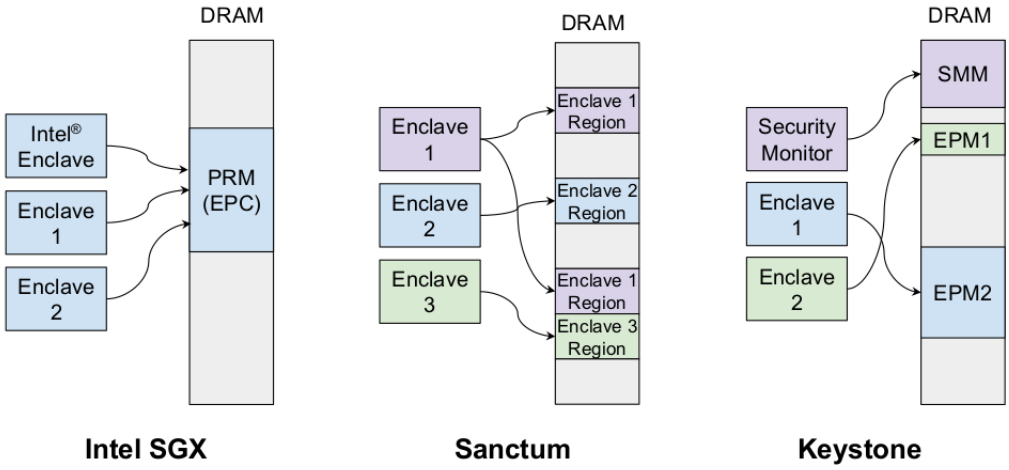


Fig. 2.13 Memory management in SGX vs Sanctum vs Keystone (Picture credit: [10])

More recently, CheriOS [144] introduces the concept of "Foundations" to CHERI architecture [7] that are similar to enclaves. A foundation starts with no capabilities referring to it and no capabilities in it referring to other data. The nanokernel assigned IDs for entering a founded code. Each foundations memory can be locked and measured for attestation process. A very interesting feature of this system is the ability to create a hierarchy of trust by creating nested foundations. CheriOS TCB contains a nanokernel that gives reservations (allocating capabilities) and ensures secure propagation of capabilities.

AMD SEV [145] utilizes cryptography mechanisms to isolate guest VMs from each other and from the host untrusted hypervisor. It allows the memory contents of a VM to be transparently encrypted with a key unique to the VM. The memory controller contains a hardware AES-128 engine for encryption/decryption of DRAM. The key management and secure data transfer between the host hypervisor and guest VM memory are handled by the SEV firmware running AMD Secure Processor. SEV only protects guest VMs against a benignly malicious hypervisor and is shown to be easily broken [146]. Recently, Intel Trust Domain Extensions (TDX) [27] are introduced to achieve a similar goal as AMD SEV through Intel Multi-Key Total Memory Encryption (MKTME) technology. MKTME is a technology that allows transparent memory encryption in upcoming Intel platforms. It uses a new instruction (`PCONFIG`) for key setup and selects a key for individual pages by repurposing physical address bits in the page tables.

Utilizing these TEEs could be beneficial in various use cases like OS kernel protection [147], auditing [148], security reference monitors [149], web services [150–152], secure payments [153–156], databases [157], autonomous vehicle control [158], and privacy-preserving machine learning [152, 159–163]. However, the right abstractions for securely and properly utilizing TEEs/enclaves are still not clear; particularly when combining them in the same SoC or integrate other hardware features inside enclaves (e.g., memory safety hardware features). For example, depending on the use cases, software vendors provide deployment frameworks for running

diverse workloads within enclaves. Some applications, like confidential data analytics, benefit from userspace enclaves, while other use cases like micro-services and lightVMs benefit from hypervisor-assisted isolation of trusted domains (TDs).

Hence, we see deployment frameworks, such as Red Hat's Enarx or Amazon Nitro enclaves, which support running the same binary within enclaves in different privilege layers. These systems are inspired by Haven [164] that ports Drawbridge [165], a Windows library OS, inside an SGX enclave. Since a large portion of applications system support is provided by 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. Similarly, Graphene-SGX [66] ports the Linux-based Graphene library OS in an enclave. Scone [67] tries to reduce TCB by porting musl libc and a portion of Linux Kernel Library (LKL)[5]. Scone exposes larger external interfaces. However, the asynchronous system call mechanism slightly improves the performance overhead. TrustShadow [166] also protects unmodified applications from the host OS following the overshadow system [126]. It traps applications exceptions and systemcalls inside TrustZone, and after parameter marshaling, passes the calls to the OS and verifies the output. TrustShadow's slowdown for each systemcall varies, though in the worst case, the slowdown is around 3x for signal handling calls. The same approach will not have a reasonable overhead on SGX, mainly because TrustZone allows shared memory, and SMC calls are less expensive than SGX Ocalls.

These designs force developers to run large code in a single enclave, resulting in inefficient and over-privileged enclaves. This leads to wide range of new attacks due to exploiting in-enclave vulnerabilities [41, 42], insecure interactions with outside [73, 38, 42], or misusing enclaves to escalate privileges [43, 44, 167]. Currently there is no systematic way to neither detect nor protect against these threats.

As another approach, various TEE partitioning frameworks are proposed to split applications into two trusted and untrusted components; such as Intel's SGX SDK [136], Microsoft's Open Enclave [53], Google's Asylo [54], OP-TEE [33], and Keystone [35]. There are also language-specific partitioning frameworks such as Civet [55] for porting Java classes into an SGX enclave, TLR (Trusted Language Runtime) [65] for running portions of C# applications inside TrustZone, and Glamdring [168] a compiler for partitioning applications into SGX enclaves via code annotation. However, none of these fully consider the complex attack surface originating from insecure interactions between the host OS, userspace processes, and enclaves. Civet uses dynamic taint-tracking to control the flow of objects on enclave interfaces but can not help for proper privilege separation and against more complex attacks (e.g., horizontal privilege escalation (HPE) attacks [44]). It also relies on a specific language. Moreover, handling the key issue of over-privileged enclaves requires in-address space compartmentalization, which is not considered in these works.

[5] sgx-lkl https://github.com/lsds/sgx-lkl.

Additionally, while TEE-assisted applications are typically structured around a compartmentalized design, existing TEE environments force them to an "all-or-nothing" trust model that wastes this knowledge. There have been attempts to provide more flexible TEEs (e.g., multiple enclave types [35, 37])), finer-grained isolation (e.g., in-enclave libraries [60, 169]), additional functionality (e.g., shared memory [170], multi-processing [62]). However, All of them come at the cost of notable hardware change, significant slowdown, dependency to specific programming language, or a large TCB. This is not surprising because any additional functionality or extensible isolation granularity requires bridging the *semantic gap* or lack of sufficient system view across privilege boundaries [43, 44]. This requires the globally centralized TEE implementation to map high-level programmatic constructs, that could belong to different privilege layers, to low-level hardware protection.

In summary, existing TEE solutions have significant security and performance limitations originating from hardware (e.g., small protected memory, lack of isolated peripheral) and system stack. They provide a fixed security abstraction where the enclave is the only unit of isolation. Thus, TEEs have *(i)* no support for fine-grained isolation or secure resource sharing, *(ii)* no systematic way to detect and defend against sensitive data and control flow tampering *within* and *across* enclave boundaries, *(iii)* no support for securely integrating with other compartment types or hardware security features on the same SoC, or migrating to other TEEs. As a result, using such monolithic TEEs has increased the attack surface to worsen existing vulnerabilities or even expose applications to new attack vectors, as we explain in section 2.3. We show how our extensible compartmentalization mechanism could entirely resolve or significantly improve these issues.

2.2.6 Intra-address space compartmentalization

Many software attacks target sensitive content in an application's address space, usually through remote exploits, malicious third-party libraries, or unsafe language vulnerabilities. Processing highly sensitive data in a single large compartment (e.g., process or enclave) leads to real threats that require effective protection against: *(i)* attackers can exploit vulnerabilities in less secure parts of the code to leak information, escalate privileges, or take control of the application or even the host. *(ii)* an application's secret data (e.g., private keys or user passwords) can be leaked in the presence of untrusted code parts or compromised third-party libraries like OpenSSL [171]; *(iii)* privileged functions or modules can be misused to access private content [172]; *(iv)* applications written in memory-safe languages such as Rust or OCaml are vulnerable via unsafe external libraries that jeopardize all other safety guarantees [173, 174]; and *(v)* in multithreaded use cases, attackers can exploit vulnerabilities (e.g., TOCTOU or buffer overflows) so the compromised thread can access sensitive data owned by other threads [175]. This whole class of attacks could be avoided by providing a practical way to enforce the least privilege within a shared address space.

Hence, many software or hardware-based techniques propose intra-address space (mostly in-process) isolation. Also, the importance of in-address space security threats results in significant improvement in hardware support for efficient memory isolation [69, 56, 4, 7]. However, simple APIs for utilizing such hardware features are not effective due to the complexity of attacks as well as various hardware limitations [57, 176] in security and performance. For example, hardware virtualization features are used for in-process data encapsulation by Dune [177] by using the Intel VT-x virtualization extensions to isolate compartments within user processes. However, overall, the overheads of such hardware virtualization-based encapsulation are not practical for IoT/mobile applications.

ERIM [57], light-weight contexts (lwCs) [59] and secure memory views (SMVs) [178] all provide in-process memory isolation and have reduced the overhead of sensitive data encapsulation on x86 platforms. ARMLock [179] is a software fault isolation (SFI)-based solution that offers lower overhead utilizing ARM memory domains. Similarly, Shreds [58] provides new programming primitives for in-process private memory support. Burow et al. [180] also leverage the Intel MPK and memory protection extensions (MPX) to isolate the shadow stack. Libmpk [176] focuses on resolving the limitation of supporting only 16 domains in MPK hardware by scheduling protection keys for Intel MPK. It still requires expensive PTE updates if more than 16 keys are used. Similarly, Donky [181] proposes a scalable design for introducing memory domains into RISC-V architecture.

These systems mainly focus on efficient memory isolation (e.g., fast domain switches or supporting a large number of memory domains) inside a single address space. Such features are important for effective compartmentalization, but such merely isolation techniques are not enough for enabling extensible privilege separation and secure sharing based on mutual distrust, particularly in a hetero-compartment environment. For example, recently, the growing attack surface of enclaves also led to several proposals for in-enclave isolation and more flexible trust models. A few systems focus on expensive SFI-based in-enclave isolation to reduce the attack surface [169, 62]. However, there is no solution to target security threats within and across these different compartment types. For instance, using dispersed compartments we could protect cross-compartment interactions and provide secure resource/data sharing (including shared memory) and safer multi-threading. Further, our defense-in-depth mechanism does not follow a fixed one-way trust model and considers mutual-distrustful situations (e.g., when an enclave is compromised and could be a threat to the host applications or systems).

2.2.7 Information flow control systems

Information Flow Control (IFC) goes beyond access control. It is a primary data-centric approach that not only restricts program access to data but also controls the propagation of data. It aims to guarantee non-interference across security contexts. In early definitions of IFC systems, entities or objects are associated with secrecy and an integrity level, depending on the sensitivity of the information they access [182]. The secrecy levels do not allow entities within less confidential

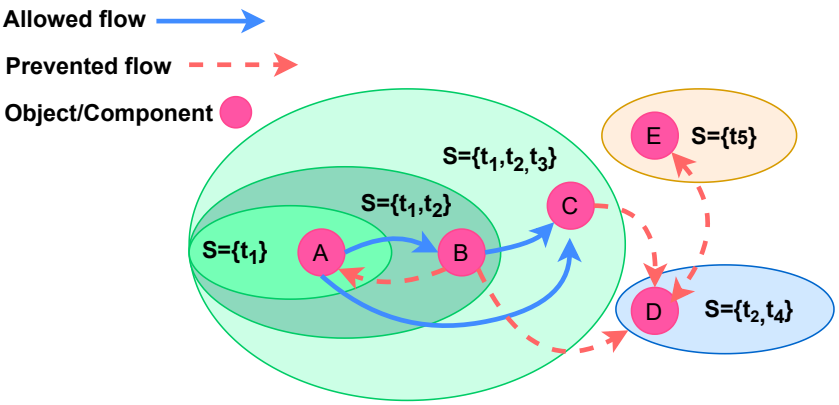


Fig. 2.14 Secrecy information flows

levels to access higher secrecy objects, which guarantees "the no read up, no write down" in the Bell-LaPadula principle. For example, in an IFC system with three secrecy levels, top-secret, secret, and public, entities in the top-secret level can receive information flows from all other levels, while a public entity can only access public information flow. Similarly, the integrity levels guarantee "the no read down, no write up" Biba principles to ensure objects are modified and verified only by authorized entities.

Many applications and static analysis systems use multi-level classic IFC to track data flow through a system and restrict any violation of their security policy from any code when operating on data. However, static analysis is not adequate or practical in complex applications, such as web applications, where the security policy may depend on user data at runtime or where execution entities (e.g., user or process) can join or leave the system arbitrarily. Hence, different variants of dynamic IFC are introduced to resolve shortcomings of static IFC systems (particularly scalability and efficiency issues) [183–186].

Variants of dynamic IFC can be enforced at either software or hardware abstractions. Particularly *granularity* of policy enforcement and *complexity* of the target system are two key factors determining the practicality of IFC mechanisms. For example, at the programming language level, these techniques assign explicit security policies—or labels—to every variable and within operations between them [187, 184, 188, 189]. While at the OS level, dataflows are enforced within OS kernel objects such as processes and files [190–192]. Similarly, it can be enforced within various hardware components [193]. No single IFC mechanism provides the most efficient or practical solutions for these different use cases.

Information Flow Control models

IFC models at core are a set of relatively simple rules over system's selective entities or objects. However, there are various IFC and labeling models depending on the enforcement granularity, needed policies, performance, and scalability requirements. Particularly, the implementation of complex security policies through IFC in a practical and expressive way is non-trivial.

However, basically most IFC systems check dataflows based on a partial ordering of security contexts, generally as a lattice, that defines where data can flow legitimately. This is enforced by assigning unique tags and labels (i.e., a set of tags) to monitored entities. Often each object can be assigned with secrecy and integrity labels. For example, under simplest model, in Figure 2.14, object B can receive information from A since $A \subseteq B$, while B (and also C, D, E) can not sent/access A since $B \nsubseteq A$. Similary, Figure 2.15 shows simple integrity-only dataflows, where a flow from C to A and B is possible since $A, B \subseteq C$. The flows can be controlled either *explicitly* or *implicitly*. In explicit flows, the security principal makes data flow from one level to the other through a controlled way of requesting and then assigning labels. On the other hand, in implicit flows, the labeling changes happen in the background as the data flows through the system. For instance, when an untainted process accesses a tainted file, the process itself becomes tainted automatically.

Besides, practical systems often need some information to flow from high to low (for example, to release secrets or use an untrusted network). Such unsafe transfer of information should only be allowed if the data goes through a controlled downgrading process [194]. Various systems proposed different ways of managing labels, extending IFC rules, and handling the downgrading process. Here we only summarise some of these systems with a focus on compartmentalization.

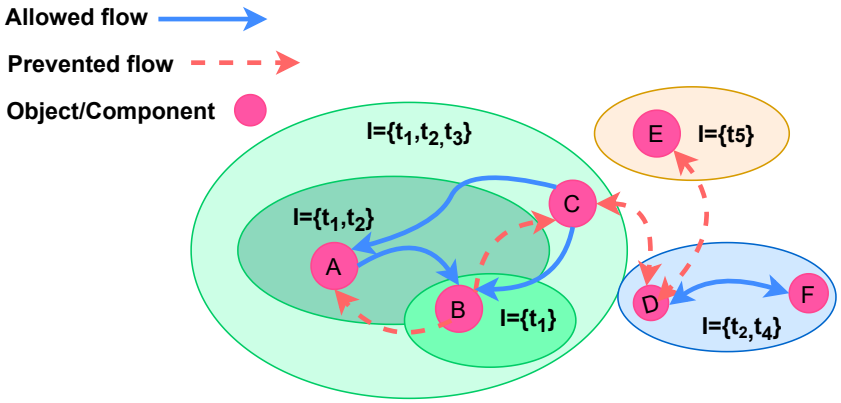


Fig. 2.15 Integrity information flows

Language-level taint tracking

Lots of previous work exist on IFC-based static and dynamic analysis and taint tracking [195–197, 189, 198, 199]. Language-based IFC extends existing programming languages by labeling variables with security attributes. Compilers use the security labels to generate security checks based on programmer-defined security regions. These languages indirectly compartmentalize the program by partitioning safe dataflows from unsafe ones that could significantly improve the security of programming languages.

Jif [195] is the first language, an extension to Java, that provides static checking of information flow using the decentralized label model. Motivated by limitations of the Java sandboxing

framework (e.g., insecure sharing mechanism) and other IFC systems, the decentralized model proposed a more comprehensive security model for different data manipulation by applications with different security requirements, even in situations of mutual distrust. The model allows users to control their information flow without imposing the rigid constraints of a traditional multilevel security system. The decentralized label model also introduced a more flexible notion of declassification to arbitrarily weaken information flow restrictions since strict control could be too restrictive for real-world applications. Declassification in earlier systems was performed only by a trusted subject, such as a centralized authority of a universally trusted principal.

Consequently, traditional IFC and the notion of a universally trusted principal are not practical and reasonable in decentralized environments. Hence, the decentralized model permits every security principal (or data owner) to declassify their own data rather than requiring a trusted subject. In this model, a principal may weaken only the policies it has provided and thus may not directly compromise data it does not own. Hence, the decentralized model can support policies from potentially different sources that are suitable for decentralized systems.

Similar to other IFC models, the decentralized label model defines a set of rules that programs must follow in order to avoid information leaks. In Jif, these rules are checked statically based on the user's annotations (labels) that describe the allowed dataflows. In conjunction with regular Java type declarations, these labels form an extended type system. Then Jif programs are type-checked at compile time to ensure that they are type-safe and that they do not violate information flow rules. Jif is used in building more compartmentalized frameworks such as SIF [200] (Servlet Information Flow), a web framework with explicit support for confidentiality and integrity information security policies. SIF showed that utilizing a language that is designed based on principals of mutual distrust is notably useful for tracking the flow of information within a web application, and information sent to and returned from the remote client.

However, these languages are limited and can not scale well. Besides their large overhead, they are cumbersome to work with from various perspectives, such as variable-level code annotations and security reasoning, and restrictive interactions with OSs, other languages, or libraries. Also, these language-based IFC systems trust the whole OS and provide no guarantees against security violations on system resources, such as files and sockets.

That is why there are few proposals on unifying programing language and OS mechanisms for enforcing IFC. For example, Laminar [189] provides a common security abstraction and labeling scheme based on decentralized IFC for Java program objects and OS resources. In Laminar, developers express complete security policies by annotating their Java code, and then the Laminar virtual machine (VM) and OS enforce these policies. Thie results show that, on average, about 10% code modification is required, and the performance overhead could be up to 56% depending on the use case. Laminar adopts the decentralized IFC principles of JIF combined with the labeling model of Flume [191], a DIFC OS. The Laminar's LSM (Linux security module) governs information flows through all standard OS interfaces, including

through devices, files, pipes, and sockets. The OS regulates interactions between different processes that access the labeled or unlabeled system resources. OS enforcement applies to all applications, preventing un-labeled or non-Laminar applications bypassing the DIFC restrictions. The Laminar VM regulates information flow between Java heap objects within the address space. TaintDroid [198], a widely-used data privacy tracking system on the Java-Android framework (with average overhead of 32%), is another example of such systems. These systems show the potentials of proper IFC software stacks; that could significantly improve the security of the systems by supporting a wide range of complex security or privacy policies.

OS-level IFC

Among early OS-based solutions, LOMAC and IX added traditional IFC and multi-level security to Unix-based OSs [201, 202]. However, as we discussed earlier (§2.2.7 and §2.2.3) such systems rely on inflexible centralized policy decisions and are not practical for supporting complex security policies.

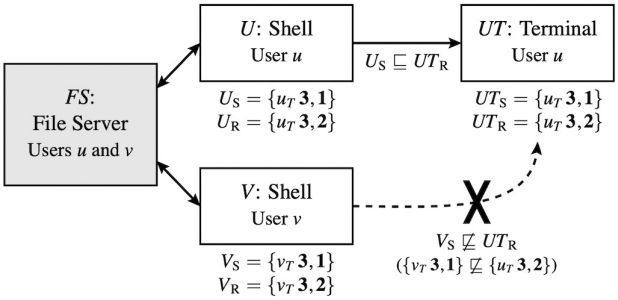
Asbestos [203] is the first OS that was designed based on the principles of language-level DIFC (inspired by Jif) to enable any process to create non-hierarchical security categories (called tags) dynamically. Asbestos brings the privilege concept into the DIFC security model. So, for example, a process with privilege for a tag can bypass the $\star-$ property with respect to that tag either by declassification or by raising the security clearance of other processes. Asbestos aimed for *unprivileged* and *large-scale server applications* (scale of thousands users). Unprivileged means its applications could define and change security policies without requiring approval from a privileged authority. To achieve this, the OS prevents processes from violating the established policies, whether or not they are compromised.

The Asbestos kernel followed a small, non-preemptive, and single-threaded process design with no support for symmetric multiprocessors or shared memory. Its IPC was based on asynchronous message passing (similar to microkernels such as Mach). Each message is addressed to a single port, and messages sent to a port are delivered to the single process with receive rights for that port; this is initially the process that created the port, but receive privileges are transferable and are determined through label checks. Conventional mechanisms such as pipes and file descriptors are emulated using the message passing. For example, to read a file, the client sends a READ message to the file server's port and awaits the corresponding `READ R` reply (inspired by Plan 9's 9P protocol [116]). Besides traditional services (e.g., memory management), the kernel provides syscalls for supporting tag, label, and event process functionality while checking their DIFC checks within the processes and syscalls.

To track information flow, the Asbestos kernel applies labels to processes. Each process has two labels, a tracking label, and a clearance label. The tracking label records the information flow a process has observed so far; the clearance label bounds the maximum tracking label a process is allowed to achieve. A given per-process security policy may require one, two, or many tags to implement (up to 2^{61} tags). In their model, A label specifies a sensitivity level for

Level	Usage
$\star$	Privilege for the tag
0	High integrity for the tag
1	Default tracking level—no restriction for the tag
2	Default clearance level, used in taint tracking
3	High secrecy for the tag

(a)



(b)

Fig. 2.16 DIFC in Asbestos: (a) Label sensitivity levels, (b) simplified process labeled communication.

each tag. Normal sensitivity levels range from 0 to 3. Information can flow freely from 0 up to 3; the reverse direction represents declassification that requires having the privilege. To elaborate more, they introduced the concept of handle privileges that are represented by levels, which are members of the ordered set $[\star,0,1,2,3]$; in send labels, $\star$ is the lowest or most privileged level, and 3 is the highest or least privileged level. The default levels lie in between; they are 1 for send labels and 2 for receive labels. A label, then, is just a function from handles to levels. For example, $\{h_1 0, h_2 1, 2\}$ shows two h_1 and h_2 handles and their levels. Comparing two labels, means comparing each of their components (e.g., $L_1 \sqsubseteq L_2$ iff $L_1(h) \leq L_2(h) \forall h$.) For example, Figure2.16 shows a simplified labeled communication in Asbestos. The processes U and V are tainted with u_T and v_T, and their receive labels allow them to receive the data of their respective users. Since user u's terminal process, UT, has the same labels as U, U can send messages to UT (because $U_S \sqsubseteq UT_R$), but V cannot, since $V_S(vT) > UT_R(vT)$, and neither can any other process that has seen v's data.

The Asbestos labeling model is mainly intended for decentralized compartments: *(i)* $\star$ sensitivity level represents declassification privilege with respect to a compartment. *(ii)* when sending a message, a process can supply additional restrictive discretionary labels on top of the process labels maintained by the kernel; these labels can be forwarded to the receiving application for further analysis. *(iii)* their per-process separate send and receive labels with different defaults for future compartments, support policies that transitively prevent two processes from communicating without unnecessarily restricting either process's ability to communicate with the rest of the system.

Despite their minimal custom kernel design and limited evaluation of real-world applications, Asbestos showed not only traditional access control forms (discretionary and nondiscretionary) could be mapped to their labeling model and be enforced by proper DIFC checks, but also their labels could track and limit the flow of information within the system- and application-defined compartments. They also showed that the ability to declassify data in a single compartment

is similar to possession of a discretionary capability. Therefore, the resulting system could support capability-like and traditional MLS policies and application-specific isolation policies with decentralized declassification, all through a single unified mechanism.

HiStar [192] was directly inspired by Asbestos but varied in providing a new OS kernel design from scratch for system-wide and fine-grained DIFC enforcement. The HiStar kernel is organized around six object types: a segment (a variable-length byte array similar to a file), an address space (a mapping from virtual memory to segment object names), a network device (which can send and receive packets), a thread (a set of CPU registers, along with the name of an address space object), a gate (an IPC mechanism), and a container (a directory-like object in which all other objects reside). HiStar assigned a unique 64-bit object ID and a label that is used to control information flow to or from each object. It stored all state accessible to user processes in kernel objects. Upon receiving any request on system objects, the kernel compares the labels of the currently executing thread and the objects being accessed to decide whether the operation should be permitted. While it is not possible to interpose on every read and write to memory-mapped files, the kernel remembers all active memory mappings and invalidates them when it suspects access should no longer be allowed (e.g., when a thread's label changes).

Unlike Asbestos, HiStar could support intra-address space security policies. Every running thread in HiStar has an associated address space object containing a list of $VA \rightarrow$ <S, offset, npages, flags> mappings. VA is a page-aligned virtual address. $S =< C, O >$ is a container entry for a segment to be mapped at VA. offset and npages can specify a subset of S to be mapped, flags specifies read, write, and execute permission. Each address space A has a label L_A, to which the HiStar label rules apply. Thread T can modify A only if $L_T \sqsubseteq L_A \sqsubseteq L_T$, and can observe or use A only if $L_A \sqsubseteq L_T$. When launching a new thread, one must specify its address space and program counter. As a result, HiStar could support policies for secure sharing system objects (e.g., shared memory) inside a single address space. However, HiStar suffers from key issues, including the hard-to-use labeling model, the non-POSIX kernel, and application incompatibility.

Flume [191] instead proposed process-level DIFC as a minimal extension to the Linux kernel, making DIFC work with the languages, tools, and OS abstractions already familiar to programmers. It also introduced a cleaner label system (which HiStar have later adopted). To help programmers reason about labels on standard Unix abstractions, Flume proposed *endpoint* abstraction. It represents each resource with a process to communicate as an endpoint, including pipes, sockets, files, and network connections. A process can define what subset of its privileges should be exercised when communicating through each endpoint. Uncontrolled channels are also modeled as endpoints that exit the DIFC system. Hence, Flume ensures that no process can have both an uncontrolled channel and access to private data it cannot declassify. Flume is mostly built in userspace with a few small kernel patches for implementation convenience

Secrecy dataflow p to q	$\{S_p - D_p \subseteq S_q \cup D_q\}$
Integrity dataflow p to q	$\{I_q - D_q \subseteq I_p \cup D_p\}$
Safe data flow p to q	$\{S_p - D_p \subseteq S_q \cup D_q\} \wedge \{I_q - D_q \subseteq I_p \cup D_p\}$
Safe label change L to L'	$\{L' - L\}^+ \cup \{L - L'\}^- \subseteq C_p$

Table 2.3 Summary of DIFC dataflow rules in Flume.

and portability. This design choice causes a relatively large overhead compared to deeper kernel integration.

On the other hand, Flume's approach to decentralized IFC adds a simple new representation of how processes maintain and use decentralized privilege in a way that is intended to be easier to understand and use. Krohn et al. also proves non-interference in Flume's labeling model [204]. Its labeling model uses tags and labels to track data as it flows through a system. Each principal represents an entity with security interests (e.g., processes) over multiple objects which can also be of interest to other principals. Principals use tags and labels to control the dataflows. They express confidentiality and integrity over objects (e.g., file) by assigning them secrecy or integrity tags. So for example when Principal P_1 assigns a secrecy-tag to object O_1, it marks it as a confidential object that can not be read or accessed by unauthorized principal P_2. Labels are sets of tags and each principal has only two labels; a secrecy label S_p to keep all secrecy-tags and integrity label I_p for integrity-tags. If $t \in S_p$ then the system assumes that p has seen some private data tagged with t so it can not reveal the data or propagate it to another principal that does not have t in its secrecy label.

In Flume, every principal 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, plus and minus, per tag. So t^+ and t^-, 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 it can not 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 which p has both t^+ and t^- capabilities (full ownership).

Adding a tag to a secrecy label and removing a tag from an integrity label 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 so it must not be exported to public. Table 2.3 summarizes the DIFC rules for enforcing safe dataflows and label changes. Note that in Table 2.3, $L\{\}^{+/-}$ mean that the label set contains tags with `plus/minus` capabilities. So $L\{\}^+$ only contains tags with plus capability and $L\{\}^-$ only contains tags with minus capability.

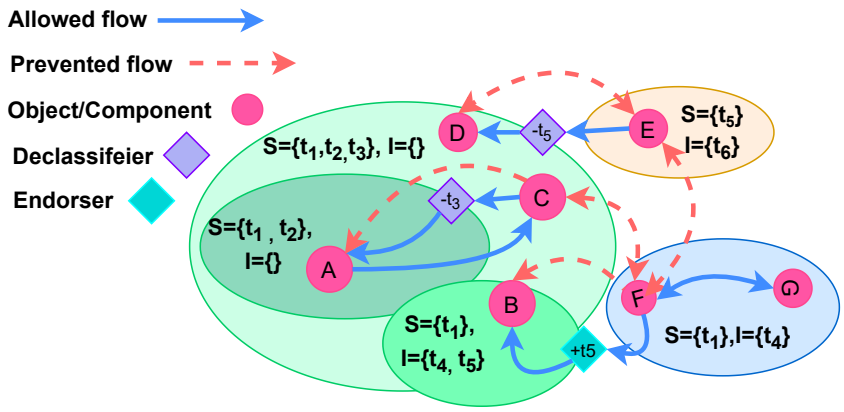


Fig. 2.17 Controlled downgrading: declassification and endorsement to enable dataflows

Figure 2.17 shows how controlled downgrading works through declassifiers and endorsers in a DIFC system. For example, dataflows from A to C are possible, while C requires going through a declassifier to remove t_3 for enabling dataflows to A. Enabling controlled downgrading makes DIFC mode more attractive for handling complex policies specially in networking use cases.

Therefore, many other IFC systems [205, 198, 206, 207, 190] inspired by these systems and their security model. These works showed the importance and possibility of combining proper OS abstractions with a usable labeling model to enable fine-grained and high-performant information flow control within complex systems and applications, which inspired this thesis.

Architectural IFC

Many proposals exist for hardware-based information flow tracking to achieve various goals [208–212, 207, 193]. Suh et al. [208] proposed one of the first system architectural mechanisms for dynamic information flow tracking. Its IO-specific and byte granularity tagging used a one-bit tag to indicate whether the corresponding data block is authentic or spurious. Raksha [212] proposed hardware support to enable transparent and fine-grain management of security tags for dynamic information flow tracking (DIFT). It mainly focuses on improving the performance of label checking and binary compatibility. Then, Loki [207] showed how hardware support for tagged memory allows could partially help enforce HiStar's labels to reduce the amount of fully trusted kernel code (by 50%). Loki's tagged memory simplifies security enforcement by associating security policies with data at the lowest level in the system–in physical memory. Its security monitor, LoStar, translates userspace labels into tags on the physical memory containing the respective data. As a result, Loki's tagged memory mechanism could directly map and enforce policies without trusting the kernel. For example, a memory page representing a file descriptor could be tagged to be accessible only to the processes that have been granted access to that file descriptor. Likewise, the private memory of a process's address space can be tagged to ensure that only threads within that particular process can access that memory. Unix user IDs

are also mapped to labels and then translated into tags and enforced using the same hardware mechanism.

Moreover, HDFI [211] proposed a fine-grained dataflow isolation mechanism and memory protection, focusing on minimizing hardware changes. HDFI enforces isolation at machine word granularity by virtually extending each memory unit with an additional tag. It does not require modifying register files, ALU, main memory, or the bandwidth between cache and main memory. Hence, it requires much less physical memory dedicated to storing tags compared to other tagged memory solutions (e.g., CHERI). Some systems enforce information flow policies at lower-level for securing hardware design. For example, Caisson [213] is a new Hardware Description Language (HDL) to enable the creation of synchronous hardware designs that are statically verifiable as secure. HyperFlow [193] is a processor architecture designed for timing-safe information-flow security. It extends the RISC-V instruction set with instructions for information flow control to remove all disallowed information flows between security levels, including timing channels.

This thesis focuses on properly utilizing existing hardware security features to enable effective compartmentalization instead of introducing new hardware primitives and architecture. We believe this approach can benefit broader use cases and a much more comprehensive range of applications that rely on commodity hardware.

2.3 Motivating examples

Here we further explain the shortcomings of existing compartmentalization techniques by analysing modern real-world applications. We show how even compartmentalized applications suffer from non-trivial attack vectors that could not easily be identified or mitigated.

2.3.1 Cryptography libraries

Cryptographic libraries are responsible for securing all connected devices and network communication, yet have been a source or victim of severe vulnerabilities. Given these libraries' critical role, a single vulnerability can have a tremendous security impact. In 2014, OpenSSL's Heartbleed vulnerability [171] enabled attackers to access many servers' private data (up to 66% of all websites were vulnerable). More recently, GnuTLS suffered a significant vulnerability allowing anyone to passively decrypt traffic (CVE-2020-13777). Lazar et al. [214] studied 269 cryptographic vulnerabilities, finding that only 17% of the vulnerabilities they studied originated inside the cryptographic libraries, with the majority coming from improper uses of the libraries or interactions with other codebases. However, recent studies show that about 27% of vulnerabilities in cryptographic software are cryptographic issues, and the rest are system-level issues, including memory corruption and interactions with the host or other applications/libraries [215].

Furthermore, they show a direct correlation between the increasing complexity of cryptography libraries and the increasing rate of their vulnerabilities. For example, they find approximately

Vulnerability	Description	Crypto Library
CVE-2021-3711	Memory corruption in the API function EVP_PKEY_decrypt()	OpenSSL
CVE-2021-3450	Improper access control and bypass certificates	OpenSSL
CVE-2021-23840	Buffer overflow in EVP_CipherUpdate, EVP_EncryptUpdate and EVP_DecryptUpdate	OpenSSL
CVE-2016-6309	Memory corruption and code execution in statem/statem.c	OpenSSL
CVE-2021-20232	Memory corruption in lib/ext/pre_shared_key.c	GnuTLS
CVE-2020-13777	Authentication bypass in TLS 1.3	GnuTLS
CVE-2017-5334	Memory corruption in the gnutls_x509_ext_import_proxy	GnuTLS
CVE-2016-4456	Improper access and corrupt arbitrary files in the filesystem	GnuTLS
CVE-2020-36177	Out-of-bounds write in wolfcrypt/src/rsa.c	WolfSSL
CVE-2021-3336	Man-in-the-middle attacks on TLS 1.3 servers	WolfSSL
CVE-2019-11873	Buffer Overflow in DoPreSharedKeys in tls13.c	WolfSSL
CVE-2017-8854	Buffer overflow when accessing a malformed temporary DH file	WolfSSL
CVE-2018-9127	Improper access control over wildcard certificates	Botan
CVE-2021-24115	Control-channel and side-channel over decoding/encoding	Botan
CVE-2016-2196	Heap-based buffer overflow in the P-521	Botan
CVE-2021-3345	Heap-based buffer overflow in cipher/hash-common.c	Libgcrypt
CVE-2021-40528	Cross-configuration attack	Libgcrypt

Table 2.4 A few selective vulnerabilities in popular cryptography libraries.

22K LoC code changes between major versions of OpenSSL, and at least one vulnerability occurs per 1K LoC change. Unfortunately, despite significant progress in adapting best security practices and improving code quality [216], these libraries are still susceptible to a wide range of attacks (see Table 2.4).

As a result, historically, these libraries are considered essential use cases for evaluating isolation and compartmentalization techniques [114, 57, 48, 168, 176]. Privatrans [114] integrated process-based privilege separation into OpenSSL so that SSL services could protect their privileged RSA private key operations. It partitioned the library so two sensitive functions (`RSA_private_encrypt` and `RSA_private_decrypt`) will be executed in a trusted monitor, while everything else will be inside a slave process. This limited partitioning also adds about 46% overhead due to transferring data between the slave and monitor. More recently, with the progress in hardware-assisted security, there are more efficient ways for compartmentalizing complex cryptography libraries. For example, Erim [57] uses Intel MPK's 16 memory domains to efficiently isolate the crypto functions and session keys (libcrypto) due to their low-cost domain switching. It adds about 4.8% on the throughput of NGINX when isolating all session keys, up to 6.3x, 13.5x, and 3x lower than the overhead of similar protection using SFI (with Intel MPX), lwCs [59] and Intel VT-x, respectively. However, despite more efficient isolation techniques, these systems have a large TCB (including the OS) and do not consider many attack vectors from unsafe systems interactions, privilege management, and resource/data sharing.

