ABSTRACT

Title of Dissertation: EXECUTION ENVIRONMENTS FOR RUNNING LEGACY APPLICATIONS IN MULTI-PARTY TRUST SETTINGS

Stephen Herwig
Doctor of Philosophy, 2021

Dissertation Directed by: Professor Dave Levin
Department of Computer Science

Applications often assume that the same party owns all of the application's resources, and that these resources require the same level of privacy. This assumption no longer holds when organizations outsource applications to a third-party cloud, or when the application requires access to not only public content, but private configuration, such as authentication and keying material. The result of this broken assumption is that applications either must be re-written to accommodate each new security posture, or used as-is, accepting that one party exposes private data to another.

In this dissertation, I argue the following thesis: *it is possible to run legacy application binaries with confidentiality and integrity guarantees that reflect a multi-party trust setting.* I support this thesis through the design, implementation, and evaluation of two distinct application-level virtualization layers that handle trust concerns on behalf of the application: conclaves and SecureMigration. Conclaves

assume the availability of Intel SGX secure hardware enclaves and extend prior work in developing runtimes that execute legacy applications within an enclave.

In contrast, SecureMigration does not use secure hardware, but rather composes information flow control with process migration to execute a process across multiple physical machines owned and operated by distinct principals, while shielding each principal's sensitive portion of the process from its peers.

# EXECUTION ENVIRONMENTS FOR RUNNING LEGACY APPLICATIONS IN MULTI-PARTY TRUST SETTINGS

by

Stephen Herwig

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2021

## Acknowledgments

My wife, Angie, originally encouraged me to pursue a PhD, and never stopped. The first two years of my PhD were rocky; it is because of Bobby that I stayed in graduate school, and because of Dave that I graduated. Graduate school is a long endeavor; I want to thank now all of the people who helped me along the way.

I first want to thank my advisor, Dave Levin. On a practical level, Dave originated the thesis for this dissertation. On a more personal level, Dave's only agenda is to bring out the best in his students, regardless of the circumstances. While I learned a lot from Dave about how to approach problems and communicate ideas, the most enduring lesson is how to stay positive and keeping working hard, even when either may seem difficult.

I next want to thank Bobby Bhattacharjee. I took Bobby's graduate-level computer networks course (CMSC711) when I was a part-time student. At the time, I was having trouble finding my footing, and strongly considered dropping out of grad school. Bobby's course reignited my interest in systems research. More importantly, Bobby offered me a full-time position in the Systems Lab, which provided me with a stable and supportive research home, and from which I met Dave.

Dave, Bobby, and Neil Spring ran the Systems Lab for a time, and I would be remiss if I did not acknowledge Neil, who played a pivotal role in my early work involving DNS [1,2]. I always will be grateful that he went out of his way to review my last minute submission [3] to the BSDCan Conference.

I want to thank my fellow students in the Systems Lab (in roughly chronological order): Ramakrishna Padmanabhan, Matthew Lentz, James Litton, Zhihao Li, Katura Harvey, and Richie Roberts, as well as the undergraduates in the Breakerspace Lab: George Hughey, Michael Reininger, and Jayson Hurst. I collaborated with Zhihao on an earlier work [4] involving geographical-avoidance routing in Tor [5]; with Katura, Richie, and George on my earlier work involving botnets [2]; and with Michael and Jayson on a later project [6] that used conclaves. All members provided moral support and technical advice over the course of my PhD.

I also want to thank Christina Garman and Deepak Garg for being close collaborators on the work that comprises this thesis, and for serving on my committee. I benefited greatly from having two collaborators that were supportive and easily approachable, and who addressed problems with a slightly different background and viewpoint than myself or Dave. Additionally, I thank David Van Horn for serving on both my proposal and dissertation committees, and Gang Qu for serving on my dissertation committee. All have provided valuable feedback and support during the process.

I want to thank James Purtilo for helping with my admission to graduate school, and both James and A. Udaya Shankar for assisting me in the first two years of my PhD.

Last, but not least, I want to extend my sincere thanks to my family. Throughout my life, my parents have been my most important role model, helping me through some very difficult times and teaching me the values of hard work and the importance of faith. My wife, Angie, unconditionally supported me through even the most difficult times. My daughter, Grace, and her little sister on the way, in their own way, always helped me to keep things in perspective. I love you all with all my heart.

This dissertation was supported by the following NSF awards: TWC-1564143, CNS-1901325, CSR-1409249, and SaTC-1816802.

This dissertation involved collaborative efforts with the following people:

*Chapter 4*: My co-authors for conclaves [7] are Christina Garman and Dave Levin. I would also like to thank the reviewers and the Graphene-SGX team [8].

*Chapter 5*: My co-authors for SecureMigration [9] are James Larisch, Deepak Garg, and Dave Levin.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1: Introduction

Software is often written with one trust model in mind but, over time, users wish to deploy it in ways the original designers did not anticipate. For instance, current web servers were originally designed under a *monolithic* trust model: whoever was running the server was also the owner of the content, the machine it was running on, and the cryptographic keys it was using. Over time, these assumptions have become invalidated, and web servers are typically deployed in a *distributed* trust model: most websites today are hosted by third-party providers like content delivery networks (CDNs) or cloud service providers [10]. The principal running the server (the CDN) is no longer the principal that owns the content or cryptographic keys (the website owner).

When the trust assumptions underlying a piece of software change, it leads to one of two broad outcomes:

**Option 1: Weaken trust models.** First (and sadly most common), users can adapt their trust assumptions to match those of the old software's. For example, web servers need access to the cryptographic keys to run, and so the majority of website owners today *give their secret keys away* to their hosting providers [10]. In essence, users trust these third parties because decades-old software did not assume there would be third parties.

**Option 2: Redesign and re-implement.** Second, developers can adapt the software to meet the new trust models. Keyless SSL [11] is a new protocol introduced in the wake of Heartbleed [12] that allows CDN customers to maintain sole ownership of their secret keys. During the TLS handshake, the CDN-run web server effectively issues an RPC to a customer-run "key server," sending it the client's portion of the handshake to be signed or decrypted by the secret key. This relatively straightforward protocol adapted web servers to more accurate trust models, but the cost in doing so has been extensive. It required significant changes to the underlying web servers, as well as an extensive standardization process, including multiple attacks identified against its initial design [13, 14].

In a similar vein, developers could make the original software so highly configurable that it can adapt to future deployment considerations without requiring changes to the code itself. Sendmail is the exception that proves the rule; it supports a highly configurable set of deployments, but required creating its own sophisticated configuration programming language to do so [15].

In short, prior approaches have largely left us with the choice of undergoing significant engineering efforts—which may not be feasible or possible, if source code is not available—or simply defaulting to weaker security goals.

Recent approaches have sought to sidestep issues of distributed trust models by moving all sensitive components into (monolithic) trusted hardware [7, 16–19]. Part of my dissertation extends this line of work to support a richer set of applications without modification. However, wide-scale deployment of trusted hardware is not yet a reality, and it is not yet clear existing designs are trustworthy [20]. Thus, my dissertation also addresses the problem without assuming the use of trusted hardware.

## 1.1 Thesis

*It is possible to run legacy application binaries with confidentiality and integrity guarantees that reflect a multi-party trust setting.*

I define *legacy applications* as off-the-shelf binaries, in contradistinction to source code. In this dissertation, I assume that the source code for the applications is not available, thus precluding techniques that would modify or otherwise re-compile the source. On occasion, I must relax this assumption slightly, and clearly state when I do. The *confidentiality and integrity guarantees* mean that parties do not leak data and cannot tamper with another's. For this dissertation, I assume that side-channel attacks are out of scope, with defenses against side channels being an avenue for future work. The term *multi-party trust setting* means that the application (or the deployment thereof) requires resources from multiple (distrustful) parties.

In this dissertation, I focus on the multi-party sub-problem of *secure remote computation*: executing software on a remove computer owned and operated by an untrusted party, without leaking data to that party, and without that party being able to interfere in the software's execution.

My work in this dissertation focuses on applying ideas from operating system designs that emphasize a distributed and application-specific execution environment. These designs help to construct and enforce isolation boundaries that reflect the application's principals and their trust assumptions. Since the execution environment is transparent to the application, it may be modified, partitioned, or distributed across domains of varying trustworthiness, so as to reflect multi-party security goals. Transparency with respect to the application lends this approach to the important and practical property of enabling *post-hoc* refinements to an application's security and privacy guarantees, without changing the application proper.

## 1.2 Contributions

To support my thesis, I have designed, implemented, and evaluated two systems that serve as an execution environment for legacy applications: conclaves (Chapter 4) and SecureMigration (Chapter 5). Conclaves assume the availability of Intel SGX secure hardware enclaves and extend prior work in developing runtimes that execute legacy applications within an enclave. In contrast, SecureMigration does not use secure hardware, but rather combines ideas in information flow control and process migration to execute a process across multiple physical machines owned and operated by distinct principals, while shielding each principal's sensitive portion of the process for its peers.

This dissertation is structured as follows:

***Chapter 2: Background***

As a motivating and recurring example of a multi-party trust setting, I describe content delivery networks (CDNs) and discuss the current security implications of CDNs. I argue that CDNs exemplify broader trust issues endemic to cloud computing. I review prior work that addresses these issues, and place prior work into context with respect to the contributions of my own work. I divide prior solutions based on their requirement for secure hardware.

***Chapter 3: Goals and Assumptions***

I outline the common goals for conclaves and SecureMigration. Additionally, I discuss the spectrum of threat models for these systems.

***Chapter 4: Conclaves***

Conclaves extends prior work on SGX-based library operating systems (libOSes) by supporting a broader set of legacy services: namely, multi-process, shared resource, applications. My extensions result in a distributed system reminiscent of a microkernel, where each kernel service (for instance, a filesystem or shared memory) runs in a separate enclave and mediates that service's shared resources among the application's enclaved processes. With conclaves, the trust policy is code-centric, as the parties specify which processes comprise the system, as well as which processes may interact with one another.

***Chapter 5: SecureMigration***

SecureMigration is a form of application-level virtualization that composes fine-grained taint tracking with process migration to dynamically partition a legacy application across physical trust domains (physical machines). Each domain may pin private resources (memory, files, networking) such that the process must first migrate to that domain before computing with the private data. When a process is executing in a domain, the private data of its peer domains remain confidential. With SecureMigration, the trust policy is data-centric, as the parties specify the initial ownership of private data.

***Chapter 6: Conclusion and Future Work***

I conclude by revisiting the contributions of this work. I discuss immediate next steps and challenges for conclaves and SecureMigration, and describe future uses cases for both. Additionally, I motivate scenarios that relate, and potentially compose, the two systems. Finally, I take a step back from my own work and comment on the larger challenge of decoupling applications from a specific security posture.

# Chapter 2: Background

In this chapter, I explore the problem of executing old applications in new trust settings by providing an overview of a common, running example, of content delivery networks (CDNs). I describe the security implications of CDNs and describe how the CDN use case exemplifies the more general problem of secure remote computation. I highlight prior approaches for allowing users to delegate their computation to third parties while at the same time shielding their sensitive data from these parties.

## 2.1 Content Delivery Networks

CDNs, like Akamai [21] and Cloudflare [22], are third-party services that host their customers' websites (and other data). Virtually all of the most popular websites (and a very long tail of unpopular websites) use one or more CDNs to help reliably host their content [10]. Historically, CDNs have been thought of as a massive web cache [23], but today's CDNs play a critical role in achieving the performance and security that the web relies on [24].

We identify four key roles that fundamentally define today's CDNs, and their enabling technologies:

**Low latency to clients:** The primary driving feature of CDNs is their ability to offer low page-load times for clients visiting their customers' websites.