Therefore, several works proposed TEE-assisted partitioning to protect sensitive keys and cryptography operations, even from potentially malicious host system software such as OS. For example, Glamdring [168] partitions LibreSSL[6] to protect the confidentiality of the private key of the root certificate of the LibreSSL CA inside an SGX enclave. Glamdring places a sub-set

[6]https://www.libressl.org

of LibreSSL into the enclave: *(i)* the entropy/random number generator, *(ii)* the RSA and Big Numbers module, and *(iii)* the X509 module, which stores the certificates. The functionality placed outside the enclave includes: *(i)* the TLS/SSL modules for secure communication, *(ii)* digest algorithms (MD5, SHA256), and *(iii)* cryptography protocols unrelated to certificate signing (DSA, AES). Glamdring places 22% of LibreSSL LoC, 17% of functions, and 16% of global variables inside the enclave. The partitioned LibreSSL also exposes 171 ecalls (calls to the enclave) and 613 ocalls (calls from outside the enclave). Similarly, SGX SSL[7] is another solution that ports crypto libraries inside an enclave. It also adds between 0.3x-0.8x performance overhead over the original versions.

However, there are many new attack vectors that these solutions do not consider. For example, multiple attacks are possible through the host-enclave interface [38, 42]. Previous work shows that existing mitigations based on sanitizing the TEE interfaces are failing due to the vast and complex attack surface [44, 42]. Also, attackers can take advantage of inadequate address space isolation within an enclave to launch ROP attacks [39, 40, 43] for extracting cryptography keys or bypass remote attestation. Additionally, insecure shared memory buffers are essential to attack vectors for extracting secrets or compromising RPC interfaces [42]. Moreover, a compromised or malicious third-party TA can collect sensitive data [45, 46] and leak them using OS facilities such as files, network sockets, or pipes. Hence, naive TEE-based compartmentalization could lead to a less secure outcome with significant cost and overhead.

2.3.2 Web services

Almost all enterprises have been widely dependent on web services, particularly after the rise of cloud computing, which has increased online storage and enabled the easy processing of vast amounts of data in many domains. Modern web applications are no longer a simple collection of pre-assembled HTML pages, scripts, and style/media files. Instead, they now rely on complex designs and software stacks on both client and server sides. Consequently, web applications are vulnerable to a wide range of attacks that can compromise confidential data, bring down critical systems, or otherwise wreak havoc on our lives.

Any security issue in critical web services could significantly affect millions of users due to the scale of adoption. For example, recently, several remote code execution (RCE) vulnerabilities in Apache Log4j and JNDI, referred to as "Log4Shell", (CVE-2021-44228, CVE-2021-45046, CVE-2021-44832) gained broad attention due to its severity and potential for widespread exploitation. The list of potential victims covers nearly 30% of the world's web servers. That means Twitter, Amazon, Microsoft, Apple, IBM, Oracle, Cisco, Google, and one of the world's most popular video games, Minecraft, could be open to attacks. Hence, more than 3.7 million attempts were made to exploit the vulnerability in a few days, including an attack on the Belgian defense ministry and many attacks to install cryptocurrency "mining" software on victims'

[7] https://github.com/intel/intel-sgx-ssl

Vulnerability	Description	Web service
CVE-2021-44228	Takeover of log messages and execute arbitrary code loaded from JNDI LDAP servers	Apache Log4j
CVE-2021-4104	Unauthorised write access to the Log4j configuration	Apache Log4j
CVE-2021-45046	Thread privilege escalation and crafting malicious input data using a JNDI Lookup pattern	Apache Log4j
CVE-2021-44790	Buffer overflow in the mod_lua multipart parser	Apache http
CVE-2019-0211	Arbitrary code execution and privilege escalation of child process/thread from parent	Apache http
CVE-2019-0215	Bypass configured access control restrictions via a bug in mod_ssl	Apache http
CVE-2020-35452	Stack overflow in mod_auth_digest	Apache http
CVE-2021-23017	Unauthorised affect and control over worker process	NGINX
CVE-2018-16845	Controlling worker processes via bug in the ngx_http_mp4_module	NGINX
CVE-2017-7529	Memory overflow in nginx range filter module	NGINX
CVE-2018-16843	Excessive memory consumption	NGINX
CVE-2019-11072	Overflow via a malicious HTTP GET request	Lighttpd
CVE-2018-19052	Path traversal attack in mod_alias.c	Lighttpd
CVE-2021-2394	Bug in core module of fusion middleware and takeover of WebLogic Server	Weblogic
CVE-2020-14859	Bypass access control in console component and takeover of WebLogic Server	Weblogic
CVE-2020-5398	Reflected file download (RFD) attack	Weblogic
CVE-2020-11023	Untrusted code execution	Weblogic
CVE-2019-17571	Arbitrary code execution via bug in SocketServer class in Log4j	Weblogic
CVE-2017-5638	Arbitrary code execution via erro handling bug in the Jakarta parser	Apache Struts
CVE-2020-17530	Remote code execution when using OGNL evaluation on raw user input	Apache Struts

Table 2.5 Few selective vulnerabilities in some popular web services.

computers. As another example, Apache Struts is a mainstream web framework widely used by Fortune 100 companies in education, government, financial services, retail, and media. A remote command injection vulnerability in Apache Struts (CVE-2017-5638) led to Equifax's data breach, exposing the personal information of 147 million people. The company has agreed to a global settlement with the Federal Trade Commission, the Consumer Financial Protection Bureau, and 50 U.S. states and territories. The settlement includes up to $425 million to help individuals affected by the data breach. These attacks could be avoided or significantly reduced by properly compartmentalizing third-party dependencies and untrusted components. Table 2.5 shows only a few such vulnerabilities in various web services.

Despite a large amount of work on securing web services [217–219, 48, 57], their attack surface remains vast due to the lack of proper compartmentalization, fine-grained isolation, and suitable access control mechanisms. Among previous work, OKWS [217] provided a set of tools to enforce the principle of least privilege in web services so that the different components of the system distrust each other. It assumes that vulnerabilities in complex web services are inevitable and presents several simple guidelines for protecting sensitive site data in the worst-case scenario by limiting the propagation of vulnerabilities between key components in the system via process-based sandboxing. It provides a set of libraries and helper processes so web developers can build Web services as independent, stand-alone processes, isolated almost from the file system. The core libraries provide basic functionality for receiving and responding to HTTP requests, accessing data sources, composing an HTML response, and logging the results to disk. A process called OK launcher daemon, or *okld*, launches custom-built services and relaunches them when they crash. A process called OK dispatcher, or *okd*, routes incoming requests to appropriate Web services. A helper process called *pubd* provides Web services with

limited read access to configuration files and HTML files stored on the local disk. Finally, a dedicated logger daemon called *oklogd* writes log entries to the disk. Wedge [48] proposed a simpler privilege separation mechanism in Apache/OpenSSL with a focus on protecting private keys. It partitioned Apache 1.3.19, so worker threads encapsulate unprivileged code and have no direct access to the private key maintained and processed inside a separate process. As mentioned earlier (§2.2.3), such isolation techniques lack extensible policies and add enormous performance and refactoring costs.

More recently, various hardware-assisted techniques have been suggested for securing web services while providing smaller isolation overhead or TCB size. Several proposals use intra-address space isolation for protecting private keys [57, 176, 59] in different web servers. On the other hand, various systems propose coarse-grained isolation by porting web services such as NGINX entirely or partially inside TEEs such as SGX enclaves.

Existing TEEs do not support running such large applications out-of-the-box; developers have to use TEE frameworks to port applications to an enclave [164, 66, 168, 136, 220] or use in-enclave LibOSs such as SGX-LKL [221] or Graphene-SGX [66]. First, this leads to additional TCB inside the enclave, ranging from 20K-2M LoC, that increases the application attack surface [38, 42]. Second, they pay notable overhead due to hardware limitations of TEEs (e.g., limited physical memory) or unsupported dependencies (e.g., language runtimes, multithreading, optimization tools), which slows down the application's performance by 10-1000%. Lastly, several programming constructs are either unavailable or untrusted inside enclaves. Running an application which was designed with a trusted OS assumption as-is inside an enclave leads to various attacks stemming from insecure interactions with the untrusted OS and other untrusted enclaves (e.g., Iago attacks [73], TOCTOU [42], AsyncShock [41], Boomerang [43], HPE [44], malware-enclave [167]). More importantly, these attacks are non-trivial to defend; they require careful and application-specific defenses.

2.3.3 Data analytics

Modern data processing services hosted in edge/cloud environments are constantly attacked by malicious entities. This includes attacks on different processing stages such as databases, storage systems, edge/cloud platforms, or ML frameworks/models. Encryption-based mechanisms can provide strong and relatively efficient protection for data at rest and in transit [222–224]. However, this is not sufficient when dealing with untrusted parties and platforms. For example, compromised host systems could leak unencrypted data during processing or decrypt sensitive data in memory.

As an example, consider a cloud-based ML service that performs training and inference over large datasets. Such a service does not want to share its sensitive data, model, or inference with the cloud provider or other untrusted entities executing the application. At the same time, the cloud provider has to protect itself from getting attacked by the service. A TEE-based solution can resolve this by running the ML service in an enclave so that the remote platform can verify

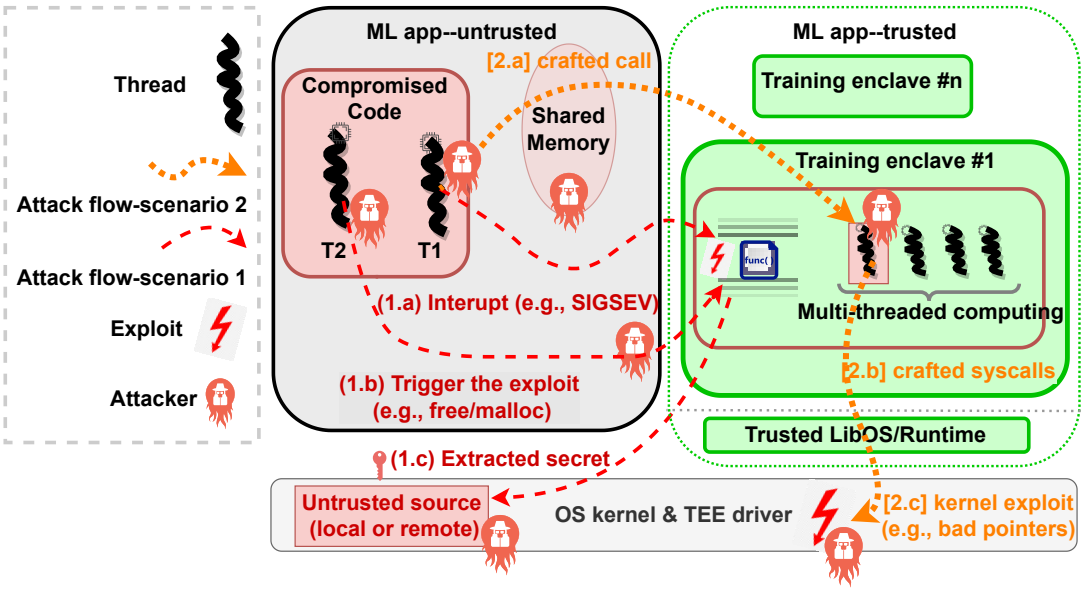


Fig. 2.18 Unsafe information flows in TEE-assisted ML.

that the computation is protected on the untrusted host [225, 226, 163, 227, 228]. Although the majority of the ML computation would execute inside the enclave, such a deployment requires sharing of data via host OS-controlled resources. For instance, the program may use untrusted memory to share results across multiple enclaves due to in-enclave memory limitations, untrusted storage to persist intermediate results, and an untrusted network to receive requests and send results. The architecture of such an application is sketched out in Figure 2.18, along with the following issues.

Attack Scenario I: Enclave as a victim. Such applications lead to porting large and possibly insecure codebase to an enclave. Previous work shows attackers easily take advantage of inadequate address space isolation within the host process or enclave for extracting secrets [38–40, 43]. Additionally, the ML framework uses parallelism with concurrent algorithms to achieve high performance. Thus, it may use insecure shared memory buffers or assumes the faithful enforcement of synchronization primitives, timely delivery of event notifications, and correct ordering of computations. However, the untrusted OS controls the thread scheduler and event notification. Further, the ML code ported to execute inside the enclave may have synchronization vulnerabilities. In a non-enclave execution, this vulnerability cannot be used for any meaningful attack beyond a denial-of-service. However, an attacker can now selectively interrupt and invoke the enclave threads to trigger such vulnerabilities to induce unexpected behavior such as a use-after-free to compromise the confidentiality and integrity of the enclave (1.a and 1.b in Figure 2.18) or leak its secrets (1.c in Figure 2.18).

For example, the attacker can extract and leak enclave keys and sensitive data to untrusted memory or send it over the network [42, 38, 41]. She can also launch membership inference

attacks (MIA), to extract private training data, through malicious queries [229]. Or she could significantly reduce the model's accuracy (e.g., asynchronous gradient descent) by manipulating the threads when updating shared parameters [230].

Thus, existing TEEs can *worsen* the impact of over-privileged enclaves with memory safety or concurrency bugs because they lack sufficient control and view of the system to protect themselves from external threats. Also, most TEE systems require disabling traditional security features (e.g., ASLR), memory-safe languages/dependencies, or fuzzing tools, since in-enclave limitations do not allow supporting them. Hence, users may blindly weaken the security of their applications because of the strong security claims of TEE-based services (particularly from cloud providers [47]), which may not be entirely valid. Existing solutions do not offer adequate mitigation for such threats; they either require specific languages for the interface security [55] or suggest further privilege separation via expensive techniques like using multiple enclaves [66], SFI-based in-enclave isolation [62] or custom hardware [60].

Attack Scenario II: host as a victim. Enclaves typically communicate with the OS via a trusted runtime executing inside the TEE. These runtimes usually have a different interface to the host OS and more closely resemble an embedded system. The TEE runtimes are rarely as heavily audited as their non-enclave counterparts or have a lower-level interface tied to the hardware (e.g., controlling pagetables in ARM TrustZone). The attacker can leverage a compromised enclave to launch attacks on the host kernel. For example, previous work shows that the attacker can send malicious inputs to the enclave and gets control of the enclave logic. Next, they use the enclave runtime interface to send malicious inputs to the kernel. For instance, in Figure 2.18, the attacker can send a crafted call through unsafe shared memory (2.a) and after taking the control of an in-enclave thread (2.b), send malicious syscalls to the TEE kernel (2.c). Such specially crafted kernel invocations are sufficient to launch confused deputy attacks [43, 231]. As another example, HPE [44] attacks show the possibility of compromising other processes on the host via a malicious/misbehaving enclave.

Thus, existing TEEs may open up a more powerful attack interface to the host because it has no way to effectively monitor and protect itself from such misbehaving enclaves. Current solutions offer insufficient protection, e.g., via pointer authentication for shared memory [43] or restricting enclaves inside OS-assisted sandboxes [167]. Figure 2.18 shows the steps to carry out both these scenarios. We have therefore established that a user-level attacker can leverage such mismatches between monolithic TEEs and the host to compromise both sides.

These attacks arise because the trust model assumes fixed coarse-grained mutually-untrusted execution entities (e.g., enclave vs. host process) wherein none of them have a global view of the system. Thus, each entity can only enforce security over its own code and data. Neither the kernel nor the enclave have sufficient context information to detect such threats or decide if the interactions at the kernel-process-enclave boundaries are secure. This is also the reason that

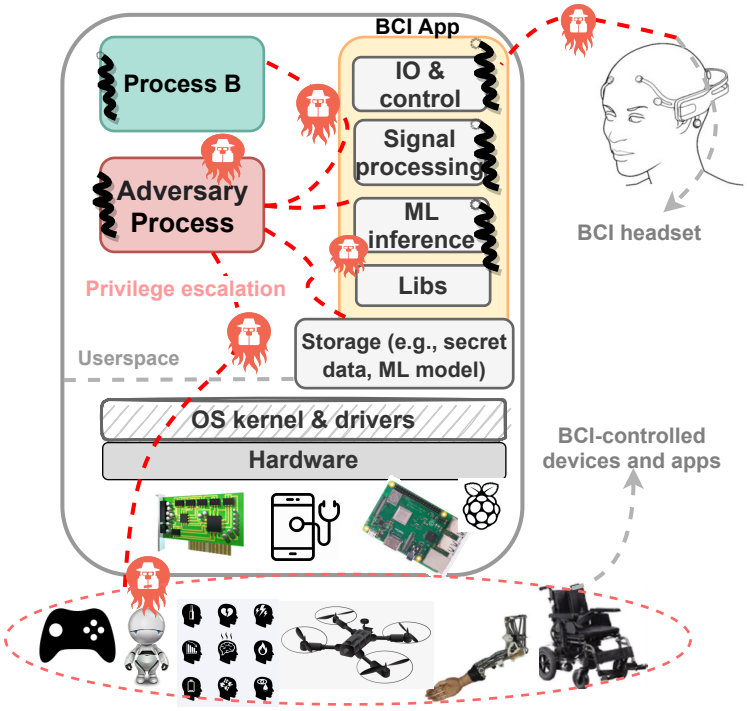


Fig. 2.19 Some attack vectors in BCI applications.

they can not effectively utilize the systems resources via sharing. Further, the coarse-grained boundaries increase the probability of vulnerabilities and over-privileged trust domains.

2.3.4 Wearable technology

The current state of mobile/IoT security threats is too big for one single general-purpose solution. As a result, recently, we have observed a surge in domain-specific techniques for improving security and privacy on mobile/IoT use cases [232–236]. To name a few, Privaros [234] enforces host-specified privacy policies on delivery drones, while McReynolds et al. tackle privacy issues on children's smart toys [237], and Aggio [235] focuses on privacy-preserving smart retail environments. Additionally, MegaMind [232] offers a solution targeted to voice assistants, SpecEye [238] proposes a privacy-preserving screen exposure detection system, and SecureSIM [239] provides an access control specific to in-SIM files.

Wearable technology is one of a highly-sensitive subset of IoT/mobile use-cases that process critical private information such as medical data. As an example, consider mobile brain-computer interfaces (BCIs) that are rapidly expanding to deliver solutions for monitoring health, mental/emotional state (e.g., focus, anxiety, or motivation), or sleep quality [240, 241]. Lately, relatively low-cost wearable BCIs are customized to support people with physical impairments or to improve smart living through BCI-assisted drones, robotic arms, wheelchairs, or mixed reality environments [242–244]. In addition, several proposals for utilizing BCI for multi-factor authentication and crypto-biometrics [245–247] are emerging. Compromising BCI-controlled devices—such as by taking control of a wheelchair or drone—is a real physical threat [248].

Also, sensitive personal information, like thoughts, sexual orientation, or religious beliefs, is under threat if security and privacy measures are not fully adopted [249].

Figure 2.19 illustrates how attackers can exploit and compromise BCI applications at different stages, and consequently steal secrets or take control of the host or BCI-controlled devices. Unauthorized access to pre-trained deep learning models on the host device could lead to the injection of malicious inputs, or the leaking of prediction model parameters or training data [229, 228]. Attackers can also combine/use traditional system-based attacks with/for ML adversarial ones to gain private information or manipulate ML predictions. Simple manipulation of BCI peripherals can alter the ML model output (e.g., from drowsy to awake) causing serious traffic accidents in EEG-based driving assistance for self-driving cars [250]. As we will explain in Chapter 5, our comprehensive security analysis on real-world BCI-mobile platforms (e.g., OpenBCI, Muse, NueroSky) led to successful prototypes of six major attack vectors, both system and adversarial ML attacks, and finding more than 300 vulnerabilities.

Proper compartmentalization and information flow tracking can significantly reduce such a large attack or help detecting various threats. Previous work (such as TaintDroid [198], Weir [251], or FlowFence [205]) come with a significant complexity and overhead, particularly when dealing with non-trivial security policies (e.g., in-process threats) and handling large number of sensitive system objects to monitor. They are also either strictly designed for Android or force the adoption of a specific programming language or hardware. Securing wearable technology such as BCIs require a strong, yet lightweight solution suitable for resource-constraints devices.

2.4 Summary

This chapter has explored the current state of ever-growing security issues in modern applications and platforms. We discussed the effect of compartmentalization on significantly reducing attack vectors and explained the profound limitations of existing solutions in diverse range of use cases. To summarise, we discussed the following key challenges this thesis resolves for enabling effective compartmentalization in hetero-compartment environments:

- Introducing the concept of hetero-compartments to the underlying system software via simple primitives while avoiding large TCB. To provide an adequate view of the system beyond fixed privilege layers and trust boundaries, our solution needs to extend existing abstractions and add new ones.

- Building and enforcing extensible and complex policy from simple security primitives. Our proposed approach must allow developers to specify and enforce extensible policies such as controlling a compartment in different address spaces or maintaining ownership of shared system objects with different compartment types (e.g., an enclave or in-process

sandbox). Hence, our solution must simultaneously support fine-grained (e.g., intra-address space) and cross-privilege policies in a practical way.

- Providing transparency by enabling developers to gain insight from the system behavior and modeling security threats. Our approach must allow developers to track critical information flows within various privilege boundaries and low-level system objects, audit compartments, and investigate non-trivial attack vectors.

- Providing such whole-system and fine-grained security mechanisms efficiently with small overhead. The proposed mechanism should not cause an impractical performance overhead that most applications can not afford. Also, it should not force large memory overhead or hardware modification, so many IoT/mobile use cases could benefit.

- Enabling flexibility to allow customisation depending on use cases. Our approach should provide enough flexibility to function with already existing technology and applications.

- Making practical programming models and userspace APIs for proper specification of compartmentalization policies. Our approach should not force developers to modify large portions of applications, use specific programming language features, or change the hardware.

Chapter 3

Effective hardware-assisted compartmentalization

This chapter formally defines dispersed compartments and presents a bottom-up description for enabling them on commodity hardware. This includes introducing three new hardware-assisted building blocks to enable: *(i)* intra-address space isolation, *(ii)* dispersed monitoring, and *(iii)* dispersed enforcement. These abstractions are key modules integrated into SIRIUS for making dispersed compartments a practical reality. However, first, we need to explore the architectural details and challenges of properly utilizing modern hardware features for privilege separation and isolation. We focus on primary hardware features in AArch32-64 and Intel x86-64 and argue their current shortcomings from security, performance, and usability perspectives (§3.1 and §3.2). Then, we will explain dispersed compartments more formally (§3.3) and present our hardware-assisted abstractions required for enabling dispersed compartments (§3.4).

3.1 Compartmentalization on TrustZone-enabled platforms

When the ARM Security Extensions (a.k.a., TrustZone) are implemented, the architecture enabled two security states, Secure state (`S`) and Non-secure state (`NS`). Each security state has its own system registers and memory address space. All code execution occurs in one of the states and operates in its own virtual memory address space. Since ARM-v8.4, all of the processor modes that are available in a system that does not implement the security extensions are available in each of the security states, including hypervisor (a.k.a., hyp) mode that was not available in the secure world before this version [5]. The current processor mode is determined by the mode field (M[4:0]) of the current program state register (CPSR). Processor mode change can be triggered by exceptions or privileged program can directly write CPSR by calling a `MSR CPSR_c,#imm` instruction, where *c* stands for the control field that includes processor mode bits and interrupt mask bits. As shown in Figure 3.1, TrustZone also defines an additional processor mode, *monitor mode*, that provides a bridge between code running in non-secure state and code

running in secure state. Secure monitor call (SMC) causes a switch to secure monitor exception (via SMC<c> #<imm>) and it is available only in privileged modes.

Execution privilege	Secure state	Non-secure state	Typical use
Highest	EL3	-[a]	Secure monitor
-	EL2[b]	EL2	Hypervisor
-	EL1	EL1	Secure or Non-secure OS
Lowest, Unprivileged	EL0	EL0	Secure or Non-secure application

a. EL3 is never implemented in Non-secure state.
b. If FEAT_SEL2 is implemented in AArch64 state, EL2 can be enabled in Secure state.

(a)

Processor Mode	Abbr.	ARMv7 Privilege Level	ARMv8 Exception Level	Security State
User	usr	PL0	EL0	Both
Supervisor	svc	PL1	EL1	Both
System	sys	PL1	PL1	Both
Abort	abt	PL1	EL1	Both
IRQ	irq	PL1	EL1	Both
FIQ	fiq	PL1	EL1	Both
Undefined	und	PL1	EL1	Both
Monitor†	mon	PL1	EL3	Secure only
Hyp‡	hyp	PL2	EL2	Non-secure only

† only implemented with Virtualization Extensions.
‡ only implemented with Security Extensions.

(b)

Fig. 3.1 (a) AArch32-64 privilege/exception levels in each security states, (b) ARMv7/v8 processor modes (after ARMv8.4 hyp mode in secure state is supported).

The fundamental mechanism that determines the security state is the SCR.NS bit in Secure Configuration Register as shown in Figure 3.2. SCR is a 32-bit read/write privilege register and part of the security extensions. It defines the configuration of the current security state and exists only in the secure world. It specifies the security state of the processor, the execution mode when handling IRQ, FIQ, or external aborts, and whether the current program status register (CPSR) bits can be modified when SCR.NS=1. For all modes other than monitor mode, the SCR.NS bit determines the security state for code execution (see Figure 3.3). Code executing in monitor mode is always executed in the secure state regardless of the value of the SCR.NS bit.

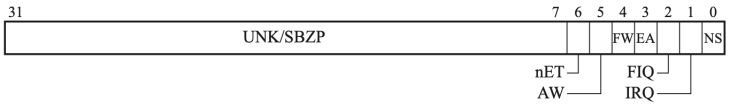


Fig. 3.2 Format of Secure Configuration Register (SCR).

The general-purpose registers and the processor status registers are not banked between the secure and the non-secure states (see Figure 3.4.(a)). Typically the registers that are not banked relate to global system configuration options that ARM expects to be common to the two security states. When execution switches between the Non-secure and Secure security states, ARM expects that the values of these registers are switched by a kernel running mostly in monitor mode. But most of the systems registers in coprocessors (e.g., CP15) and system control are banked between the secure and non-secure security states. A banked copy of a register applies only to execution in the appropriate security state. Therefore, banked CP15 registers have two copies, one secure and one non-secure (depends on the value of SCR.NS bit). The security extensions include an input signal, CP15SDISABLE, that disables write access to some of the secure registers when asserted HIGH (see Figure 3.4.(b)). Note that the CP15SDISABLE input does not affect reading secure registers, or reading or writing non-secure registers. It only

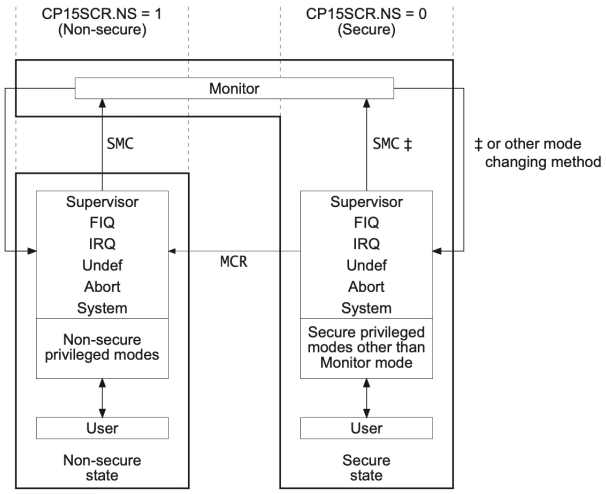


Fig. 3.3 ARM security states.

AArch32 Application Level View	AArch32 System Level View & AArch32 Relationship To AArch64								
	usr	svc	sys	abt	irq	fiq	und	mon†	hyp‡
R0	R0_usr\X0	-	-	-	-	-	-	-	-
R1	R1_usr\X1	-	-	-	-	-	-	-	-
R2	R2_usr\X2	-	-	-	-	-	-	-	-
R3	R3_usr\X3	-	-	-	-	-	-	-	-
R4	R4_usr\X4	-	-	-	-	-	-	-	-
R5	R5_usr\X5	-	-	-	-	-	-	-	-
R6	R6_usr\X6	-	-	-	-	-	-	-	-
R7	R7_usr\X7	-	-	-	-	-	-	-	-
R8	R8_usr\X8	-	-	-	-	R8_fiq\X24	-	-	-
R9 (SB)	R9_usr\X9	-	-	-	-	R9_fiq\X25	-	-	-
R10	R10_usr\X10	-	-	-	-	R10_fiq\X26	-	-	-
R11	R11_usr\X11	-	-	-	-	R11_fiq\X27	-	-	-
R12 (IP)	R12_usr\X12	-	-	-	-	R12_fiq\X28	-	-	-
SP (R13)	SP_usr\X13	SP_svc\X19	-	SP_abt\X21	SP_irq\X17	SP_fiq\X29	SP_und\X23	SP_mon\N/A	SP_hyp\X15
LR (R14)	LR_usr\X14	LR_svc\X18	-	LR_abt\X20	LR_irq\X16	LR_fiq\X30	LR_und\X22	LR_mon\N/A	-
PC (R15)	PC	-	-	-	-	-	-	-	-
APSR	CPSR	-	-	-	-	-	-	-	-
N/A	N/A	SPSR_svc \SPSR_EL1	N/A	SPSR_abt	SPSR_irq	SPSR_fiq	SPSR_und	SPSR_mon \SPSR_EL3	SPSR_hyp \SPSR_EL2
N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	ELR_hyp \ELR_EL2

† only implemented with Security Extensions. ‡ only implemented with Virtualization Extensions. A cell filled with - means the user mode register is used when referred. A cell filled with a register name means the register is banked. A cell filled with N/A means no register is available for this mode or level view.

(a)

CP15 register	Register name	Affected operation
c1	SCTLR, System Control Register	MCR p15, 0, <Rt>, c1, c0, 0
c2	TTBR0, Translation Table Base Register 0	MCR p15, 0, <Rt>, c2, c0, 0
	TTBCR, Translation Table Base Control Register	MCR p15, 0, <Rt>, c2, c0, 2
c3	DACR, Domain Access Control Register	MCR p15, 0, <Rt>, c3, c0, 0
c10	PRRR. Primary Region Remap Register	MCR p15, 0, <Rt>, c10, c2, 0
	NMRR, Normal Memory Remap Register	MCR p15, 0, <Rt>, c10, c2, 1
c12	VBAR, Vector Base Address Register	MCR p15, 0, <Rt>, c12, c0, 0
	MVBAR, Monitor Vector Base Address Register	MCR p15, 0, <Rt>, c12, c0, 1
c13	FCSEIDR, FCSE PID Register [a]	MCR p15, 0, <Rt>, c13, c0, 0

a. If the FCSE is implemented. The FCSE PID Register is RAZWI if the FCSE is not implemented.

(b)

Fig. 3.4 (a) Core registers in each mode, (b) Secure registers affected by CP15SDISABLE.

enables key secure privileged features to be locked in a known good state as an additional level of overall system security.

3.1.1 ARM VMSA

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 provides 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 Secure state (e.g., EL1&0 stage 1 MMU) and for operation in 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. Figure 3.5 shows these stages of memory system control in VMSAv8. Each MMU uses a set of address translations and associated memory properties

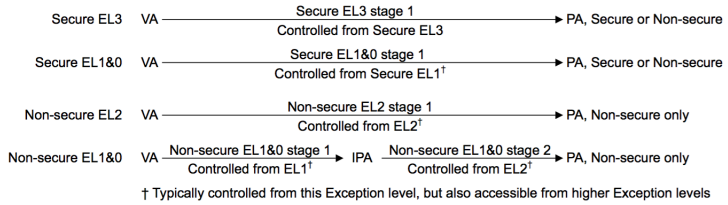


Fig. 3.5 VMSAv8 address translation and MMUs.

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 is valid for when am implementation does not include the virtualization extensions.

System Control coprocessor (CP15) registers control the VMSA, including defining the location of the translation tables. They include registers that contain memory fault status and address information. The MMU supports memory accesses based on memory sections or pages, supersections consist of 16MB blocks of memory, sections consist of 1MB blocks of memory or 64KB blocks of memory, and pages consist of 4KB blocks of memory. Operation of MMUs can be split between two sets of translation tables, defined by the Secure and Non-secure copies of `TTBR0` and `TTBR1`, and controlled by `TTBCR`. For hyp mode stage 1, The `HTTBR` defines the translation table for EL2 MMU, controlled by `HTCR`. For stage 2 translation, The `VTTBR` defines the translation table, controlled by `VTCR`. Access to a memory region is controlled by the access permission bits and the domain field in the TLB entry.

ARM memory domains (MDs)

A domain is a collection of contiguous memory regions. The ARM VMSAv7 architecture supports 16 domains, and each VMSA memory region is assigned to a domain. First-level translation table entries for page tables and sections include a domain field. Translation table entries for super-sections do not include a domain field (super-sections are defined as being in domain 0). Second-level translation table entries inherit a domain setting from the parent first-level page table entry. Each TLB entry includes a domain field. A domain field specifies which domain the entry is in, and a two-bit field controls access to each domain in the Domain Access Control Register (`DACR`). Each field enables access to an entire domain to be enabled and disabled very quickly without TLB flushes so that whole memory areas can be swapped in and out of virtual memory very efficiently. Hence `DACR` controls the behavior of each domain and is not guarded by the access permissions for TLB entries in that domain. Also, `DACR` defines the access permission for each of the sixteen memory domains (see Figure 3.6). The `DACR` is a 32-bit read/write register and is accessible only in privileged modes. When the security extensions are implemented `DACR` is a banked register, and write access to the secure copy of the register is disabled when the `CP15SDISABLE` signal is asserted high. To access the `DACR` you read or write the `CP15` registers. For example: '`MRC` $p15,0,<Rt>,c3,c0,0$ ' for reading from `DACR` and '`MCR`

Value	Access types	Description
00	No access	Any access generates a Domain fault
01	Client	Accesses are checked against the access permission bits in the TLB entry
10	Reserved	Using this value has UNPREDICTABLE results
11	Manager	Accesses are not checked against the access permission bits in the TLB entry, so a Permission fault cannot be generated

(a)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D15		D14		D13		D12		D11		D10		D9		D8		D7		D6		D5		D4		D3		D2		D1		D0	

(b)

Fig. 3.6 (a) ARM MDs access permissions, and (b) DACR register.

$p15,0,<Rt>,c3,c0,0$' for writtig to DACR. Data Fault Status Register (DFSR) holds status information about the last data fault in MDs. It is a 32-bit read/write register, accessible only in privileged modes. These registers are banked when security extensions are enabled, so we could have separate 16 domains inside TrustZone secure world as well as the normal world.

Though ARM memory domains are a useful isolation primitive in concept, the current hardware implementation and OS support suffer from significant problems that have prevented their broader adoption:

Scalability: ARM relies on a 32-bit DACR register and so supports only up to 16 domains. Allocating a larger register (e.g., 512 bits) would mean larger page table entries or additional storage for domain IDs.

Flexibility: Unlike Intel MPK, ARM-MDs only apply to first-level entries; the second-level entries inherit the same permissions. This prevents arbitrary granularity of memory protections to small page boundaries and reduces the performance of some applications [252]. Also, the DACR access control options do not directly mark a domain as read-only, write-only, or exec-only. So the higher-level VM abstraction should resolve these issues.

Performance: Changing the DACR is a fast but privileged operation, so any change of domain access permissions from userspace require a system call. This is unlike Intel MPK that makes its Protection Key Rights Register (PKRU) accessible directly from userspace.

Userspace: There is no Linux userspace interface for using ARM-MD; it is only used within the kernel to map the kernel and userspace into separate domains. In contrast, Linux already provides some basic support for utilizing Intel MPK from userspace.

Security: Though the DACR is only accessible in privileged mode, any syscall that changes this register is a potential breach that could cause the attacker to gain full control of the host kernel (e.g., through the misuse of the `put_user/get_user` kernel API in CVE-2013-6282). Also, since only 16 domains are supported, guessing other domains' identifiers is trivial, making it essential not to expose these directly to application code.

Address space identifier

The VMSA permits TLBs to hold any translation table entry that does not directly cause a translation fault or an access flag fault. To reduce the software overhead of TLB maintenance, the VMSA differentiates between *global pages* and *process-specific pages* through the Address

Space Identifier (ASID). A global virtual memory page is available for all processes on the system, and a single cache entry can exist for this page translation in the TLB. A non-global virtual memory page is process-specific, associated with a specific ASID. The ASID identifies pages associated with a specific process and provides a mechanism for changing process-specific tables without maintaining the TLB structures. Hence, multiple TLB entries can exist for the same page translation, but only TLB entries that are associated with the current ASID are available to the CPU (x86 supports a similar mechanism, called PCID). On ARMv7, the current ASID is defined by the Context ID Register (CONTEXTIDR), and on ARMv8, the ASID is defined by the translation table base registers that causes better performance compare to ARMv7. Each TTBR contains an ASID field, and the TTBCR.A1 field selects which ASID to use. If the implementation supports 16 bits of ASID, then the upper 8 bits of the ASID must be written to 0 by software when the context being affected only uses 8 bits. ASIDs/PCIDs are useful for relatively faster context switching [59] and more efficient page table isolation as shown in design of kernel page-table isolation (KPTI or PTI, previously called KAISER [253]) for mitigating Meltdown vulnerability [139].

3.1.2 Hyp mode and TrustZone virtualisation

The ARM virtualization extension is full system virtualization of all hardware resources, which include memory, CPU, devices, and also entire ISA. It not only adds a new hyp mode (EL2/PL2) in the CPU but brings a two-stage address translation to the VMSA. In stage-1, a VA is translated to an intermediate physical address (IPA), and in stage-2, the IPA is further translated to the corresponding PA. The stage-1 page table is controlled by guest OS, and the stage-2 page table is controlled by the hypervisor. TrustZone only switches CPU between the normal world and secure world through the monitor mode; other hardware resources such as memory and peripherals could also physically be splitted among these two states. Hence, it is possible to use TrustZone as a virtualization tool, but it can only host two isolated operating systems; in this case, the monitor (EL3) will serve as a hypervisor that has the highest privilege and is more security-critical than hyp mode.

The Hypervisor Call instruction (HVC, requests a hypervisor function, causing the processor to enter Hyp mode. The HVC exception return is performed by an ERET instruction, using the SPSR and ELR_hyp values generated by the exception entry and its execution transfers the immediate argument of the instruction to the HSR. The exception handler retrieves the argument from the HSR and therefore does not have to access the original HVC instruction. The virtual machine identifier (VMID) identifies the current virtual machine with its own independent ASID space. The TLB entries include this VMID information, meaning TLBs do not require explicit invalidation when changing from one virtual machine to another if the virtual machines have different VMIDs. For stage-2 translations, all translations are associated with the current VMID, and there is no concept of global entries.

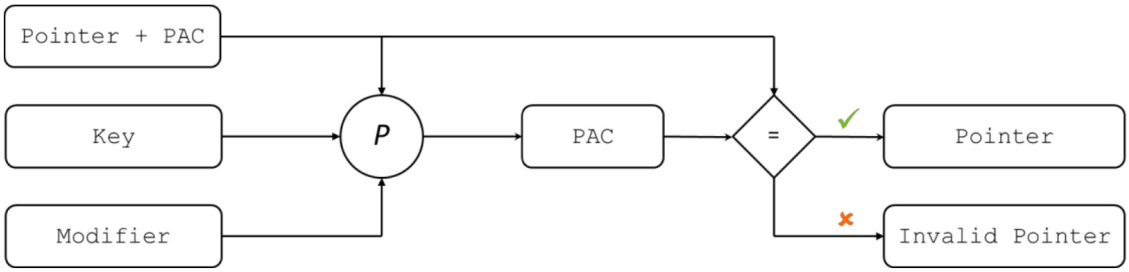


Fig. 3.7 The PAC authentication process.

Before the Armv8.4, TrustZone is not virtualizable. Hence on a virtualized environment, e.g., cloud, all VMs on the same host have to share one TEE kernel that is not only inefficient but may also cause security issues since there is no way for each VM/vendor to deploy its own TEE codebase. This means that cloud users are restricted to using the root key controlled by the vendor to sign their TAs and have to trust the only TEE kernel provided by the vendor, which is the single point of breach as shown in various security vulnerabilities in major vendors' TEE-kernel [68, 231, 254]. To resolve these issues, ARM recently introduced the support of secure EL2 inside the secure world [255, 256]. This brings the features that are available for virtualization in the non-secure state to the secure state. Secure EL2 and SMMU adds stage-2 translations to the processors' secure state and IO virtualisation. Therefore, secure virtualization enables the isolation of EL3 software from SEL1 codebase and the isolation of distinct SEL1 software components from each other. However, due to the lack of development environment, it is too early for evaluating this feature's performance and security implications.

3.1.3 Memory safety extensions

Recently, ARM introduced multiple memory safety features to ARMv8 architecture. With the current trend, soon we will observe supporting all these features in a single SoC.

Pointer authentication

ARMv8.3-A introduced the option of pointer authentication mainly for mitigating ROP attacks. Pointer authentication takes advantage of the fact that pointers are stored in 64-bit format, but not all those bits are needed to represent the address. When pointer authentication is enabled, the unused upper bits store a signature and are not treated as part of the address. This signature is referred to as a Pointer Authentication Code (PAC). The exact number of bits that are available for the PAC depends on the configured size of the virtual address space, and on whether tagged pointers are enabled. The smaller the virtual address space, the more bits are available. To protect against ROP attacks using PAC, the return address in the *LR* should be signed. This means that a PAC is added in the upper order bits of the register. Then, before returning, the return address is authenticated using the PAC. If the check fails, an exception is generated when the address is used for a branch. This change makes ROP attacks much harder to launch;

because to form the chain of gadgets, the attacker needs to know the location of those gadgets and correctly signed pointers to those locations. To get a signed pointer, it would need access to the signing gadget.

The architecture provides five 128-bit keys. Each key is stored in a pair of 64-bit System registers, including two keys (A and B) for instruction pointers, two keys for data pointers, and one key for general use. The registers that store these keys are only accessible at EL1 and above. For data and instruction addresses, the instruction used to create and check the PAC specifies whether the A key or the B key is used (x in Figure 3.8). For a particular pointer, the instruction that generates the PAC and the instruction that authenticates the PAC must agree on which key to use. The signature is formed from the address itself, the key, and a modifier, as shown in Figure 3.7.

PACIxSP	Sign LR, using SP as the modifier.
PACIxZ	Sign LR, using 0 as the modifier.
PACIx	Sign Xn, using a general-purpose register as modifier.
AUTIxSP	Authenticate LR, using SP as the modifier.
AUTIxZ	Authenticate LR, using 0 as the modifier.
AUTIx	Authenticate Xn, using a general-purpose register as modifier.
BRAx	Indirect branch with pointer authentication.
BLRAx	Indirect branch with link, with pointer authentication.
RETAx	Function return with pointer authentication.
ERETAx	Exception return with pointer authentication.

Fig. 3.8 New aarch64 instructions to support PAC.

Tagged memory extension

Memory Tagging Extension (MTE), also called memory coloring, is introduced in Armv8.5-A. Memory locations are tagged by adding four bits of metadata to each 16 bytes of physical memory (this is the Tag Granule). Tagging memory implements the lock. Hence, pointers and virtual addresses are modified to contain the key. In order to implement the key bits without requiring larger pointers, MTE uses the TBI (top byte ignore) feature of the Armv8-A Architecture. When TBI is enabled, the top byte of a virtual address is ignored when using it as an input for address translation similar to PAC design. This allows the top byte to store metadata. Memory tagging and pointer authentication both use the upper bits of an address to store additional information about the pointer: a tag for memory tagging, and a PAC for pointer authentication. Both technologies can be enabled at the same time. The size of the PAC is variable, depending on the size of the virtual address space. When memory tagging is enabled at the same time, there are fewer bits available for the PAC.

MTE adds a new memory type, Normal Tagged Memory, to the Arm Architecture. A mismatch between the tag in the address and the tag in memory can be configured to cause a synchronous exception or to be asynchronously reported. When the asynchronous mode is enabled, upon fault, the PE updates the `TFSR_EL1` register. Then the kernel detects the change during context switching, return to `EL0`, kernel entry from `EL1`, or kernel exit to `EL1`. When enabled[1], a call to `malloc()` will allocate the memory and assign a tag for the buffer. The returned pointer will include the allocated tag. If software using the pointer goes beyond the limits of the buffer, the tag comparison check will fail. This failure will allow us to detect the overrun. Similarly, for use-after-free, on the call to `malloc()` the buffer gets allocated in memory and assigned a tag value. The pointer that is returned by `malloc()` includes this tag. The C library might change the tag when the memory is released. If the software continues to use the old pointer, it will have the old tag value, and the tag-checking process will catch it.

Morello's hardware capabilities

The Morello project aims to introduce CHERI ISA (Version 8) to Armv8-A architecture while providing backward compatibility. It implements 129-bit CHERI capabilities with compressed bounds, which compromise memory consumption and bounds precision. General-purpose registers, certain system registers, and certain special-purpose registers are extended to 129 bits to hold capabilities. A Program Counter Capability(PCC) extends the existing Program Counter(PC) to be a capability, providing validity, permission, bounds, and other checks on instruction fetch, along with some ambient permissions on specific classes of instructions. To prevent capability forgery, bit 128 of a capability containing the capability tag, stored in a separate location that is not accessible by normal load and store instructions. The other 128 bits of the capability are stored in regular memory locations. The Morello architecture extends the VMSA with new permissions in page table entries to control access to capabilities in memory, and also to track the writing of capabilities to memory. A group of TLB registers, `TTBRy_ELx`, and Capability Control Registers, `CCTLR_ELx`, are added (the value of x and y depends on the relevant translation stage and the translation table). When the Morello architecture is implemented, MMU capability access controls provide control of access to valid capabilities in memory. For the purpose of MMU capability access controls, an atomic access is treated as both loading and storing a capability.

A memory location can be marked as faulting stores of valid capabilities; then a store of a valid capability to that location causes a capability access fault, and the write to the location does not occur. Each stage of translation for a translation regime can mark a location as faulting stores of valid capabilities. If a stage of translation for a translation regime is disabled, that stage of translation does not cause a capability access fault due to a store of a valid capability. If an exception due to a Capability access fault on a store of a valid capability is taken to `ELx`, the lowest faulting address is recorded in `FAR_ELx` or the exception is reported as a write in

[1]MTE is currently supported by LLVM

ESR_ELx.WnR during capability access fault on a store of a valid capability as part of an atomic access. A memory location can also be marked as tracking stores of valid capabilities. For each stage of translation, the following registers contain hardware use control bits for the block and page descriptor fields used by the Morello architecture.

Hardware use control bit	Translation stage	Corresponding Block and Page descriptor bit
TCR_ELx.HWU62	Stage 1	LC, bit 62
TCR_ELx.HWU61	Stage 1	LC, bit 61
TCR_ELx.HWU60	Stage 1	SC, bit 60
TCR_ELx.HWU59	Stage 1	CDBM, bit 59
VTCR_EL2.HWU61	Stage 2	LC, bit 61
VTCR_EL2.HWU60	Stage 2	SC, bit 60
VTCR_EL2.HWU59	Stage 2	CDBM, bit 59

Fig. 3.9 ARM Morello ISA changes in each stage of address translation.

Morello software and hardware stacks are actively under development[2,3], adding promising compartmentalizaton features to future devices.

3.2 Compartmentalization in presence of userspace enclaves

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. As we briefly mentioned in Section 2.2.5, 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.

All the memory operands used by the Intel SGX instructions are interpreted as offsets within the data segment (DS). The segment-override prefix on SGX instructions is ignored, and operand size is fixed for each enclave instruction. Any attempt by software executing inside an enclave

[2]https://www.arm.com/why-arm/architecture/cpu/morello
[3]https://www.linaro.org/blog/porting-common-linux-tools-into-morello-architecture/

to modify the processor's segmentation state (e.g., via MOV seg register, POP seg register, LDS, far jump, etc; excluding WRFSBASE/WRGSBASE) results in the generation of a #UD. As with segmentation, enclaves abide by all the paging policies set up by the OS, but they can be more restrictive than the OS. All the memory operands passed into SGX instructions are interpreted as offsets within the DS segment, and the linear addresses generated by combining these offsets with DS segment register are subject to paging-based access control if paging is enabled at the time of the execution of the leaf function. Since the ENCLU[EENTER] and ENCLU[ERESUME] can only be executed when paging is enabled, and since paging cannot be disabled by software running inside an enclave (recall that enclaves always run with CPL=3), enclave execution is always subject to paging-based access control. The Intel SGX access control itself is implemented as an extension to the existing paging modes.

Generally, an EPC (Enclave Page Cache) page is only accessed by the owner of the executing enclave or an instruction that is setting up an EPC page. Pages in the EPC can either be valid or invalid, where every valid page in the EPC belongs to one enclave instance. On implementations in which EPC memory is part of system DRAM, the contents of the EPC are protected by an encryption engine. The EPCM is a secure structure used by the processor to track the contents of the EPC. The EPCM holds one entry for each page in the EPC. The format of the EPCM is micro-architectural and is implementation-dependent. However, the EPCM contains the linear address through which the enclave is allowed to access the page, and the specified read/write/execute permissions on that page. The EPCM structure is used by the CPU in the address-translation flow to enforce access control on the EPC pages.

However, SGX2 allows allocating a new page to an already initialized enclave by invoking the EAUG leaf function (summarised in Table 3.1). Typically, the enclave requests that the OS allocates a new page at a particular location within the enclave's address space. Once allocated, the page remains in a pending state until the enclave executes the corresponding EACCEPT leaf function to accept the new page into the enclave. SGX2 also allows restricting the EPCM permissions associated with an enclave page using the EMODPR leaf function. This operation requires the cooperation of the OS to flush stale entries to the page and to update the page-table permissions of the page to match. Extending the EPCM permissions associated with an enclave page is accomplished directly by the enclave using the EMODPE leaf function. After performing the EPCM permission extension, the enclave requests the OS to update the page table permissions to match the extended permission. Permission extension does not require enclave threads to leave the enclave as TLBs with stale references to the more restrictive permissions will be flushed on demand, but to allow forward progress, an OS needs to be aware that an application might signal a page fault.

3.2.1 Interactions with memory protection features

SGX hardware extensions are not designed for secure and high-performant integration with other types of compartments (e.g., host processes or intra-process sandboxes) and hardware

Instruction	Privilege mode	Description
ENCLS[EAUG]	Supervisor	Allocate EPC page to an existing enclave.
ENCLS[EMODPR]	Supervisor	Restrict page permissions.
ENCLS[EMODT]	Supervisor	Modify EPC page type.
ENCLU[EACCEPT]	Userspace	Accept EPC page into the enclave.
ENCLU[EMODPE]	Userspace	Enhance page permissions.
ENCLU[EACCEPTCOPY]	Userspace	Copy contents to an augmented EPC page and accept the EPC page into the enclave.

Table 3.1 SGX2 memory management instructions

memory isolation features (e.g., MPX, MPK, TXT) that could improve the security of in-enclave code or host applications in general. For instance, SGX hardware support for confidentiality and integrity does not guarantee the actual security of enclaves, especially when memory corruption vulnerabilities, like buffer overflow, exist inside SGX programs. Hence, it is the developers' responsibility to enable traditional memory protection features, such as bound checking or ASLR, inside unprotected linear memory of enclaves if they need proper protection. However, the hardware limitations make it challenging to efficiently utilize underlying hardware to bridge this gap.

As an example, MPTEE [257] proposes a mechanism for supporting dynamic permission enforcement inside enclave memory utilizing MPX registers for more efficient bound checking. It uses three bound registers to offer the six common memory permissions (e.i., RWX, RW, RX, R, X, non-permission) more efficiently than software-only approaches [258]. However, this still causes about 35% overhead on average and 57% in some cases. SGX2 provides two special instructions, `EMODPR` and `EMODPE`, to facilitate the dynamic permission changes. However, the overhead is not clear yet due to the lack of hardware support. Also, SGX1 is used widely and will still be the dominating version for a while. Occlum [62] also proposed an MPX-based SFI technique for supporting in-enclave processes, which adds a 10x slowdown compared to Linux processes. Similarly, EnclaveDom [61] suggests using Intel MPK domains for supporting in-enclave isolation, which shows better results than MPX-based solutions. However, they evaluated the system on simulators since currently there is no public hardware that supports both features simultaneously.

3.3 Dispersed Compartments

Dispersed compartments enable developers to control applications' sensitive partitions inside traditional isolation boundaries. More importantly, they allow compartmentalization across these boundaries for secure sharing, interactions, and data transfers. Hence, they could systematically mitigate various classes of attacks by *simultaneously* tackling two fundamental issues in existing compartmentalization techniques: *(i)* coarse isolation granularity and *(ii)* inextensible security policies. We define a compartment as *dispersed* when it supports mutually distrustful policies for other compartments, either within the *same* or *different* address spaces. This thesis mainly

focuses on TEE/enclave-based compartments to design and evaluate our hetero-compartment model and system for cross-boundary features.

To formally describe the principles of dispersed compartments, we first model each system resource (e.g., memory or file) as an *object* to establish a unified view of resources across privilege boundaries. Each dispersed compartment can be specified by $SPEC\langle P, C\{T, O\}\rangle$ that describes a set of policies P over dispersed compartment C, which contains a set of threads T and other system objects O. P defines integrity, confidentiality, and sharing policies over the needed system objects. Each hetero-compartment (e.g., process or enclave) can specify its own *SPEC* for using dispersed compartments. Then given one or multiple *SPEC*s, our monitoring and enforcement mechanisms should satisfy the following properties:

- **P0** Given a *SPEC*, one or multiple dispersed compartments could be created such that compartment C_i contains a set of threads T_{C_i} and other arbitrary systems objects O_{C_i} i.e., $C_i = \{T_{C_i} \cup O_{C_i}\}$.
- **P1** For dispersed compartment C, no other compartment's thread $\{T' \in C'\}$ can access C objects without a specified sharing policy P_i that is aligned with P_C, so $\{P_i \in P_C\} \wedge \{P_i(T') \in P_{C'}\}$.
- **P2** We assume a hetero-compartment environment has at least two kernels or system runtimes responsible for compartment execution and resource management (e.g., Linux kernel and TrustZone OS). Dispersed compartments can operate and enforce the specified security policies without trusting the host kernel K_h.
- **P3** Authorized threads of each dispersed compartment C can dynamically change their own policy $P_C \rightarrow P'_C$ defined over objects; this includes adding, removing, and sharing objects to/from C.
- **P4** The policy P_C over objects must be enforced at all times (during the execution and rest). This includes after an object is accessed/shared or during unexpected events. During the process of dynamically changing policies, it is possible to have policy inconsistency between dispersed compartments, which could lead to killing compartments trying to enforce the old policy. However, this will not cause sensitive information leakage.

3.3.1 Sensitive system objects

The security principles of dispersed compartments can be applied to any security-sensitive object. However, for designing a practical system, particularly in a hetero-compartment environment, it is essential to decide about supporting which system objects carefully. To this end, we examined 41 existing TEE-assisted open-source applications, either TrustZone- or SGX-enabled, to emulate a hetero-compartment environment (summarised in Table 3.2). As a result, we

Application name	Description	TEE	Use case
ltzvisor	TrustZone-assisted Hypervisor	TZ	Reference monitor
LibSEAL	SEcure Auditing Library for internet services	SGX	Auditing
darknetz	Darknet DNN framework	TZ	Data analytics
sgx-spark	In-enclave Apache Spark	SGX	Data analytics
shadow-box	Kernel Protector	TZ	Reference monitor
SGX-Tor	Tor anonymity network in the SGX	SGX	Web app
TaLoS	TLS Termination Inside Enclaves	SGX	SSL/TLS
MQT-TZ	TrustZone Enabled MQTT Broker	TZ	Data analytics
optee-sks	Library for Secure Key Services	TZ	Key management
Enclave EVM	Enclave Ethereum Virtual Machine	SGX	Blockchain
fabric-optee-chaincode	Hyperledger Fabric chaincode execution	TZ	Blockchain
fabric-private-chaincode	In-enclave Chaincode Execution	SGX	Blockchain
self-healing_FreeRTOS	A self-healing FreeRTOS	TZ	Reference monitor
graphene-httpd	In-enclave httpd	SGX	Web apps
graphene-nginx & redis	In-enclave nginx & redis	SGX	Web apps
rustZone-backed-Bitcoin-Wallet	An embedded Bitcoin wallet	TZ	Blockchain
keyvault	Library for generating, storing and distribute secret keys	TZ	Key management
tzMon	security framework for a mobile game application	TZ	Reference monitor
SGX_SQLite	SQLite database inside an enclave	SGX	Databases
sgx-lkl-MySQL	In-enclave MySQL	SGX	Databases
SGX-OpenSSL	SGX SSL cryptographic library	SGX	SSL/TLS
mbedtls-SGX	A SGX-friendly TLS stack	SGX	SSL/TLS
sgx-ra-tls	Integrate SGX remote attestation into the TLS	SGX	Attestation
WolfSSL	WolfSSL with SGX	SGX	SSL/TLS
slalom	Private Execution of ML	SGX	Data analytics
TresorSGX	Securing storage encryption	SGX	Databases
SGX-LKL tensorflow & pytorch	In-enclave tensorflow & pytorch	SGX	Data analytics
stealthdb	Extension to PostgreSQL leveraging enclaves	SGX	Databases
bolos-enclave	Ledger BOLOS Enclave	SGX	Blockchain
Anonify	A blockchain-agnostic execution environment	SGX	Blockchain
SecureKeeper	In-enclave ZooKeeper	SGX	Web apps
node-secureworker	In-enclave Java Scripts	SGX	Web apps
SGX-Log	Securing System Logs	SGX	Reference monitor
custos	OS Auditing	SGX	Reference monitor
sgxjail	Enclave sandboxing	SGX	Reference monitor
TEE-TLS-delegator	TLS sign delegator	TZ	SSL/TLS
SGX-pwd	passwords distribution	SGX	Key management
Panoply-Tor	In-enclave Tor	SGX	Web apps
openenclave-tpm	Virtual tpm	TZ	Attestation

Table 3.2 The full list of TEE-enabled applications that we analyzed for choosing systems objects to support.

identified various security threats within applications, TAs, enclaves, and across trust layers, which leads to sensitive data compromise

We looked at a diverse range of applications as well as existing attacks on hetero-compartment use cases. As summarised in Table 3.3, 92% of TEE-enabled applications we analyzed suffer from vulnerabilities where at least three system objects of memory, threads, or IPC/RPC (including network sockets, files, pipes, or shared memory) were involved (e.g., as a target object or for propagating vulnerabilities). Therefore, to significantly reduce the attack surface of such applications, enabling principles of dispersed compartments over these system objects is a must.

3.3.2 Threat Model

We consider two threat models based on trusting the host OS or not. By default, we assume that the attacker controls the OS as well as the host applications executing in userspace. It can access and corrupt the host kernel and process resources (e.g., memory, threads structures, network, files, locks). An attacker can also exploit existing vulnerabilities inside an enclave and

	App Category	Number	Thread	Memory	IPC/RPC	Priv
TEE-enabled Applications	Reference monitor & Auditing	8	◐	●	●	●
	Web apps	7	●	●	●	●
	Data analytics	5	●	◐	●	●
	Key management	4		◐	◐	◐
	Attestation	2		●	◐	◐
	Databases	4	●	◐	●	●
	SSL/TLS	5	◐	●	●	●
	Blockchain	6	◐	●	●	●
	HPE [44]	95			●	●
	BOOMERANG [43]			●	●	◐
	COIN Attacks [42]	10	●	●	●	◐

● Object/feature is involved, ◐ partially involved Priv: Inadequate Privilege separation
Table 3.3 Summary of analyzing TEE-enabled applications and known attacks.

leverage them to access and export secrets from the enclave. We assume application developers correctly specify complete and sound security policies through our userspace API and tools. We consider each dispersed compartment a unit of isolation, so our system must defend against cross-compartment attacks. It restricts the attacker from propagating the vulnerability to other dispersed compartments. In this model, vulnerability propagation and full-system compromise are far harder, even if one of the kernels is compromised. Hence, our TCB only contains a security monitor (SSM) running at the highest privilege mode on the system (EL3 on ARM, M-mode on RISC-V, SMM on x86_64), and a TEE runtime.

In the second (and weaker) threat model, we allow developers to trust the host OS to enforce each dispersed compartment's security policies. Note that in both models, we trust the enclave runtime. We again assume a userspace attacker in both the secure and normal worlds who could gain full control of a thread inside the host application or TA/enclave. The attacker could use OS services, memory operations, and spawn more threads up to the resource limits. Covert or side-channel attacks at the software or micro-architectural level are out-of-scope [141, 138, 259–261, 139, 140, 262].

3.3.3 Dispersed Monitoring

Achieving **P0-P4** requires a new monitoring mechanism to enable *checking* policies across traditional privilege boundaries. Dispersed monitoring provides an adequate view of the system for each dispersed compartment, but it is not responsible for ensuring policy enforcement based on our threat model. Unlike traditional centralized systems, our monitoring mechanism requires tracking information flows between different isolation and trust boundaries. It is our main mechanism for introducing the concept of hetero-compartment into the underlying system. For example, dispersed monitoring enables the system to check dispersed compartments security policies for mutually untrusted processes on the host, within enclaves, or within sensitive

interactions between these two. The monitoring mechanism verifies policies based on checking system object labels and capabilities. However, it fully trusts that these object credentials are not compromised and reports any policy violations based on this assumption. Hence, the monitoring is not responsible for guaranteeing the integrity of our labeling mechanism or removing untrusted components, such as the host kernel (**P2**), from the TCB. These are the responsibilities of our dispersed enforcement mechanism.

Consider the following pseudocode (listing 3.1) for creating a dispersed compartment between the host application and its enclave. We will describe details of our API in chapter 5, but briefly, in this pseudocode, on the application side, the programmer creates a labeled compartment $C1$ (L2) and a thread t_a (L3) and associates them with each other (L4). The compartment $C1$ metadata is then passed to the enclave initialisation portion of the application. We omit the details of initialisation of the enclave processes as it is not important to understanding the operations described here.

```
1    init_application(): // app initialisation
2    c1 = s_create(S_LABEL); //create a dispersed compartment
3    t_a = s_clone(); //create a thread object
4    s_add(c1, t_a); //add the thread object to c1
5    e = s_create_enclave(c1); //passes c1 metadata to the enclave
6
7    // ...rest of main application
```

Listing 3.1 Pseudocode of creating a dispersed compartment between an app and enclave

Dispersed monitoring enables the enclave side to have a unified view of $C1$ with the host process. It allows both sides to monitor their security policies over $C1$ and any associated objects. For instance, now the enclave could share a memory region with only $C1 \rightarrow t_a$ and not with any other compartment or threads inside the host application. As shown in pseudocode 3.2, enclave can define its shared memory region (shm_{ae}) as read-only (L5) to $C1 \rightarrow t_a$. Dispersed monitoring ensures that the underlying system can track and check fine-grained policies between various compartment types. However, enabling dispersed monitoring securely and efficiently is challenging and requires hardware-assisted building blocks that we discuss later in this chapter (§3.4).

```
1  enclave_main(s_compartment c1):
2    c2 = s_create(S_LABEL); //create an in-enclave compartment
3    shm_ae = s_vao_create(size); // create a VA (virtual address space) object
4    s_add(c2, shm_ae); //define shm owner as C2
5    s_grant(c1, c1->t_a, shm_ae,RO); // make shm read-only and share it with c1->ta
6
7    // ...rest of enclave code
```

Listing 3.2 Pseudocode of sharing a memory region through dispersed compartments

3.3.4 Dispersed Enforcement

The last essential part of achieving the security principles of dispersed compartments is to provide a *decentralized* mechanism for enforcing security rules on compartments and the underlying system objects. Dispersed enforcement ensures the system security guarantees based on our two threat models and compartmentalization specifications. This mechanism securely integrates multiple local dispersed monitors into a unified system to ensure the whole-system security guarantees. It ensures all essential system objects are covered, and any policy violation or unsafe dataflows can be stopped in the system. This requires guaranteeing the integrity of our labeling mechanism and ensuring untrusted components, which are specified in the threat model, can not manipulate our monitoring or enforcement mechanisms.

For instance, in a TEE-based system, we have at least two OS kernels (or system runtimes) that each manages its own system objects. Therefore, dispersed monitoring requires each kernel to have a small monitoring module. A local enclave monitor launches enclaves and manages in-enclave dispersed compartments. It checks dispersed compartment rules on in-enclave system objects (e.g., memory, threads, RPCs). If the enclave has two privilege modes (e.g., user and kernel), this local monitor can be executed with higher privilege than the enclave-bound program logic. Similarly, the host kernel local monitor manages dispersed compartments inside userspace processes, outside the enclave address spaces, and checks security policies over the process system objects.

Therefore, these local monitors should be unified to ensure the proper protection of dispersed compartments. Dispersed enforcement ensures that the host kernel could be securely removed from the TCB (**P2**), local monitors are updated when dealing with dynamically changing security policies (**P3**), and the policies are enforced at all times of execution (**P4**). Achieving these properties in a practical way requires novel abstractions for utilising the underlying hardware.

3.4 Hardware-assisted abstractions

In this section we present a bottom-up architecture of SIRIUS for enabling dispersed compartments on commodity hardware focusing on AArch32-64 and x86-64. We explain three hardware-based abstractions to enable: *(i)* intra-address space isolation, *(ii)* efficient communication and context switching to implement dispersed monitoring, and *(iii)* SIRIUS security monitor (SSM) that runs in highest privilege mode and is responsible to ensure dispersed enforcement in a secure and high-performant way.

3.4.1 Virtual address space objects

Efficient and secure intra-address space isolation is an essential building block of enabling dispersed compartments. To this end, we introduce virtual address space objects (VAOs) as a new abstraction designed based on the distributed nature of dispersed compartments and their extensible trust model. We implemented VAOs on ARMv7 and x86-64. The VAOs abstraction

aims to enforce the least privilege for dispersed compartments memory accesses via the following principles and assumptions on the underlying hardware.

Fine-grained strong isolation: All dispersed compartments' threads of execution should be able to define their security policies and trust specifications to protect their sensitive resources inside one or multiple VAOs selectively.

Performance: VAOs operations, including launching, running, changing access permissions, and sharing across threads, should cause minimal overhead. Moreover, untrusted parts of applications (i.e., VAOs-independent) should not suffer any overhead.

Efficiency: VAOs should be lightweight and practical for embedded devices running on a few megabytes of memory or slow ARM CPUs. Moreover, the system should support an adequate number of VAOs for different use cases.

Compatibility: VAOs should be portable on various popular hardware features and platforms. Moreover, it is difficult to provide strong security guarantees with no code modifications, and VAOs are no exception. Therefore, VAOs should be implemented without extensive complexity and hardware changes so existing applications can be ported easily.

We define a VAO as a *labeled* and *contiguous* range of virtual memory. Any memory address owned by dispersed compartments can only belong to one VAO. In this way, a large shared memory space such as the heap can be divided into several distinct sets of isolated VAOs. For example, a dispersed compartment can create a private VAO that is only accessible by one of its threads or a partially shared memory VAO such that only threads with explicit privileges can access it. In general, an unauthorized dispersed compartment's thread cannot tamper with a VAO even if there exists a defect in the code of the thread.

VAOs security policy management and isolation enforcement are separated mechanisms for two reasons: first, achieving the compatibility goal by mapping the isolation mechanism to different hardware back-ends; second, achieving a more efficient implementation by moving most of the security policy management into the system software to avoid hardware changes as well as avoiding redundant parts. To this end, we introduce virtual segment descriptors that are used for the security policy management of VAOs. These descriptors are labeled instead of every VAO's memory page to avoid large labels that cause large overhead. They also keep the permissions and other metadata for memory layout of each VAO such as its base address and length as shown in listing 3.3.

Then VAO isolation mechanisms map the labels and the associated virtual segment descriptors to actual hardware isolation mechanism for enforcement. VAOs have only one secrecy label that is a set of unique and randomised tags. Privileges are represented in forms of two capabilities θ^+ and θ^- per tag θ for adding or removing tags to/from labels. The information flow from an execution unit α to VAO θ is allowed only if $SL_\alpha \subseteq SL_\theta$, and integrity flow if $IL_\theta \subseteq IL_\alpha$. When an execution unit has θ^+ capability for VAO θ, it gains the privilege to only access VAO θ *with only the permission set by its owner* (e.g., read, write, or execute). The access privileges

Feature	ARM Memory Domains	Intel MPK
Per process domains	16	16
Access control register	DACR (2 bits per domain, privileged register)	PKRU (2 bits per domain, userspace register)
Access rights	No-access, Full access, MMU default	No-access, write-disable, MMU default
Paging modes	2-level paging (bits 8:5, level 1 entries)	4-level paging (bits 62:59 of PDPTE)
Address space privilege	Privileged & usersapce	Userspace only
Specific page fault	Domain fault	PK fault
Kernel virtual memory API	No support	Limited support (`pkey_mprotect`, `pkey_alloc`, `pkey_free`, and `mmap`)

Table 3.4 ARM MDs vs Intel MPK.

to each VAO can be different; hence, two threads can share a VAO, but each with a different access privileges. Having a θ^- capability lets an execution unit to declassify VAO θ. The declassification allows the thread to modify the VAO memory layout (by adding/removing pages to it), changing permissions, or copying the content to untrusted sources. Unsafe operations like declassifying VAOs or by endorsing a VAO as high-integrity require the thread to be an owner or an authority (`acts-for` relationship). We describe details of integrating the VAO policy management and abstraction to different OS kernel virtual memory abstractions in Chapter 5.

```
1  struct vao_struct {
2      int vao_id;
3      struct label *vao_label;
4      struct mutex vao_mutex;
5      struct vm_segment *vao_range;
6      DECLARE_BITMAP(vseg_Read, vseg_MAX); //operation bitmaps
7      DECLARE_BITMAP(vseg_Write, vseg_MAX);
8      DECLARE_BITMAP(vseg_Execute, vseg_MAX);
9      DECLARE_BITMAP(vseg_Allocate, vseg_MAX); };
```

Listing 3.3 Simplified structure of VAOs

Naively mapping VAOs virtual segment descriptors to underlying hardware does not provide a practical solutions. For example, translating VAOs metadata to MMU descriptors leads to large overhead for per-thread context switches, VAO permission changes, and ensuring the enforcement of security policies dynamically. To resolve these, we first propose a lazy context switching technique using hardware based TLB tagging. Second we partition the dedicated virtual memory of VAOs from the rest of the userspace and kernel as well as major memory operations on them (e.g., `mmap`,`mummap`,`mprotect`,`alloc`,`free`) to bypass the regular kernel virtual memory abstraction and replace them with our simpler and more efficient kernel abstraction (§4.2.4).

Mapping VAOs to hardware features like memory domains (e.g., ARM MD or Intel MPK) could increase the performance, particularly when changing VAOs permissions, but the abstraction also requires to resolve the following hardware limitations.

Scalability: ARM relies on a 32-bit `DACR` register and so supports only up to 16 domains. Similarly MPKs support 16 domains due to only dedicating 4 bits in the page table entries (see Table 3.4). Allocating a larger register (e.g., 512 bits) would mean larger page table entries or additional storage for domain IDs.

Flexibility: Unlike Intel MPK, ARM-MDs only apply to first-level entries; the second-level

entries inherit the same permissions. This prevents arbitrary granularity of memory protections to small page boundaries and reduces the performance of some applications [252]. Also, the DACR access control options do not directly mark a domain as read-only (see Table 3.5), write-only, or exec-only. So our higher-level VM abstraction should resolve these issues.

Performance: Changing the DACR is a fast but privileged operation, so any change of domain access permissions from userspace require a system call. This is unlike Intel MPK that makes its Protection Key Rights Register (PKRU) accessible directly from userspace.

Userspace: There is no Linux userspace interface for using ARM-MD; it is only used within the kernel to map the kernel and userspace into separate domains. In contrast, Linux already provides some basic support for utilizing Intel MPK from userspace. Our new abstraction should consider such differences.

Security: MPK permission changes are not privilege operations, so any userspace adversary can change the permissions. On the other hand, the DACR is only accessible in privileged mode; yet any syscall that changes this register is a potential breach that could cause the attacker to gain full control[4]. Also, since only 16 domains are supported, it is trivial to guess other domains' identifiers, making it essential to not expose these directly to application code.

Mode	Bits	Description
No Access	00	Any access causes a domain fault.
Manager	11	Full accesses with no permissions check.
Client	01	Accesses are checked against the page tables
Reserved	10	Unknown behaviour.

Table 3.5 ARM MDs access permissions.

Later in Chapter 4, we explain details of how our VAO system abstraction resolve such issues to efficiently support different hardware features for enforcing security requirements on multiple platforms (§4.2.4).