*How they achieve this:* CDNs achieve low latencies via a massive, global network of *multi-tenant edge servers*. Edge servers act primarily as reverse proxy web servers for the CDN's customers: to handle client requests, edge servers retrieve content from the customers' *origin servers*, and cache it so they can deliver it locally. CDNs direct client requests to the edge servers in a way that balances load across the servers, and that minimizes client latency—often by locating the "closest" server to the client. There are many sophisticated means of routing clients to nearby servers, involving IP geolocation, IP anycast, and DNS load balancing—but these specific mechanisms are outside the scope of this dissertation.

Edge-network services like CDNs therefore derive much of their utility from the fact that they have servers close to most clients. To this end, CDNs deploy their own data centers, and deploy servers within other organizations' networks, such as college campuses, ISPs, or companies. Indeed, today's CDNs have so many points of presence (PoPs) that they often are within the *same* network as the clients visiting their sites. To support such proximity without an inordinate number of machines, CDNs rely on the ability to host multiple tenants (customers) on their web servers at a time.

**Manage customers' keys:** As the web moves towards HTTPS-everywhere [25], customers increasingly rely on CDNs to store their HTTPS certificates and the corresponding secret keys, so that they can accept TLS connections while maintaining low latency to clients.

*How they achieve this:* CDNs manage their customers' keys in a variety of ways: sometimes by having their customers upload their secret keys, but typically by simply generating keys and obtaining certificates on their customers' behalf [10,26]. Many CDNs combine multiple customers onto single "cruiseliner certificates" under the same key pair—these customers are not allowed to access their own private keys, as that would allow them to impersonate any other customer's website on the same cruiseliner certificate [10]. A recent protocol, Keyless SSL [11], has been proposed to address this; we describe this in more detail in §2.4.2.

**Absorb DDoS traffic:** CDNs protect their customers by filtering DDoS traffic, keeping it from reaching their customers' networks.

*How they achieve this:* CDNs leverage economies of scale to obtain an incredible amount of bandwidth and computing resources. Their customers' networks block most inbound traffic, except from the CDN. Thus, attackers must overcome these huge resources in order to impact a customer's website.

**Filter targeted attacks:** An often overlooked but critical feature [24] of today's CDNs is the ability to filter out (non-DDoS) attack traffic, such as SQL injection and cross-site scripting attacks.

*How they achieve this:* Unlike with DDoS traffic, the primary challenge behind protecting against targeted attacks is *detecting* them. CDNs achieve this by running *web-application firewalls (WAFs)*, such as ModSecurity [27]. WAFs analyze the plaintext HTTP messages, and compare the messages against a set of rules (often expressed as regular expressions [28]) that indicate an attack. Edge servers only permit benign data to pass through to the customer's origin server.

### 2.1.1 Security Implications of CDNs

Simultaneously fulfilling these four roles—low latency, key management, absorbing large attacks, and blocking small attacks—inherently requires processing client requests on edge servers. In the presence of HTTPS, however, this processing requires edge servers to have at least each TLS connection's session key, if not also each customer's private key.

It is therefore little surprise that CDNs have amassed the vast majority of private keys on the web [10,26]. This has significant implications on the trust model of the PKI and the web writ large: today's CDNs could arbitrarily impersonate any of their customers—and recall that virtually all of the most popular websites use one or more CDNs [10].

Even if one were to assume a trustworthy CDN, the need to store sensitive key materials on edge servers introduces significant challenges. CDNs have historically relied on a combination of their own physical deployments and deployment within third-party networks, such as college campuses. To protect their customers' keys, some CDNs refuse to deploy HTTPS content anywhere but at the data centers they have full physical control over [24]. However, as the web moves towards HTTPS-everywhere, this means that such CDNs can no longer make as much use out of third-party networks. In short, without additional protections for private and session keys on edge servers, the move towards HTTPS-everywhere represents an *existential threat* to edge-network services.

## 2.2 Secure Remote Computation

CDNs are just one example of the broader problem of organizations outsourcing their applications or infrastructure to a third party despite exposing private data about themselves in the process. The general problem is that of *secure computation*: executing software while maintaining the confidentiality of some portion of the computation, such as its inputs or outputs. In this thesis, I specifically focus on the subdomain of *secure* ***remote*** *computation*: executing software on a remote computer owned and operated by an untrusted party, without leaking data to that party, and without that party being able to interfere in the software's execution.

For both conclaves and SecureMigration, the driving use case is a customer outsourcing their web hosting to a CDN (or, more generically, a cloud operator). As such, the evaluations for both systems focus on running a popular webserver under configurations that maintain either the confidentiality or integrity of distinct customer resources. However, the problem setup and the systems themselves readily transfer to outsourcing other types of services, such as DNS, mail, or VPN servers, while shielding the sensitive data of either the customer or their clients.

Next, I review prior work that addresses solutions to either the CDN-case specifically, or secure remote computation in general.

## 2.3 TEE-based Solutions

*Trusted execution environments* (TEEs) provide hardware protections for running small trusted portions of code with guarantees of confidentiality and integrity. Applications can be guaranteed that code executed within the TEE was run correctly and that any secrets generated during execution remain safely within it as well.

A wide range of TEEs are available today, with varying functionalities. In this dissertation, I focus on Intel's Software Guard Extensions (SGX).

### 2.3.1 Intel SGX

Intel's SGX provides a new mechanism for trusted hardware and software as an extension to the x86 instruction set [29,30]. A program called an *enclave* runs at high privilege in isolation on the processor in order to provide trusted code execution, while an untrusted application can make calls into the enclave. While these enclaves can be statically disassembled (so the code running in the enclave is not private), once an enclave is running, its internal state is opaque to any observer (even one with physical access), as are any secrets generated.

Enclave memory resides in a special region of system memory called the Enclave Page Cache (EPC). Code and data from multiple enclaves can reside within the EPC, but each EPC page is owned by only a single enclave and this owner is the only one allowed to access the page. An on-chip Memory Encryption Engine (MEE) [31] encrypts and integrity protects the EPC; the MEE decrypts enclave memory when the memory is brought into the CPU's cache. In current SGX-capable systems, the EPC is 128 MiB, of which 93 MiB is usable (the rest is used to store metadata for integrity protections).

Enclaves must be measured and signed by their creator and cannot run without this signature, and the enclave state is checked against this measurement before running. An enclave can also cryptographically *attest* to its current state, in order to prove that it correctly executed code [32, 33]. Another feature is the ability to cryptographically *seal* data to be used across multiple invocations of an enclave [33, 34]. SGX also provides such features as trusted time and monotonic counters [35,36]. However, an enclave currently has no access to networking functionality itself, so it must rely on the untrusted application for all network interactions. In fact, enclaves are prohibited from making any system calls, so these must be proxied through the untrusted OS as well.

### 2.3.2 Running Legacy Applications on SGX

Various works use SGX as a mechanism for achieving shielded execution of unmodified legacy applications. These works generally differ in how much of the application's code runs within the enclave.

At one extreme, TaLoS [17] ports the LibreSSL library to SGX so that the application terminates TLS connections in an enclave; the rest of the application remains outside the enclave, unchanged. At the other extreme, SCONE [18] moves the entire C library into the enclave. Haven [19] and Graphene [16] carry this approach further by implementing kernel functionality in an enclave by means of a library operating system (libOS). libOSes refactor a traditional OS kernel into a user-land library that loads a program. The program's C library is modified to redirect system calls to the libOS, which in turn either services the calls internally or calls into the untrusted OS when the host's resources are needed. Aurora [37] extends the libOS from the SGX enclave to System Management Mode (SMM) by running device drivers in SMM memory.

Many applications, including a CDN's webserver, involve multiple processes, and of these works, only Graphene supports forking and executing new processes within enclaves. However, Graphene's support for shared state among multiple enclaves, such as a read-write file system and shared memory, is limited. We discuss these limitations in §4.1 and our extensions to Graphene in §4.2.

Other work [38] provides frameworks for developing *new* software that takes advantage of SGX, whereas our interest is in supporting *legacy* applications.

### 2.3.3 Side-Channel Attacks

We must address the recent rise of side-channel attacks against SGX, including the speculative execution attack Foreshadow [20, 39]. This attack allows for the extraction of not only the entire SGX enclave's memory contents but also the attestation and sealing keys. We note that this attack would break the security guarantees that we provide with conclaves. Intel has stated that SGX is explicitly designed to not deal with side-channel attacks in its current state and leaves handling this up to enclave developers [40, 41]. Regardless, Intel has released both microcode patches and recommendations for system level code that at the current time address Foreshadow and known related attacks [39, 42, 43]. There is also ongoing research to address both speculative execution as well as other cache-based side-channel attacks on SGX and in general [43–46]. We consider protections against such side-channel attacks to be outside of the scope of this work and rely on these defenses.

## 2.4 TEE-less Solutions

### 2.4.1 HTTP Solutions

Several systems have proposed that the origin server digitally sign their data [23, 47] or embed cryptographic hashes directly into HTML [48, 49], which clients can then verify. Such approaches ensure provenance, freshness, and integrity of web assets served by a proxy—without requiring the proxy to store the origin server's private key. However, they do not provide for confidentiality, nor do they allow for CDN services such as media transcoding and web application firewalls. Moreover, they place the origin on the critical path, thereby increasing latency and making them more susceptible to attack.

### 2.4.2 TLS Solutions

Other approaches allow origin servers to retain ownership of their private keys by changing the server-side implementation of TLS. SSL Splitting [50] leverages the fact that a TLS stream comprises data records and authentication records (MACs), and develops a new protocol in which the origin sends the authentication records and the proxy merges them with the data records to form the complete TLS stream. In essence, this implements the above HTTP solutions in TLS, and thus suffers from the same limitations of requiring the origin server to be on the fast path.

Cloudflare's Keyless SSL [11] takes advantage of the fact that TLS only uses the website's private key in a single step of the TLS handshake. Like SSL Splitting, Keyless SSL keeps the master private key off of, and unknown to, the proxy, but unlike SSL Splitting, Keyless SSL does not provide for content provider endorsement of the content the proxy serves. Neither SSL Splitting nor Keyless SSL provides for the protection of the session keys from the CDN provider.

Another line of work modifies TLS to allow for the interception of traffic by middleboxes [51–53]. This is contrary to our desire to support legacy applications; it is not clear how these solutions would be integrated with tools such as WAFs.

### 2.4.3 Program Partitioning

Program partitioning is a technique for automatically breaking a monolithic program into multiple communicating subprograms for the purpose of separation of privilege [54, 55]. For instance, Privtrans [54] uses developer-supplied source-level annotations of sensitive data and code, along with static analysis and source-level translation, to partition the source code into trusted and untrusted programs. The fundamental difference between our work and these prior efforts is that they require access to (and the ability to alter) the application's source code, whereas we operate on binaries.

Jif/split [56] is one of the earliest *language-leve*l approaches for using program partitioning to protect the confidentiality of data within an application comprising heterogeneously trusted hosts. Jif/split extends Java with explicit program annotations and security types that specify information flow control restrictions, and the compiler and run-time system reject programs that violate the restrictions. Civet [57] uses a similar approach to partition a Java program into untrusted portions and trusted portions, where the trusted portions execute within an enclave. TinMan [58] applies taint tracking (based on TaintDroid [59]) and process migration (based on Comet [60]) within the Java virtual machine itself, so as to keep private data off of a smartphone and isolated to a trusted server. Unlike Jif/split, TinMan is mostly transparent to the Java application, requiring modest changes to the application's UI for selecting handles to the private data.

While language-level approaches naturally express security according to the semantics of the program, conclaves and SecureMigration are instead language neutral. An interesting area of future work would be to combine program partitioning with the information flow control features of SecureMigration to partition a program along likely boundaries, using taint tracking to ensure that there is no data leakage.

### 2.4.4 Cryptographic Solutions

Fully homomorphic encryption (FHE) [61,62], functional encryption (FE) [63], and secure multiparty computation (SMC) [64] offer potential building blocks for secure remote computation. In broad terms, FHE allows for arbitrary computations over encrypted data, while FE and SMC are restricted to computing a plaintext outputs from encrypted inputs. In particular, SMC allows *multiple* parties to cooperatively compute a function over their private inputs such that the output remains private. However, each of these approaches suffers from drastic performance overheads, while violating my goal of supporting legacy applications.

Various techniques [65–68] focus on the special case of searchable encryption so as to support specific uses cases, such as deep packet inspection (DPI), while still maintaining the privacy of the data. In general, these approaches require changes of some sort to the endpoint(s), suffer from performance overheads, and do not achieve the rich and varied features we require to support arbitrary applications.

# Chapter 3: Goals and Assumptions

In this chapter, I enumerate the common design goals for both conclaves and SecureMigration. I also define the threat models that each system assumes.

## 3.1 Goals

At a high-level, my goal is to make heterogeneous trust a concern of the application's runtime execution environment, rather than a concern of the application developer. I distill my overarching goal down to three specifics requirements:

1. **Preserve confidentiality of private data:** The execution environment *must* maintain the confidentiality of the private keys and private data sets of each party.

2. **Ensure data integrity:** The execution environment *must* preserve the integrity of each party's data. This requirement is moot for the honest-but-curious threat model, but essential for the byzantine faulty model (see §3.2).

3. **Require minimal changes to existing applications:** The execution environment *must* remain transparent to the application, and thus require few, if

any, changes to the application. This requirement also implies that trust *must* be expressible as a policy configuration.

Collectively, these requirements decouple trust from the application logic and express trust as a policy, with the execution environment enforcing the policy.

## 3.2 Threat Models

Trust is an expectation that a party or application operates in its stated purpose, and, in particular, does not purposefully leak private data. The goal of an adversary is to leak the private data of another party involved in the computation.

In this thesis, I make the idealized assumption that the parties trust the application proper, and that the application does not leak data via control flows or other side channels. In other words, the parties know the application, but may not trust the hosting platform. Likewise, I assume that all parties faithfully run the components of the execution environment (with conclaves, secure hardware guarantees this assumption; for SecureMigration, I must explicitly make this assumption).

Outside attackers are within the threat model. However, I assume that both the application and the execution environment are non-buggy, and thus an adversary cannot gain a thread of execution in either (the adversary may still have a thread of execution on the application's machine, though).

I allow myself to assume several threat models.

### 3.2.1 Honest-but-curious

The weakest threat model I assume is *honest-but-curious*: each principal faithfully executes the application, as well as the components and protocols that comprise the execution environment, but may inspect and analyze the system and its data for the purposes of trying to recover the private data of its peers. Stated differently, this model allows an adversary to act as a passive attacker within the system.

Honest-but-curious is the only threat model I assume for SecureMigration, as, in the absence of secure hardware, it is difficult to guard against prevarication of the execution environment, or tampering of the application proper. For conclaves, this threat-model allows for faster implementations of the execution environment which guarantee only confidentiality, but not integrity.

Admittedly, these are fairly weak assumptions, but ones which allow me to make initial progress towards my novel goals, and which are nevertheless realistic in many instances. For instance, my assumptions hold in cases where the same principal controls all resources and uses either conclaves or SecureMigration for physical isolation as a means of defense-in-depth. Moreover, an honest-but-curious trust model is significantly more strict than the fully-trusted model of today's CDNs (which are often the sole owners of their customers' secret keys [10]).

### 3.2.2 Byzantine faulty behavior

In this more extreme threat model, the entity hosting the hardware can deviate arbitrarily from the protocol, alter any software running in an untrusted environment on that hardware, and passively monitor traffic, and actively interact with the application. Nonetheless, we assume attackers cannot violate basic assumptions of cryptography or trusted hardware. In the context of a CDN deployment, a website may wish to adopt this model for CDNs whom they do not trust. Likewise, CDNs may assume this threat model when using edge-network servers that they do not personally host or have physical control over [24].

# Chapter 4: Conclaves

In this chapter, I present my work on the design and implementation of conclaves. Conclaves extend prior systems for executing legacy applications in Intel SGX enclaves to support applications that are multi-process and that share resources, such as the filesystem or memory, among the processes. To support shared resource abstractions, conclaves create a distributed execution environment, reminiscent of a microkernel, where each shared resource is provided by a user-space "kernel server" running in a an enclave, and where the enclaved application processes transparently issue their system calls to these kernel servers. In this way, the kernel servers mediate all access to the shared resources and ensure consistency.

Using conclaves, we[1] present the design of the first truly "keyless CDN," which we call Phoenix. Phoenix performs all of the quintessential tasks of today's CDNs—hosting web servers, applying web application firewalls, performing certificate management, and more—without requiring CDNs to gain access to sensitive key material, and without having to change legacy web applications. We present a detailed design and implementation of Phoenix, and evaluate it on Intel SGX hardware. Our results indicate that conclaves scale to support multi-tenant deployments with

[1]This work involved collaborations with Christina Garman and Dave Levin.

modest overhead (~2–3× for many configurations).

The rest of this chapter is organized as follows. I present the design of conclaves and of Phoenix in §4.1, and their implementation in §4.2. I present our evaluation in §4.3 and conclude in §4.4.

## 4.1 Design

At a high level, our approach is to deploy CDNs in enclaves. However, doing so in a manner that permits multi-tenancy and support for legacy applications is challenging. Prior work on SGX libOSes [16, 18, 19] make it possible to run legacy applications within an SGX enclave, but all of them either lack multi-process support completely, or only support multiple processes in a restricted environment. Conversely, we aim to be able to support dynamic scaling up and down of web servers, tenant configurations, and security postures.

To address these challenges, we introduce a new architectural primitive that we call a *conclave*: in essence a container of enclaves. As we will show, conclaves permit flexible deployment configurations and achieve security in multi-tenant settings. We first describe the conclave design, and then how we compose them to build the first "keyless CDN," Phoenix.

### 4.1.1 Conclaves Design

The conclave design extends a libOS to support shared state abstractions among multiple processes. Recall from §2.3.2 that libOSes expose traditional OS kernel services within an enclave, and either handle the system calls themselves or, when necessary (e.g., to send a network packet), hand them off to the untrusted OS. Graphene [16] supports the critical system calls `fork` and `exec` by automatically spawning a brand new enclave, and performing a checkpoint-and-migration (essentially copying the first enclave's memory pages into the second). Graphene further offers some support for these separate processes (enclaves) to communicate with one another over pipes, and implements signals, semaphores, message queues, and exit notifications as RPCs over these pipes. In other words, Graphene essentially turns a traditional multi-process application into a "distributed system" of enclaves, along with some basic plumbing to allow them to communicate with one another.

However, two important multi-process abstractions that Graphene does not support with confidentiality and integrity guarantees are a read-write filesystem, and shared memory. Graphene's sole filesystem, `chrootfs`, is modeled as a restricted view of the host's filesystem. Graphene does not support shared memory at all (neither anonymous nor file-backed).

Conclaves extend upon this prior design by leaning into the distributed system nature of it. We implement kernel services as *kernel servers*; applications act as clients, connecting to and issuing requests to kernel services—via pipes or TLS network connections. The kernel servers also run atop the libOS. Our design is effectively that of a multi-server microkernel system, similar to GNU Hurd [69] or Mach-US [70], in which shared resource abstractions are implemented as a set of enclaved daemons shared by all processes in the system.

#### 4.1.1.1 Conclave Kernel Servers

Using the NGINX web server [71] as a guide (as software representative of a CDN edge server), we identified five key shared resources: files, shared memory, locks/semaphores, cryptographic keys, and time. For flexibility in deployment configurations, we implement four servers to manage these resources[2]:

**fsserver** The fsserver provides a file system interface to user applications. Much like a remote file system, the fsserver performs strict access control to restrict access only to the relevant enclaves. We discuss how this access control is provisioned in §4.1.2.2. NGINX uses the file system for storing cached and persistent web resources.

**memserver** The memserver provides an interface for creating, accessing, manipulating, and locking shared memory. NGINX uses shared memory for storing usage statistics, metadata for the on-disk HTML caches, and state for TLS session resumption.

**keyserver** The keyserver is an SGX enclave rendition of a hardware-security module (HSM): the keyserver stores private keys and performs any private key cryptographic operations. Like Keyless SSL [11], this not only maintains the confidentiality of the private key with respect to an untrusted host, but also isolates the key to an address space distinct from the application's, thereby guarding against critical memory disclosure vulnerabilities, such as Heartbleed [72].

**timeserver** Given that the components of a conclave must authenticate one another, we need trusted time to guard against attacks that trick the conclave into

[2]Due to the common pattern of using locks with shared memory, the memserver manages both.

accepting expired certificates. Unfortunately, SGX itself does not provide trusted time. Its SDK [35] provides features [36] for retrieving coarse-grained, monotonic time through a protected clock provided by Intel's Converged Security and Management Engine (CSME), but not all processors support it [73].

Instead of relying on the CSME, we simply design a remote, signed timestamping server. The timestamping server runs outside of an enclave, on a remote trusted machine (e.g., at the CDN's customer). The timeserver's purpose is not to provide fine-grained precision to the conclaved processes, but rather to serve as an integrity check of the time those processes receive from the untrusted host.

In §4.2, we detail several variants of each of these kernel servers, covering various trade-offs between performance and security. While we have found that these four kernel servers suffice for NGINX—and, we believe, for a wide range of networked applications—it is possible that other applications may need more. For instance, some applications may require an enclaved network stack (as a type of TUN/TAP device), or virtualization of the CPU to handle fairness concerns.

#### 4.1.1.2 Conclave Images

Conclaves bundle the SGX microkernel runtime and application suite into a deployable and executable image, reminiscent of a traditional container image. When the conclave is executed, the first enclave process that is executed is an init process, which executes the kernel servers and the specified application proper. From that point, the application can fork, spin up new applications, and so on.

### 4.1.2 Phoenix Design

Conclaves provide a multi-process runtime for running multi-process legacy applications within SGX enclaves. Phoenix addresses a number of remaining questions concerning how the customer and CDN operator deploy and provision the combined runtime and application suite.

The core problem Phoenix solves is that the runtime and application need various assets—in particular, keying material—in order to successfully and securely execute. These assets must be delivered in a manner that is shielded from CDN inspection or tampering. Furthermore, as one of our goals is to not burden the customer with running additional services, we, paradoxically, must have the CDN manage the provisioning of these assets on behalf of the customer. Finally, Phoenix's design must allow for multi-tenant deployments. We address each of these in turn.

We present a high-level overview of Phoenix's design in Figure 4.1. Its design spans three principles: (1) the CDN customer, who must run the origin server as they do today, as well as an agent for provisioning conclaves, (2) the core CDN servers, which make and enforce decisions of where exactly to deploy customers' content, and (3) the CDN edge server itself, which receives the majority of the changes.