3.4.2 Communication channels for dispersed compartments

As another essential building block of enabling dispersed compartments, we provide a hardware-assisted abstraction for enabling secure and high-performant communication channels between dispersed compartments together and with other types of compartments. Our abstraction utilises the underlying hardware to provide: *(i)* fast cross-compartment switching, *(ii)* a novel shared memory and RPC mechanism that follows dispersed compartment principles for controlling and tracking sensitive dataflows, and *(iii)* a mechanism for enabling dispersed monitoring (§3.3.3) over cross-privilege system objects.

[4] An occasion that has happened once already through the misuse of the put_user/get_user kernel API (CVE-2013-6282)

Cross-compartment context switching

Efficient cross-compartment is essential for achieving practical dispersed compartments and good performance. Dispersed compartment context switching mechanism varies depending on two circumstances: first, the type of communication channel, and second, whether they are switching inside the same address space or to another address space that is also platform-dependent. Unlike traditional thread switching, the virtual memory space does not remain the same during dispersed compartments switching. However, compared to processes or enclaves, dispersed compartments are more lightweight execution units with faster context switching.

For an explicit switch, a dispersed compartment's thread may invoke `d_switch` system call to switch to another dispersed compartment's thread. Such a switch atomically stores and changes the labels and capabilities, the VAOs mappings, file table entries, permissions, instruction and stack pointers of the thread. Note that multiple threads may execute simultaneously within the same dispersed compartment. We describe details of integrations of dispersed compartments with traditional kernel threads, and how our modified kernel handles multithreading and `fork/clone` process in the presence of dispersed compartments in Section §4.2.3.

Our abstraction follows a lazy switching approach, which means during an explicit switch, it ensures no VAO virtual segment descriptors are translated and loaded to actual page tables unless the caller thread is actually accessing the VAO. Also, the abstraction relies on hardware support for TLB tagging (e.g., ASID, PCID) to distinguish dispersed compartment pages that belong to different page tables while reducing the number of TLB flushes. Our x86-64 development platform supported 12-bits of PCID, enabling 4096 different page tables to be distinguished, while our ARM implementation supports 8 bits of ASID and 256 tagged TLBs. Whenever a dispersed compartment's thread is switching and actually accessing another dispersed compartments VAOs, the abstraction maps the associated virtual segment descriptor to create an active page table that is ready to be changed, and the kernel sets the `TTBCR/CR3` register to a value containing the ASID/PCID and the address of the first page directory entry of the target VAO. Any cached TLB entries that share this ASID/PCID are considered valid and may be used. Importantly, the entire TLB does not have to be flushed upon a context switch since entries belonging to other ASID/PCIDs are considered invalid by the hardware. This facility reduces the cost of context switches by reducing the frequency of TLB flushes, e.g., up to 98% [263]. The performance improvements could be different based on the underlying hardware or OS kernel versions.

When dispersed compartments are switching between shared VAOs and relying on hardware memory domains, our cross-compartment switching mechanism only enables a fast domain-specific permission change, and the cost is low as context switches between POSIX threads (e.g., $0.9\mu s$ per ARM MD- based VAO switch). When switching to a dispersed compartment located within an TEE/enclave, we need extra steps of checking and updating security policies between the two worlds that varies depends on the TEE system as we will explain in §4.3.

Shared memory channels

he security principles of dispersed compartments enable fine-grained protected sharing of the system objects while preserving ownership. Hence, dispersed compartment threads benefit from mutually distrustful communication channels based on guarded shared VAOs. Each dispersed compartment's thread can create one or multiple VAOs and share them with other dispersed compartments' threads, whether they are running in the same or different address space. Also, dispersed compartments enable dynamic policy changes over shared VAOs, such as revoking the privileges, permissions, or memory layout by the owner/authorized thread. By utilizing hardware domains, our abstraction enforces such dynamic policies more quickly and efficiently than software-only mechanisms.

In general, sharing memory between an enclave and its host application is unsafe and not supported by most TEEs, including SGX enclaves. However, our novel abstraction resolves this issue by enabling each enclave or application thread to preserve its control over shared VAOs. For example, a dispersed compartment inside an enclave e could create a memory object M_1 as read-only and grants the access capability (t^+) to the application thread p. Hence, only p can read from M_1, but it cannot change M_1 layout or permissions since it does not have the declassification capability and $t^- \notin D_p$. Also, other threads in the same process or enclave cannot access M_1 without having the right tag and capabilities in their labels.

Safe RPC interactions

We provide an RPC mechanism for dispersed compartments to communicate cross address spaces, e.g., compartments inside TEE and the host application. It ensures labeling and tracking of transferred data and messages by mapping them to dedicated VAOs. For monitoring the safety of RPC messages between an application and TEE, our RPC mechanism relies on a small hardware-assisted monitor that runs on the highest hardware privilege layer.

As an example, consider a scenario on TrustZone when a dispersed compartment's thread p inside the host application with a secrecy-tag i to its label L_p and privilege set D_p. To communicate with enclave e, the RPC mechanism first transfers messages to the security monitor via a specific `SMC` call. The monitor checks for message safety, and if the flow is approved, the TEE kernel updates e labels with $L_e\{j,i\}$ and $D_e\{j\}$. It should be noted that both threads have each other secrecy tags for bidirectional communication, but with only `plus` capability. The monitor checks the safety of all RPC calls between two worlds. So no unauthorized thread can jump to an enclave entry. The monitor drops unauthorized messages, and the normal kernel kills the violating thread. We describe details of our RPC implementation in §4.3.

Monitoring system objects across privilege layers

Our communication channels are built on top of our low-level dispersed monitoring abstraction that checks whether dataflows are secure based on dispersed compartments security policies. It is responsible for high-performant checking of labeled system objects when they cross

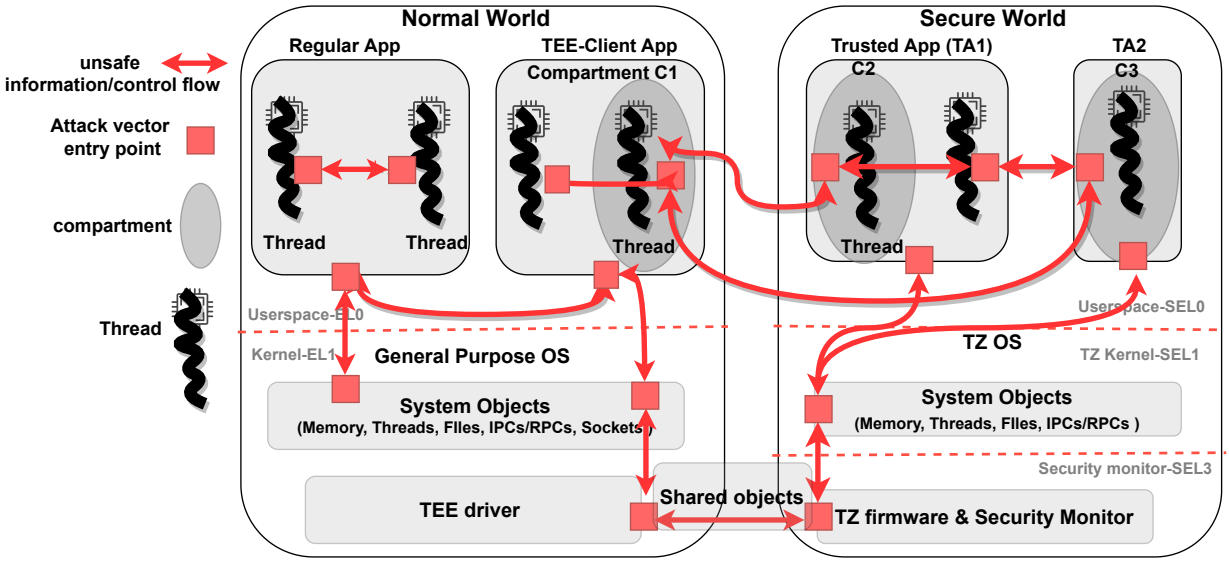


Fig. 3.10 Propagation of sensitive system objects across hardware privilege layers.

various isolation boundaries and hardware protection layers. For example, as demonstrated in Figure 3.10, to be able to monitor the propagation of system objects such as files or VAOs between `EL0, EL1, EL3,SEL1,SEL0` we need hardware support for secure sharing and updating metadata (e.g., labels and capability lists). This includes using the hardware partitioning for building mutually distrustful monitoring modules, and ensure secure exception handling and privilege switching between those modules. We will explain details of implementing this abstraction and efficiently resolving this challenging problem in §4.3.

3.4.3 Hardware-assisted dispersed enforcement

To achieve the design goals of dispersed enforcement, we added a new hardware-assisted security monitor, SSM, to provide: *(i)* fine-grained isolation and security policy enforcement of dispersed compartments beyond the traditional privilege boundaries, and *(ii)* on-demand resource and control management between our two threat models and securely remove the host OS from the TCB when demanded.

Since dispersed compartments are extensible entities, SSM ensures each dispersed compartment could maintain its control over its own objects even when the object is located beyond the dispersed compartment's isolation boundary. For local labeled of each dispersed compartment, our system relies on local monitors within each OS kernel for enforcing safe dataflows. Hence, SSM requires a secure and fast mechanism for integrating these local monitors and providing a *global* and *trusted* view of the whole system objects and security policies for all dispersed compartments.

To achieve this, SSM first ensures secure initialization and execution of verified images and dispersed compartments binaries, particularly inside the TEE environment to maintain their threat model. This involves a hardware-based local and remote attestation mechanism. SSM

also contains an isolated file system and persistent storage, called SIRIUS's trusted file system (STFS), that can not be accessed by unauthorized system software, including the host OS. On ARM architecture, this is achieved via TrustZone-based resource partitioning and on x86-64 via SGX protected memory. As we explain the details in Chapter 4.3.3, STFS ensures the enforcement of security policies over attestation keys, persistent system objects, and during unexpected events such as power loss or rebooting the system.

Moreover, sharing dispersed compartment resources and data while achieving **P2** (removing the kernel from TCB) is non-trivial. Previous works are designed for a single trusted kernel [192, 191] or language runtime [264] that is not suitable for our threat model. SSM resolves this by decoupling policy checking, enforcement, and functionality across multiple monitors. It ensures that despite checking security rules over the host kernel's internal objects based on the application specification, SIRIUS does not trust the kernel to enforce the rules (e.g., jumping over the security hooks) or modify labels (e.g., destroying or copying label). Hence, SSM maps all the labels and capability lists to private memory regions and ensures that the kernel has read-only access. This way, the kernel cannot perform unsafe label operations without being detected.

Our system also relies on the SSM to protect the local monitors and further monitor the integrity of the kernel to ensure it does not bypass the security-sensitive interfaces and security hooks for checking the safe dataflows. In our ARM implementation, this includes a TrustZone-based real-time data and control flow kernel integrity enforcement. We will show how this mechanism can intercept critical events and examine their impact on the security of the system before allowing them to get executed(§4.3). When combined, our technique enables the dispersed monitors to securely enforce specified policies across the system in a distributed fashion).

3.5 Summary

This chapter has introduced dispersed compartments (§3.3) and formally explained their security principles for effective compartmentalization in hetero-compartment environments. It thoroughly explained dispersed compartment threat model and requirements. We started with a bottom-up description of enabling dispersed compartments on commodity hardware and OSs. Hence we introduced three novel hardware-assisted abstractions. These abstractions are building blocks of implementing dispersed compartments on commodity hardware with a focus on AArch32-64 and Intel x86-64 (§3.4). The three abstractions utilize the underlying hardware features, e.g., protection domains and isolated execution, to enable: *(i)* intra-address space isolation for dispersed compartments, *(ii)* dispersed monitoring and cross-compartment communications, and *(iii)* dispersed enforcement of security policies while achieving high-performant. We have provided architectural details and limitations of modern hardware security features for privilege separation and isolation as technical background for explaining these hardware abstractions (§3.1 and §3.2). We describe the details of how these building blocks are implemented in Chapter 4,

where we demonstrate how our whole-system compartmentalization framework is implemented and utilizes these hardware-assisted abstractions.

Chapter 4

Bridging the HW/SW semantic gap

This chapter describes the system side of SIRIUS, which implements and enables dispersed compartments principles. We first overview SIRIUS overall architecture and systems modules §4.1. Then we explain the SIRIUS's host OS design and implementation, including our hardware-assisted abstractions, extensions, and Linux kernel modifications (§4.2). Similarly, we next discuss SIRIUS's TEE stacks (§ 4.3) for both TrustZone- and SGX-based TEEs, where we describe our implementation of SSM and how it achieves dispersed monitoring and enforcement. Finally, we finish this chapter by evaluating the performance of SIRIUS system modules (§4.4).

4.1 SIRIUS's architecture overview

SIRIUS is designed to enable dispersed compartments, particularly in a hetero-compartment environment. To this end, SIRIUS contains multiple systems components to integrate dispersed compartments within or across five different types of compartment boundaries and their combinations: processes, intra-processes, SGX enclaves, TrustZone TAs, and intra-TAs. Hence, it needs to provide a unified system to enable dispersed compartments on at least two OSs and achieve dispersed monitoring and enforcement between their system objects. In this section, we present an overview of the SIRIUS components before explaining the details of each of them.

4.1.1 Revisiting the core security model

Dispersed compartments principles (i.e., **P0-P4** in §3.3) are too high-level to *directly* be mapped to the underlying systems for enforcement. Hence, SIRIUS at its core needs an enforceable security model to convert userspace policies into it automatically. There are several security models that could form the base of our desired model. Particularly, we considered access controls (e.g., ACL, DAC, MAC), object capabilities, and information flow control which are widely-used models in systems security. Table 4.1 summarises the suitability of different security models to achieve dispersed compartment principles.

Between these models, both object capabilities and DIFC models could be a potential security model for mapping dispersed compartment policies to the underlying system objects. Hardware-

Security mechanisms	Security Principles				
	P_0	P_1	P_2	P_3	P_4
ACM [105, 265, 266, 107]	◐	◐	-	-	-
Object capability[109, 7, 52]	●	●	◐	◐	◐
IFC[182, 201, 202, 267]	◐	◐	◐	◐	◐
DIFC-OSes [195, 192, 191]	●	◐	◐	●	◐
SIRIUS	●	●	●	●	●

● practical, ◐ large overhead - partial/not feasible

Table 4.1 Feasibility and practicality of using existing security models to achieve dispersed compartments security principles.

based capability systems could provide finer-grained building blocks for a minimally privileged system and are suitable for designing decentralized trust models and extensible systems (**P1, P2**). However, as we discussed earlier (§2.2.2), this fine-granularity comes from complex hardware and system stack modifications. Also, developers need to specify all authorized operations per object-capability; this requires comprehensive knowledge of various system objects and continuous manual updates. In this model, subjects can also pass their object capabilities to other subjects, who can then propagate those directly (**P3**). Thus, it is not easy to control the propagation and audit dispersed compartments. More importantly, it is well known that capability revocation is expensive and can be unmanageable for large systems (**P4**) [102, 268, 7].

Therefore, as we explained in Chapter 2.2.7, core concepts of DIFC offer a more suitable security model for achieving dispersed compartment principles, particularly **P4** that demands dispersed compartments to be able to track and control privileges (e.g., revoke permissions) at all times of execution. Hence, we borrowed the core concept SIRIUS's security model from previous DIFC systems (c.f. Flume [191], Aeolus [190]) to translate the dispersed compartment userspace policies into invariants that a decentralized runtime can check.

In SIRIUS's model, each security principal represents an entity with security interests (e.g., enclaves, threads, processes) over multiple objects, which can also be of interest to other principals. Principals use tags and labels to control the data as it flows through a system. For example, by assigning them secrecy or integrity tags, they express confidentiality and integrity over objects (e.g., VAOs, files, threads). So, for example, when principal P_1 assigns a secrecy tag to object O_1, it marks it as a confidential object that can not be read or accessed by unauthorized principal P_2. Labels are sets of tags, and each principal has only two labels; a secrecy label S_p to keep all secrecy tags and an integrity label I_p for integrity tags. If $t \in S_p$, then the system assumes that P have seen some private data tagged with t, so it can not reveal the data or propagate it to another principal that does not have t in its secrecy label.

Controlling the dataflows is based on specific partial orders of labels. Hence, every principal can enable or restrict a flow by adding or removing tags from labels if they have the *capability*—a right to operate —to do so. When a thread creates a tag, it has authority for that tag. Subsequently,

authority can be delegated to other threads via grants that can also be revoked. SIRIUS maintains the delegation hierarchy for each tag, which form a directed acyclic graph. SIRIUS also allows transitive revocation that removes a particular link from the principal hierarchy or delegation graph only if the thread has the authority. We use two capabilities per tag, t^+ and t^-, which 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 tagged with t, but it can not 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 (that is represented by $D_p = \{t | t^+ \in C_p \wedge t^- \in C_p\}$).

Adding a tag to a secrecy label and removing a tag from an integrity label 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 so it can not be exported to public memory via `memcpy`, network, IPC, files, etc. SIRIUS then follows DIFC rules for enforcing safe dataflows and label changes as described in Chapter 2.2.7. Our userspace API hides these details by providing a higher-level interface for defining dispersed compartment policies and automatically mapping them to labeled compartments and system objects.

4.1.2 Userspace components

Figure 4.1 illustrates high-level architecture of SIRIUS and the three stages of development, build, and deployment. For example, as 4.1 shows, a developer can specify dispersed compartments within an enclave or host application **(1)**. The SIRIUS's build system compiles the source code and create enclave and application binaries **(2,3)**. Then SSM with the help of SIRIUS local monitors parse the specification, convert them to DIFC principles, label objects, isolate compartments, and update them at runtime **(4,5,6)**.

Hence, SIRIUS contains a userspace framework and API for developing dispersed compartments and expressing their security requirements (static and dynamic security policies) in different computing environments like traditional host processes or TEEs/enclaves. SIRIUS APIs allow defining dispersed compartments interfaces and safe communications between compartments. Then, SIRIUS systems modules handle all the semantics complexity of security enforcement. We describe details of the userspace framework and how it facilitates compartmentalizing real-world applications in Chapter 5.

Our build-system receives SIRIUS-enabled source code and configuration files as inputs and cross-compiles the code to generate executable binaries that contain calls to the SIRIUS API and dispersed compartments, and loads all together into an associated userspace execution environment. For example, in Figure 4.1, it creates one enclave binary that loads to into the TEE

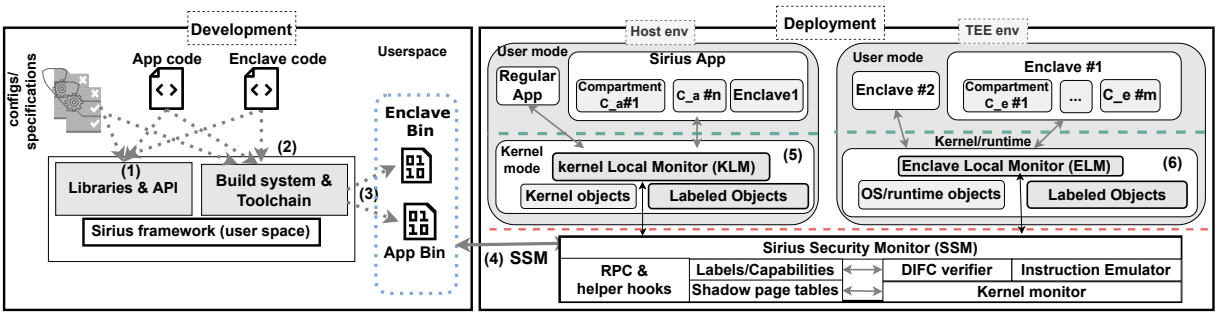


Fig. 4.1 SIRIUS high-level architecture.

environment and one application binary that loads and runs as a userspace process. However, either binary now could contain dispersed compartments and their isolated system objects.

4.1.3 Systems components

Providing simplicity for developers complicates the SIRIUS systems stacks. SIRIUS enables dispersed compartments on two OSs and contains three runtime components that operate in different portions of the host and TEE environments, as shown in Figure 4.1. To properly convert userspace dispersed compartment policies to the underlying SIRIUS's DIFC-based security model and enforce them, SIRIUS requires to introduce dispersed compartments and their hardware-assisted abstractions (§3.4) by extending and modifying corresponding OS kernels. Although the kernel is the best place to check dataflows within kernel objects efficiently, it is a huge codebase exposing over 300 syscalls. Enforcing 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 a huge performance overhead. This is particularly difficult due to SIRIUS supporting intra-address space policies that existing OSes typically do not directly support.

Besides, it needs three dispersed runtime monitors: First, a host kernel local monitor (KLM) manages dispersed compartments in userspace processes and checks DIFC rules over the host's system objects. Second, an enclave local monitor (ELM) launches enclaves, manages in-enclave dispersed compartments, and checks DIFC rules on the enclave's system objects (e.g., VAOs, threads, RPCs). If the enclave has two privilege modes (e.g., user and kernel), then ELM can execute at a higher privilege than the enclave-bound program logic. Third, the SSM runs in the highest privilege mode and ensures partitioning of hardware resources, secure interactions of local monitors, and strong enforcement of policies.

4.2 Enabling SIRIUS on Linux

This section describes our implementation of SIRIUS on Linux kernel that requires modifying existing abstractions (e.g., VFS), adding new hardware-assisted abstractions (e.g., for imple-

menting VAOs), and introducing a new local monitor to implement SIRIUS security model while avoiding conflicts with existing Linux security features. Our Linux kernel implementation adds 22 new syscalls, and is standalone and it can be configured to provide many of dispersed compartments properties on systems with no TEE support. Although we prototyped SIRIUS on Linux-based systems, a similar approach could be used on any other system.

4.2.1 Design principles

Our work on enabling SIRIUS on Linux kernel has fundamental distinctions with previous DIFC OSs. First, it follows a distributed model to monitor dataflows locally and across host-TEE privilege boundaries. We also need to separate policy enforcement and policy management to be able to remove the kernel from TCB when required.

Second, our primary design goal is achieving SIRIUS strong security guarantees while providing good performance. We are not only required to support intra-address space isolation of dispersed compartments but also to enable dispersed monitoring without excessive overhead. Unfortunately, the Linux kernel is not designed to facilitate these features efficiently. That is why previous works are limited to process-level isolation [191] or had to build a non-POSIX kernel from scratch to label intra-address space objects efficiently [269] or support more secure multithreading. Also, any control switch and data exchange to/from TEE is expensive; our technique must reduce this cost to make dispersed monitoring and enforcement practical, particularly for IoT use cases.

Additionally, moving almost all complexity of dispersed compartments and DIFC from the userspace to the kernel and SIRIUS's monitors (for better programmability and performance), should not lead to extreme architectural changes of the kernel. It also should not conflict with the existing security features of the kernel. We achieved this with less than $10K$ LoC kernel changes, which was a non-trivial and challenging task.

4.2.2 Kernel local monitor (KLM)

SIRIUS partitions the Linux kernel to dynamically redirect every call on a tagged object to a separated path toward KLM. KLM is implemented from scratch and as an extension to the Linux security module (LSM) that can be enabled by configuring the kernel with `CONFIG_EXTENDED_LSM_DIFC`. It is implemented at a lower level than LSM, and any existing LSMs could be enabled on top of it. Figure 4.2 shows how KLM design avoids conflicts with existing Linux kernel security features like DAC, LSM, and `seccomp` filtering. KLM implements DIFC principles and adds a few new syscalls and security hooks for managing label operations, capability lists, and checking dataflows. It traps and handles SIRIUS userspace API calls to the kernel. In the threat model that the host OS is fully trusted, the KLM enforces safe dataflows for the kernel objects. However, when the host OS is not trusted, KLM only handles the dataflow *checking* operations and collaborates with SSM to remove the kernel from TCB. In that model,

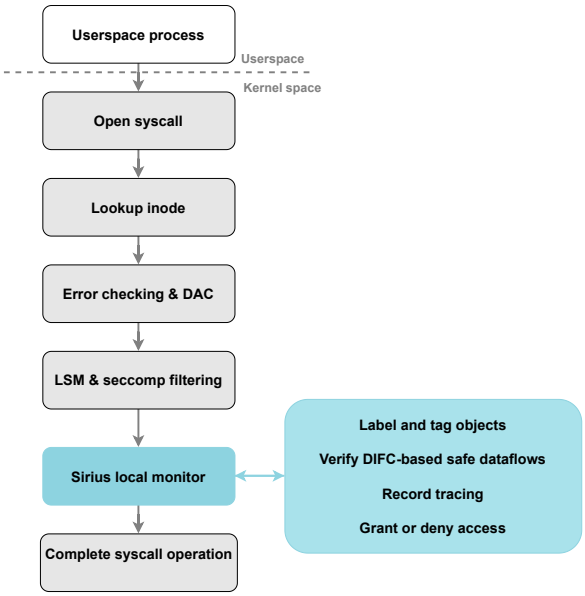


Fig. 4.2 KLM example on the open system call.

when the KLM wants to *change* dataflows by creating or updating labels and capability lists, SSM would be notified (§4.3.3).

Representing tags, labels, & capabilities

As listing 4.1 summarises, tags and their corresponding capabilities (`capability_t`) are presented by unsigned integers. Tags are generated from a monotonically increasing counter and randomized to prevent an attacker from making inferences based on the counter value.

For every capability we use two bits for representing plus (`PLUS_CAPABILITY`)or minus (`MINUS_CAPABILITY`) types. When labeling any object or execution entity, KLM initialises and adds their tags to `object_security_struct->label_struct` and capabilities to `cap_segment` list. We added two helper new syscalls, `alloc_label` and `set_task_label`, for facilitating the process of converting userspace policies and labeling dispersed compartments threads and system objects.

```
#ifdef CONFIG_EXTENDED_LSM_DIFC
    // labels and capabilities related variables & data structs should be here
    typedef uint64_t label_t;
    typedef uint64_t capability_t;
    typedef capability_t* capList_t;
//.............................................
    #define LABEL_LIST_BYTES 512
    #define LABEL_LIST_LABELS (LABEL_LIST_BYTES / sizeof(label_t))
    #define LABEL_LIST_MAX_ENTRIES (LABEL_LIST_BYTES / sizeof(label_t)) - 1
    /*cap lists max size */
    #define CAP_LIST_BYTES 512
    #define CAP_LIST_CAPS (LABEL_LIST_BYTES / sizeof(capability_t))
    #define CAP_LIST_MAX_ENTRIES (CAP_LIST_BYTES / sizeof(capability_t)) - 1
    /* Use the upper two bits for +/- capabilities*/
    #define PLUS_CAPABILITY  (1<<30)
```

```
16      #define MINUS_CAPABILITY (1<<31)
17      /* only 29 bits are used for each tag value */
18      #define CAP_MAX_VAL     (1<<29)
19      #define CAP_LABEL_MASK (0xFFFFFFFF ^ (PLUS_CAPABILITY | MINUS_CAPABILITY))
20  //.............................................
21      struct tag {
22        struct list_head next;
23        unsigned long content;
24        int type;
25      };
26      struct label_struct {
27          label_t sList[LABEL_LIST_LABELS]; //secrecy label
28          label_t iList[LABEL_LIST_LABELS]; //integrity label
29      };
30      struct cap_segment{
31        struct list_head list;
32        capability_t caps[CAP_LIST_CAPS];
33      };
34      struct object_security_struct {
35        struct label_struct label;
36        struct rw_semaphore label_change_sem;
37      };
38  //.............................................
39  #endif //CONFIG_EXTENDED_LSM_DIFC//
```

Listing 4.1 Simplified representation of tags, labels, and capabilities in SIRIUS-enabled Linux

Partitioning sensitive paths

KLM implements a single set of DIFC rules that checks secrecy and integrity dataflows of given objects, including VAOs and threads. For each dispersed compartment, tool's proxy in the syscall exception handler traps and redirects the calls to KLM (L3, code 4.2). KLM performs a lookup over tagged objects (L6). Depending on the object and thread's labels and capability lists, it checks DIFC rules (L8-13).

For each object, we extend its main data structure with a pointer to SIRIUS's metadata that holds its label (e.g., `inode->s_obj_label`) and capabilities. When creating a labeled object, KLM assigns the metadata and forwards the call to the kernel (or TEE/enclave local monitor) for creating the object. When the kernel is not trusted, each metadata is mapped `read-only` to protected memory by the SSM, via a shadow address space, so the kernel cannot manipulate it.

```
1  /// syscall handler ///
2  SYSCALL_DEFINE(x,const type __user *, u_obj,...){
3       if (!check_flow_allowed(&current->s_task,u_obj,obj_type)) do_x(obj,...);}
4  /// inside KLM ///
5  check_flow_allowed(*s_task,u_obj,obj_type){
6      s_obj_label* obj= decode(u_obj,obj_type);
7      if (s_task==NULL && s_obj_label==NULL) return 0;
8      if(s_obj_label->slist!=NULL) {return check_secrecy_flow(s_task,s_obj_label->slist)}
9      if(s_obj_label->ilist!=NULL){return check_integrity_flow(s_task,s_obj_label->ilist)}
10     else {return check_safe_flow(s_task,s_obj_label}} // both secrecy & integrity flows
```

Listing 4.2 Pseudocode of dataflow checking by KLM

Security hooks

SIRIUS extends LSM with 29 new security hooks that are placed in necessary places inside various kernel abstractions to track and control dataflows within tagged objects. Some of the hooks are object-specifics, for example, listing 4.3 shows a pseudoco of `check_tasks_labels_allowed` hook that checks whether the dataflow between two Linux `tasks` are allowed. Similarly, KLM provides security hooks for other kernel objects such as `inode, file, vao, socket, pipe` and etc as we will explain later for each modified abstraction. We will also explain SSM-specific interfaces for collaborating with KLM in 4.3.3.

```
1  static int difc_tasks_labels_allowed(struct task_struct *s_tsk,struct task_struct *d_tsk)
2  {
3    const struct cred *scred;
4    const struct cred *rcred;
5    struct task_security_struct *tsec;
6    struct task_security_struct *rsec;
7    int unlabeled_source_tsk, unlabeled_dest_tsk;
8      //........initialisation checks
9    scred = get_task_cred(s_tsk);
10     if (!scred) {
11       difc_lsm_debug(" no cred!\n");
12         return -ENOMEM;
13     }
14     //........more initialisation checks
15   // check both tasks are labeled first
16   unlabeled_source_tsk = is_task_labeled(s_tsk);
17   unlabeled_dest_tsk = is_task_labeled(d_tsk);
18     //........more checks
19   if(!unlabeled_source_tsk && !unlabeled_dest_tsk)
20   {
21     difc_lsm_debug(" both tasks are labeld! lets check difc allowance then\n");
22     return check_labaling_allowed(&tsec->label, &rsec->label);
23   }else{
24     //difc_lsm_debug(" one of the tasks is not labeld\n");
25     return -1;
26     }
27 }
28
29 static struct security_hook_list sirius_hooks[] __lsm_ro_after_init = {
30 //......... the rest of sirius security hooks ................
31     LSM_HOOK_INIT(check_tasks_labels_allowed, difc_tasks_labels_allowed),
32 };
```

Listing 4.3 Pseudocode of difc_tasks_labels_allowed hook in KLM

4.2.3 Tasks, fork, & clone

Unlike Linux threads, dispersed compartments threads do not share resources by default; each dispersed compartment thread may have multiple tagged system objects attached. To tag a kernel thread we added a new structure,`task_security_struct`, to `task_struct->cred`. Listing 4.4 shows `task_security_struct` that supports both forms of explicit and floating labeling models with a separate set of labels. To protect against concurrency issues, each label structure

and task_security_struct use separate mutex and rw_semaphore. We also had to add new LSM security hooks for labeling threads including difc_cred_alloc_blank, difc_cred_free, difc_cred_prepare and difc_cred_transfer as replacements for cred_alloc_blank, cred_free, cred_prepare and cred_transfer legacy LSM hooks respectively, so we could support both type of threads.

```
1  struct task_security_struct {
2  #ifdef CONFIG_EXTENDED_LSM_DIFC
3      struct label_struct label; //each task has a secrecy or integrity label
4      int type;  //special tag: fthread=1 (floating) ethread=2 (explicit) not_labeld=3
5      struct list_head slabel;
6      struct list_head ilabel;
7      struct list_head olabel;
8  #endif  //CONFIG_EXTENDED_LSM_DIFC//
9
10 #ifdef CONFIG_EXTENDED_FLOATING_DIFC
11     pid_t pid;
12     uid_t uid;
13     struct tag* seclabel; /* Secrecy label  */
14     struct tag* poscaps; /* plus capabilities */
15     struct tag* negcaps; /* minus capabilities */
16 #endif //CONFIG_EXTENDED_FLOATING_DIFC//
17     struct mutex lock;
18 };
```

Listing 4.4 Security structure of a labeled dispersed compartment kernel thread.

There is no concept of inheriting credentials and capabilities by default, for example, in the style of fork, as this makes reasoning about security difficult [115]. The parent thread can explicitly create a child with specified labels as an argument of SIRIUS's new syscall s_clone as an alternative to clone for dispersed compartments (see listing 4.5). We modified copy_creds and copy_process to disallow cred inheritance by allocating an empty cred per dispersed compartment's thread.

```
1  SYSCALL_DEFINE6(s_clone, const char __user *, label, unsigned long, clone_flags, unsigned
       long, newsp,
2       int __user *, parent_tidptr,
3       int __user *, child_tidptr,
4       unsigned long, tls)
```

Listing 4.5 Interface of s_clone syscall for creating dispersed compartments threads.

For enabling dataflows between dispersed compartments threads, it must be through transferring capabilities. Each owner or authority thread (that has the right capabilities) can grant or revoke privileges to/from another thread internally via KLM's difc_transfer_caps security hook (for userspace API, this is done by s_grant and s_revoke). The owner thread can also restrict any access or modifications of its object state by calling difc_access_disable security hook, which temporarily alters the object's tag until difc_access_enable is called by the thread. This is particularly useful for fast intra-process access restrictions when adapting untrusted code or libraries.

4.2.4 VAO kernel abstraction

To achieve sufficient isolation, we need to extend the kernel virtual memory abstractions with a novel mutually distrustful model that lets each dispersed compartment's thread protect its own VAOs (virtual address space objects) from untrusted parts of the same address space as well as other threads and processes. As we briefly explained in Chapter 3.4.1, each VAO is a *labeled* and *contiguous* range of virtual memory that its isolation can be enforced by different hardware features. Simply providing POSIX memory management within hardware-based memory domains (e.g. `malloc` or `mprotect`), without proper compartmentalization semantics, is inadequate. SIRIUS provides relatively small kernel services that would be integrated with its userspace framework for efficient and protected memory management and multithreading (§5.1). Note that VAOs are not protected against covert channels based on shared hardware resources (e.g., a cache). Systems such as Nickel [262] or hardware-assisted platforms such as Hyperflow [193] could be a helpful future addition for side-channel protection on VAOs.

Each VAO 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 dispersed compartments threads would significantly impact system performance, as all the memory operations related to page tables should maintain strict consistency. Instead, we extend `mm_struct` to embed VAO metadata within it as lightweight protected regions in the same address space as shown in listing 4.6. It stores a per-thread `pgd_t` and other metadata for memory management, fault handling, and synchronization.

```
1  struct mm_struct {
2  ...
3  #ifdef CONFIG_EXTENDED_LSM_DIFC
4      struct vao_struct *vao_metadata[VAO_MAX];
5      atomic_t num_vao; /* number of vaos */
6      /*vao Page tables per threads.*/
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_DIFC//
13 ... };
```

Listing 4.6 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. Hence, we extended kernel's `check_and_switch_context` to reload VAO-based page tables and flush related TLB entries if one of the switching threads owns a VAO as shown in listing 4.7. Here, we tag these private page tables via ASID/PCID or map them to hardware memory domains if available for reducing TLB flushes. During this

lightweight switch, the virtual page tables are loaded into the TTBR register on AArch and CR3 on x86 when the dispersed compartment's thread needs to do memory operations inside a VAO.

```
1  static inline void vao_check_and_switch_context(struct mm_struct *mm,
2                struct task_struct *tsk)
3  {
4    if (unlikely(mm->context.vmalloc_seq != init_mm.context.vmalloc_seq))
5      __check_vmalloc_seq(mm);
6
7    if (irqs_disabled())
8      /*
9       * cpu_switch_mm() needs to flush the VIVT caches. To avoid
10      * high interrupt latencies, defer the call and continue
11      * running with the old mm. Since we only support UP systems
12      * on non-ASID CPUs, the old mm will remain valid until the
13      * finish_arch_post_lock_switch() call.
14      */
15     mm->context.switch_pending = 1;
16   else
17     cpu_switch_mm(mm->pgd_vao[tsk->vao_id], mm);
18 }
```

Listing 4.7 VAO-based task context switch in ARM-Linux.