#### 4.1.2.1 Bootstrapping Trust

We first address how the conclave, viewed as a distributed system, establishes the trust of each member node, whether kernel server or application process. This

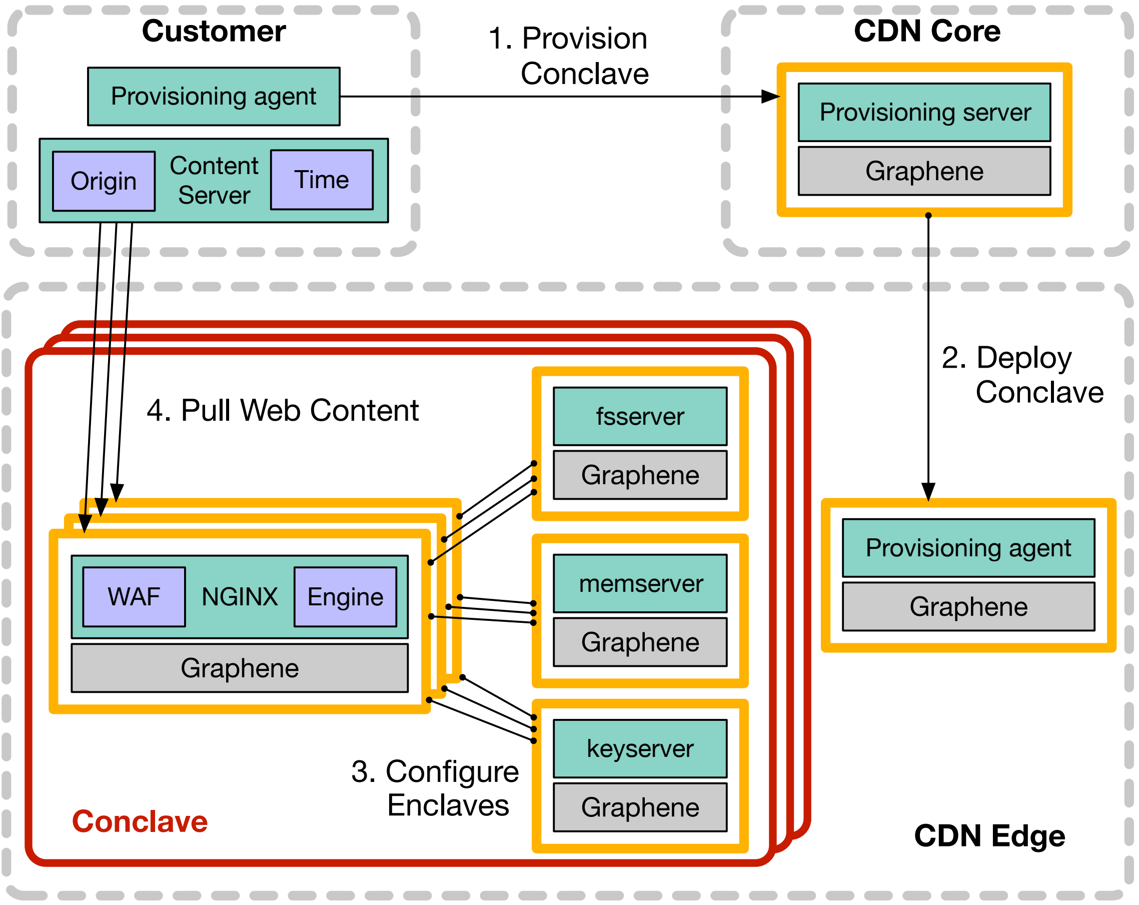


Figure 4.1: Architectural design of Phoenix. Multiple enclaves (yellow boxes) reside in a logical conclave (red boxes), permitting multiple processes and multi-tenant deployments. The CDN Edge and Core servers run on untrusted hosts.

is a chicken-and-egg problem of establishing a secure channel between two nodes without first provisioning these nodes with, say, private keys and certificates for mutual authentication.

The standard approach for establishing a secure channel in an SGX setting is to use SGX as a root of trust and enclave attestation as a form of authenticated identity, and to merge this form of attestation into the establishment of the shared channel secret. To that end, Phoenix follows closely from the work of Knauth et al. [74], which integrates attestations with TLS by adding the SGX quote as an X.509 certificate extension. This has the effect of making channel establishment and SGX attestation occur together, atomically, with respect to the channel protocol. Certificate validation can thus be extended to examine these new extensions.

Adding new certificate extensions, of course, is not the full story. In this setup, the enclave generates an ephemeral key pair. SGX quotes cover the enclave image, the enclave signer, non-measurable state, such as the enclave mode (e.g., debug vs. production), and, optionally, any additional data (user data) the enclave wants to associate with itself. The trick for ensuring the atomicity of attestation and secure channel establishment is for the enclave to specify as user data a hash of the ephemeral public key. Since the key pair is created within the enclave, and since only an enclave can get a valid quote, such user data binds the key pair to the enclave. The enclave then generates a self-signed certificate for this ephemeral public key, which includes the aforementioned extensions for the quote and Intel Attestation Service (IAS) verification.

In our conclave setup, the attestation is a local attestation, and validation of the quote is based on a list of valid attestation values in the manifest. Specifically, the manifest specifies a graph of which processes can establish secure channels with one another.

#### 4.1.2.2 Provisioning Server and Provisioning Agents

Having bootstrapped trust within the conclave, our next challenge is the delivery of sensitive assets to the conclave. Phoenix has the init process spawn a process called the *provisioning agent* that communicates remotely with a *provisioning server* operated by the CDN. The provisioning agent periodically beacons to the provisioning server, and downloads and installs any new conclave assets.

The provisioning agent and server both run in an enclave, and use essentially the same method for secure channel establishment as what we described for channel establishment within the conclave. The main difference is that the quote is generated and validated using SGX's remote attestation protocol, rather than the local attestation protocol.

At this point, we have recursed nearly to the base case; all that is needed for end-to-end asset encryption is for the customer to post assets to the provisioning server.

#### 4.1.2.3 Key Management

The last thing we must address is how Phoenix enables the CDN to manage its customers' keys. Today, CDNs manage their customers' keys in a handful of ways [10, 26]; customers can generate their own keys and upload them to the CDN, or they can delegate all key and certificate management to the CDN. When CDNs manage their customers' certificates, they often put multiple customers on a single "cruise-liner certificate" [10], under a single key pair.

Phoenix supports all of these configurations by shifting them into the (enclaved) provisioning server. When customers wish to upload their keys, they establish a secure connection from their provisioning agent to the CDN's provisioning server. When the CDN manages its customers' keys, the provisioning server generates key pairs and runs Let's Encrypt's [75] ACME protocol [76]—from within the enclave—to request the certificates. The provisioning server can then load this data onto edge servers however it sees fit, by connecting to provisioning agents running in enclaves on the edge servers (see Figure 4.1). The end result is that, unlike today, the CDN will never learn the secret keys. In fact, when the CDN manages its customers' keys, *no one* learns them, as they will forever reside within one or more enclaves.[3]

### 4.1.3 Deployment Scenarios

Phoenix's conclave-based design permits a diverse range of deployment options to support varying threat models like those described in §3.2. There are two dimensions for describing edge server deployments: First, a deployment can be *single-tenant* or *multi-tenant*, based on whether there is one or more customers on

[3]Recall our threat model is that the CDN's code within the conclaves is trusted and does not leak data.

a given edge server (physical or virtual). Second, a given customer's deployment can be *fully-enclaved* or *partially-enclaved*, based on whether all or just a specific subset of components are executed in an enclave. The provisioning agent and server design handle these use cases uniformly. Throughout the design of Phoenix, we have described the *single-tenant, fully-enclaved* deployment. Below, we discuss two other potential deployments.

*Single-tenant, partially-enclaved* deployments handle an honest-but-curious attacker wherein the customer trusts the CDN with everything but the private key. In this deployment, only the keyserver and provisioning agent reside in the conclave. This configuration is similar to Keyless SSL, but without requiring modifications to the application or TLS.

*Multi-tenant* deployments multiplex customers at one of three places. First, the CDN operator can trivially place a proxy server (for example, an HAProxy [77]) on the edge server; the proxy examines the SNI field of the client request and proxies to the relevant conclave. In other words, this strategy reduces to running *single tenant, fully-enclaved* conclaves for many customers. Second, if the application is conducive to multiplexing, then the CDN operator can run an instance of the application in an enclave, with the application's configuration reflecting the customer partitions; each customer then runs their own conclave of kernel servers. As an example, NGINX can run multiple virtual servers; the resources for each virtual server are mounted on mountpoints within the application that point to each customer's respective kernel servers. Finally, the kernel servers themselves can multiplex the resources of several customers. These represent different trade-offs: more multiplexing can increase the attack surface, but requires less resources to achieve high performance (SGX can incur significant overhead in switching between enclaves on a given CPU).

## 4.2 Implementation

We implement conclaves and Phoenix as extensions to the open-source Graphene SGX libOS [16]. In this section, we present details of this implementation. We have made our code and data publicly available so that others can continue to build off this work.[4]

### 4.2.1 Kernel Servers

We implement the fsserver, memserver, and keyserver as single-threaded, single-process, event-driven servers that communicate with the application's Graphene instances over a TLS-encrypted stream channel. In the case of a TCP channel, we disable Nagle's algorithm due to the request-response nature of the RPCs. The timeserver uses a datagram channel. Each server is independent.

**fsserver** For our file server, *nextfs*, we extend lwext4's [78] userspace implementation of an ext2 filesystem into a networked server. nextfs uses an untrusted host file as the backing store, similar to a block device. We develop three variants of this device to accommodate different security postures, and a fourth for comparison purposes.

- **bd-std** stores data blocks in plaintext, without integrity guarantees. This serves

[4]Our code may be found at `https://phoenix.cs.umd.edu`.

as a baseline in our evaluation.

- **bd-crypt** encrypts each block using AES-256 in XTS mode, the de facto standard for full-disk encryption [79, 80]. We base each block's initialization vector on the block's ID. This, too, lacks integrity guarantees, and is thus suitable only for an honest-but-curious attacker.

- **bd-vericrypt** adds integrity guarantees to bd-crypt, thus providing authenticated encryption. It does so by maintaining a Merkle tree over the blocks: a leaf of the tree is an HMAC of the associated (encrypted) block, and an internal node the HMAC of its two children. To keep the memory needs of the enclave small, bd-vericrypt consults a serialized representation of the tree in a separate file, rather than use an in-memory representation. The root of the Merkle tree exists both on the file and in enclave memory; the HMAC key exists only in enclave memory. As an optimization for reducing reads and writes to the Merkle tree file, bd-vericrypt maintains an in-enclave LRU-cache of the tree nodes. bd-vericrypt is the appropriate choice in a Byzantine threat model.

**memserver** We implement shared memory as filesystems that implement a reduced set of the filesystem API[5]: `open`, `close`, `mmap`, and `advlock` (`advlock` handles both advisory locking and unlocking). In our shared memory filesystems, files are called *memory files*, and either represent a pure, content-less lock, or a lock with an associated shared memory segment. Memory files are non-persistent: they are created on the first open and destroyed when no process holds a descriptor to the

[5]Graphene does not have a unified filesystem and memory subsystem, and thus `munmap` is not currently available as a filesystem operation.

file and no process has the associated memory segment mapped.

We implement three versions of shared memory. Each stores a canonical replica of the shared memory at a known location (either a particular server or file). Upon locking a file, the client "downloads" the canonical replica and updates its internal memory maps. On unlock, the client copies its replica to the canonical.

- **sm-vericrypt-basic** uses an enclaved server to keep the canonical memory files in an in-enclave red-black tree.

- **sm-vericrypt** implements a memory file as two untrusted host files: a mandatory lock file, and an optional segment file. When a client opens a memory file, the sm-vericrypt server creates the lock file on the untrusted host, and the Graphene client maps (`MAP_FILE|MAP_SHARED`) the lock file into untrusted memory. The client then constructs a ticketlock structure over this untrusted shared memory. Since the untrusted host may manipulate the ticketlock's turn value, a shadowed, trusted turn number is maintained by the enclaved sm-vericrypt server. After the client has acquired the lock, the client makes an RPC to the server to verify the turn number. The server thus acts as a trusted monitor of the untrusted monotonic counter.

  If a client `mmap`s the memory file, the server creates the associated segment file on the untrusted host. When the client subsequently locks the file, the client makes a `lock` RPC to server, which returns the keying and MAC tag information for the segment. The client copies the untrusted memory segment into the enclave, and uses AES-256-GCM to decrypt and authenticate the data. When a client unlocks the file, the client generates a new IV, copies an encrypted version of its in-enclave memory segment into the untrusted segment file, and makes an `unlock` RPC to the server, passing along the new IV and MAC tag.

- **sm-crypt** assumes the untrusted host does not tamper with data. As such, sm-crypt uses AES-256-CTR instead of AES-256-GCM, and does not need an enclaved server to monitor the integrity of the ticketlock and IV.

**keyserver** We implement the keyserver as two components: the keyserver proper, and an OpenSSL engine ("Engine" in Figure 4.1) that the application loads as a shared library; the engine proxies private key operations to the keyserver. Unlike the fsserver and memserver clients, the key client operates at the application layer, outside of Graphene.

OpenSSL's engine API requires the caller (in our case, NGINX) to provide an `RSA` object, which contains the secret key. To avoid having to expose the key, we modified OpenSSL to populate `RSA` objects with dummy keys that instead serve as identifiers that the keyserver uses to look up the real keys it stores securely.

To reduce the number of connections and avoid a dependency on the memserver for lock files, our engine maintains the property that all keys for the same keyserver, within the same process, share a single connection. This requires that the engine detect forking by the application, which we achieve by also associating process IDs with the `RSA` objects.

**timeserver** We modify the Graphene system call handlers for `getttimeofday`, `time`, and `clock_gettime` to optionally proxy application calls to a remote, trusted, timestamp signing server. The use of such a timeserver, and the related parameters, such as the timeserver's public key, are specified by the Graphene user (here, the content provider), and hard-coded into Graphene's configuration. As a freshness guarantee, each request includes a new, random nonce, generated by the Graphene system call handlers. The timeserver, in turn, returns an RSA signature over a message consisting of the current time concatenated with this nonce.

Our timeserver approach resembles Google's roughtime protocol [81]; future work would fully port the roughtime protocol to Graphene to reduce the need for a trusted timeserver by instead tolerating some fraction of misbehaving servers. Note, however, that, in the SGX setting, both our approach and roughtime are best efforts; an untrusted host that identifies the traffic between the Graphene client and timeserver could, for instance, "slow down" time by delaying the responses.

### 4.2.2 Graphene Modifications

We have modified Graphene to add missing functionality and increase performance.

**Exitless System Calls** For potential performance gains, we merge Graphene's exitless system call patch [82]. The patch is an optimization, similar to the solution proposed elsewhere [18, 83, 84], that enables enclaves to issue system calls without first making an expensive enclave exit and associated context switch to the untrusted host process.

For every SGX thread, the exitless implementation spawns an untrusted (outside of the enclave) RPC thread that issues system calls on behalf of the SGX thread.

The RPC and SGX threads share a FIFO ring buffer for communicating system call arguments and results. To issue a system call, the SGX thread enqueues the system call request, and waits on a spinlock for the RPC thread's response. To conserve CPU resources, SGX threads only spin on the spinlock a set number of times (by default, 4096 spins) before falling back to sleeping on a futex (the `futex` call is a normal ocall).

**BearSSL** We integrate the BearSSL library [85] into Graphene to provide the TLS connections between the Graphene clients and kernel servers, and to verify the timeserver's response. The library is well-suited to a kernel environment, as it avoids dynamic memory allocations, and has minimal dependencies on the underlying C library. For performance, we use BearSSL's implementations based on x86's AES-NI, PCL MUL, and SSE extensions, which helped to expose stack mis-alignment bugs in Graphene.

**File Locking System Calls** Graphene does not currently support file locking. As our memservers required this feature, we added an `advlock` (advisory lock) file system operation; applications invoke the operation through a reduced set of locking/unlocking flags to the `fcntl` system call.

### 4.2.3 NGINX Modifications

**Shared Memory Patch** NGINX uses shared memory to coordinate state among the worker processes that service HTTP(S) requests. On most systems, it uses `mmap` to create shared, anonymous mappings. NGINX encapsulates each mapping as a *named zone*. For allocating in shared memory, NGINX overlays a slab pool over the zone's shared memory.

To coordinate concurrent allocations and frees on the pool, as well as modifications to the user data structures allocated from the pool, each pool has an associated mutex. On systems with atomic operations, the mutex is implemented as a spinlock over a word of the shared memory, optionally falling back to a POSIX semaphore for long, blocking lock operations. On systems without atomic operations, the mutex is implemented as a lock file.

To have NGINX follow the semantics of our shared memory design, we create a small patch (~300 lines) that changes the creation of shared memory and the associated mutex. In particular, we implement shared memory by having `mmap` map a path obtained by concatenating the filesystem root with the zone name. To force the use of a lock file, we disable atomics. NGINX's lock file path is the name of the zone concatenated with a prefix that may be specified in the NGINX configuration file (`nginx.conf`), thus allowing us to easily have the lock file be the very same file that is mapped.

**Request Lifecycle** When NGINX operates as a caching server, it runs four processes by default: (1) a master process that initializes the server and responds to software signals, (2) a configurable number of worker processes that service HTTPS requests, (3) a cache manager, and (4) a cache loader.

Figure 4.2 shows the lifecycle of an NGINX worker process serving an HTTPS request, and the resultant RPCs to the enclaved kernel servers. Note that each

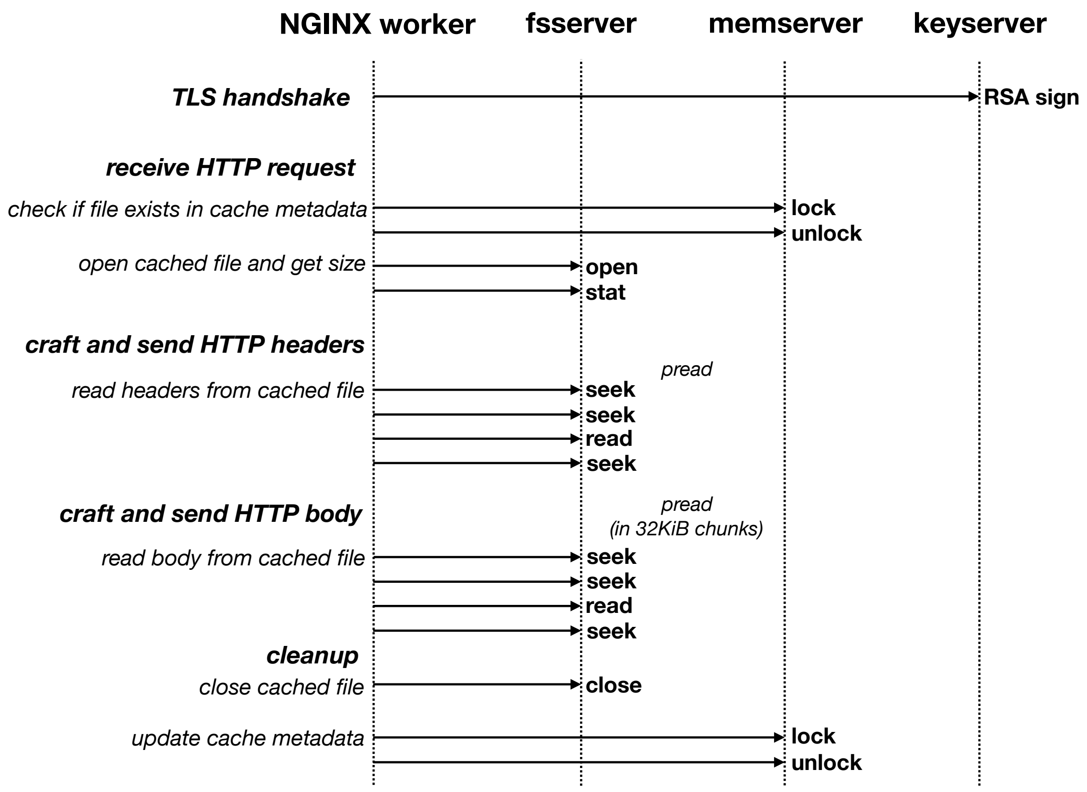


Figure 4.2: An NGINX worker servicing an HTTPS request for cached content, and the resultant kernel server RPCs.

request requires two critical sections involving the metadata. Also, NGINX reads the cached content using the `pread` system call, which Graphene's virtual file system (VFS) layer implements as a sequence of `seeks` and a `read` to the underlying filesystem.

## 4.3 Evaluation

We evaluate the performance of NGINX 1.14.1 running within a Phoenix Conclave. We seek to understand (1) the performance costs of the various aspects of the conclave design and implementation, (2) how performance scales with multi-tenancy, and (3) the performance impact of a WAF.

We perform our tests on the Intel NUC Skull Canyon NUC6i7KYK Kit with 6th generation Intel Core i7-6770HQ Processor (2.6 GHz) and 32 GiB of RAM. The processor consists of four hyperthreaded cores and has a 6 MiB cache.

We use ApacheBench to repeatedly fetch a file 10,000 times over non-persistent HTTPS connections (each request involves a new TCP and TLS handshake) from among 128 concurrent clients.[6] We run ApacheBench on a second NUC device connected to the conclave's NUC via a Gigabit Ethernet switch. For the benchmarks, the origin server is another instance of NGINX running on the conclave's NUC.

We examine three conclave configurations: (1) *Linux-keyless*: NGINX running on normal Linux and using a keyserver, (2) *Graphene-crypt*: NGINX running on Graphene and using a bd-crypt fsserver, sm-crypt for shared memory, and the keyserver, and (3) *Graphene-vericrypt*: NGINX running on Graphene and using a bd-vericrypt fsserver, sm-vericrypt for shared memory, and a keyserver. These correspond to a Keyless SSL analog, a conclave deployment for data confidentiality, and a conclave deployment for both data confidentiality and integrity, respectively. We compare these conclaves to the status quo of NGINX running on standard Linux (simply denoted as *Linux*). We omit using the timeserver.

For each benchmark that uses the nextfs fileserver, we use a 128 MiB disk image. As a baseline, we configure NGINX to use a small shared memory zone of 16 KiB to hold the web cache metadata (enough for  125 cache keys). §4.3.2 presents a sensitivity analysis on the size of the shared memory zone.

[6]That is, the command `ab -n 10000 -c 128`

### 4.3.1 Standard ocalls vs. exitless

To determine the optimal ocall method for our application, we first compare the performance of standard vs. exitless versions of Graphene-crypt. We present HTTPS throughput and latency results for each version as part of Figure 4.3.

Surprisingly, the exitless version performs worse across the board. Although both perform similarly with a single NGINX worker, the standard ocall version exhibits expected performance gains as new workers are added, whereas exitless generally worsens with additional workers. In a conclave environment, increased contention on the kernel servers, as well as contention among the SGX and RPC-queue threads, magnify the RPC latency overheads, which in turn causes exitless to exit the spinlock and make a futex ocall.

Based on these results, we use standard ocalls in all instances of Graphene (both on the Graphene-hosted NGINX processes, and the kernel servers) for the remainder of the macro-benchmarks.[7]

### 4.3.2 Single-Tenant

Figure 4.3 shows request latency and throughput results for the four configurations. Due to the RSA private key operation in the TLS handshake, Linux becomes CPU-bound at four workers (our test machine has four physical cores) and saturates the Ethernet link for tests with a 100 KiB payload and more than one NGINX worker. Linux-keyless shows that the concurrency of the keyserver levels off with

[7]§4.3.5 shows that exitless performs better than standard ocalls for low-latency calls, but degrades for high-latency calls.

| **Segment Size** | **16 KiB** | **100 KiB** | **1 MiB** | **10 MiB** |
|---:|---:|---:|---:|---:|
| **# Cache Keys** | 125 | 781 | 8,000 | 80,000 |
| **Throughput** | 437.76 | 320.36 | 133.25 | 9.71 |
| **Latency** | 292.40 | 399.54 | 960.58 | 13,184.09 |

Table 4.1: Effect of increasing the size of NGINX's shared memory segment for cache metadata. We use Graphene-crypt with one NGINX worker, and fetch a 1 KiB file. Throughput is the mean requests served per second; latency is the client-perceived latency (ms).

two workers, and thus that the two NGINX worker configuration of Linux-keyless is an upper-bound on the performance we can hope to achieve with the other conclave configurations. Linux with two or more workers beats all conclave configurations.