We extended the kernel's memory management module (i.e., mm codebase) with VAO-based paging operations such as allocating, deallocating, mapping, unmapping, tracking, and permission management. As a result, we added 12 new syscalls as summarised in Table 4.2. We later describe details on how SIRIUS's userspace framework uses these syscalls in §5.1. We implemented `vao_ops` and `vao_mem_ops` as multi-purpose syscalls for either initialisation or cleaning up process. Note that all these operations are only allowed after successfully passing SIRIUS's security hooks for ensuring safe information flows. When creating a VAO, users can define the type of isolation enforcement, for example, via hardware domains or MMU-backed private page tables. SIRIUS userspace framework uses these syscalls to create/destroy VAOs and convert dispersed compartments policies by labeling VAOs.

To keep track of VAO-mapped memory ranges and add `vao_mmap/mumap` to manage the memory layout of each VAO, we first modified `mmap.c` inside the kernel. This includes ensuring the virtual memory partitioning of non-SIRIUS processes from memory dedicated to VAOs. We extended `do_mmap` call to check whether the caller thread is labeled and has the right capabilities to change the memory layout of specific address space. Also, it checks for any overlapping virtual memory ranges by keeping lists of dedicated virtual memory ranges to SIRIUS VAOs. Similarly we extended `madvise` syscall and `mm/memory.c` operations to be be aware of VAOs ranges and check for unauthorised operations by labeled threads. When using ARM-MDs for isolation enforcement, during `do_mmap` we set `DACR` register based on the VAO label.

Moreover, for handling larger number of VAOs than supported hardware domains we implemented a VAO caching mechanism and added two syscalls (`vao_cache` and `vao_cache_evict`) for using it. Whenever creating a VAO, if there is a free hardware domain, it maps pages to

VAO-related syscalls	Description
vao_ops	requesting some operations on a VAP such as registering, initialising, creating, and cleaning up
vao_mem_ops	requesting labeling operations on VAO memory block
vao_get	get some helper information about a VAO
vao_set	update some helper information about a VAO
vao_alloc	allocate memory from a VAO range
vao_free	deallocate memory from a VAO range
vao_mprotect	change permissions of a VAO
vao_mmap	map an address space or fd to a VAO
vao_munmap	unmap a VAO's address space or fd
vao_cache	caching a VAO when more than supported hardware domains are used
vao_cache_evict	evicting a VAO from the cache when a hardware domain is available
vao_exec	execute a binary inside a VAO

Table 4.2 SIRIUS syscalls for managing VAOs.

that domain and places the VAO metadata to the cache. When a VAO already exists in the cache, further access to it is fast. When there is no free hardware domain, we have to evict one of the VAOs from the cache and map the new VAO metadata to the freed hardware domain; this requires storing all the necessary information for restoring the evicted VAO, such as its permission, address space range, and label. The caching process can be further optimized by tuning the eviction rate and suitable caching policies similar to libMPK [176].

We extended `mm/mprotect.c` with implementation of `vao_mprotect`, SIRIUS's new syscall for managing permissions of dispersed compartments' VAOs (see listing 4.8). We also modified `mprotect` syscall for checking safe permission changes by labeled threads. The kernel memory fault handler is also extended (`handle_mm_fault`) to specially manage page faults in VAO regions, so a VAO privilege violation results in the handler killing the violating thread.

```
1  SYSCALL_DEFINE4(vao_mprotect, unsigned long, start, size_t, len,
2      unsigned long, prot, int, vao)
3  {
4      return do_mprotect_vao(start, len, prot, vao)
5  ;}
```

Listing 4.8 Interface of vao_mprotect syscall for changing permissions of dispersed compartments VAOs.

When SSM is enabled for monitoring the kernel, for example on the TEE-enabled SIRIUS, SSM page tables dual-map parts of the physical address spaces with the kernel to create shadow address space regions such that the kernel has read-only access to them. SSM uses these shadow regions to map critical parts of the kernel (e.g., the .text, read-only data sections, page-table base addresses) as read-only. The SSM is thus notified if an attacker tries to disable or jump over SIRIUS security hooks by tampering with the kernel code or injecting an unauthorized kernel module. Hence, we extended the kernel MMU operations so that the SSM could monitor events that lead to violating shadow regions like changing page tables or disabling the MMU by its instruction emulator that traps sensitive instructions within the regions (e.g., MCR, LDC,

STP). We also place SSM hooks for event-driven monitoring of the kernel's static invariants (e.g., syscall tables, exception vectors). We explain SSM specific changes in Chapter 4.3.3.

4.2.5 File objects

We modified the VFS layer to enforce thread's security policies within all operations on `inode`, `file`, and VFS address space objects; these kernel abstractions are used to perform operations on unopened files and file handles. As in previous work [191], we assume that write implies read in some file operations; during a write, we can also learn information about the file, for example, its size. SIRIUS continuously checks every operation on `inodes`, which includes files and directories. Therefore, for example, a thread that changed its security label may not be able to read from or write to an open file. Similarly, it checks all operations on tagged files to ensure no thread could operate on them without having the right label and capabilities. This includes any operation that changes file content or other file attributes such as its existence, location, size, or linkage type.

Therefore, the label of an `inode` protects its contents and its metadata (stored in `inode->i_security` structure). In a typical filesystem tree, secrecy increases from the root to the leaves. To ensure writing a new entry in a parent directory does not disclose secret information, we disallow a thread with secrecy label $S\{x\}$ from creating a file with the same secrecy label in an unlabelled directory since it leaks information through the filename. KLM checks and lets a thread with non-empty labels S_p, I_p create a labeled file or directory with labels S_d, I_d, if the label change is safe and the thread can write to the parent directory with its current label. When a dispersed compartment creates a labeled file or directories, they inherit the security context of their creator.

KLM extends the LSM with a few new file-specific security hooks for verifying safe information flows. For example, listing 4.9 shows pseudocode of `difc_inode_set_security` that is used internally for labeling file objects. Similarly, we added a new `inode_permission` security hook to check 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`; hence, we modified the kernel `namei` and placed KLM's security hooks to disallow unauthorized information flow at early lookup stages.

We also extended the `open` syscall with two new flags (`SLABEL` and `ILABEL`) that a thread can use to create a labelled file (e.g. `O_CREAT | SLABEL`) and added our new `file/inode_permission` security hooks on necessary places to disallow unauthorised file operations like `open/close`, `read/write`, `stat`, `seek`, `link`, and so on. A malicious thread may also try to map a labeled file to an address space object via `writepage`. SIRIUS checks that labeled files are only be mapped to protected VAOs with the right labels via `vao_mmap`.

To enable SIRIUS on all file systems as well as kernel drivers we had to extend them with with KLM hooks for either labeling files or checking DIFC rules. However these modifications

are small. For example, listing 4.10 shows the few changes to `dev_mkdir` for creating labeled directory in `tmpfs`-based `/dev`.

```
1  static int difc_inode_set_security(struct inode *inode, const char *name,void *value,
       size_t size, int flags)
2  { struct inode_difc *isec;
3    struct label_struct *user_label;
4    struct tag* new_tag, *t;
5    //initialisation and more variables
6   if(value==NULL)
7    {   tag_content =get_random_long() ;
8    //more initialisation and checks here//
9   }else{ user_label = value;// difc_copy_user_label(value);
10         if(!user_label)
11         {difc_lsm_debug(" Bad user_label\n");
12         return -ENOMEM;}
13         sec_num= (user_label->sList[0] );
14         integ_num=(user_label->iList[0]);}
15   if ( sec_num || integ_num) {
16     if (sec_num && value != NULL) {
17       for(i; i<=sec_num; i++){
18       new_tag = kmem_cache_alloc(tag_struct, GFP_NOFS);
19       isec->type = TAG_EXP;//user_label->sList[sec_num+1];
20       new_tag->content = user_label->sList[i];
21       list_add_tail_rcu(&new_tag->next, &isec->slabel);
22       }}
23 // check all other possibilities ......
24     else if(integ_num && value != NULL) {
25 //update integrity labels
26       }}
27     else {difc_lsm_debug("inode label type is not clear!\n");}}
28   kfree(user_label);
29   return 0;}
```

Listing 4.9 Pseudocode for difc_inode_set_security KLM hook.

```
1  #ifndef CONFIG_EXTENDED_LSM_DIFC
2  static int dev_mkdir(const char *name, umode_t mode)
3  #else
4  static int dev_mkdir(const char *name, umode_t mode,void* label)
5  #endif {
6      struct dentry *dentry;
7    struct path path;
8    int err;
9    dentry = kern_path_create(AT_FDCWD, name, &path, LOOKUP_DIRECTORY);
10   if (IS_ERR(dentry))
11     return PTR_ERR(dentry);
12   #ifndef CONFIG_EXTENDED_LSM_DIFC
13     err = vfs_mkdir(d_inode(path.dentry), dentry, mode);
14   #else
15     err = vfs_mkdir(d_inode(path.dentry), dentry, mode,label); //use sirius vfs interface
16   #endif /*CONFIG_EXTENDED_LSM_DIFC */
17 // check for errors
18   return err;}
```

Listing 4.10 Using SIRIUS VFS interface for labeling directories in tmpfs-based /dev.

For persistence labeling, SIRIUS stores files' labels in the extended attributes. However, when SIRIUS-TEE is enabled, SSM stores them in the secure storage if the file is an enclave-shared/owned object. We later explain how SSM manages shared objects in §4.3.3.

4.2.6 Networking in SIRIUS

SIRIUS extends the kernel to control dataflows within any communication channels in the system. It can proxy these channels dropping messages as appropriate. For example, when thread p sends data to q, or vice-versa, SIRIUS checks the corresponding endpoint labels, silently dropping the data if it is unsafe according to DIFC rules. A receiving thread cannot distinguish between an unsent message, and a message dropped because it is unsafe; therefore, it does not leak information. The endpoints of labeled sockets are mutable, and p and q can change the labels so long as they maintain safe dataflows. Verifying that a thread p has a safe socket label requires information about p's labels, but not information about q's. Thus, if a thread attempts to change a mutable endpoint's label unsafely, the system can safely notify the target thread of the failure and its specific cause.

KLM introduces a few security hooks for enabling safe communications between two threads if they have the right labels and capabilities. For example, SIRIUS does not allow a labeled thread to `connect` to a socket unless that thread has the declassification capability for the accessed secrets. As a result, bidirectional messaging can be conflicting when both threads are labeled. Hence, we added a mechanism for enabling explicit and temporarily bidirectional messaging that is possible when both threads have full ownership of their objects (both plus and minus capabilities). Then using `difc_enable_biendpoints` taints both threads with the same temporary tag and enables the communication.

Similar to file systems, we extended the kernel to enforce DIFC in networking operations like `create`, `listen`, `connect`, `sendmsg`, or `recvmsg`. This was done by placing KLM's security hooks in those functions and at the end of the lookup process, for example, in `sockfd_lookup_light` kernel interface. All operations for unlabelled threads and unlabelled objects follow the traditional Linux access control mechanisms, so applications that do not use SIRIUS do not require any modifications.

4.2.7 SIRIUS Pipes

SIRIUS also controls dispersed compartments communications through pipes. As files pipes and their message buffer are associated with `inodes`. Hence, SIRIUS's pipe labeling is similar to files, and on every operation, the security context of a pipe is checked against the security context of the thread of reading or writing the pipe. We also extended `pipe` syscall with new (SLABEL and ILABEL) flags for facilitating the creation of labeled pipes. A process may read or write to a pipe so long as its labels are compatible with the label of the pipe. Similar to sockets, message delivery over a pipe in SIRIUS is unreliable and done silently. This is because

an error code due to an incorrect label or a full pipe buffer can leak information, so messages that cannot be delivered are just dropped.

SIRIUS ensures that reads from a labeled pipe are non-blocking and readers cannot depend on an explicit end-of-file (EOF) notification if the writer can change labels to prevent unauthorized information flow. As in most DIFC OSs, SIRIUS only delivers EOF notifications if the notification constitutes a permitted information flow.

4.2.8 SIRIUS-assisted TEE driver

Our implementation of SIRIUS on Linux is standalone. When there is no TEE (or other types of compartments) involved, SIRIUS can be used to break down applications' monolithic design and over-privileged components. This is particularly true for mobile and IoT devices that can not adequately benefit from various hardware-assisted security features (e.g., userspace enclaves) due to resource constraints. Hence, SIRIUS framework and set of OS abstractions could enable partitioning applications using dispersed compartments, only within and across userspace processes boundaries

However, SIRIUS's compartmentalization capacity is fully disclosed in hetero-compartment environments. On TEE-enabled systems, SIRIUS securely enables sharing and tracking fine-grained resources between an enclave and its host application that existing TEEs do not support. There are also some TEE/enclave-specific system objects such as RPC that should be protected. For this purpose, we modify TEE/enclave kernel drivers to be extended with dispersed compartments principles. These drivers are platform-dependent. For example, TrustZone and SGX kernel drivers have different architecture and security models, as our modification of each. We explain details of our extensions to TrustZone and SGX drivers in Chapter 4.3.2 and 4.3.4 respectively.

4.3 SIRIUS TEE system

TEEs or secure enclaves form important class of hardware-assisted compartments in our hetero-compartment model of computing. Since currently widespread TEE hardware includes ARM TrustZone and Intel SGX, we focused SIRIUS design and implementation on theses two environments. This section describes SIRIUS TEE system stacks.

4.3.1 Architecture overview

We have implemented SIRIUSdispersed compartments, monitoring, and enforcement on commodity ARM (32/64-bit) hardware with enabled security extensions (TrustZone). However, SIRIUS does not follow the traditional specification of TrustZone TEEs (e.g., Trusty [153], OP-TEE [33]) and instead uses the hardware to efficiently implement dispersed compartments. We also chose ARM to ensure that SIRIUS has a lightweight design suitable for mobile and IoT usecases where data security is paramount.

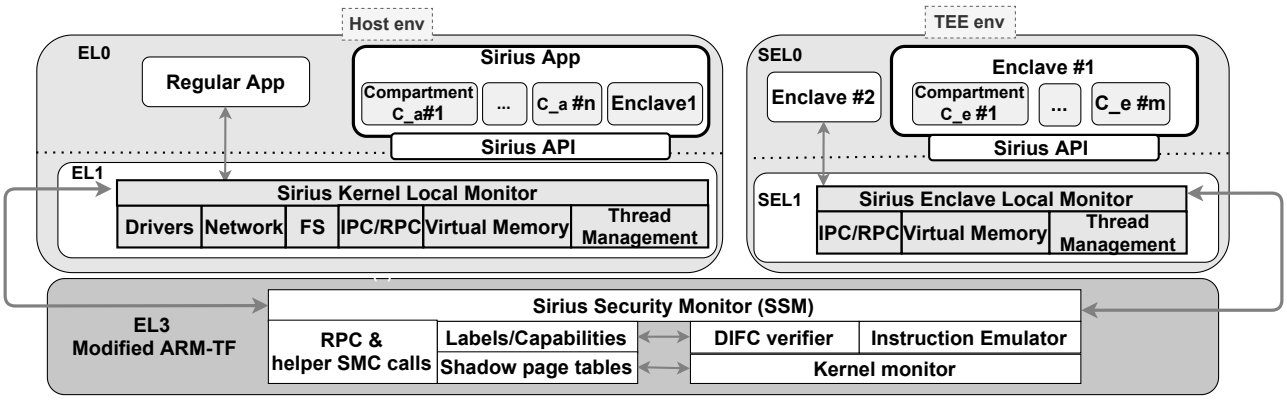


Fig. 4.3 SIRIUS architecture on ARM.

In TrustZone, the secure world has a higher privilege than the normal world. Thus, malicious TAs (trusted apps) or enclaves could lead to compromising the entire system, as shown in previous attacks (e.g., Boomerang [43]). That is why enabling dispersed compartments on ARM significantly improves its current limited security model. To this end, we implemented SIRIUS's TEE kernel, Linux drivers, and SSM as shown in Figure 4.3; application dispersed compartments are represented as C_a and enclave compartments as C_e. SIRIUS stack contains SSM (EL3), SIRIUS's kernel and local monitor (EL1), TrustZone kernel and its local monitor (SEL1) that runs enclaves dispersed compartments in secure world userspace (SEL0).

SIRIUS TrustZone OS is built on top of OP-TEE (V3.4) core kernel, which is a popular open-source TrustZone kernel. We implement the SSM by modifying the ARM-TF (trusted firmware) V2.4 [270]. Since TrustZone by default does not provide hardware-based attestation services like SGX, we integrated Microsoft's vTPM (virtual TPM) [271] into SSM for resolving this limitation, as we explain later in § 4.3.3.

The high-level lifecycle of TAs/enclaves in SIRIUS is not hanged much unless explicitly using dispersed compartments API; only the API semantics and underlying system are aware of DIFC principles. For example, when an application thread p calls `s_create_enclave` (that is part of our enclave API) to spawn an enclave (SIRIUS userspace API is explained in §5.1), from userspace perspective the enclave creation process is not changed. The host OS transfers the request to the TEE runtime, where it verifies and loads enclave binary to an isolated memory after allocating its resources.

However, in SIRIUS, the underlying system first enables a labeled bidirectional channel to ensure safe dataflows. To this end, SIRIUS Linux TEE driver creates a random secrecy-tag x and adds it to the host thread's secrecy label S_p and ownership list D_p. The driver then transfers the message to the SSM via an SMC call. The SSM creates and persistently stores a new tag y for the enclave and notifies the TrustZone kernel to assign both tags to a new enclave userspace thread e, by updating its empty labels to $S_e\{x,y\}$ and $D_e\{y\}$. The SSM enforces message safety

from p to enclave e by checking that $S_p - D_p \subseteq S_e \cup D_e \wedge I_e - D_e \subseteq I_p \cup D_p$, which verifies both secrecy and integrity flows (§2.2.7).

The SSM then passes the y^+ capability to the TEE driver for updating S_p to enable safe bidirectional calls. Note that both threads have each other's secrecy tags but with only the `plus` capability. The TrustZone kernel is the only authority for declassifying (via `s_declassify` API) an enclave tag as well as all shared objects between the two worlds. Both kernels check RPC requests' safety between the two worlds, including function calls via `s_ecall/ocall`, and ensure that no unauthorized thread can jump to an enclave entry. Both kernels drop unauthorized messages and kill the violating thread silently.

4.3.2 The kernel, driver, & ELM

Compared to the Linux kernel, SIRIUS TrsutZone OS has a much less complex design (it provides about 50 syscalls) and supports fewer system objects. We mainly added new abstractions for threading, memory management, shared memory, and RPC functionalities based on dispersed compartments and SIRIUS principles.

Memory management.

The current implementation of SIRIUS only supports short-descriptor translation table format, available on ARMv7-A and ARMv8-A AArch32. SIRIUS uses 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.

Memory layout between small and large translation tables is configured by `TTBRC` register. `TTBR1` points to the large translation table. `TTBR0` points to a small translation table when `SEL0` userspace mapping is active and to the large translation table when no userspace mapping is currently active. The translation tables have certain alignment constraints; for example, the alignment of the physical address has to be the same as the size of the translation table. Due to the alignment constraints, the translation tables are statically allocated to avoid fragmentation of memory. Each kernel thread has one small L1 translation table, and each TA context has a compact representation of its L1 translation table.

TA-specific page tables are managed with the page table cache. When the context of a TA is unmapped, all its page tables are released with `pgt_free` call. All page tables needed when mapping a TA are allocated using `pgt_alloc`. A fixed maximum number of translation tables are available in a pool. One thread may execute a TA that needs all or almost all tables, particularly when a TA has many dispersed compartments. This can block TAs from being executed by other threads. To ensure that all TAs eventually will be permitted to execute `pgt_alloc` temporarily frees eventual tables allocated before waiting for tables to become available.

Each TA binary is partitioned into four parts that are header, init, hashes, and pageable. The header is only used by the loader. In the kernel initialization phase, the loader loads the complete binary into memory and copies what follows the header and init size. The pager is initialized as early as possible during boot in order to minimize the "init" area of the binary. The global variable `tee_mm_vcore` describes the virtual memory range that is covered by the level 2 translation table supplied to `tee_pager_init`. To add physical pages into secure world, `tee_pager_add_pages()` takes the physical address stored in the entry mapping the virtual address (defined by `vaddr` and `npages` parameters) entries and map new pages when needed.

In SIRIUS TrustZone kernel, 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 dispersed compartments VAO abstraction, SIRIUS labels virtual and shared memory MOBJs and uses ELM security hooks (similar to KLM interface) to check and control information flows over them. It also adds ELM DIFC checking hooks on all sensitive MOBJs interfaces.

Thread handling.

SIRIUS allows a static number of threads, that is configurable, to be able to support running jobs in parallel inside a TrustZone TA. On memory constrained devices, these threads could be expensive, mainly because of the execution stack size. Each thread context has a label and list of attached VAOs as shown in listing 4.11.

```
1  enum thread_state {
2    THREAD_STATE_FREE,
3    THREAD_STATE_SUSPENDED,
4    THREAD_STATE_ACTIVE,
5  #ifdef CONFIG_EXTENDED_DIFC
6      THREAD_STATE_LABELED
7  #endif //CONFIG_EXTENDED_DIFC//
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_DIFC
18     struct label_struct label; //keeping secrecy and integrity labels
19     struct mutex m; //for thread-safe label operations
20     struct vao_struct *vao_list[VAO_MAX]; // keeping the list of VAOs
21 #endif //CONFIG_EXTENDED_DIFC//
22   void *rpc_arg;
23   struct mobj *rpc_mobj;
24   struct thread_specific_data tsd;
25   };
```

Listing 4.11 Thread labeling data structures in SIRIUS TrsutZone kernel.

There are handlers for different purposes that could support fast calls, FIQ and PSCI calls. Three primitives are used for the synchronization of threads and CPUs: `spin-lock`, `mutex`, and `condvar`. A `condvar` is similar to a `pthread_condvar_t` in the `pthreads` standard, only less advanced. Condition variables are used to wait for some condition to be fulfilled and are always used together with a mutex. Once a condition variable has been used together with a certain mutex, it must only be used with that mutex until destroyed.

The kernel handles switches of world execution context based on `SMC` exceptions and interrupt notifications. Interrupt notifications are `IRQ/FIQ` exceptions which may also imply switching of world execution context: normal world to secure world or secure world to normal world. SIRIUS provides optional features for enabling DIFC checking within interrupt handlers to avoid an attacker from interrupting an in-enclave dispersed compartment thread. However, depending on use cases, this feature could be expensive so it can be disabled.

Dispersed compartment scheduling, SMC, & RPC.

In-enclave dispersed compartment threads that are launched by a normal world application terminate when the kernel returns to the normal world with a service completion status. So their scheduling depends on the normal world process, though their execution can also be interrupted by a native interrupt. In this case, the native interrupt is handled by the secure kernel interrupt exception handlers, and once served, the kernel returns to the execution of the in-enclave thread. A dispersed compartment thread execution can also be interrupted by a foreign interrupt. In this case, the kernel suspends the thread and invokes the normal world through the RPC service. The dispersed compartment threads will resume only once the normal world invokes the secure kernel with the RPC service status. An in-enclave thread execution can lead the TrustZone kernel to invoke a service from the normal world, such as accessing a file, getting the current time, etc. In that case, the thread is first suspended and then resumed during remote service execution.

When a dispersed compartment's thread is interrupted by a foreign interrupt, and when the secure kernel invokes a normal world service, the normal world gets the opportunity to reschedule the running applications. The in-enclave thread will resume only once the host application is scheduled back. Thus, here also, an in-enclave thread execution follows the scheduling of the normal world caller context. To provide simplicity inside the TEE environment, we do not rely on a complex scheduling system like the Linux kernel. Each in-enclave dispersed compartment is expected to safely track a service that is invoked from the normal world and should return to it with an execution status.

The secure world kernel supports two kinds of notifications to make the normal world aware of some events; synchronous notifications delivered with `RPC_CMD_NOTIFICATION` and asynchronous notifications delivered with a combination of a non-secure interrupt and a fast call from the non-secure interrupt handler. The secure world can wait for a notification to arrive from the normal world. This allows the calling thread to sleep instead of spinning when waiting

for something, for instance, when a thread waits for a mutex to become available. Synchronous notifications are limited by depending on RPC for delivery; hence, secure interrupt handlers or another atomic context cannot use synchronous notifications. Asynchronous notifications use a platform-specific way of triggering a non-secure interrupt. This is done with `itr_raise_pi()` in a way suitable for a secure interrupt handler or another atomic context.

RPC services are mainly built on top of ARM `SMC` calling convention.`SMC`s are categorized in two flavors: fast and yielding. For fast SMCs, the kernel will execute on the entry stack with IRQ/FIQ masked until the execution returns to the normal world. For yielding SMCs, the kernel allocates or resumes a kernel thread for that requested service from the normal world then unmasks the IRQ and FIQ lines. When the kernel needs to invoke the normal world from a foreign interrupt or an RPC, it masks IRQ and FIQ and suspends the kernel thread. Both fast and yielding SMCs end on the entry stack with IRQ and FIQ masked, and the kernel invokes SSM through a `SMC` to return to the normal world.

As we explained in §3.4.2, SIRIUS ensures that dispersed compartment RPC objects are tracked and verified based on DIFC principles. 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. RPC is initiated with a call to `thread_rpc()` which uses ELM interface to label the RPC if required by dispersed compartments policies and saves the state in a way that when the thread is restored, it will continue at the next instruction as if this function did a normal return. CPU switches to use the temporary stack before returning to the normal world.

The Linux driver.

Our TEE Linux driver is a small loadable kernel module that provides a few low-level services such as shared memory management, communication channels, file system, and secure storage to both kernels and the userspace framework. It uses KLM security hooks and interfaces to monitor safe dataflows over shared system objects between the two kernels.

For every bidirectional service between the two kernels, we assigned two threads that their execution is tied together and is under the Linux kernel scheduling decision. The driver is responsible for allocating chunks of shared memory from the normal world that is used to transfer data between both worlds. Hence, the shared memory is allocated and managed by the non-secure world. Depending on the dispersed compartments policy, the shared memory objects could be assigned from VAO abstraction or traditional kernel memory allocators. The shared memory is either used internally for data exchange between the two kernels or externally via userspace request.

It also provides RPC services for using Linux file systems or secure storage inside TrustZone. Typical secure storage in TrustZone is implemented by encrypting memory blocks that are stored in the normal world file system. For example, OPTEE uses a secure storage key (SSK), enclave storage key (ESK), and file encryption key (FEK) for encrypting a persistent storage area in its boot file system. The per-device SSK is generated as a function of the unique hardware key

and chip ID. The SSK must be stored in secure DRAM (protected by TZASC) or in fuses that are not accessible by the normal world and will be used to derive the ESK. Only verified and signed TAs could have access to the encrypted storage. We also added a baremetal file system into the secure world as a replacement to encrypted storage. This is done by modifying and porting littlefs[1] block device file system which is designed for microcontrollers.

4.3.3 TrustZone-based SSM

The main job of SSM is ensuring the security guarantees of dispersed enforcement. Hence, its responsibilities vary depending on the two threat models. In our relaxed threat model, when the host OS is part of the TCB, SSM handles secure initialization of SIRIUS, verifies binaries, checks local monitors, and ensures safe communications between different privileged worlds (in our implementation means normal world and TEE world), and handles restoring security policies and labeled objects from Persistent storage. However, when SIRIUS is configured to remove the host OS from the TCB, SSM is required to handle additional responsibilities for monitoring the Linux kernel to avoid various direct and indirect attacks. On ARM, SSM is implemented as an extension to Trusted Firmware-A (TF-A)[2] that is a widely used firmware for initializing secure world resources, supporting secure boot requirements, power state coordination interface (PSCI), SMC calling convention, and PSA (platform security architecture) update specification.

First, SSM contains a standalone DIFC module and RPC implementation between the two worlds. It compares both sides' labels for protecting the ownership and security policies over shared objects and checks dataflows over RPC requests. Hence, it provides a few SMC calls to request services from both sides, such as allocating more pages to shared regions or storing/restoring persistent labels.

Second, SSM relies on the TrustZone address space controller (TZASC) to manage private shadow address space regions mapped to SSM page tables, which cannot be accessed/modified by the Linux kernel since it runs in a lower privilege than `EL3`. TZASC allows a TrustZone system to configure security access permissions for each address region and controls data transfer between the CPU and dynamic memory controller (DMC). These memory regions then are dual mapped with the Linux kernel providing the kernel read-only access when our strict threat model is enabled and read-write access otherwise. We place labels and capabilities through our custom slab allocator to these regions. Therefore, when the regions are read-only for the kernel, any attempt to modify labels and capabilities (e.g., declassification) would cause an alarm and be trapped inside SSM. Hence, an attacker cannot manipulate dispersed compartments dataflow specifications by changing or removing labels and capabilities.

When deprivileging the kernel, SSM must protect dispersed compartments against multiple attack vectors through several techniques. First, it ensures that a kernel rootkit cannot disable or jump over KLM security hooks. To achieve this, after secure boot, SSM monitors the critical

[1] https://github.com/littlefs-project/littlefs

[2] https://github.com/ARM-software/arm-trusted-firmware

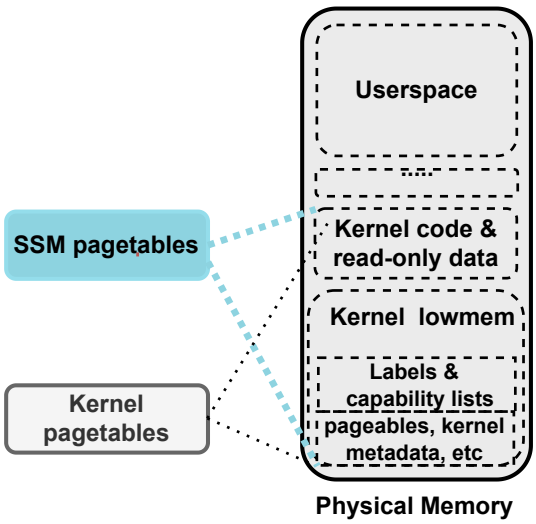


Fig. 4.4 Simplified dual page table mapping on parts of the kernel to SSM page tables.

kernel invariants and hardware instructions to ensure kernel code integrity, prevent injecting malicious code to the kernel or modifying KLM, and prevent privilege escalation from userspace. SIRIUS maps sensitive kernel sections, including `.text` and `lowmem`, as read-only as shown in Figure 4.4. `lowmem` describe the physical memory that is constantly mapped by the kernel. The kernel allocates memory from the `lowmem` when it needs to create new objects such as translation tables. The kernel code also is the only memory range that has the privileged execution permission, and it is mapped as read-only. SSM also ensures that userspace virtual memory ranges are mapped as Privileged eXecute Never (PXN). The kernel `lowmem` is mapped as non-executable privileged memory, so it cannot be accessed by unprivileged core or user processes.

Then SSM's instruction emulator scans for malicious page-table modifications (e.g., altering memory layout or dual memory mappings) or attempts to disable the MMU [272, 147]. To protect against double mapping, SSM provides several security hooks, placed inside the kernel memory management, to record the state of every page of the physical memory. Whenever a new virtual-to-physical mapping is about to be created, SSM will be notified and checks the new access permission given to the physical memory page against the stored state to verify that there is no violation to the memory protection. The state of physical frames is stored in an array called the `ssm_ph_map` where each entry of the array corresponds to a 4KB physical page and holds some status bits as well as a counter to indicate how many virtual page mappings each physical page has. This way, SSM marks physical pages that are currently used by kernel code or translation tables as protected, and any request to create a writable mapping of these pages is rejected. SSM also placed security hooks for intercepting modifications of syscall and exception tables and ensures W+X protection so no other memory can be executable with kernel privilege.

Given the small number of KLM hooks, put together, these mechanisms are sufficient to ensure that the kernel is executing the expected code and SSM catches runtime violations.

To prevent an attacker from redirecting or overwriting the data pointers (e.g., via kernel memory overflows) for bypassing the security hooks, we implemented a simple DFI (data flow integrity) technique, similar to KENALI [273], using SSM shadow regions. SIRIUS forces KLM interface pointers to be within the region's range. This stops attacks that map pointer to outside memory. The SSM event monitoring traps and handles the DFI violations. Because the module is small, $\approx 5K$ LoC, it is practical and fast enough to implement these integrity checkings. For more speedup, in the future, SIRIUS can enforce DFI using hardware support for pointer integrity, such as MPX or ARM's pointer authentication [274, 275]. Given the performance overhead of these hardware features, this approach offers better performance on ARM than x86-64.

Our implementation extends the Linux kernel and firmware and it is hard to guarantee that our modifications are bug-free. However, our SSM-based kernel monitoring restricts attacks that change the kernel code/data at runtime. Note that, SIRIUS also includes misbehaving enclaves in its threat model where a malicious enclave tries to launch attacks on the kernel [43]. However, SSM ensures enclaves and their resources are tagged and cannot access or manipulate kernel state. Enclave cannot bypass these checks since corrupted dispersed compartments will be caught by the enclave runtime.

As a trusted component, SSM runs with the highest privilege and does not trust any lower-privilege components (runtimes, host OS, etc.). Though the SSM can have vulnerabilities, it has low TCB ($\approx$ 3.8K LoC) and low code complexity since it only performs the necessary monitoring. As we explained in our threat model, SIRIUS trusts the hardware, so micro-architectural side-channel attacks are out of scope.

4.3.4 SIRIUS SGX runtime

The main difference between SIRIUS AArch stacks and x86-64 is in TEE systems, including SSM functionalities. Porting the SIRIUS Linux kernel required straightforward engineering (mainly in VAO abstractions and MMU registers). For SIRIUS-SGX implementation, we only implemented the relaxed threat model, as our full ARM-based implementation is sufficient for evaluating the overhead of SSM functionalities. Hence, for example, shadow memory regions are not implemented as well as the kernel data- and control- flow monitoring features. Note that all SSM functionalities on TrustZone could be achieved using Intel VT-x as well. However, for SIRIUS-SGX, we have focused more on implementing a minimal proof-of-concept to evaluate SIRIUS usability and performance for hardening and tracking information flows in the host-enclave boundary.

Therefore, we have built SIRIUS-SGX stack on top of Intel SGX driver[3]and SDK[4]. The driver handles enclave initialisation and resource management and provides few `ioctl` calls for the SDK, for example to manage memory (via `SGX_IOC_ENCLAVE_ADD_PAGE/PAGE_REMOVE`). That is why it is a good place to integrate dispersed compartments management functionalities and integrate VAO-based memory management for shared memory.

However, most of the SSM functionalities, except kernel monitoring services, are ported into SGX Platform Software (PSW), which directly communicates with the driver and is responsible for initializing and loading an enclave memory image, handling enclave exceptions, attestation services, and executing ecalls/ocalls. We similarly ported ELM DIFC module as a standalone library to the SDK that provides proxy calls for managing dispersed compartments and labeling shared resources similar to TrustZone implementation only with less functionality since we currently do not support in-enclave memory isolation.

4.4 Evaluation

In this section, we focus on evaluating SIRIUS system components via microbenchmarks. In Chapter 5, we explain SIRIUS userspace framework and utilize it for evaluating the entire stack on real-world applications.

Setup. We evaluated using a Raspberry Pi 3 Model B [276] that uses a Broadcom BCM2837 SoC with a 1.2 GHz 64-bit quad-core ARM Cortex-A53 processor with $32KB$ L1 and $512KB$ L2 cache memory. We also use NXP's i.MX7Dual boards with two Cortex-A7 CPUs for better evaluation of hardware-protected TrustZone DRAM/SRAM. For evaluating our x86 implementations, we use Intel Core(TM) i7-6820HQ CPU. We use the term *native* for unmodified applications or benchmarks running on an unmodified Linux kernel. We use two sets of microbenchmarks, LMbench 3.0 [277] and a custom benchmark for evaluating SIRIUS overhead on different functionalities. As baseline, we run an unmodified Linux kernel version 4.19.42 and glibc version 2.28.

4.4.1 SIRIUS stacks size

Our Linux kernel extension only adds $\approx 10K$ LoC (summarised in Table 4.3), of which the KLM is $\approx 6.4K$ new LoC and the VAO abstraction modifies $\approx 2.5K$ LoC within the virtual memory layer. The remaining changes are mostly done to VFS and networking layers. On ARM, SIRIUS adds 3.8K LoC for SSM and 2.5K LoC for ELM. Our implementation inherits 53.5K of TCB from OPTEE core kernel and trusted firmware code.

Table 4.4 shows SIRIUS adds smaller code base and memory overhead compared to popular LSMs like SELinux and AppArmor despite significantly stronger security guarantees. This makes it more suitable for IoT/edge use cases.

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

	SSM	TEE system	Linux kernel	Userspace
AArch	3.8K	5.2K	10K	3K
x86-64	1.5K	3K	9.5K	3K

Table 4.3 Total LoC added/modified by SIRIUS.

Overhead	SElinux	AppArmor	SIRIUS
Base Memory(KB)	5365	6348	984
LoC	21266	11918	8782
Number of Hooks	224	68	29

Table 4.4 Existing OS security features are more expensive and much more limited than SIRIUS.