Table 4.1 shows a sensitivity analysis on the shared memory zone size for NGINX's cache metadata, using Graphene-crypt. Performance diminishes disproportionately faster than the increases in memory sizes, and request latency exceeds 1 sec past 1 MiB.

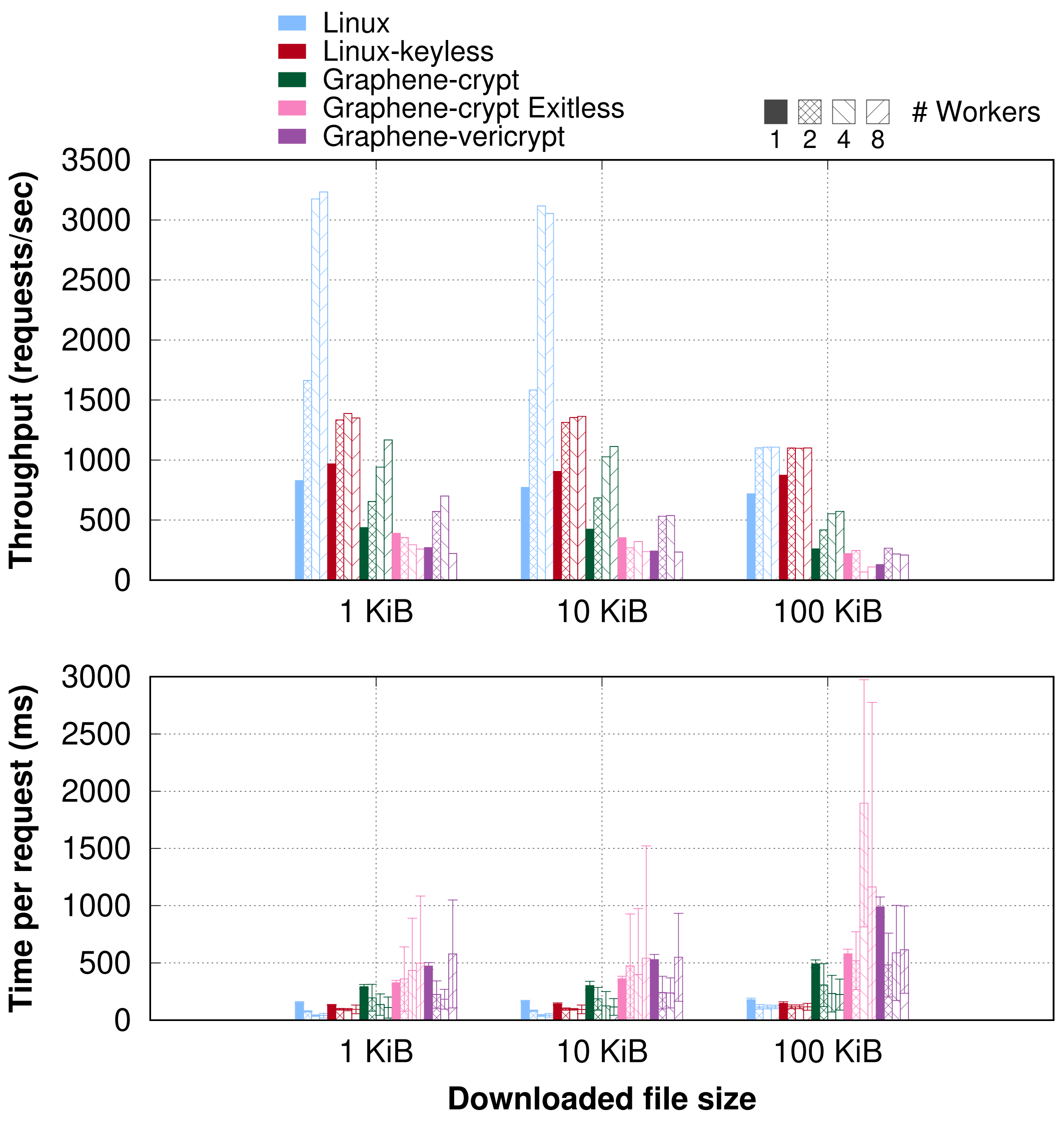


Figure 4.3: Throughput and latency for single-tenant configurations. The legend indicates the number of NGINX worker processes. We include the standard deviation of the latencies as error bars.

### 4.3.3 Scaling to Multi-tenants

We evaluate two approaches to multi-tenancy: (1) *shared nothing*, in which each customer runs their own conclave, including an enclaved instance of NGINX, and (2) *shared NGINX*, where each customer runs their own enclaved kernel servers, but share an enclaved version of NGINX: the NGINX configuration file multiplexes the customer resources. Specifically, the NGINX configuration file defines a virtual server for each customer; each virtual server's cache directory, shared memory zone for the cache metadata, and TLS private key point to separate instances of the fsserver, memserver, and keyserver, respectively. We compare these approaches to the status quo of running a single NGINX instance with a virtual server for each customer. We run each NGINX instance with four worker processes (in the shared nothing case, this means each tenant receives four workers processes; in the shared NGINX and Linux case, the tenants are multiplexed on four total workers). We direct ApacheBench tests concurrently against each tenant.

Figure 4.4 compares the mean latency and aggregate throughput of these three deployments, scaling the number of tenants from one up to six. After an initial dip at two tenants, Linux is able to increase throughput with modest increases to request latency; shared NGINX Graphene-crypt maintains a more-or-less constant overall throughput at the cost of increasing latencies, while the shared nothing configuration is unable to maintain throughput past two tenants.

For the conclave deployments, we also measure the number of SGX paging events using the kprobe-based technique from Weichbrodt et al. [86, 87]. For both the shared-nothing and shared-NGINX deployments of Graphene-crypt, these events remain under 10,000 up to four tenants; at six tenants, the shared NGINX deployment incurs on average 10,507 SGX paging events, whereas shared nothing incurs a staggering 4.35 million as the working sets of 48 enclaved processes contend for the limited 93 MiB of EPC memory.

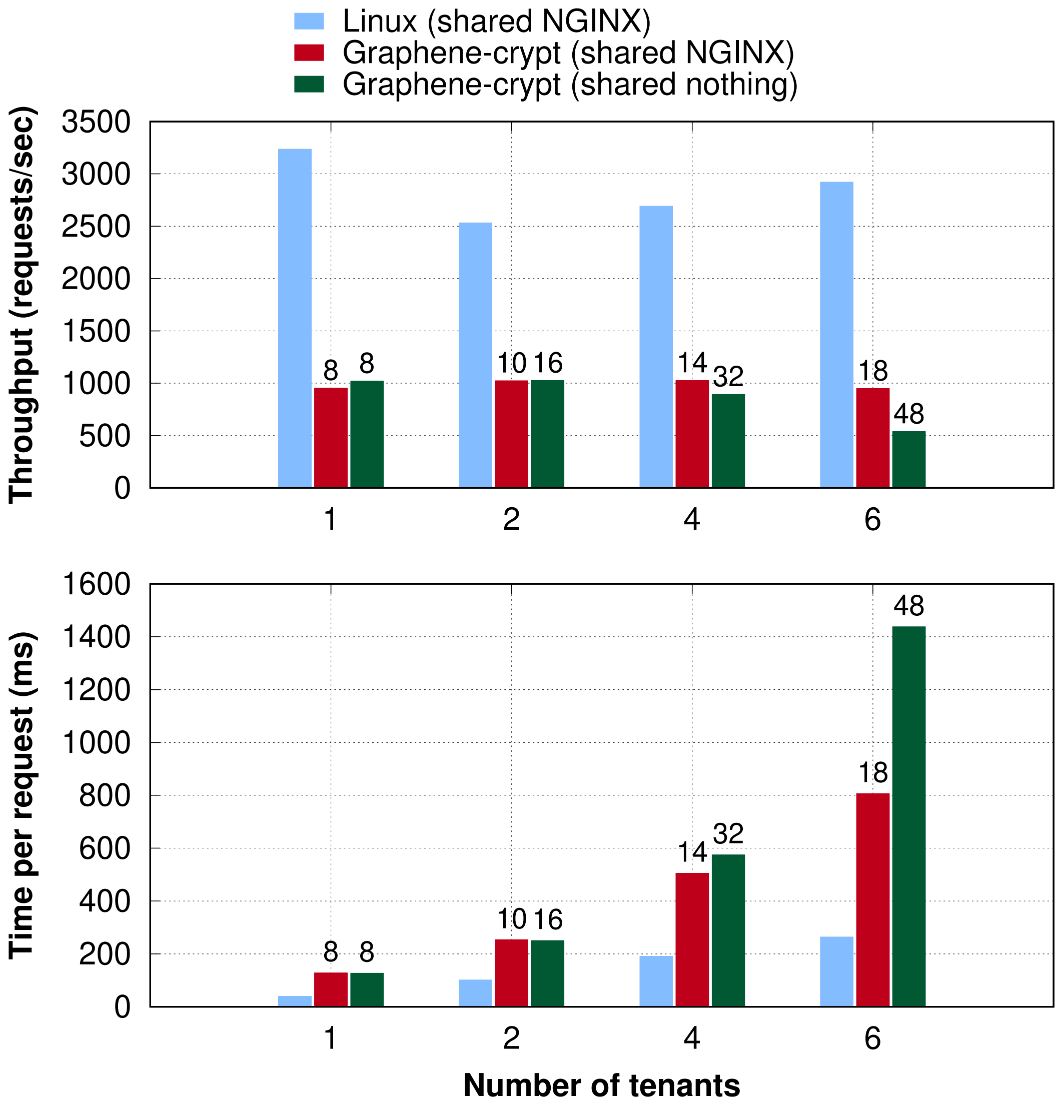


Figure 4.4: Multitenancy scaling. Throughputs are the aggregate throughput across all customers, and latencies are the mean latencies across customers. Above the bars, we indicate the number of enclaves in each configuration.

### 4.3.4 Web Application Firewall

Finally, we evaluate the cost of running the ModSecurity web application firewall (WAF) in tandem with NGINX. Each of our ModSecurity rules examines the request's query string for a unique, blacklisted substring. We increase the number of rules and observe the effect on the server's HTTPS request throughput and latency in Figure 4.5 for normal Linux and Graphene-crypt, both running as standalone, non-caching, servers. We see that just enabling ModSecurity results in a 5% decrease in throughput for Linux, and 16% decrease for Graphene-crypt. At 1000 rules, the relative costs for Linux and Graphene-crypt converge, as the throughput of each is 2/3 of that when ModSecurity is off, and latency is 1.5× slower. At 10,000 rules these relative costs increase substantially, to just 14% of the throughput and 7× the latency compared to when ModSecurity is disabled.

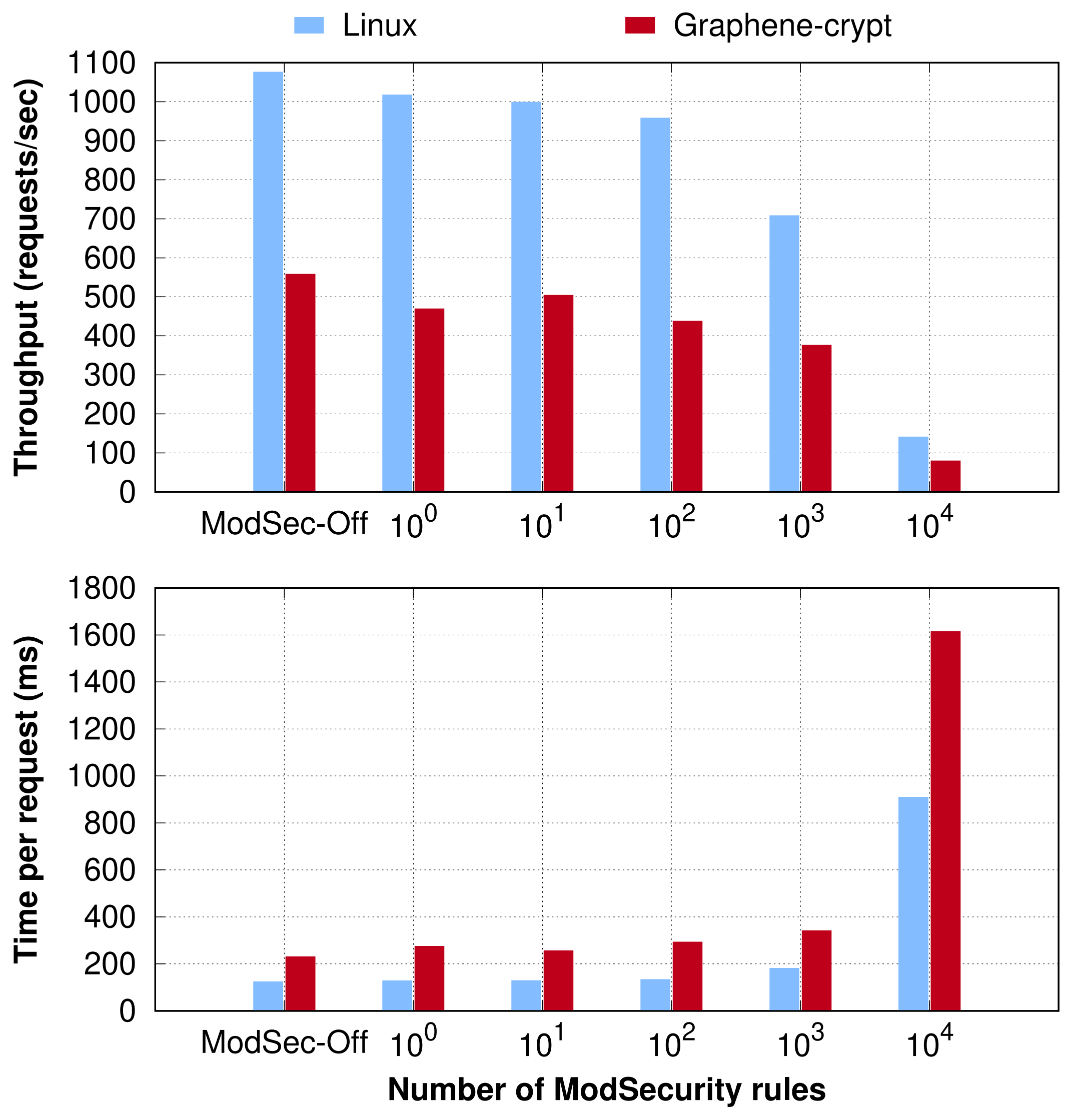


Figure 4.5: Effect of ModSecurity rule count on NGINX performance. NGINX runs with a single worker, and we fetch a 1 KiB file.

### 4.3.5 Micro-benchmarks

We now evaluate the various subcomponents of a Phoenix conclave to provide a more fine-grained explanation of our performance results. For each micro-benchmark, we compare the performance of the component running in three environments: outside an enclave (non-SGX), inside an enclave with normal system calls (SGX), and inside an enclave with exitless system calls (exitless). Each micro-benchmark tool runs on the same machine as the component we are testing.

#### 4.3.5.1 Remote Procedure Calls

To understand the cost of the RPC mechanism used by the kernel servers, absent from any additional server or client-specific processing, we design an experiment [8] where a client issues an RPC to download a payload 100,000 times, and compute the mean time for the RPC to complete. We vary the payload size from 0-bytes to 1 MiB.

Figure 4.6 shows that, in general, SGX incurs a much higher latency overhead than exitless but that this gap narrows as the payload size increases, and that at 1 MiB payloads, exitless actually performs worse than normal ocalls.

Higher payload sizes result in greater latencies for the underlying system call; if this latency exceeds the spinlock duration, the spinlock falls back to sleeping on the futex, effectively having spun in vain. For payload sizes of 0 through 100 KiB,

[8] For an apples-to-apples comparison between SGX and non-SGX environments, we benchmark at the application layer. This differs slightly from conclaves, where the kernel servers are also implemented at the application level, but the fsserver and memserver clients are subsystems of Graphene.

exitless falls back to the futex less than 30 times for both the server and client; in contrast, for the 1 MiB case, nearly every RPC uses the futex (on average, 91,285 times for the server, and 97,881 for the client).

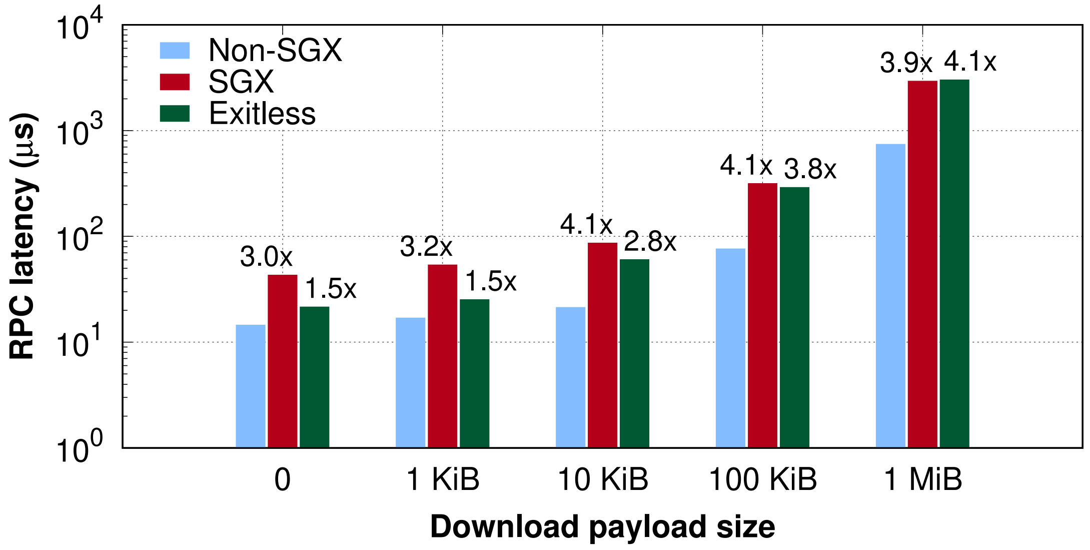


Figure 4.6: RPC latency versus payload size. The numbers above the bars are overheads compared to non-SGX.

#### 4.3.5.2 Kernel Servers

**fsserver** We use `fio` [88] to measure the performance of sequential reads to a 16 MiB file hosted on a nextfs server, over 10 seconds; each read transfers 4096 bytes of data. `fio` runs inside an enclave, uses exitless system calls, and invokes read operations from a single thread.

Figure 4.7 shows the read latencies for each variant of the filesystem. Compared to bd-std, bd-crypt adds relatively small overheads, whereas bd-vericrypt shows nearly an order of magnitude slow down due to the Merkle tree lookups, dependent on the size of the tree's in-enclave LRU cache.

Figure 4.8 shows the associated throughput. For comparison, the enclaved versions of bd-crypt and bd-vericrypt have 20$\times$ and 97$\times$ less throughput, respectively, than Linux's standard ext4 filesystem (954 MiB/s, on our test machine).

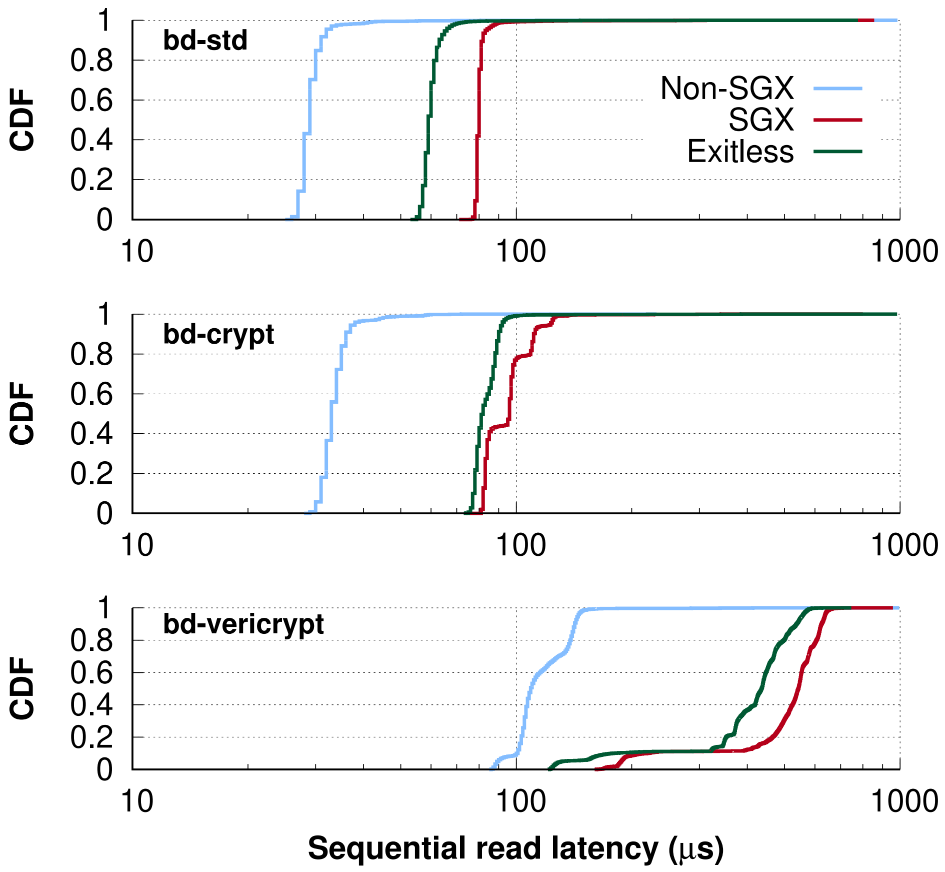


Figure 4.7: CDFs of read operation latency ($\mu$s) for a 10-second test that repeatedly reads 4096-bytes from a nextfs server, for each block device implementation.

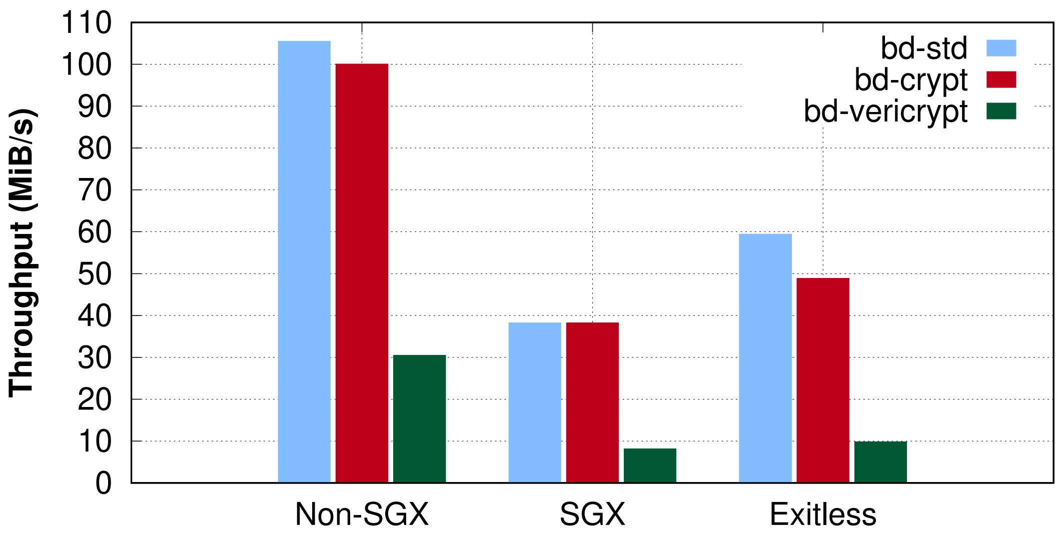


Figure 4.8: Total throughput for a 10-second test that repeatedly reads 4096-bytes from a nextfs server, for each block device implementation.