4.4.2 Effects on non-compartmentalized execution

On ARM SIRIUS incurs a fixed overhead of 1.2% and 6.1% for modified ARM-TF initialization and kernel boot respectively. This is when SIRIUS is configured with full security features and the kernel is outside the TCB. The fixed overhead is mainly due to SSM page table setup for shadow address spaces, placing SSM monitoring hooks, mapping kernel image to protected regions, and kernel image hashing. On x86, since SSM features are not fully implemented, the overall overhead of booting the modified Linux is 1.1%.

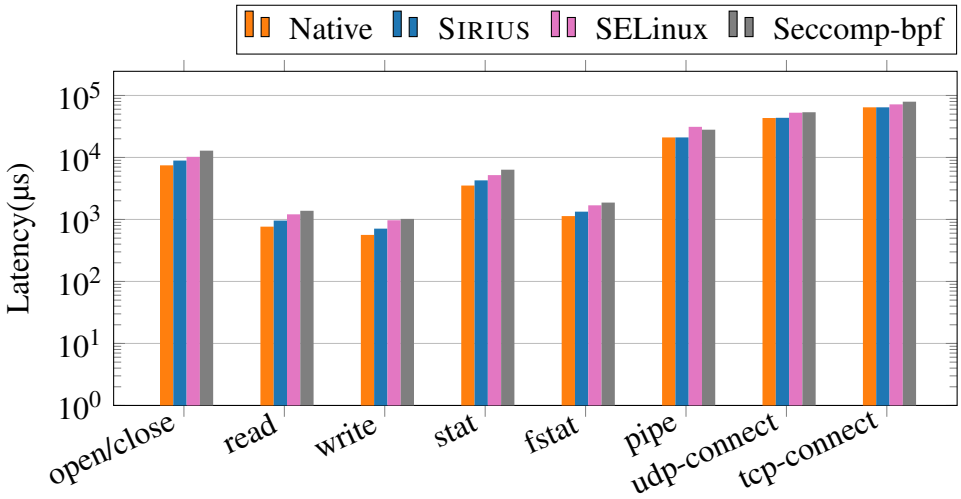


Fig. 4.5 Running Lmbench: overall SIRIUS-ARM overhead (NXP board).

We changed several Linux sub-systems, including virtual memory, file system, networking, task management, and pipes. We empirically show that SIRIUS checks do not add large overhead to the entire system at runtime. We use a system stress test benchmark (LMBench) to measures the worst case overheads per sub-systems as summarised in Figure 4.5. On ARM, SIRIUS adds $\approx$ 16% latency overhead for FS, $\approx$ 0.6% for networking, and $\approx$ 0.2% for IPC benchmarks. Linux file system abstractions are basis of other Linux subsystems like Network, pipes, and IPCs. That is why to make sure we could monitor dataflows across the entire system we added

SIRIUS security hooks on almost all critical paths in FS operations which are by far wider than other Linux abstractions. Hence, the overhead of enabling SIRIUS on all Linux file systems is larger than other Linux services. SIRIUS modifies `fork` to ensure a child thread won't inherits its parents labels and capabilities that adds 0.1% overhead.

On x86, SIRIUS performs better for two reasons: first, the Linux subsystems on modern x86-64 are more optimized and second, SSM strict security features to monitor the kernel are currently not enabled. Therefore, SIRIUS adds $\approx 9\%$ latency overhead for FS, $\approx 0.3\%$ for networking, and $\approx 0.25\%$ for IPC benchmarks.

Running LMbench shows that existing limited Linux security features like SELinux and Seccomp also cost more–their worst case scenario adds 42–80% slowdown.

SIRIUS TrustZone stacks, without enabling dispersed compartments, provide slightly better performance than legacy OPTEE mainly because of replacing its security mechanism with a single address space model of protection that eliminates the need for expensive multiple process-level or enclave-level isolation. For example, SIRIUS provides a secure shared memory mechanism without using encrypted message passing. Our evaluation shows 16% speedup compare to OPTEE (see Table 4.5). The results are average of 2000 runs. We also compared SIRIUS with CSR-based pointer verification that is used for BOOMERANG attack protection, that results in 31% speedup by SIRIUS VAO approach. Note that SFI-based in-enclave isolation techniques adds at least $10\times$ slowdown [62] or need custom hardware [60].

	Latency (µs)	
	OP-TEE	SIRIUS
create enclave	99.82 ± 0.02	93.95 ± 0.01
delete enclave	30.02 ± 0.01	30.10 ± 0.01
enclave calls (ecall+ocall)	22.68 ± 0.01	20.14 ± 0.03
shared memory	9.58 ± 0.02ms	8.16 ± 0.01ms
file system (read/write)	375.8 ± 0.3 ms	34.68 ± 0.01ms

Table 4.5 Average overhead comparison of some of enclave operations in SIRIUS vs OP-TEE.

Also, SIRIUS enables secure sharing of systems resources (e.g., FS) without the need for encrypted channels. Table 4.6 shows that SIRIUS-based FS protection is $\approx 11\times$ faster than encrypted Linux FSes.

This is not the same with our SGX-based implementation. For legacy enclave-based applications (with no dispersed compartments enabled) that rely on SGX SDK, SIRIUS adds about 3% slowdown for ecall/ocall operations on Linux file system. Note that our current implementation does not optimize SIRIUS-SGX system stacks as we did for TrustZone. So the results might be better with a more mature implementation.

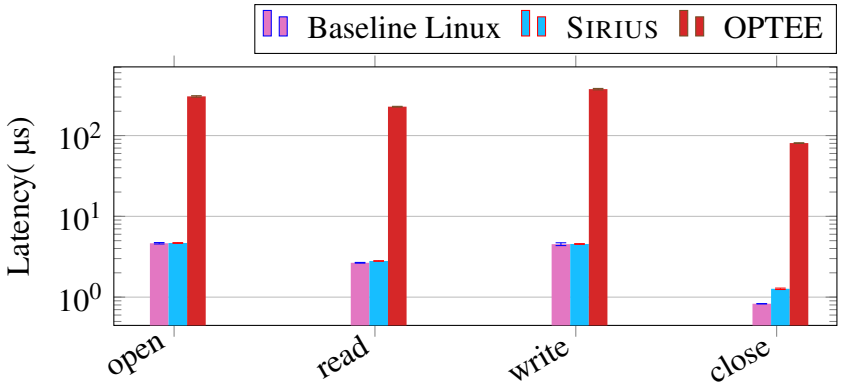


Fig. 4.6 SIRIUS's file protection vs OP-TEE secure storage.

4.4.3 Dispersed compartments operations

We tested the cost of creating and joining dispersed compartment threads on Linux using our new `s_clone` and `s_wait` syscalls. We also run `pthread` and `fork` microbenchmarks on the baseline kernel. Table 4.6 shows the average latency (µs) of 100000 runs with 1MB and 2MB heap sizes. Forking is far more expensive than baseline threads with shared address space. The dispersed compartment threads are a bit slower than `pthreads` due to the overhead of VAO-based isolation and DIFC checking. Hence, the `s_clone` is 1.6*x* slower than `pthread_create` and is ≈ 5.9x faster than standard `fork`. On x86 `s_clone` is 1.2*x* slower than `pthread_create` and ≈ 6.5x faster than standard `fork`.

Operation	fork	pthread	s_clone
Launch (1MB)	280.24	31.17	51.80
Join (1MB)	832.45	1.10	3.78
Launch (2MB)	331.40	31.51	51.85
Join (2MB)	1126.69	1.13	3.82

Table 4.6 Average latency overhead of SIRIUSdispersed compartment threads on ARM-v7.

Another important abstraction that effects dispersed compartments isolation performance is VAOs. Table 4.7 tests the cost of creating and mapping pages to VAOs using `vao_mmap` when VAOs are directly mapped to hardware domains, as compared to virtualized VAOs when there is no free hardware domain and requires evicting VAOs from the cache. The results show that when there is a free hardware domain, the performance improves by 4.9% compare to the virtualized one. Note that creating VAOs is usually a one-time operation at dispersed compartments initial phase. Since we did not have an x86 platform with both MPK and SGX available, our implementation of VAOs on x86 does not support MPK for optimization. Therefore, on x86 `vao_mmap/munmap` add about 6.8% overhead compared to baseline.

Operation	Overhead	stddev
vao_mmap/munmap	10.01%	+- 0.15%
MD-based vao_mmap/munmap	4.8%	+- 0.17%

Table 4.7 Cost of creating VAOs when directly mapped to ARM-MDs vs virtualized mapping that requires VAOs caching.

Changing VAO permissions and memory allocation operations inside VAOs have the most impact on runtime overhead of dispersed compartments. We evaluated the performance comparison of `vao_mprotect` vs glibc `mprotect` based on permission flags. Since ARM-MDs do not have flexible access control options, we cannot benefit from a control switch of domains using the DACR register for all possible permission flags such as the RO, WO, and EO variants. Our results show that on average `vao_mprotect` is 1.17x faster than `mprotect` for no access (PROT_NONE) or RW permissions (PROT_READ | PROT_WRITE), but 1.3x slower for read/write/execute-only options that are emulated (see Table 4.7).

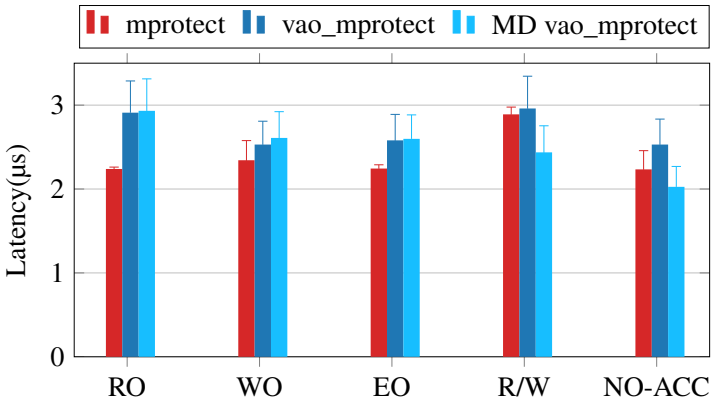


Fig. 4.7 The effect of ARM-MDs on performance of VAO-based permission changes.

Allocating memory using `vao_malloc` is on average 1.08x faster than glibc `malloc` for blocks $\leq 64KB$ and introduces a small overhead (8.3%) for larger blocks ($> 64KB$) as demonstrated in Figure 4.8. This cost can be optimized by using high-performance memory allocators like snmalloc[5]. We report the average of running microbenchmarks 20000 times and show how utilizing VAOs provides small overhead for memory allocation and permission changes.

VAO codebase & memory overhead: Another factor towards the usability of VAOs is the codebase size, which is important both from a security perspective and the resource limitations of small IoT devices. We implemented VAOs as a Linux kernel extension with no dependency on any third-party library. As Table 4.8 shows, on ARM devices it adds less than $4.5K$ LoC in total to both the kernel ($\approx 2.5K$) and userspace ($2K$). It adds $7KB$ to the kernel image size and adds $204KB$ for kernel slabs at runtime. The userspace library only needs $\approx 10KB$ of memory.

[5]https://github.com/microsoft/snmalloc.git

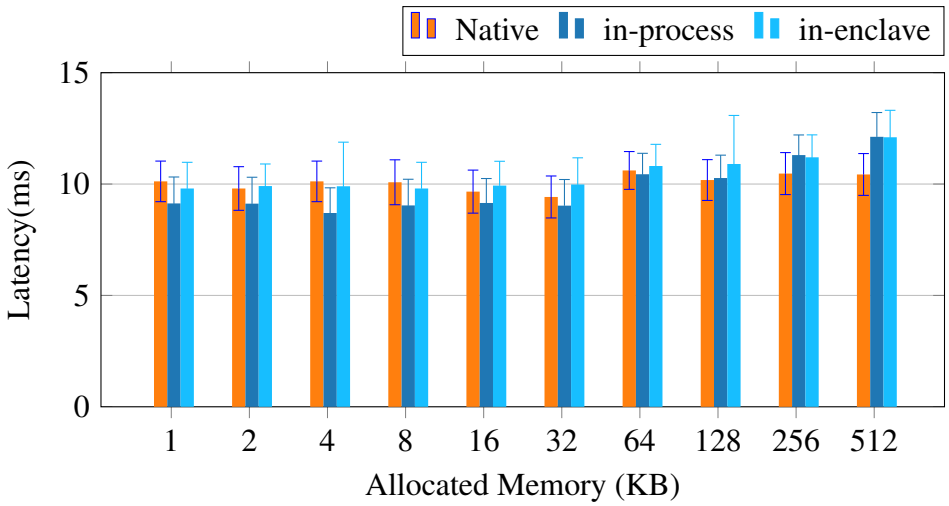


Fig. 4.8 Cost of `s_malloc/s_free` within a VAO on ARMv7.

These results show that the VAOs memory footprint is extremely low and suitable for many resource-constrained uses.

Overhead	Linux Kernel	Userspace
Added LoC	2.5K	2K
Static Memory footprint	static(7KB) slab(204KB)	Static(10KB)

Table 4.8 VAOs codebase size and Memory footprint in the ARM-Linux Kernel and userspace.

Moreover, for every dispersed compartment, the overall size of its label would effect the performance of SIRIUS operations. Figure 4.9 shows that operations on associated tagged-objects cost increases linearly by increasing the size of the associated threads' labels. But for regular (non-tagged) objects the cost is constant.

We evaluate other dispersed compartment operations in Chapter 5, since they are quite integrated to SIRIUS userspace framework.

4.5 Summary

In this chapter, we have explained SIRIUS system stacks that make dispersed compartments practical on commodity hardware. We have described how the Linux kernel should be extended with our new security model and hardware-assisted abstractions (§4.1 and §4.2). We have also discussed SIRIUS's TEE stacks (§ 4.3) for both TrustZone- and SGX-based TEEs, including our implementation of SSM, and explained how SIRIUS achieves dispersed monitoring and enforcement. We have explained the security of SIRIUS components and how it protects potential threats on each security-sensitive module. Finally, we evaluated various SIRIUS system services via microbenchmarks (§4.4). In our evaluation, here we have only focused on the overhead of the systems side (without userspace). In Chapter 5, we focus on SIRIUS userspace framework for deploying dispersed compartments. We use SIRIUS API to reduce attack surfaces in real-

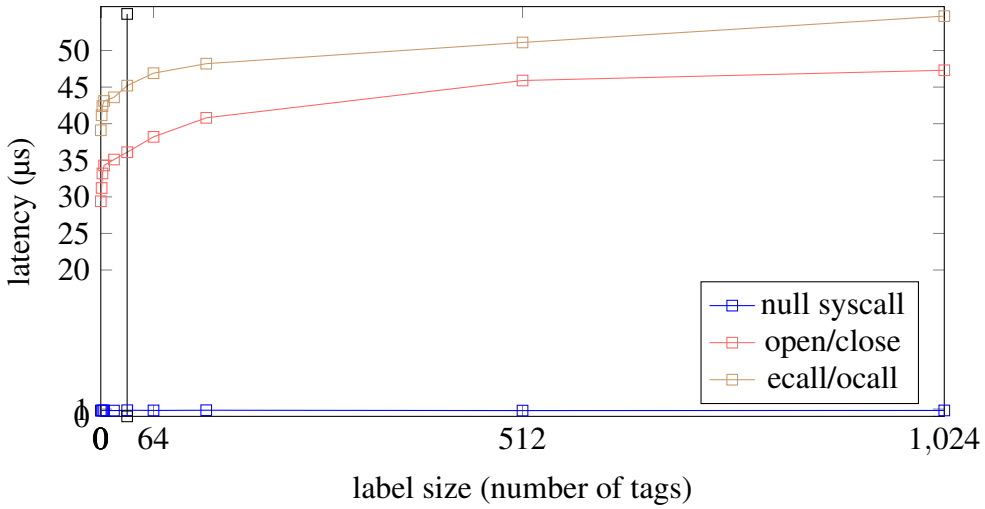


Fig. 4.9 Effect of label size on performance of SIRIUS-enabled systems and dispersed compartment operations.

world use cases, and then we will evaluate SIRIUS whole stacks from security, usability, and performance perspectives.

Chapter 5

Utilizing SIRIUS

This chapter describes SIRIUS userspace framework that enables a wide range of applications to utilize dispersed compartments for reducing their attack surface. It first overviews SIRIUS API and userspace stack (§ 5.1) for compartmentalization and attack investigation. Then we explain how dispersed compartments are used in various real-world use cases (§ 5.2); where we use and evaluate different SIRIUS functionalities on a diverse set of applications: LevelDB, OpenSSL, Apache Httpd, TrustZone Darknet, Wearable BCIs, and data distribution service.

5.1 SIRIUS userspace framework

SIRIUS introduces a new userspace library for defining and managing dispersed compartments within applications. This consists of compartment management calls and associated functions to add, remove, and share objects from them. A crucial feature of our programming interface is that it supports checking application security requirements for mutually untrusted processes on the host or within an enclave. We deliberately designed our userspace framework to be programming language agnostic for easy integration via adding extra annotations to existing applications and not requiring modifications to third-party libraries.

5.1.1 The API

SIRIUS full interface supports dispersed compartments principles over a rich set of system objects: address spaces, threads, files, sockets, and IPC/RPCs handles, all of which aim to match developer intuition with existing POSIX interfaces. Note that SIRIUS internally checks labels on dispersed compartments and their communication channels to ensure attackers cannot misuse or forge the API calls for unauthorized communication. For example, an attacker thread cannot launch MITM attacks to corrupt the API parameters between dispersed compartments. Each call invocation passes parameters via tagged thread, IPC, shared memory, or RPC messages. Hence, the attacker cannot directly change the values of API parameters in labeled thread calls or shared memory because it does not have the right capabilities.

API	Description
Compartment Control	
s_create (SL/IL)->cid	create a dispersed compartment
s_add (cid, obj,...)-> ret	add object to cid
s_remove (cid, obj,...)-> ret	remove obj from cid
s_cleanup (cid)-> ret	dispersed compartment cleanup
s_grant/revoke (tid,obj,priv)	grant/revoke privileges
s_acc_enable/disable (obj)	enable/disable object access
s_declassify/endorse (obj)	declassify/endorse an object
Object Management	
s_clone(&fn,...)->cid	create a new dispersed compartment thread
s_execv(vao,bin...)	execute binary in a VAO
s_wait(&stat)	wait for other dispersed compartment threads
s_vao_create(hw_mode)	create a VAO
s_malloc/free(_id,size)	allocation from a VAO
s_mprotect(_id,...)	change permissions of a VAO
s_mmap/munmap(_id,...)	change layout of a VAO
open/socket/pipe(SL/IL)	create labeled file/socket/pipe
Enclave calls	
s_create/delete_enclave (cid,...)	create/delete a SIRIUS enclave
s_ecall/s_ocall(cid, ...)	call from/to an enclave

Table 5.1 Simplified SIRIUS programming interface.

SIRIUS provides 22 new API calls to enable expressing compartments' requirements (see Table 5.1). The API is built on top of SIRIUS syscalls, so developers do not have to deal with the details of underlying DIFC concepts. SIRIUS API enables easy management of dispersed compartment and seamless labeling of each dispersed compartment's single or group of objects. Any thread can create a dispersed compartment via `s_create` and assign/remove single or multiple objects to it via `s_add`, `s_remove` or config files, which leads to automatically initializing and labeling the objects. Developers are not forced to assign or update policies only over a group of objects; SIRIUS supports per-object operations as well. For each object, an authorized dispersed compartment can do operations like labeling, declassification, endorsement, sharing, and revoking capabilities. A dispersed compartment can securely share its objects with other threads via `s_grant` or by enabling inheritance during `s_clone`, which leads to automatically sharing all of its attached objects. Similarly a dispersed compartment can revoke its capabilities via `s_revoke` at runtime. Listing 5.1 demonstrates a simple usage of the APIs.

SIRIUS implements userspace VAO-based memory management on top of its VAO syscalls (§4.2). The security of this API is very important for the isolation of dispersed compartments objects. As a simple example, attackers can misuse the API for changing the memory layout of other dispersed compartments VAOs or unauthorized memory accesses. The VAOs interface needs *(i)* to provide isolation within a single dispersed compartment; *(ii)* to be flexible and secure for sharing between dispersed compartment threads, and *(iii)* to restrict all unauthorized permission changes or memory mappings modification of allocated VAOs. Note that previous

in-address space isolation mechanisms such as ERIM [57] or libMPK [176] does not offer such security guarantees since their focus is more on performance and virtualization of the hardware protection keys.

```
void grant_test (void) {
    cid= s_create(SLABEL);//create a disperesd compartment
    vao_id = s_vao_create(DEFAULT, MEMDOM_READ|MEMDOM_WRITE);// create a VAO
    s_add(cid,vao,vao_id) // add the VAO to cid
    tid1 = s_clone(cid,&fn ,...);
    s_grant(tid1, vao_id,MEMDOM_READ | MEMDOM_WRITE); //give tid1 RW privileges
    s_wait(&status); // wait for threads
    s_cleanup(cid); //cleanup cid compartment
}
void revoke_test (void) {
    cid= s_create(SLABEL);//create a disperesd compartment
    vao_id = s_vao_create(DEFAULT, MEMDOM_READ|MEMDOM_WRITE);// create a VAO
    s_add(cid,vao,vao_id) // add the VAO to cid
    tid1 = s_clone(cid, vao_id,&fn ,...);//inherited VAO privileges
    s_revoke(tid1, vao_id, MEMDOM_WRITE); //revoke write privilege from tid1
    s_wait(&status); // wait for threads
    s_cleanup(cid); //cleanup cid compartment}
```

Listing 5.1 Example code of using s_grant and s_revoke API.

As Table 5.1 demonstrates, SIRIUS library supports a familiar API for memory management within a VAO, including `s_malloc` and `s_free` for memory management; which is implemented as a custom memory allocator similar to HeapLayer [278]. This allocates memory from an already mapped VAOs using `vao_mmap` syscall. For each VAO, the allocator keeps two lists of free blocks from the head and tails of the VAO region that is used when searching for free blocks of memory. The library also provides a userspace implementation for isolating and loading a binary into a VAO, `s_execv(const vao* v, const char *path,...)` API, for execution. To this, the loader parses the elf binary and uses `vao_execv` syscall for final security checks before loading and executing it.

```
static int fn(void* arg) {
  s_execv(mem_obj[1],zlib,deflateInit,Z_DEFAULT_COMPRESSION);
}
void zlib_example(void) {
  int cid= s_create(SLABEL); // create a dispersed compartment
  int fd= open(tmpname,O_CREAT);
  s_add (cid,persistent_file,fd);// add a file to cid
  mem_obj[1] = s_vao_create(DEFAULT, MEMDOM_READ|MEMDOM_WRITE);// create a VAO
  mem_obj[2] = s_vao_create(HW_MD,MEMDOM_READ); //create a MD-based VAO
  s_add(cid,vao,memory_obj[1],memory_obj[2]) // add both VAOs to cid
  // map VAOs to the dispersed compartment threads
  tid1 = s_clone(cid, mem_obj[1],&fn ,...);
  tid2 = s_clone(cid, mem_obj[2],...);
  s_wait(&status); // wait for tid1 and tid2
  s_cleanup(cid); //cleanup cid compartment
}
```

Listing 5.2 Using VAOs for isolating zlib within a dispersed compartment

The psudocode 5.2 shows how a dispersed compartment can isolate an untrusted third-party library like zlib using the VAO interface. The main thread creates two VAOs and maps them to `cid`, a dispersed compartment, (L8-10). The owner thread can explicitly give its VAOs to `tid1` and `tid2` threads with specific permissions (L12-13). Then `tid1` isolates `zlib` library inside `mem_obj[1]` (L1-3); so potential `zlib` corruption would not lead to information leakage outside its VAO and to other parts of the dispersed compartment.

The API also allows each dispersed compartment to protect its objects against untrusted parts of the code (e.g., unsafe third-party libraries) through the `s_access_enable/disable` calls. This calls temporarily disable any thread to access a SIRIUS-supported system objects; even if the thread already gained the privilege, after `s_access_enable` by the owner, it would lose its privileges until the object owner call `s_access_disable`. The code sample 5.3 shows how a dispersed compartment (`cid`) protects its `private_blk` when the there is an unsafe computation (L6-L8) that its potential vulnerabilities might affect the security of `private_blk`.

```
1  void temp_access_disable_test (void) {
2      cid= s_create(SLABEL);//create a disperesd compartment
3      vao_id = s_vao_create(DEFAULT, MEMDOM_READ|MEMDOM_WRITE);// create a VAO
4      s_add(cid,vao,vao_id) // add the VAO to cid
5      private_blk = (char*) vao_malloc(vao_id, len); // allocate memory from vao
6      //... before entering unsafe computations on private_blk ....//
7      s_acc_disable(vao,vao_id);
8      //... untrusted computations on private_blk ....//
9      s_acc_enable(vao,vao_id);
10     //... trusted computations ....//
11     //... cleanup....//
12     vao_free(private_blk); //free private_blk pages
13     s_cleanup(cid); //cleanup cid compartment
14     }
```

Listing 5.3 Example code of using s_access_enable/disable API.

We have ported SIRIUS userspace framework with small modifications to TrustZone TAs and SGX enclaves. Besides mapping the libraries' backend to different runtime services, a few APIs are also changed. For SIRIUS-TEE framework, the API allows controlling dispersed compartments resources inside the enclave/TA as well as sharing resources located outside their TEE-protected address space. For shared system objects, we implemented a few `ocall` interfaces to SIRIUS libraries outside the enclave. Note that SIRIUS provides a specifically designed `ecall/ocall` interface for dispersed compartments in addition to the conventional interface; that both its syntax and semantic are different.

The main difference between the two TEE APIs is the lack of SIRIUS support for in-enclave VAO isolation in our SGX implementation. Note that this can be easily resolved on a hardware platform with MPK or SGX2 support. Therefore, for TrustZhene, SIRIUS supports the VAO API for in-TA isolation of dispersed compartments and provides a specially VAO-assisted shared memory interface. However, for SGX, only VAO-based shared memory is supported. A dispersed compartment inside SGX enclave still can express security policies over other

in-enclave system objects such as threads, RPCs, files (since we ported the same bare-metal FS as TrustZone to SGX enclave), and all the Linux system objects that SIRIUS supports for controlling shared resources and communicating with external dispersed compartments.

For example, sample code 5.4 shows how an in-enclave dispersed compartment, `enclave_cid` uses in-enclave SIRIUS open call for creating a labeled file from our in-enclave baremetal file system (BMFS). The call will be redirected to ELM for DIFC checking (the label is hold in `BMFS` data structure). On the other hand, for creating a shared labeled file from the Linux FSs it uses `ocall_open` which redirected to the SIRIUS userspace on the host. Note that for dispersed compartments all RPC objects are internally tagged to ensure safe dataflows between the two SIRIUS libraries.

```
1  /* Allocate memory for ramdisk */
2  size_t buf_size = BMFS_MINIMUM_DISK_SIZE;
3  void *ramdisk_buf;
4  /* Setup the file system structure. */
5  struct BMFS p_fs;
6  /* Setup the ramdisk */
7  struct BMFSRamdisk p_disk;
8  struct BMFSFile p_file;
9  const char* file_name="/tmp/test.txt";
10
11 //open a file from in-enclave BMFS file system
12 int open(struct BMFS *fs,struct BMFSFile *file,const char *path)
13 {   if ((fs == BMFS_NULL) || (file == BMFS_NULL) || (path == BMFS_NULL))
14     return BMFS_EFAULT;
15   return open_file(fs, file, path);// path to sirius monitor for labeling and DIFC
         checking
16 }
17
18 void ecall_fs_test (void){
19     enclave_cid= s_create(SLABEL);//create a disperesd compartment
20     // initialise in-enclave FS
21     bmfs_init(&p_fs);
22     p_fs->label=SLABEL;
23     s_add(enclave_cid,bmfs,p_file) // add p_file to enclave_cid
24     int ret=open(p_fs,p_file,file_name);//open a labeld file from in-enclave BMFS
25     int fd= ocall_open(tmpname,O_CREAT|SLABEL);// open a labeled file from Linux FS
26     s_add (enclave_cid,persistent_file,fd);// add fd to enclave_cid
27 }
```

Listing 5.4 Example code of using SIRIUS-SGX in-enclave FS vs Linux FS.

5.1.2 Attack investigation

As we discussed earlier, one of the primary goals of this dissertation is to facilitate the secure adjustment to the contemporary hetero-compartment environment. The lack of proper systems for attack investigation within various compartments is one key obstacle to this goal. That is why we emphasized finding a compartmentalization solution with both extensibility and audibility features, so developers could use dispersed compartments for defense-in-depth mechanisms and for finding and investigating potential security threats when designing secure applications.

SIRIUS by design enables an extensible tracing capability for a proper set of objects with sufficient granularity, POSIX-friendly API, and minimal syscalls addition/changes. It already contains a tracing system and policy management that run inside the OS kernel or TEE/enclave system stacks. Hence SIRIUS current implementation can be used for attack investigation on five compartment types (and their combinations): process, in-process, SGX enclave, TrustZone TA, and in-TA. Additionally, unlike existing tracing and security analysis frameworks that add significant slowdown even for covering one isolation boundary, SIRIUS's system abstractions and hardware-based optimizations reduce the overhead by orders of magnitude.

```
1   cid= s_create(SLABEL);//create a dispersed compartment
2  //add fname to start tracing secrecy violations
3  //use ILABEL for integrity violations
4  s_add(cid,file,fname);
5  //or use open(fname,O_CREAT|SLABEL);
6  if(!access(fname,W_OK)) {
7  //Sirius reports the attacker's TOCTOU e.g., by using symlink to redirect fname
8  f = fopen(fname,"w+");
9  operate(f);
10 ...}
11 else{...}
```

Listing 5.5 Simple use of SIRIUS for tracing TOCTOU vulnerability on a compartment's file.

```
1  //-------------Enclave side--------------
2  void ecall_heap_leak(struct eData* data){
3      char temp[] = "kioasdinkadssasdkjhsdaklj";
4      char *buf;
5      int ret;
6      ecid= s_create(SLABEL);//launch a dispersed compartment
7      int mobj = s_vao_create(DEFAULT, MEMDOM_READ|MEMDOM_WRITE);// create a VAO
8      s_add(ecid,vao,mobj) // add the VAO to cid
9      //change malloc -> s_malloc
10     data->msg = (char*) s_malloc(mobj,strlen(temp));
11     memcpy(data->msg, temp, strlen(temp));
12     //replaced by Sirius version of memcpy
13     data->len = strlen(temp);
14     data->left = strlen(temp);
15     while(data->left > 0){
16     buf = data->msg + data->len - data->left;
17     ocall_write_out(&ret, buf, data->left);
18     if(ret <= 0) return;
19     data->left -= ret; } }
20 //-------------App side--------------
21 int ocall_write_out(char *buf, int left)
22 { printf("%c\n", buf[0]);
23   printf("%d\n", left-1); return 1;}
24 int main()
25 {   //init enclave .....//
26     acid= s_create(SLABEL);//launch a dispersed compartment
27     int a_mobj = s_vao_create(DEFAULT, MEMDOM_READ|MEMDOM_WRITE);// create a VAO
28     struct eData* data = (struct eData*) s_malloc(a_mobj,sizeof(struct eData));
29     ret = ecall_heap_leak(eid, data)
30     //cleanup enclave......//}
```

Listing 5.6 Pseudocode of using SIRIUS to detect enclave heap leakage

Source/Project	Attacker	Victim	Description	Compartments Objects	Policy Violation	LoC
COIN [42]	host app	enclave	heap info leak	$C1\{t_e, mo, ocall\}$ $C2\{t_a, ocall\}$	$t_e \not\longrightarrow ocall(mo) \not\longrightarrow t_a$	5
COIN [42]	host app	enclave	malicious calls	$C1\{t_e, mo_e, rpc\}$ $C2\{t_a, mo_a, rpc\}$	$t_a^{\otimes} \not\longrightarrow ecall(mo_a)$ $t_e \not\longrightarrow ocall(mo_e)$	9
COIN [42]	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 [279]	host app	enclave	stack info leak	$C1\{t_e^1, mo^1\}$ $C2\{t_e^2, mo^2\}$	$t_e^1 \not\longrightarrow mo^2(memcpy(mo^2, mo^1, size))$	21
SGX$_S$*QLite* [280]	host app	enclave	malicious calls	$C1\{t_a^1, f_e\}$ $C2\{t_a\}$	$t_a^{\otimes} \not\longrightarrow f_e$	8
SGX-Tor [151]	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 [151]	host app	enclave	concurrent calls	$C1\{t_e\}$ $C2\{t_a\}$	$t_a^{\otimes} \not\longrightarrow \{t_e\}$	4
Boomerang [43]	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 [33]	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 [33]	TA	TA	memory leak	$C1\{t_e\}$ $C2\{t_a, mo\}$	$t_e \not\longrightarrow mo$	11
OPTEE [33]	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 [38]	host app	enclave	enclave mem corr	$C1\{t_e, mo_e^1, rpc_e\}$ $C2\{t_a, rpc_e\}$	$t_e^{\otimes} \not\longrightarrow mo_e^1$ $t_a^{\otimes} \not\longrightarrow t_e$	12
async [41]	host app	enclave	concurrency	$C1\{t_e, mo_e, rpc_e\}$ $C2\{t_a, mo_a, rpc_e\}$	$t_a^{\otimes} \not\longrightarrow rpc_e$ $t_a^{\otimes} \not\longrightarrow mo_e \wedge mo_a$	18
HPE [44]	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 5.2 Utilizing SIRIUS for investigating vulnerabilities across privilege boundaries. We use subscripts *e* for enclave objects, *a* for app, *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 violation.

When SIRIUS policies are in place, any runtime attacks that violate the policies are detected and flagged. We demonstrate the effectiveness of our automatic runtime-checks in detecting previous attacks on a diverse range of attacker models and compartment boundaries. For example As listing 5.5 shows, simply adding `fname` to a dispersed compartment (L4), causes SIRIUS to report when an attacker (any unauthorised execution thread) exploiting a TOCTOU vulnerability by redirecting the file via `symlink` (between L6 and L8).

As another example, listing 5.6 shows how with small modification we can detect heap leakage inside an enclave. In this example when the enclave calls `ocall_write_out` and passes `data` to outside, we can detect heap leakage by bounding and allocating `data` from a protected VAO.

Therefore, we integrated SIRIUS into some of the existing attack codebases on TrustZone- and SGX-based applications to evaluate the practicality of SIRIUS in detecting them. As summarised in Table 5.2 we picked various vulnerabilities, ranging from privilege management issues (e.g., HPE [44] attacks) to memory vulnerabilities (e.g., COIN [42]) and unsafe interfaces (e.g., Asyncshock [41]). SIRIUS logs the executing and dataflow violations that immediately terminate the program execution, thus detecting all of them. The main limitation of our evaluation is that we already knew the potentially vulnerable parts of the code and focused SIRIUS tracing functionality on that parts. Therefore, we believe building better SIRIUS-assisted userspace tools focusing on attack investigation could be a beneficial future work.

5.2 compartmentalizing real-world applications

We evaluated the practicality and performance of SIRIUS by hardening real-world applications against various security issues.

5.2.1 OpenSSL & Appache Httpd

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. Hence, it significantly benefits from isolating its sensitive content in separate compartments to mitigate information leakage attacks (e.g., Heartbleed). In addition, many web services, including Apache httpd, use OpenSSL for various operations like TLS protocol. If compromised, the attacker can leak all private keys and certificates.

Due to limited enclave memory in ARM TEEs, it is currently impractical to run the entire Apache and OpenSSL stacks (more than $45MB$ RSS and $750K$ LoC) as a single enclave [33]. But using dispersed compartments, we could enable resource sharing from the host to selectively execute and protect essential parts of the stack inside an enclave. Out of the total $750K$ LoC ($533K$ OpenSSL, $210K$ Apache), we modified $\approx 2K$ LoC to refactor and port essential dependencies inside our TEE stack that is the same as any TEE system. SIRIUS only required $\approx 400LoC$ for creating and managing necessary compartments. Hence our alternative approach for partitioning, instead of porting the whole stack inside TEE, reduces the TrustZone memory footprint by $3\times$.

Our modified TEE-assisted httpd protects all private keys, session keys, and certificates and operations on them from any unauthorized thread by defining trust boundaries in both normal and enclave worlds. It forbids a malicious enclave thread from transferring secrets through uncontrolled channels to another enclave, or via untrusted memory, or via a file or networking sockets. A malicious httpd worker thread also cannot compromise the enclave by crafting RPC requests or modifying shared memory or even by gaining root privilege[1] unless also compromising the host kernel and security kernel to obtain the right labels. It also provides in-depth mutually distrustful isolation of stored data, metadata, and binaries on the host filesystem for both enclaves and httpd process.

We use SIRIUS to compartmentalize the webserver and the OpenSSL library. We first modified OpenSSL `libcrypto` to utilize dispersed compartments for protecting private keys from potential information leakage by storing the keys in protected memory pages. Depending on the threat model, each key could be protected inside a separate VAO, or all keys could be placed inside a single VAO. Using multiple VAOs provides stronger security while adding more overhead due to the cost of caching VAOs. To enable dispersed compartments inside OpenSSL, first, all the data structures that store private keys, such as `EVP_PKEY`, needed VAO-based protected heap memory allocation. This meant replacing `OpenSSL_malloc` with`s_malloc` and using `s_mmap` at the initialization phase for creating one or multiple (per session) VAOs to store private keys. After storing the keys, access to dispersed compartments is disabled by calling `s_acc_disable`. Only functions inside the dispersed compartment that require access to private keys (e.g., `EVP_EncryptUpdate` or `pkey_rsa_encrypt/decrypt`) can access the keys

[1] See CVE-2019-0211 or CVE-2019-0217, among others.

by calling `s_acc_enable`. Modifying OpenSSL required fairly small code changes and added only 281 LoC.

We then define two in-enclaves dispersed compartments ($C1$ and $C2$). $C1$ is a single-threaded OpenSSL (`libcrypto`) compartment to run cryptographic operations, such as `EVP_Encrypt`, `EVP_DecryptUpdate`, or `pkey_rsa_encrypt/decrypt`). $C1$ also stores private keys and certificates in our TEE-based secure storage. $C2$ is the attestation compartment that runs only in the initialization phase and should not access webserver secrets. We define $C3$ as a non-enclave compartment within httpd. We only authorize $C3$'s thread to communicate with the enclave's $C1$. We limit the web server's access to the protected memory and files between the web server's $C3$ and enclave's $C1$. When the webserver starts, it creates $C3$ dispersed compartment that can now launch the enclave dispersed compartments and *only* that $C3$'s thread has the right label to interact with them. It then grants the OpenSSL compartment direct RPC access to transfer secrets to the storage compartment and proceeds to revoke its own access to the secure storage.

Note that the webserver needs to enforce mutual distrust between the enclave-assisted dispersed compartments and the normal world. The in-enclave OpenSSL compartment only needs access to the information required to establish a session key and no other user data. Only a subset of Webserver threads that handles OpenSSL operations are authorized to communicate with the enclave dispersed compartments and access to the protected shared memory between the webserver and enclave compartments. We also label the private keys files for the TLS negotiations to not be accessible to webserver threads.

Figures 5.1 shows the overhead of ApacheBench applied against the original OpenSSL library on a baseline ARM kernel and the SIRIUS-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 SIRIUS-enabled httpd adds $\approx 10.8\%$ overhead on multithreading benchmark. When SSM is configured to monitor the kernel integrity, this overhead increases to $\approx 15\%$. This is a very reasonable overhead for an application that now gains fine-grained isolation with defense-in-depth layers to protect its secrets against threats from both the normal and secure world, which was not possible without SIRIUS.

5.2.2 LevelDB

Authenticated and secure data storage is an important computing paradigm for a wide range of applications. That is why various cryptography-based as well as enclave-based techniques are introduced to protect them [157, 281] for example, by hosting all sensitive data (tables, indexes, queries, and other intermediate states) in enclave memory. However, similar to other TEE-assisted solutions, they lack protection against insecure shared resources and non-trivial attack vectors.

To this end, we utilized SIRIUS for improving the security of Google's LevelDB, a fast key-value store and storage engine used by many applications as a backend database. LevelDB uses blocks as the base unit of data and is based on the implementation of Google Bigtable's

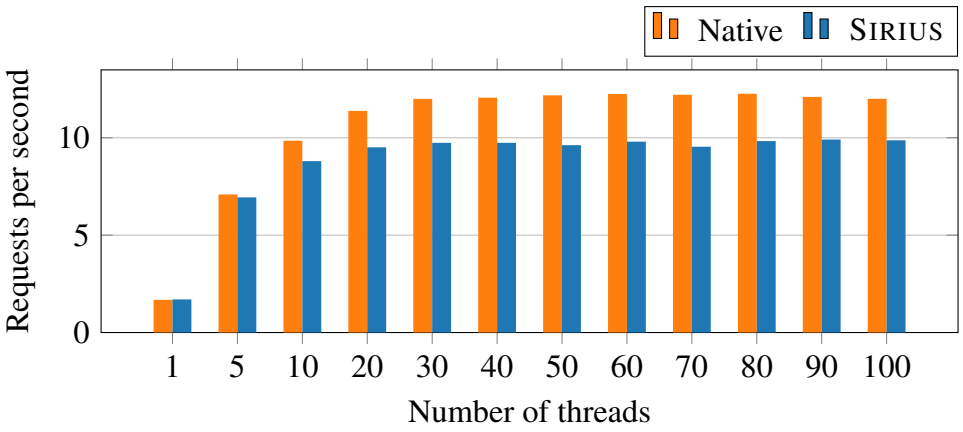


Fig. 5.1 Effect of multithreading on the performance of SIRIUS-assisted httpd, evaluated on Raspberry Pi 3.

tablets. We considered two scenarios; LevelDB without using any enclave (suitable for IoT use cases) and then hardening an enclave-assisted one.

For non-enclave LevelDB, we used dispersed compartments for in-process privilege separation. LevelDB supports multithreading for both concurrent writers to insert data into the database as well as concurrent read to improve its performance. However, there is no privilege separation between threads, so threads can not communicate securely with the database and protect their private content from other threads.

We first modified LevelDB to integrate dispersed compartments secure multithreading model when each thread has its own VAO-based private storage that other threads cannot access. We replaced the LevelDB threading backend (`env_posix`) that uses `pthreads` with dispersed compartment-aware threading, where each dispersed compartment thread creates an isolated VAO to protect its private storage and sensitive computations. We used the LevelDB `db_bench` tool (without modification) for measuring the performance overhead of dispersed compartments.

We generate a database with 400K records with 16-byte keys and 100-byte values with a raw size of 44.3MB. The number of reader threads is set to 1, 2, 4, 8, 16, and 32 threads for each successive run. The threads operate on randomly selected records in the database. The results summarized in 5.2 and 5.3 show how multithreading can improve the performance of LevelDB, and utilizing dispersed compartments adds a small overhead on write (5%) and read (1.98%) throughput. Native LevelDB multithreading on Raspberry Pi is not adjusted well. That is why in 5.2, increasing the number of threads does not lead to consistent performance improvement. For our evaluation, we decided to use unmodified LevelDB as our baseline.

We then focused on using SIRIUS only for securing the untrusted interface for SGX-enabled LevelDB, where the entire engine is located inside a single enclave and causes about 2.5x slowdown on LevelDB `db_bench`. Here we only targeted a single threaded use case. Since most computation is an inside enclave and only switches the execution outside enclave for file access (as shown in listing 5.7), we created a dispersed compartment inside the enclave for labeling

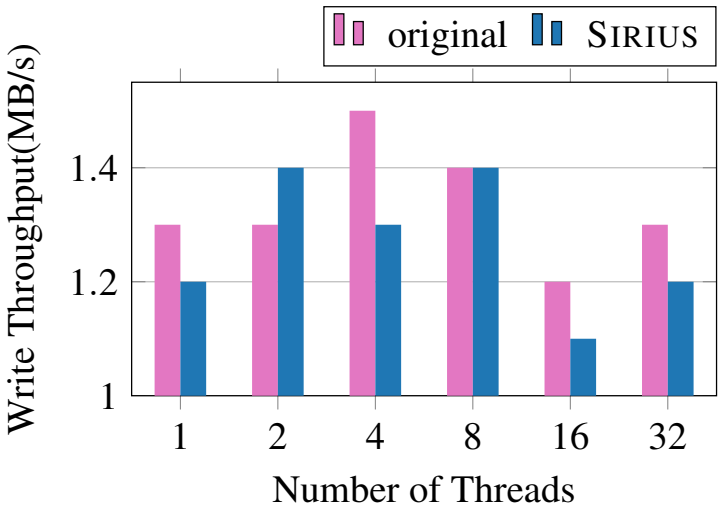


Fig. 5.2 LevelDB: performance overhead of SIRIUS-based multithreading compare to pthread-based in terms of write throughput (5%) on Raspberry Pi.

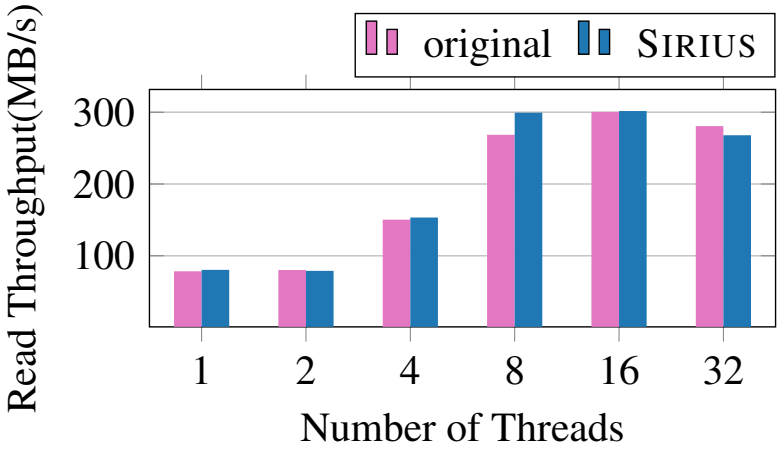


Fig. 5.3 LevelDB: performance overhead of SIRIUS-based multithreading compare to pthread-based in terms of read throughput (1.98%) on Raspberry Pi.

the associated files and directories and RPC objects to avoid unauthorized enclave calls from the host process. Also, its Arena and LRU cache backend allocate shared memory read buffers within the untrusted host process (via `ocall_allocate*` in listing 5.7).

We utilized SIRIUS's VAO based memory management to ensure all shared memory buffers are owned and controlled by the in-enclave dispersed compartment. SGX SDK uses `edl` (Enclave-Definition Language) files and description format for defining enclave interface[2]. SIRIUS does not require any modification on `edl` tool or enclave interface format. These security features required less than 100 LoC modifications on interface layers and add only 8.4% additional overhead.

[2]https://github.com/intel/linux-sgx

```
/* Enclave.edl - Top EDL file. */
enclave {
from "sgx_tprotected_fs.edl" import *;
include "user_types.h" /* buffer_t */
untrusted {
 void ocall_print_string([in, string] const char *str);
 void ocall_file_exists([in, string] const char *str, [out] int *res);
 void ocall_create_dir([in,string] const char *str);
 void ocall_delete_file([in,string] const char *str);
 void ocall_rename_file([in,string] const char *str1,[in,string] const char *str2);
 void ocall_fopen([in,string]const char *fname, [in,string] const char *mode, [out] long *
     res);
 void ocall_getchildren([in,string]const char *fname, [out] long* res, [out] int* size);
 void ocall_checkchildren(long res, int i, [out,size=100] char* filename);
 void ocall_fwrite([in,size=ncount]const char *data, int psize, int ncount, long f, [out]
     int *res);
 void ocall_fread([out,size=ncount] char *data, int psize, int ncount, long f, [out] int *
     res);
 void ocall_fread_outside(long data, int psize, int ncount, long f, [out] int *res);
 void ocall_fflush(long f, [out] int *res);
 void ocall_fclose(long f, [out] int *res);
 void ocall_feof(long f, [out] int *res);
 void ocall_fseek(long f, int offset,[out] int *res);
 void ocall_pthread_create(int type, long ls);
 void ocall_get_filesize([in,string] const char* str, [out] uint64_t *res);
 void ocall_allocate_specific([out] long  *res, int size);
 void ocall_delete(long mem);
 void ocall_allocate([out] long  *res, int flag);
 void ocall_allocate3([out] long  *res,[out] long  *res1,[out] long  *res2); }; };
```

Listing 5.7 SGX LevelDB ocalls in the .edl (enclave definition language) file.

5.2.3 TEE-based ML framework

SIRIUS can mitigate various fine-grain attacks (e.g., scenarios I and II in Section 2.3.3) on ML frameworks for existing TEEs. A developer can use dispersed compartments to significantly reduce the attack surface we outlined for TEE-assisted ML frameworks and restrict the propagation of vulnerabilities.

For example, as Figure 5.4 shows, the interface attacks in Scenario I [41, 42], where the enclave is the victim, can be avoided by the developer forming A_{SPEC} with a policy P_{C1} over application's dispersed compartment $C1$ and E_{SPEC} with a policy P_{C2} over in-enclave dispersed compartment $C2$. By adding the application thread t_a to $C1$ and enclave thread t_e to $C2$, and an RPC object rpc_{ae} to both compartments, only these two threads are allowed to communicate securely and no other thread ($t2 \notin \{C1 \cup C2\}$) can interfere or access their objects. This stops the attacks because the attacker thread $t2$ is prohibited from interacting with the enclave, similar to Thread 2 restricted dataflow in Figure 5.4 (1.a).

To avoid over-privileged enclaves, the programmer can specify a separate in-enclave dispersed compartment $C3$ dedicated to attestation and private keys. $C3$ contains a dedicated enclave thread t_{att}, an in-enclave memory object m_{keys}, and file storage f_{keys}, and another RPC

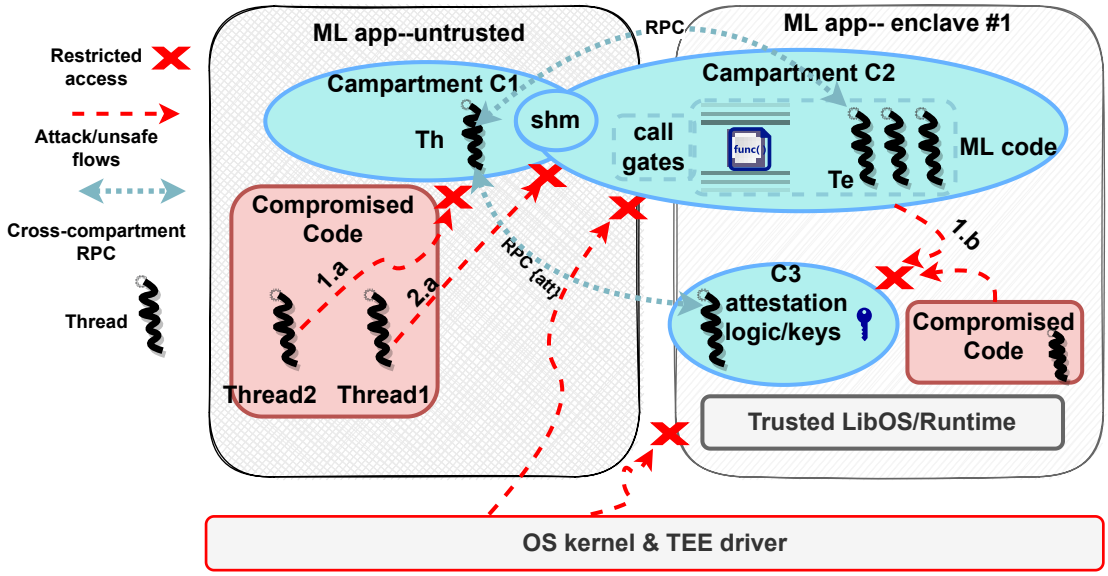


Fig. 5.4 Simplified SIRIUS-assisted compartmentalization for hardening ML.

object rpc_{att}. Now even if the in-enclave ML logic has vulnerabilities, they cannot manipulate the execution of the attestation logic or access/corrupt the private enclave keys (as shown in 1.b restricted dataflow in Figure 5.4).

The developer can also avoid the Scenario II attacks [43], when the attacker can leverage a compromised enclave to launch attacks on the host kernel, (Figure 5.4, 2.a), by restricting unauthorised access to enclave, TEE driver calls, and shared resources. For example to avoid Boomerang the developer adds a VAO-based shared memory object shm_{ae} to $C1$ and $C2$ such that only t_a and t_e can access it. By specifying the enclave's t_e as the owner of shm_{ae}, it allows the in-enclave dispersed compartment's thread to modify the memory but restricts t_a to read-only access. This policy allows the ML logic to securely use shared memory to and from the host and enclave while ensuring that no unauthorized thread can access or modify it. Thus the attacker cannot send specially crafted messages or bad pointers to the kernel through shm_{ae}.

To evaluate the performance of SIRIUS for enforcing such policies, we modified DarkneTZ [228], which already partitions ML layers to run inside TrustZone to avoid membership inference attacks (MIA) [229]. We compartmentalize DarkneTZ with only minor modifications (318 LoC). Since Darknet is heavily multi-threaded, we modify its classifier (`classifier.c`) to launch dispersed compartment $C1$ thread for communicating with in-enclave ML dispersed 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 avoiding concurrency and interface attacks [38, 41, 230].

We assign another enclave dispersed compartment $C3$ for protecting all sensitive resources located in the host OS such as (`/cfg`), (`/models`), and (`/data`). We evaluated the performance using AlexNet, a benchmark used by prior works [228]. We train a model with four layers

Operation	Native	SIRIUS
training	75.234 s	78.486 s
pre-trained	6.8753 s	7.3987 s
inference	33.23 µs	36.32 µs

Table 5.3 Latency overhead of Darknet ML framework.

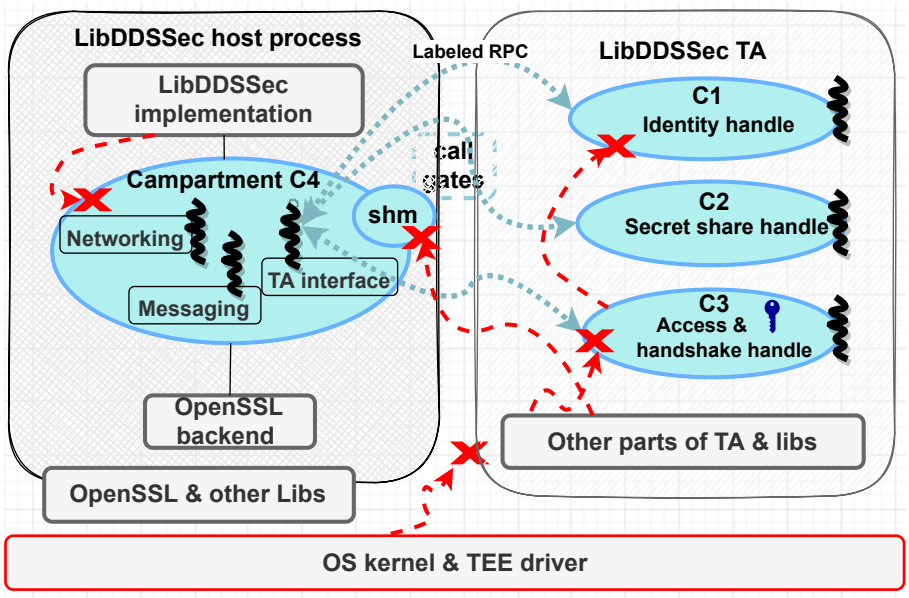


Fig. 5.5 Attack surface reduction for LibDDSSec-based applications using SIRIUS's dispersed compartments.

outside and one layer inside an enclave using standard CIFAR-100 dataset [282]. Compared to native, SIRIUS 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 (Table 5.3).

SIRIUS outperforms OPTEE-based DarkneTZ by $\approx 5.6\%$ and on average adds $\approx 9.2\%$ overhead compare to baseline Darknet, despite the improved layers of isolation and amount of data flowing across the normal and secure worlds.

5.2.4 Secure DDS

The Data Distribution Service (DDS) for IoT and real-time systems (e.g., autonomous vehicles or medical devices) is an open networking middleware protocol and API standard. It implements a publish-subscribe communication pattern for real-time and embedded systems that allows participants to send and receive data, events and commands among the nodes transparently. Nodes that produce information (publishers) create *topics* (e.g., temperature, location, pressure) and publish *samples*. DDS delivers the samples to subscribers that declare an interest in that topic. DDS is layered on top of the UDP to make use of multicasting. Multicasting allows a node to subscribe to a router. The sending node sends only one message to a router with the

message's information to be sent to all the subscribers. The router distributes this message to all subscribers of this topic. DDS uses multicasting to discover the different participants for a topic. After that, it uses unicast and direct messages directly to any participants.

TEE-assisted DDSs, such as ARM's LibDDSSec, are proposed for security-sensitive IoT applications. LibDDSSec follows the DDS security model that defines the users of the system, the objects that are being secured, and the operations that are to be restricted. It relies on TEE services for the confidentiality of the data samples, the integrity of the data samples and the messages that contain them, authentication of DDS writers and readers, authorization of DDS writers and readers, message/data origin authentication, and non-repudiation of data. All these services are implemented in a single TA with about 122 interface calls between the TA and the host process.

We used SIRIUS to compartmentalized LibDDSsec to harden handlers for authentication, protecting data samples, secret sharing, and certificate operations, which all require secure interactions with the normal world. We created three dispersed compartments (C_1, C_2 and C_3 in Figure 5.5) inside its TA to separate identity handle (IH), shared secrets handle (SSH), handshake handle (HH). This way, a vulnerability in one of these stages won't be propagated through the entire TA. These required minor modifications, less than 200 LoC, to `dsec_hh.c`, `dsec_ih.c`, and `dsec_ssh.c` .

We also created a dispersed compartment (C_4) on the host for ensuring safe shared memory and RPC calls with C_1, C_2 and C_3. The changes ensure that all shared data from other nodes are protected while being processed via protected VAOs and while at rest (via labeled storage). Hence, only dispersed compartments with the right capabilities can exchange safety-sensitive messages, control messages, critical system data (e.g., emergency start/stop), and sensor data (e.g., temperature, laser, camera). Hence, we also modified `dsec_ca.c` to replace the RPC and shared memory with SIRIUS-protected operations. C_4 is also responsible for safeguarding the message interpretation routines and networking handles so an attacker could not send malformed RTPS packets to flood a target host with unwanted traffic or cause information exposure. Hence, C_4 allocates a VAO dedicated to messages and ensures only its authorised thread can transfer messages through labeled sockets. C_4 restricts any node from leaking private content through uncontrolled channels by labeling associated sockets and files.

We evaluate the overhead of SIRIUS by running LibDDSSec benchmarks on unmodified OPTEE compared to SIRIUS TEE stack that shows SIRIUS slightly improves the overhead by 0.05%. This gain is mainly due to alternative communication channels in SIRIUS compared to OPTEE.

5.2.5 Wearable BCIs

Wearable BCIs are an example of security-sensitive IoT/mobile use cases with resource-constrained and increasing attack surfaces. As we discussed earlier(§2.3.4), these use cases now suffer from both system-side and ML adversarial attack vectors (and their combinations). We

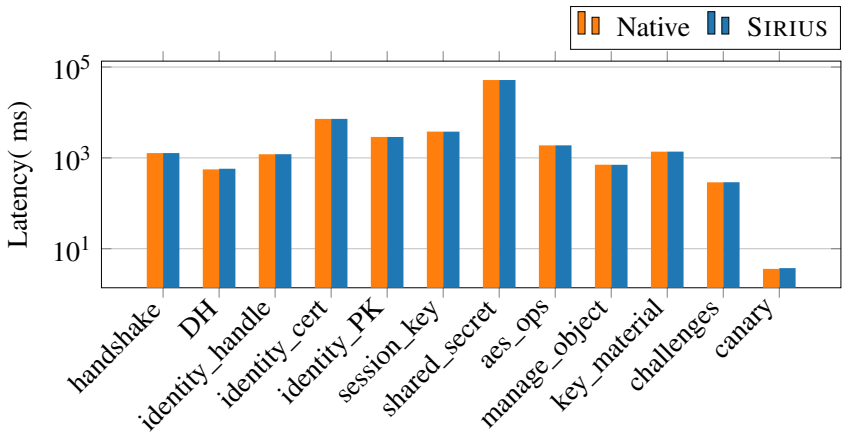


Fig. 5.6 Latency overhead of LibDDSSec benchmark.

observe that SIRIUS could be beneficial in detecting and mitigating many of such attack vectors by carefully modeling and mapping them into the underlying system objects (e.g., files, memory regions, device peripherals) and then monitoring the information flows within objects.

There are different ways to use SIRIUS. On one hand, *BCI framework/SDK* developers can integrate it into their codebase via small modifications. On the other hand, *BCI application* developers that need to rely on insecure and closed-source BCI third-party libraries, can utilize it to track information flows within all untrusted modules alongside their own applications' sensitive dataflows. Here we use SIRIUS with no TEE involved only as a *domain-specific* information flow tracking system, for reducing both system-based and ML adversarial attack surface in a lightweight manner. We consider the following attack vectors:

(AV1) Sniffing, Spoofing & Man-in-the-middle. Due to unencrypted and insecure BLE (Bluetooth Low Energy) connections with the BCI host devices, numerous sniffing attacks are possible. For example, we implemented several sniffers in all the considered BCI devices to capture and record all transmitted packets, MAC addresses, and connection parameters. Gaining this information facilitates the launch of more complex attacks such as MITM. Via the MITM attack, we could easily intercept and record all data sent between the headset and device and, additionally, compromise the device integrity by altering the data. One of our attack prototypes hijacks and alters communications between the headset and phone using an unauthorized Raspberry Pi acting as the BLE peripheral device.

(AV2) Inadequate isolation & access control. All BCI applications and devices we investigated lack adequate access control mechanisms. By launching an adversary process, we could easily access and modify the application stored data and deep learning model on the file system. Then we could leak secrets (e.g., predicted mental state of the user) to the outside, alternating control peripherals/signals (e.g., to change the direction of a BCI-controlled drone), or maliciously target the deep learning model by sending crafted queries [229, 230]. Similarly,

we could compromise the applications through insecure interactions with the host, for example, via inter-process communication or shared memory.

(AV3) Privilege escalation. Wearable BCI platforms suffer from poor privilege management. They do not use any isolation mechanism to protect their resources from other applications, nor separate privileges within their different components (e.g., by isolating untrusted libraries). Moreover, some require higher privileges such as "root-access" for their operations (e.g., OpenBCI's user interface), which can be misused by attackers to take control of the host. We implemented several privilege escalation attacks to leak information and successfully launched confused deputy attacks to compromise other host applications and even the host OS through an insecure BCI application. Additionally, to show the importance of in-application compartmentalization, we analyzed third-party libraries such as BrainFlow[3]. As expected, by exploiting the vulnerabilities within untrusted dependencies, we could gain unauthorised privileges.

(AV4, AV5) Adversarial ML Current wearable BCI devices are at early stages of adopting deep learning algorithms. They are, however, rapidly following the trend of enabling AI on mobile/wearable environments (e.g., smart watches). Given a clean signal $\mathbf{X}$, an adversarial attack introduces small perturbations such that the prediction for $\mathbf{X}$ and $\mathbf{X}^{adv}$ differs. Therefore to test usability of SIRIUS, we prototyped few attacks on real-world BCI devices. First, Fast Gradient Sign Method (FGSM) [283], a single step attack which aims to find the adversarial perturbations by moving in the opposite direction to the gradient of the loss function $L(\mathbf{X}, y)$ w.r.t. the signal. Second, Projected Gradient Descent (PGD) [284], which is a stronger variant of FGSM that applies it iteratively.

(AV6) Peripheral injection black-box attack. Given the lack of security features in our studied BCI platforms, we launched an AV1 to hijack the headset-phone communication and created a black-box attack by introducing a signal delay as adversarial perturbation (without accessing the deep learning model). Detecting this category of attacks is not straightforward since delays in the signal can also happen naturally, therefore, we need a tracing system that can identify malicious behaviour. For example, code sample 5.8 shows how we can monitor `tty` ports on OpenBCI.

```
1  //*** BCI App side ***//
2  s_create(SLABEL);//enable tracing in this compartment
3  //use SLABEL for secrecy violations
4  s_add("/dev/tty.OpenBCI-DM00DRM0",file,SLABEL);
5  //..... rest of the code
6  //*** attacker side ***//
7  if __name__ == '__main__':
8      port = '/dev/tty.OpenBCI-DN0096XA'
9  //....initialisation
10 //Sirius detects an access
11     board = bci.OpenBCIBoard(port=port, scaled_output=False, log=True)
12 //..... rest of the code
```

Listing 5.8 SIRIUS-assisted tracing of OpenBCI `tty` port accesses.

[3]https://brainflow.org/

Table 5.4 A subset of simplified policies used for IFC monitoring on our attack PoCs. We use subscripts $\otimes$ for attacker system objects, b for BCI app, and a for any host app. $\longrightarrow$ denotes a permitted information flow, while $\not\longrightarrow$ represents an information flow policy violation. Here $P\{\}$ represents a set of processes/threads, $F\{\}$ a set of files, $S\{\}$ a set of sockets (and their binding ports), $M\{\}$ a set of VAOs, $IPC\{\}$ a set of any object used for inter-process communication (e.g., pipes or shared memory). For clarity, we omit the set annotation $\{\}$.

Attacks	**Tainted objects** (O_b)	**Secrecy policy** ($\forall obj \in O_b$)	**Integrity policy** ($\forall obj \in O_b$)
AV1	$F_b\{f_{in}, f_{out}, f_{dev}\} \cup S_b\{port_{in}, port_{out}\} \cup IPC_b$	$obj \not\longrightarrow P_\otimes \wedge P_a$	$P_\otimes \wedge P_a \not\longrightarrow obj$
AV2	$P_b \cup F_b \cup S_b \cup IPC_b \cup M_b$	$obj \not\longrightarrow P_\otimes \wedge P_a$	$P_\otimes \wedge P_a \not\longrightarrow obj$
AV3	$P_b \cup F_b \cup S_b \cup IPC_b \cup M_b$	if $\exists obj \in O_b : obj \longrightarrow P_a$ then $P_a \not\longrightarrow P_\otimes$	if $P_a \longrightarrow obj, \exists obj \in O_b$ then $P_\otimes \not\longrightarrow P_a$
AV4	$P_b \cup F_b\{f_{dev}, f_{in}\} \cup S_b\{port_{in}\}$	$obj \not\longrightarrow P_\otimes$	$P_\otimes \not\longrightarrow obj$
AV5	$P_b \cup F_b\{f_{model}, f_{dev}, f_{in}\} \cup S_b\{port_{in}\}$	$obj \not\longrightarrow P_\otimes$	$P_\otimes \not\longrightarrow obj$
AV6	$P_b \cup F_b\{f_{dev}\} \cup S_b\{port_{in}\}$	$obj \not\longrightarrow P_\otimes$	$P_\otimes \not\longrightarrow obj$

We successfully detect and avoid these six AVs by employing SIRIUS with reasonable effort and overhead. Further, we connect different BCI devices to Raspberry Pi to evaluate all of them on the same baseline, running on the same SIRIUS-enabled Linux kernel. Tables 5.4 and 5.5 summarise our specified information flow policies, modifications, and results on each BCI platform.

OpenBCI-based platforms. Instead of modifying separate OpenBCI applications, we opted for modifying the underlying open-source library, Brainflow, which is used by most applications. After creating a dispersed compartment in the initialization phase (via `s_create`), we modify several parts of BrainFlow for tainting and tracking various objects. This includes modifying the modules that handle connections to the BCI headset to enable labelling the serial/BLE ports, sockets, and worker threads. Additionally, we labeled any file that contains the headset information, metadata, and logs. This required small modifications for example, in `board_shim`, `brainflow_get_data`, and `brainflow_filter`. To avoid data leaks at runtime (e.g., via memory corruption attacks), we assigned a VAO for the headset information and another one for mapping the log file. Similarly, we labeled the ML model and mapped it to a separate tainted memory region. Table 5.5 summarises the overhead introduced by our changes.

Muse & NeuroSky-based platforms. Many Muse applications depend on BrainFlow too, therefore the same modifications as OpenBCI are required. Additionally, some Muse applications use Muse-lsl library[4] for streaming, visualizing, and recording EEG data. We integrated dispersed compartments to different parts of muse-lsl such as `muse` backend, `record`, and `stream` to taint all sensitive threads, files, and communication ports with small modifications as shown in Table 5.5. NeuroSky does not officially provide Linux-based libraries, so most of its Linux-based applications use python-mindwave[5] library. Therefore, we integrated SIRIUS to this library by applying modifications similar to the muse-lsl ones.

[4] https://github.com/alexandrebarachant/muse-lsl
[5] https://github.com/akloster/python-mindwave

Table 5.5 Overhead of integrating SIRIUS to BCI platforms.

BCI platform	Modified/added LoC	Slowdown
OpenBCI	0.02%	14.2%
Muse	0.01%	12.1%
NeuroSky	0.01%	12.7%

Utilizing LibSirius for ML adversarial attacks. For all the BCI uses cases we rely on the deep learning framework described in § 2.3.3. Through dispersed compartments, we could detect the aforementioned adversarial attacks by carefully modeling and mapping the attack vectors into the underlying system objects (e.g., threads, files, memory regions, device peripherals) and then monitoring the objects. For instance, we tracked information flows during white-box attacks (AV4 and AV5) by tainting the BCI process and model file f_{model} in the file system as well as the memory blocks (e.g., $M_b\{m_i,\ldots,m_j\}$ in Table 5.2) assigned to the model (e.g., upon the `mmap`). This enables the system to detect any information flow policy violation ($obj \not\longrightarrow P_{\otimes}/P_{\otimes} \not\longrightarrow obj$ in Table 5.2) which allows the attack to get access to the deep learning model in the first place. Furthermore, we know that the attacker needs to re-inject the perturbed signal therefore we could track this injection by monitoring the BCI device ports (e.g., $S_b\{port_{in}\}$ in Table 5.2). These are just a set of examples on how we used SIRIUS for tracing ML adversarial attacks, which can be easily extended to future unseen attacks.

5.3 Summary

This chapter has explained how SIRIUS can be utilized for reducing the attack surface in a diverse range of real-world applications. It first has described the SIRIUS's userspace framework and API (§ 5.1) for compartmentalization and attack investigation. It then has explained how dispersed compartments are used in various real-world use cases (§ 5.2) such as LevelDB, OpenSSL, Apache httpd, TrustZone Darknet, DDS, and Wearable BCIs. We have shown SIRIUS achieves its goal in significantly reducing non-trivial attack vectors while adding small overhead and application changes.

Chapter 6

Conclusion

This dissertation presented a fundamental approach for securing the future of hetero-compartment computing, where applications can securely and efficiently benefit from various forms of compartments such as processes, enclaves/TEEs, sandboxes, and even intra-address space compartments simultaneously. We introduced a novel compartmentalization mechanism that is extensible to manage the complex requirements of hetero-compartment environments. Furthermore, this thesis discussed the importance of properly utilizing hardware security features to reduce the large attack surface in modern applications while needing slight application modification, performance overhead, and TCB. Thus, it proposed several hardware-assisted abstractions for achieving these features and enabling effective isolation, extensibility, and auditability.

6.1 Thesis summary

We analyzed modern applications' security issues, architectures, and compartmentalization techniques to shape the dissertation foundations. Our findings showed that existing compartmentalization solutions have fundamental limitations. They are particularly unsuitable for hetero-compartment environments, where they fail to reduce non-trivial attack surfaces, especially those caused by cross-compartment interactions and data exchanges. Hence naive compartmentalization in modern complex computing environments could introduce new attack vectors, as explained in Chapter 1 and Chapter 2. We proposed a principled approach based on mutual distrust and extensibility to target single or hetero-compartment environments as an alternative to existing solutions. Our privilege separation and isolation mechanism enable application security policies to be defined and enforced *within* and *across* different isolation boundaries (independent to address space), while remaining flexible in the face of diverse threats and changing hardware requirements.

To this end, we have introduced the principles of dispersed compartments, monitoring, and enforcement as a unified compartmentalization model. Using these security primitives enables developers to bridge semantic gaps across heterogeneous compartments. This thesis has then presented the design and implementation of SIRIUS, a full OS-based compartmentalization

framework for enabling dispersed compartments on commodity hardware, focusing on ARM and x86-64 platforms. SIRIUS resolves three fundamental limitations of the state-of-the-art compartmentalization techniques: *(i)* not supporting sufficient fine-grained security policies for either isolation or resource sharing, *(ii)* the inability to extend and audit compartments beyond a single privilege layer (e.g., userspace) or address space boundaries (e.g., process or enclaves), and *(iii)* impractical performance overhead, hardware modification, or programming language dependency.

To reach these goals, in Chapter 3, we formed SIRIUS security model based on the principles of decentralized IFC, which enables more flexible security policies based on mutual distrust, monitoring compartments, and easier extensibility. We also designed the system based on two threat models, with and without the host OS as the TCB component, to allow developers to adapt to different use cases (§3.3). However, during the early design of SIRIUS, we noticed that achieving all these goals without the help of hardware security features and the right system abstractions is not feasible. Therefore, we have proposed three hardware-assisted abstractions as building blocks of SIRIUS to enable: *(i)* inter- and intra-address space privilege separation, *(ii)* efficient interactions between heterogeneous compartments, and *(iii)* a high privilege security monitor to ensure dispersed enforcement in a secure and high-performant way (§3.4).

Chapter 4 describes SIRIUS systems stack and the integration of its security model and hardware-based abstractions together into a hetero-compartment environment. We described how SIRIUS modifies the host OS kernel and TEE kernel/system to label and enforce security policies over a wide range of system objects. We showed that integrating SIRIUS security primitives into the Linux kernel and TEE stacks removed the significant performance overhead and language dependency limitations of previous decentralized IFC-based systems. Moreover, our relatively small Linux kernel modifications showed that SIRIUS is practical for many IoT/mobile use cases with resource constraints. Besides the newly added kernel monitors, e.g., KLM, most of SIRIUS kernel changes are applied to virtual memory management, file systems, and network abstractions (§4.2). We also showed that SIRIUS's unified security model, which means one single set of rules for all object types, enables similar system changes on TEE systems (§4.3) or other compartmentalization systems. SIRIUS incorporation with existing TrustZone- and SGX-based systems, as the two leading TEE models, showed that dispersed compartments could be efficiently enabled in various hetero-compartment environments (§4.4).

Finally, in Chapter 5, we described SIRIUS userspace framework and API for utilizing dispersed compartments within applications. It allows developers to define and reason about security requirements between/within mutually untrusted heterogeneous compartments. Our framework currently supports six compart types, i.e., processes, SGX enclaves, TAs, in-process, in-enclave, and in-TA. SIRIUS userspace framework is designed to be programming-language agnostic for easy integration via adding extra annotations to existing applications and not requiring modifications to third-party libraries (§5.1). This is shown by compartmentalizing

and auditing real-world use cases on the edge-cloud environment. We targeted a diverse range of use cases for proper evaluation and optimization, including web services, databases, cryptography libraries, data distribution services, wearables, and machine learning frameworks. We have shown that SIRIUS userspace API activates fine-grained and flexible policies for these applications with relatively small code changes and significant performance benefits (§5.2).

6.2 Compartmentalization extensibility & auditability

Extensibility and auditability are essential to reduce modern applications' attack surface significantly. Being able to investigate potential attack vectors plays a vital role in designing defense-in-depth mechanisms, which is notably lacking in hetero-compartment environments. In this dissertation, we enabled both features simultaneously. This way, our unified compartmentalization framework could deliver simplicity and provide multiple security benefits for developers. However, as we have explained SIRIUS details, providing this simplicity is complex. We have decided to port most of this complexity to the system stacks, like the OS kernel, to achieve stronger security and gain better performance.

An alternative design would be separating these two functionalities and implementing a userspace-only attack investigation framework. Since there is no existing work that properly supports TEEs, or hetero-compartment environments in general, estimating the cost of this approach is hard. Especially in integration with hypervisor-based TEEs such as Intel Trust Domain Extension (TDX) [27] and AMD SEV [145]. These systems will introduce a similar semantic gap as other TEEs in the hypervisor layer that are not explored yet, and we hope our work help to investigate the gap more and design a more secure systems stack. Our experience with building SIRIUS has shown a sweet spot for adopting decentralized IFC models to enable a unified framework for supporting mutually-distrustful compartments running inside different hardware privilege levels. However, this approach was not practical without *(i)* our hardware-based dispersed monitoring and enforcement and *(ii)* our kernel extensions to reduce the overhead of labeling within both kernels' abstractions, and in particular, within address space objects.

6.3 Fine granularity & mutual distrust

To properly achieve dispersed compartments security principles, supporting mutually-distrustful security policies inside a single address space is essential. In this thesis, we tried several hardware-based memory protection features and showed their varying impact on performance overhead and hardware compatibility. Fortunately, the current trend in adding tagged memory and hardware capabilities into ARM architecture may improve SIRIUS performance and support for more IoT devices. Here the OS support and system abstractions for utilizing these memory safety features are major deciding factors. For instance, the current lack of support and potential design for CHERI capabilities in Linux-based systems is a crucial limitation. That is why we

think the recent efforts on Linux-compatible RISC-V intra-address space isolations and memory protection keys [36, 37, 181] are likewise promising alternatives.

This dissertation has not explored programming language-based solutions for enabling dispersed compartments, which may be even higher-performant in languages like Rust. However, we believe forcing developers to a specific programming language or paradigm can not scale and is a significant constraint for effective compartmentalization.

References

[1] John E Gaffney Jr. Metrics in software quality assurance. In *Proceedings of the ACM'81 conference*, pages 126–130, 1981.

[2] ObjectscriptQuality. Cyclomatic complexity. https://objectscriptquality.com/docs/metrics/cyclomatic-complexity, 2015.

[3] CWE over time. https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cwe-over-time#vuln-type-change-by-year-title, 2022.

[4] ARM. Architecture reference manual; ARMv7-a and ARMv7-r edition. https://static.docs.arm.com/ddi0406/c/DDI0406C_C_arm_architecture_reference_manual.pdf, 2012. Access Date : 2020-5-26.

[5] ARM. Architecture reference manual: ARMv8 for ARMv8-a architecture profile. *ARM Limited, Dec*, 2017.

[6] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX explained: An empirical study of Intel MPX and software-based bounds checking approaches. *arXiv preprint arXiv:1702.00719*, 2017.

[7] 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. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy (SP)*, pages 20–37. IEEE, 2015.

[8] Aaron Grattafiori. Understanding and hardening Linux containers. *Whitepaper, NCC Group*, 2016.

[9] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, pages 857–874, 2016.

[10] Towards an open-source, formally-verified secure enclave. https://keystone-enclave.org/files/dawn-nsf-2018-v5.pdf. Access Date : 2018-09-20.

[11] Jack StewardJack. The ultimate list of internet of things statistics for 2022. https://findstack.com/internet-of-things-statistics/, 2021.

[12] Kevin Collier. Baby died because of ransomware attack on hospital. https://www.nbcnews.com/news/baby-died-due-ransomware-attack-hospital-suit-claims-rcna2465, 2021.

[13] List of data breaches. https://en.wikipedia.org/wiki/List_of_data_breaches, 2018.

[14] Lewis Morgan. List of data breaches and cyber attacks in October 2017 – 55 million records leaked. https://www.itgovernance.co.uk/blog/list-of-data-breaches-and-cyber-attacks-in-october-2017-55-million-records-leaked/, 2017.

[15] Cyber security breaches survey 2018. https://www.gov.uk/government/statistics/cyber-security-breaches-survey-2018, 2018.

[16] Junjue Wang, Brandon Amos, Anupam Das, Padmanabhan Pillai, Norman Sadeh, and Mahadev Satyanarayanan. A scalable and privacy-aware iot service for live video analytics. In *Proceedings of the 8th ACM on Multimedia Systems Conference*, pages 38–49. ACM, 2017.

[17] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[18] Anil Madhavapeddy, KC Sivaramakrishnan, Gemma Gordon, and Thomas Gazagnaire. An architecture for interspatial communication. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 716–723. IEEE, 2018.

[19] Partha Das Chowdhury, Joseph Hallett, Nikhil Patnaik, Mohammad Tahaei, and Awais Rashid. Developers are neither enemies nor users: They are collaborators. In *IEEE Secure Development Conference 2021*, 2021.

[20] Jeff Vander Stoep. Queue the hardening enhancements. https://security.googleblog.com/2019/05/queue-hardening-enhancements.html, 2019.

[21] Ryan Levick. We need a safer systems programming language. https://msrc-blog.microsoft.com/2019/07/18/we-need-a-safer-systems-programming-language/, 2019.

[22] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. Checked c: Making c safe by extension. In *2018 IEEE Cybersecurity Development (SecDev)*, pages 53–60. IEEE, 2018.

[23] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: a safe dialect of c. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[24] Check Point. Cyber security report. https://www.checkpoint.com/downloads/resources/cyber-security-report-2021.pdf, 2021.

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

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

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

[28] Intel. 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, 2016.

[29] Charles García-Tobin. 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, 2021.

[30] NXP. ASUG-i.MX Android security user's guide. https://www.nxp.com/docs/en/user-guide/IMX_ANDROID_SECURITY_USERS_GUIDE.pdf, 2022.

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

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

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

[34] Global Platform. TEE client api specification. https://globalplatform.org/specs-library/, 2010.

[35] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, 2020.

[36] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. Timber-v: Tag-isolated memory bringing fine-grained enclaves to risc-v. In *NDSS*, 2019.

[37] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. *cure*: A security architecture with customizable and resilient enclaves. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[38] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D Garcia, and Frank Piessens. 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*, pages 1741–1758, 2019.

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

[40] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The guard's dilemma: Efficient code-reuse attacks against intel SGX. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1213–1227, 2018.

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

[42] Mustakimur Rahman Khandaker, Yueqiang Cheng, Zhi Wang, and Tao Wei. 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*, pages 971–985, 2020.

[43] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. Boomerang: Exploiting the semantic gap in trusted execution environments. In *NDSS*, 2017.

[44] Darius Suciu, Stephen McLaughlin, Laurent Simon, and Radu Sion. Horizontal privilege escalation in trusted applications. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

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

[46] Marion Marschalek. The wolf in SGX clothing. *Bluehat IL (Jan 2018)*, 2018.

[47] Jatinder Singh, Jennifer Cobbe, Do Le Quoc, and Zahra Tarkhani. Enclaves in the clouds: Legal considerations and broader implications. *Communications of the ACM*, 64(5):42–51, 2021.

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

[49] Douglas Kilpatrick. Privman: A library for partitioning applications. In *USENIX Annual Technical Conference, FREENIX Track*, pages 273–284, 2003.

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

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

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

[53] Microsoft Corporation. Open enclave SDK. https://github.com/openenclave/openenclave, 2019. Access Date :2019-08-12.

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

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

[56] Intel. Intel® 64 and ia-32 architectures software developer's manual, 2019.

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

[58] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-grained execution units with private memory. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 56–71. IEEE, 2016.

[59] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-weight contexts: An OS abstraction for safety and performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 49–64, 2016.

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

[61] Marcela S Melara, Michael J Freedman, and Mic Bowman. Enclavedom: Privilege separation for large-tcb applications in trusted execution environments. *arXiv preprint arXiv:1907.13245*, 2019.

[62] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 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*, pages 955–970, 2020.

[63] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. {Provably-Safe} multilingual software sandboxing using {WebAssembly}. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1975–1992, 2022.

[64] ARM. Arm morello. https://developer.arm.com/architectures/cpu-architecture/a-profile/morello, 2021.

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

[66] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, 2017.

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

[68] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. vtz: Virtualizing arm TrustZone. In *In Proc. of the 26th USENIX Security Symposium*, 2017.

[69] ARM. ARM architecture reference manual Armv8, for Armv8-A architecture profile documentation. https://developer.arm.com/docs/ddi0487/latest, 2018. Access Date : 2020-5-26.

[70] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanović. The risc-v instruction set manual, volume i: User-level isa, version 2.1. *RISC-V foundation*, 2016.

[71] ML inference in TrustZone with TVM. https://github.com/ez2take/tvm_inside_op-tee, 2022.

[72] Jinpeng Wei and Calton Pu. TOCTTOU vulnerabilities in UNIX-style file systems: An anatomical study. In *FAST*, volume 5, pages 12–12, 2005.

[73] Stephen Checkoway and Hovav Shacham. *Iago attacks: why the system call API is a bad untrusted RPC interface*, volume 41. ACM, 2013.

[74] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, 4:308–320, 1976.

[75] Maurice H. Halstead. Advances in software science. In *Advances in Computers*, volume 18, pages 119–172. Elsevier, 1979.

[76] Allan J Albrecht. Measuring application development productivity. In *Proc. Joint Share, Guide, and IBM Application Development Symposium, 1979*, 1979.

[77] Robert Lagerström, Carliss Baldwin, Alan MacCormack, Dan Sturtevant, and Lee Doolan. Exploring the relationship between architecture coupling and software vulnerabilities. In *International Symposium on Engineering Secure Software and Systems*, pages 53–69. Springer, 2017.

[78] Ahmad Jbara, Adam Matan, and Dror G Feitelson. High-mcc functions in the Linux kernel. *Empirical Software Engineering*, 19(5):1261–1298, 2014.

[79] Yonghee Shin and Laurie Williams. Is complexity really the enemy of software security? In *Proceedings of the 4th ACM workshop on Quality of protection*, pages 47–50, 2008.

[80] Nikhil Govil. Applying halstead software science on different programming languages for analyzing software complexity. In *2020 4th International Conference on Trends in Electronics and Informatics (ICOEI)(48184)*, pages 939–943. IEEE, 2020.

[81] Institute of Electrical and Electronics Engineers. Ieee standard dictionary of measures to produce reliable software, 1989.

[82] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Acm Sigplan Notices*, volume 48, pages 461–472. ACM, 2013.

[83] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 218–233. ACM, 2017.

[84] DPDK Intel. Data plane development kit, 2014.

[85] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[86] D Elliott Bell and Leonard J La Padula. Secure computer system: Unified exposition and multics interpretation. Technical report, MITRE CORP BEDFORD MA, 1976.

[87] Part Guide. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part*, 2(11), 2011.

[88] Jack B Dennis and Earl C Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 1966.

[89] IBM. Power isa version 3.0 b. https://ibm.box.com/, 2017.

[90] Joe Devietti, Colin Blundell, Milo MK Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the c programming language. *ACM SIGOPS Operating Systems Review*, 42(2):103–114, 2008.

[91] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 189–200. IEEE, 2012.

[92] Albert Kwon, Udit Dhawan, Jonathan M Smith, Thomas F Knight Jr, and Andre DeHon. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 721–732, 2013.

[93] Ramu Ramakesavan, Dan Zimmerman, and Pavithra Singaravelu. Intel memory protection extensions (Intel MPX) enabling guide, 2015.

[94] Alexander Richardson. Complete spatial safety for c and c++ using cheri capabilities. Technical report, University of Cambridge, Computer Laboratory, 2020.

[95] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Everything you want to know about pointer-based checking. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[96] George C Necula, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 128–139, 2002.

[97] Galen Hunt, James R Larus, Martın Abadi, Mark Aiken, Paul Barham, Manuel Fahndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, et al. An overview of the singularity project. Technical report, Technical Report MSR-TR-2005-135, Microsoft Research, 2005.

[98] Rust in the Linux kernel. https://security.googleblog.com/2021/04/rust-in-linux-kernel.html, 2021.

[99] Supporting Linux kernel development in rust. https://lwn.net/Articles/829858/, 2020.

[100] Maurice Vincent Wilkes and Roger Michael Needham. The cambridge cap computer and its operating system, 1979.

[101] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[102] Henry M Levy. *Capability-based computer systems*. Digital Press, 2014.

[103] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the Linux operating system. In *USENIX Annual Technical Conference, FREENIX Track*, pages 29–42, 2001.

[104] Asaf Shabtai, Yuval Fledel, and Yuval Elovici. Securing Android-powered mobile devices using selinux. *IEEE Security & Privacy*, 8(3):36–44, 2009.

[105] Selinux tools. https://github.com/SELinuxProject/selinux/wiki/Tools. Access Date : 2019-10-4.

[106] ICIES FORDIVERSESECURITYPOL. Theflasksecurityarchi tecture: Systemsupport, 1999.

[107] Robert Watson, Wayne Morrison, Chris Vance, and Brian Feldman. The trustedbsd mac framework: Extensible kernel access control for freebsd 5.0. In *USENIX Annual Technical Conference, FREENIX Track*, pages 285–296, 2003.

[108] Robert NM Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for UNIX. In *USENIX Security Symposium*, volume 46, page 2, 2010.

[109] Jonathan S Shapiro, Jonathan M Smith, and David J Farber. Eros: a fast capability system. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 170–185, 1999.

[110] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93. IEEE, 2009.

[111] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *USENIX Security Symposium*, 2003.

[112] Google nsjail. https://github.com/google/nsjail. Access Date : 2019-09-28.

[113] Firejail. https://github.com/netblue30/firejail. Access Date : 2019-09-28.

[114] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*, pages 57–72, 2004.

[115] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. A fork () in the road. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 14–22. ACM, 2019.

[116] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in plan 9. In *Proceedings of the 5th workshop on ACM SIGOPS European workshop: Models and paradigms for distributed systems structuring*, pages 1–5. ACM, 1992.

[117] Poul-Henning Kamp and Robert NM Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, volume 43, page 116, 2000.

[118] Daniel Price and Andrew Tucker. Solaris zones: Operating system support for consolidating commercial workloads. In *LISA*, volume 4, pages 241–254, 2004.

[119] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. Containerleaks: Emerging security threats of information leakages in container clouds. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 237–248. IEEE, 2017.

[120] Sari Sultan, Imtiaz Ahmad, and Tassos Dimitriou. Container security: Issues, challenges, and the road ahead. *IEEE Access*, 7:52976–52996, 2019.

[121] Jesse Hertz. Abusing privileged and unprivileged linux containers. *Whitepaper, NCC Group*, 48, 2016.

[122] Lei Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. Deconstructing xen. In *NDSS*, 2017.

[123] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 193–206. ACM, 2003.

[124] Richard Ta-Min, Lionel Litty, and David Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th symposium on operating systems design and implementation*, pages 279–292. USENIX Association, 2006.

[125] Takahiro Shinagawa, Hideki Eiraku, Kouichi Tanimoto, Kazumasa Omote, Shoichi Hasegawa, Takashi Horie, Manabu Hirano, Kenichi Kourai, Yoshihiro Oyama, Eiji Kawai, et al. Bitvisor: a thin hypervisor for enforcing i/o device security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 121–130. ACM, 2009.

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

[127] Jisoo Yang and Kang G Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 71–80. ACM, 2008.

[128] Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for tcb minimization. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 315–328. ACM, 2008.

[129] Jonathan M McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient tcb reduction and attestation. In *2010 IEEE Symposium on Security and Privacy*, pages 143–158. IEEE, 2010.

[130] Owen S Hofmann, Sangman Kim, Alan M Dunn, Michael Z Lee, and Emmett Witchel. Inktag: Secure applications on an untrusted operating system. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 265–278. ACM, 2013.

[131] Zhi Wang and Xuxian Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 380–395. IEEE, 2010.

[132] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, David J Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, et al. Jitsu: Just-in-time summoning of unikernels. In *NSDI*, pages 559–573, 2015.

[133] Dan Williams and Ricardo Koller. Unikernel monitors: Extending minimalism outside of the box. In *HotCloud*, 2016.

[134] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B Lee. Nohype: virtualized cloud infrastructure without the virtualization. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 350–361. ACM, 2010.

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

[136] Intel Corporation. Intel software guard extensions for Linux OS. https://github.com/intel/linux-sgx, 2019. Access Date :2019-03-01.

[137] Wikipedia contributors. Software guard extensions — Wikipedia, the free encyclopedia, 2022. [Online; accessed 3-September-2022].

[138] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium. USENIX Association*, 2018.

[139] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[140] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

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

[142] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit A Seshia. A formal foundation for secure remote execution of enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2435–2450. ACM, 2017.

[143] Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry Ponomarev, Nael Abu Ghazaleh, and Ryan Riley. Iso-x: A flexible architecture for hardware-managed isolated execution. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 190–202. IEEE Computer Society, 2014.

[144] Lawrence Esswood. *CheriOS: designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor*. PhD thesis, University of Cambridge, 2021.

[145] David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption. *White paper*, 2016.

[146] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. SEVered: Subverting AMD's virtual machine encryption. In *Proceedings of the 11th European Workshop on Systems Security*, page 1. ACM, 2018.

[147] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. Hypervision across worlds: Real-time kernel protection from the ARM TrustZone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 90–102. ACM, 2014.

[148] Riccardo Paccagnella, Pubali Datta, Wajih Ul Hassan, Adam Bates, Christopher W Fletcher, Andrew Miller, and Dave Tian. Custos: Practical tamper-evident auditing of operating systems using trusted execution, 2023.

[149] Sinisa Matetic, Moritz Schneider, Andrew Miller, Ari Juels, and Srdjan Capkun. Delegatee: Brokered delegation using trusted execution environments. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1387–1403, 2018.

[150] Pierre-Louis Aublin, Florian Kelbert, Dan O'keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch. TaLoS: Secure and transparent tls termination inside SGX enclaves. *Imperial College London, Tech. Rep*, 5(2017), 2017.

[151] Seongmin Kim, Juhyeng Han, Jaehyeong Ha, Taesoo Kim, and Dongsu Han. SGX-Tor: A secure and practical tor anonymity network with SGX enclaves. *IEEE/ACM Transactions on Networking*, 26(5):2174–2187, 2018.

[152] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *OSDI*, pages 533–549, 2016.

[153] Trusty TEE. https://source.android.com/security/trusty/. Accessed: 2018-09-08.

[154] Enterprise Mobility Solutions. White paper: An overview of samsung knox™, 2013.

[155] Android keystore system. https://developer.android.com/training/articles/keystore#HardwareSecurityModule, 2019.

[156] Apple. About the security content of iOS. https://support.apple.com/en-us/HT209340. Access Date : 2020-5-27.

[157] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using SGX. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 264–278. IEEE, 2018.

[158] ARM libddssec. https://github.com/ARM-software/libddssec.git. Access Date : 2020-03-28.

[159] Karan Grover, Shruti Tople, Shweta Shinde, Ranjita Bhagwan, and Ramachandran Ramjee. Privado: Practical and secure DNN inference with enclaves. *arXiv preprint arXiv:1810.00602*, 2018.

[160] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security Symposium*, pages 619–636, 2016.

[161] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *2015 IEEE symposium on security and privacy*, pages 38–54. IEEE, 2015.

[162] Florian Tramer and Dan Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. *arXiv preprint arXiv:1806.03287*, 2018.

[163] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. Chiron: Privacy-preserving machine learning as a service. *arXiv preprint arXiv:1803.05961*, 2018.

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

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

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

[167] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. SGXJail: Defeating enclave malware via confinement. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 353–366, 2019.

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

[169] Vasily A Sartakov, Daniel O'Keeffe, David Eyers, Lluís Vilanova, and Peter Pietzuch. Spons & shields: practical isolation for trusted execution. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 186–200, 2021.

[170] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Scalable memory protection in the penglai enclave, 2021.

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

[172] Zhui Deng, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. iris: Vetting private api abuse in ios applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 44–56. ACM, 2015.

[173] Hussain MJ Almohri and David Evans. Fidelius charm: Isolating unsafe rust code. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, pages 248–255. ACM, 2018.

[174] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. Sandcrust: Automatic sandboxing of unsafe components in Rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, pages 51–57. ACM, 2017.

[175] Format string vulnerability in the Cherokee. https://www.cvedetails.com/cve/CVE-2004-1097/. Access Date : 2020-1-5.

[176] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys. *arXiv preprint arXiv:1811.07276*, 2018.

[177] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 335–348, 2012.

[178] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. Enforcing least privilege memory views for multithreaded applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 393–405. ACM, 2016.

[179] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. Armlock: Hardware-based fault isolation for ARM. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 558–569. ACM, 2014.

[180] Nathan Burow, Xinping Zhang, and Mathias Payer. Sok: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 985–999. IEEE, 2019.

[181] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain keys–efficient in-process isolation for risc-v and x86. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1677–1694, 2020.

[182] Dorothy E Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[183] Joseph A Goguen and José Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11. IEEE, 1982.

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

[185] Thomas H Austin and Cormac Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 1–12, 2010.

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

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

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

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

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

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

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

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

[194] Andrew W Roscoe and Michael H Goldsmith. What is intransitive noninterference? In *Proceedings of the 12th IEEE computer security foundations workshop*, pages 228–238. IEEE, 1999.

[195] Andrew C Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4):410–442, 2000.

[196] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.

[197] Nikhil Swamy, Brian J Corcoran, and Michael Hicks. Fable: A language for enforcing user-defined security policies. In *2008 IEEE Symposium on Security and Privacy (SP 2008)*, pages 369–383. IEEE, 2008.

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

[199] François Pottier and Vincent Simonet. Information flow inference for ml. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(1):117–158, 2003.

[200] Stephen Chong, Krishnaprasad Vikram, Andrew C Myers, et al. Sif: Enforcing confidentiality and integrity in web applications. In *USENIX Security Symposium*, pages 1–16, 2007.

[201] M. Douglas McIlroy and James A. Reeds. Multilevel security in the UNIX tradition. *Software: Practice and Experience*, 22(8):673–694, 1992.

[202] Timothy Fraser. Lomac: Low water-mark integrity protection for cots environments. In *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*, pages 230–245. IEEE, 2000.

[203] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazieres, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 17–30. ACM, 2005.

[204] Maxwell Krohn and Eran Tromer. Noninterference for a practical difc-based operating system. In *2009 30th IEEE Symposium on Security and Privacy*, pages 61–76. IEEE, 2009.

[205] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. Flowfence: Practical data protection for emerging IoT application frameworks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 531–548, 2016.

[206] Anitha Gollamudi, Stephen Chong, and Owen Arden. Information flow control for distributed trusted execution environments. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 304–30414, 2019.

[207] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. Hardware enforcement of application security policies using tagged memory. In *OSDI*, volume 8, pages 225–240, 2008.

[208] G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. *ACM Sigplan Notices*, 39(11):85–96, 2004.

[209] Jason Oberg, Wei Hu, Ali Irturk, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. Theoretical analysis of gate level information flow tracking. In *Design Automation Conference*, pages 244–247. IEEE, 2010.

[210] Jedidiah R Crandall and Frederic T Chong. Minos: Control data attack prevention orthogonal to memory model. In *37th International Symposium on Microarchitecture (MICRO-37'04)*, pages 221–232. IEEE, 2004.

[211] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. Hdfi: Hardware-assisted data-flow isolation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 1–17. IEEE, 2016.

[212] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a flexible information flow architecture for software security. *ACM SIGARCH Computer Architecture News*, 35(2):482–493, 2007.

[213] Xun Li, Mohit Tiwari, Jason K Oberg, Vineeth Kashyap, Frederic T Chong, Timothy Sherwood, and Ben Hardekopf. Caisson: a hardware description language for secure information flow. *ACM Sigplan Notices*, 46(6):109–120, 2011.

[214] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. Why does cryptographic software fail? a case study and open problems. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, pages 1–7, 2014.

[215] Jenny Blessing, Michael A Specter, and Daniel J Weitzner. You really shouldn't roll your own crypto: An empirical study of vulnerabilities in cryptographic libraries. *arXiv preprint arXiv:2107.04940*, 2021.

[216] James Walden. The impact of a major security event on an open source project: The case of openssl. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 409–419, 2020.

[217] Maxwell N Krohn. Building secure high-performance web services with OKWS. In *USENIX Annual Technical Conference, General Track*, pages 185–198, 2004.

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

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

[220] Zahra Tarkhani, Geoffrey Brown, and Steven A Myers. Trustworthy and portable emulation platform for digital preservation. In *iPRES*, 2017.

[221] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. SGX-LKL: Securing the host os interface for trusted execution. *arXiv preprint arXiv:1908.11143*, 2019.

[222] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100. ACM, 2011.

[223] Pengtao Xie, Misha Bilenko, Tom Finley, Ran Gilad-Bachrach, Kristin Lauter, and Michael Naehrig. Crypto-nets: Neural networks over encrypted data. *arXiv preprint arXiv:1412.6181*, 2014.

[224] Alhassan Khedr, Glenn Gulak, and Vinod Vaikuntanathan. Shield: scalable homomorphic implementation of encrypted data-classifiers. *IEEE Transactions on Computers*, 65(9):2848–2858, 2016.

[225] Fan Mo, Zahra Tarkhani, and Hamed Haddadi. Sok: Machine learning with confidential computing. *arXiv preprint arXiv:2208.10134*, 2022.

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

[227] Peter M VanNostrand, Ioannis Kyriazis, Michelle Cheng, Tian Guo, and Robert J Walls. Confidential deep learning: Executing proprietary models on untrusted devices. *arXiv preprint arXiv:1908.10730*, 2019.

[228] Fan Mo, Ali Shahin Shamsabadi, Kleomenis Katevas, Soteris Demetriou, Ilias Leontiadis, Andrea Cavallaro, and Hamed Haddadi. Darknetz: Towards model privacy at the edge using trusted execution environments. *arXiv preprint arXiv:2004.05703*, 2020.

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

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

[231] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. Sok: Understanding the prevailing security vulnerabilities in TrustZone-assisted tee systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1416–1432. IEEE, 2020.

[232] Seyed Mohammadjavad Seyed Talebi, Ardalan Amiri Sani, Stefan Saroiu, and Alec Wolman. Megamind: a platform for security & privacy extensions for voice assistants. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, pages 109–121, 2021.

[233] Xinyu Lei, Guan-Hua Tu, Chi-Yu Li, Tian Xie, and Mi Zhang. Secwir: securing smart home iot communications via wi-fi routers with embedded intelligence. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, pages 260–272, 2020.

[234] Rakesh Rajan Beck, Abhishek Vijeev, and Vinod Ganapathy. Privaros: A framework for privacy-compliant delivery drones. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 181–194, 2020.

[235] Albert Harris, Robin Snader, and Robin Kravets. Aggio: A coupon safe for privacy-preserving smart retail environments. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 174–186. IEEE, 2018.

[236] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. Security analysis of emerging smart home applications. In *2016 IEEE symposium on security and privacy (SP)*, pages 636–654. IEEE, 2016.

[237] Emily McReynolds, Sarah Hubbard, Timothy Lau, Aditya Saraf, Maya Cakmak, and Franziska Roesner. Toys that listen: A study of parents, children, and internet-connected toys. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 5197–5207. ACM, 2017.

[238] Zhengxiong Li, Aditya Singh Rathore, Baicheng Chen, Chen Song, Zhuolin Yang, and Wenyao Xu. Speceye: towards pervasive and privacy-preserving screen exposure detection in daily life. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, pages 103–116, 2019.

[239] Jinghao Zhao, Boyan Ding, Yunqi Guo, Zhaowei Tan, and Songwu Lu. Securesim: rethinking authentication and access control for sim/esim. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 451–464, 2021.

[240] Payongkit Lakhan, Nannapas Banluesombatkul, Vongsagon Changniam, Ratwade Dhithijaiyratn, Pitshaporn Leelaarporn, Ekkarat Boonchieng, Supanida Hompoonsup, and Theerawit Wilaiprasitporn. Consumer grade brain sensing for emotion recognition. *IEEE Sensors Journal*, 19(21):9896–9907, 2019.

[241] Sarah N Abdulkader, Ayman Atia, and Mostafa-Sami M Mostafa. Brain computer interfacing: Applications and challenges. *Egyptian Informatics Journal*, 16(2):213–230, 2015.

[242] Mashal Fatima, M Shafique, and ZH Khan. Towards a low cost brain-computer interface for real time control of a 2 dof robotic arm. In *2015 International Conference on Emerging Technologies (ICET)*, pages 1–6. IEEE, 2015.

[243] Byung Hyung Kim, Minho Kim, and Sungho Jo. Quadcopter flight control using a low-cost hybrid interface with EEG-based classification and eye tracking. *Computers in biology and medicine*, 51:82–92, 2014.

[244] Syed Rehan Abbas Jafri, Tehreem Hamid, Rabia Mahmood, Muhammad Asjad Alam, Talha Rafi, Muhammad Zeeshan Ul Haque, and Muhammad Wasim Munir. Wireless brain computer interface for smart home and medical system. *Wireless Personal Communications*, 106(4):2163–2177, 2019.

[245] Tien Pham, Wanli Ma, Dat Tran, Phuoc Nguyen, and Dinh Phung. Multi-factor eeg-based user authentication. In *2014 International Joint Conference on Neural Networks (IJCNN)*, pages 4029–4034. IEEE, 2014.

[246] Isuru Jayarathne, Michael Cohen, and Senaka Amarakeerthi. Brainid: Development of an eeg-based biometric authentication system. In *2016 IEEE 7th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, pages 1–6. IEEE, 2016.

[247] Ofir Landau, Rami Puzis, and Nir Nissim. Mind your mind: EEG-based brain-computer interfaces and their security in cyber space. *ACM Computing Surveys (CSUR)*, 53(1):1–38, 2020.

[248] Zahra Tarkhani, Lorena Qendro, Malachy O'Connor Brown, Oscar Hill, Cecilia Mascolo, and Anil Madhavapeddy. Enhancing the security & privacy of wearable brain-computer interfaces. *arXiv preprint arXiv:2201.07711*, 2022.

[249] Sergio López Bernal, Alberto Huertas Celdrán, Gregorio Martínez Pérez, Michael Taynnan Barros, and Sasitharan Balasubramaniam. Security in brain-computer interfaces: State-of-the-art, opportunities, and future challenges. *ACM Computing Surveys (CSUR)*, 54(1):1–35, 2021.

[250] Miankuan Zhu, Jiangfan Chen, Haobo Li, Fujian Liang, Lei Han, and Zutao Zhang. Vehicle driver drowsiness detection method using wearable eeg based on convolution neural network. *Neural computing and applications*, pages 1–16, 2021.

[251] Lee Badger, Daniel F Sterne, David L Sherman, Kenneth M Walker, and Sheila A Haghighat. Practical domain and type enforcement for UNIX. In *Proceedings 1995 IEEE Symposium on Security and Privacy*, pages 66–77. IEEE, 1995.

[252] Guilherme Cox and Abhishek Bhattacharjee. Efficient address translation for architectures with multiple page sizes. *ACM SIGOPS Operating Systems Review*, 51(2):435–448, 2017.

[253] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. Kaslr is dead: long live kaslr. In *International Symposium on Engineering Secure Software and Systems*, pages 161–176. Springer, 2017.

[254] Keegan Ryan. Hardware-backed heist: Extracting ECDSA keys from qualcomm's TrustZone. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 181–194, 2019.

[255] ARM. Isolation using virtualization in the Secure world, Secure world software architecture on Armv8.4, 2018.

[256] ARM. Trusted Firmware-A secure partitions, 2018. Available at: https://osfc.io/uploads/talk/paper/18/osfc_secure_partitions.pdf.

[257] Wenjia Zhao, Kangjie Lu, Yong Qi, and Saiyu Qi. MPTEE: bringing flexible and efficient memory protection to Intel SGX. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.

[258] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. SGX-Shield: Enabling address space layout randomization for SGX programs. In *NDSS*, 2017.

[259] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y Thomas Hou. Truspy: Cache side-channel information leakage from the secure world on arm devices. *IACR Cryptology ePrint Archive*, 2016:980, 2016.

[260] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre: Stealing Intel secrets from SGX enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 142–157. IEEE, 2019.

[261] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 753–768, 2019.

[262] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. Nickel: a framework for design and verification of information flow control systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 287–305, 2018.

[263] Nadav Amit. Optimizing the {TLB} shootdown algorithm with page access tracking. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 27–39, 2017.

[264] Andrew C Myers and Andrew C Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241. ACM, 1999.

[265] Secure computing with filters. https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt. Access Date : 2019-10-4.

[266] Ravi S Sandhu and Pierangela Samarati. Access control: principle and practice. *IEEE communications magazine*, 32(9):40–48, 1994.

[267] Aydan R Yumerefendi, Benjamin Mickle, and Landon P Cox. Tightlip: Keeping applications from spilling the beans. In *NSDI*, 2007.

[268] Jonathan S Shapiro and Norman Hardy. EROS: A principle-driven operating system from the ground up. *IEEE software*, 19(1):26–33, 2002.

[269] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazieres. Securing distributed systems with information flow control. In *NSDI*, volume 8, pages 293–308, 2008.

[270] ARM. Arm trusted firmware. https://github.com/ARM-software/arm-trusted-firmware/tree/master/drivers/auth/mbedtls.

[271] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon, et al. ftpm: A software-only implementation of a tpm chip. In *USENIX Security Symposium*, pages 841–856, 2016.

[272] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. Sprobes: Enforcing kernel code integrity on the TrustZone architecture. *arXiv preprint arXiv:1410.7747*, 2014.

[273] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. Enforcing kernel security invariants with data flow integrity. In *NDSS*, 2016.

[274] Marios Pomonis, Theofilos Petsios, Angelos D Keromytis, Michalis Polychronakis, and Vasileios P Kemerlis. krˆ x: Comprehensive kernel protection against just-in-time code reuse. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 420–436, 2017.

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

[276] Raspberry Pi 3 model b. https://www.raspberrypi.org/products/raspberry-pi-3-model-b.

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

[278] Emery D Berger, Benjamin G Zorn, and Kathryn S McKinley. Composing high-performance memory allocators, 2001.

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

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

[281] Judicael B Djoko, Jack Lange, and Adam J Lee. Nexus: Practical and secure access control on untrusted storage platforms using client-side SGX. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 401–413. IEEE, 2019.

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

[283] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[284] Alexey Kurakin, Ian Goodfellow, Samy Bengio, et al. Adversarial examples in the physical world, 2016.