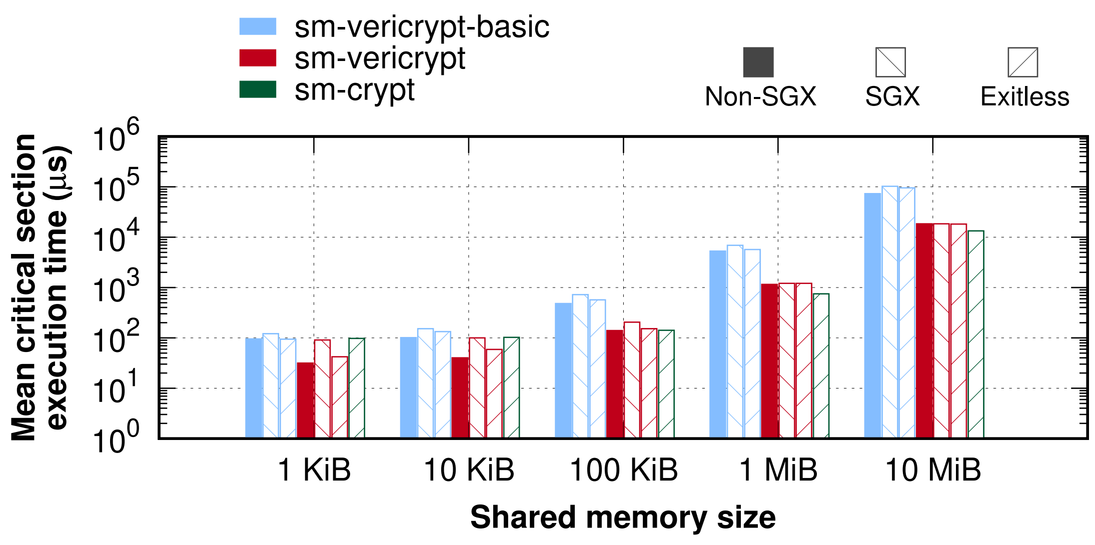


Figure 4.9: Mean wall clock time ($\mu$s) to process a critical section.

**memserver** Figure 4.9 shows the mean time for a process to evaluate a critical section (a lock and unlock operation pair) over shared memory provided by the memserver, based on 10,000 runs. We also vary the size of the memory segment to observe its effect on the run time.

We make two observations. First, since `mmap` allocates in page sizes (4096-bytes), the measurements for a 1 KiB and 10 KiB shared memory region are nearly identical; otherwise, the execution times scale linearly in accordance with the memory size. Second, starting at 100 KiB, the sm-vericrypt and sm-crypt implementations, which represent the canonical memory replica as an encrypted host file, show an order-of-magnitude improvement over sm-vericrypt-basic, which uses EPC memory to store the canonical replica and transfers the replica over interprocess communication.

| **OpenSSL** | **keyserver** | | |
|---|---|---|---|
| **non-SGX** | **non-SGX** | **SGX** | **exitless** |
| 860.92 | 933.42 | 965.32 | 932.60 |
| | (1.08×) | (1.12×) | (1.08×) |

Table 4.2: Mean wall clock time ($\mu$s) to compute an RSA-2048 signature using default OpenSSL (left) and the keyserver. The last row is overhead compared to OpenSSL.

**keyserver** To evaluate the keyserver's performance, we use the `openssl speed` command to measure the time to compute an RSA-2048 signature. For all tests, the `openssl speed` command runs outside of an enclave, and measures the number of signatures achieved in 10 seconds.

We present the results in Table 4.2. The keyserver itself uses OpenSSL's default RSA implementation; compared to the RPC micro-benchmarks in Figure 4.6, we again see that the raw time overheads are consistent with the RPC latencies.

| host time | | | timeserver | |
|---|---|---|---|---|
| non-SGX | SGX | exitless | SGX | exitless |
| 0.026 | 3.467 | 0.757 | 1,175.622 | 1,375.607 |
| | (133×) | (29×) | (45,216×) | (52,908×) |

Table 4.3: Mean wall clock time ($\mu$s) to execute `gettimeofday`. Left: retrieving time from host; Right: retrieving from (unenclaved) timeserver. The SGX and exitless designations refer to the application's environment. The last row is overhead compared to non-SGX.

**timeserver** We evaluate the timeserver by measuring the elapsed time to invoke `gettimeofday` one million times in a tight loop, and then compute the mean for a single invocation.

In Table 4.3, we list the mean time for an invocation of `gettimeofday` in Linux (non-SGX), and in Graphene, using both the host time and the timeserver. Note that non-SGX calls to `gettimeofday` are nearly free due to vDSO.[9]

The difference between the exitless and normal ocalls is roughly the round-trip cost of exiting and returning to an enclave; this is consistent with prior work [83,84, 86] that puts this cost at 8000 cycles (3.077 $\mu$s on our test machine). The timeserver cost is dominated by the signature computation; exitless calls to the timeserver actually hurt performance, as, due to the signature latency, the Graphene client fails to receive a response during the spinlock, and falls back to the more expensive futex sleep operation for every RPC.

[9]A system call implementation that uses a shared memory mapping between the kernel and application, rather than a user-to-kernel context switch.

## 4.4 Conclusion

We have presented Phoenix, the first "keyless CDN" that supports the quintessential features of today's CDNs. To support multi-process, multi-tenant, legacy applications, we introduced a new architectural primitive that we call conclaves (containers of enclaves). With an implementation and evaluation on Intel SGX hardware, we showed that conclaves scale to support multi-tenant deployments with modest overhead.

**Optimizations and Recommendations** While Phoenix is able to achieve surprisingly good performance, further potential optimizations remain, including of SGX. The multi-tenancy results in Figure 4.4 show that EPC size limits become a constraint in environments with multiple enclaved applications. Conclaves alleviate this to some extent, as the kernel servers may be run on devices separate from the application, but splitting the application itself (e.g., the NGINX workers) across machines is less tractable. Future versions of SGX should therefore investigate ways of increasing the EPC size. The cache size sensitivity results in Table 4.1 show that distributed shared memory is a challenging performance problem. Future versions of SGX should investigate features for mapping EPC pages among multiple enclaves.

While prior work has treated exitless calls as a panacea, §4.3.5 shows that they should be a per-system call policy to reflect the application's workload.

Of course, Phoenix is by no means a drop-in replacement for today's CDNs, who have specially optimized web servers and support a much wider range of features, such as video transcoding and image optimization. Rather, our results should be viewed as a proof of concept and a glimmer of hope: *it is not necessary for CDNs to have direct access to their customers' keys* to achieve performance or apply WAFs. We view Phoenix—and especially conclaves—as a first step towards this vision.

# Chapter 5: SecureMigration

In this chapter, I present my work on the design and implementation of SecureMigration.

We[1] introduce a new approach to adapting old software to new trust models, without having to rewrite (or even read) an application's code, and without trusted hardware. Our central insight is that supporting new policies that separate privileged data access across multiple principals does not require changes to *what* is executed, only *where* it is executed. For instance, in Keyless SSL [11], the same exact cryptographic operations take place as in traditional, monolithic web servers, only at a server run by the customer who owns the secret key.

In our system, SecureMigration, users (not the application developers) specify which resources (files, network accesses, etc.) should be accessible by which hosts. Our system monitors access to all system calls and, when the program tries to access a resource only available to a different machine, it migrates the process to that machine. Of course, sensitive information may "leak" into memory; we employ taint tracking, treating each sensitive resource as a source of taint, and binding all tainted memory bytes to the machines who can access the corresponding taint

[1]This work involved collaborations with James Larisch, Deepak Garg, and Dave Levin.

source.

We abstract from the process the precise host on which it is executing such that the process can transparently migrate to and execute on multiple physical hosts during its lifetime. From the process's viewpoint, it executes on a single, logical host. In reality, the process migrates to different hosts when it needs to access resources that are bound to that host.

We wish to be able to support a wide range of policies that restrict resource (data, network resources, etc.) access to specific sets of machines ("domains"). Moreover, we wish for these policies to remain *high-level* in the sense that they should operate over resources users can understand (e.g., specific files) rather than require deep knowledge about the software itself.

We have implemented this abstraction, and use it to apply distributed trust models to the NGINX [71] web server that it was never designed to support. For instance, we show that we can achieve Keyless SSL essentially "for free" by simply specifying that the CDN's host should handle all global network connections and the customer's host should be the sole owner of the secret key.

The rest of this chapter is organized as follows: I describe a design that achieves the abstraction in §5.1 and its implementation in §5.2. In §5.3, I evaluate our implementation by applying it to legacy NGINX, a popular web server, under multiple distinct deployments that NGINX was not originally designed to support. I conclude in §5.4.

## 5.1 Design

### 5.1.1 Example Use Case

To exemplify the above goals, we return to our running example of a web server being run at a semi-trusted CDN. Rather than resort to extensive re-designs [11] or trusted hardware [7], we envision a use case in which the CDN and the website owner agree on the following policy:

- The CDN handles all incoming network connections.
- The CDN also maintains proprietary access control and firewall rules.
- The website owner is the only one allowed to access its secret key.

Working with this policy, the CDN begins running a *legacy* web server, configured to use the CDN's access control policies and the website's secret key (all specified by filenames in the web server's configuration). During startup, the web server eventually seeks to access the file containing the secret key. Our system mediates all file accesses (in fact, all system calls) and checks them against the policy to see if it can be done on the current active machine (at the CDN). Realizing it cannot, our system checkpoints the program and migrates its execution to the website owner's machine. Once there, it resumes execution, accesses the local file, and treats it as a source of taint. Eventually, the process migrates back to the CDN, transferring all memory *except* those tainted (and thus pinned) at the website owner's machine. To help prevent taint from causing migrations to thrash between machines, or the process to deadlock altogether, our system allows for the declassification of taint.

Later, while handling an incoming TLS handshake at the CDN's machine, the web server seeks to access tainted memory containing the secret key. Our system monitors all memory accesses, realizes the required memory is not allowed to be local, and thus checkpoints, migrates, and restores the process at the website owner's machine. From there, it performs the required cryptographic operation and migrates back to the CDN's machine to send its response to the client.

This high-level sketch of a use case shows the potential of secure process migration: by merely expressing a high-level policy over specific network and file resources, it allows users to change the security properties of *unmodified* applications in ways the original designers did not envision. In the following sections, we will show that we are able to achieve this in practice with unmodified NGINX. Although it comes with significant performance overheads and many opportunities for future optimization, we believe it represents an important first step towards being able to significantly alter programs' security goals.

### 5.1.2 Virtualization Layer

The design of SecureMigration takes the form of a distributed, application virtual machine: the process runs atop a virtualization layer that interposes on machine instructions, memory accesses, and system calls, with this layer being replicated at each *trust domain*, which for our purposes is a physical machine owned and operated by a distinct principal. The process itself has a single address space and single, logical root filesystem; the virtualization layer at each domain ensures that the private resources are physically inaccessible to the other domains before migrating the process to a peer domain.

The virtualization layer relies on dynamic binary instrumentation (DBI)—a technique for monitoring and possibly altering the execution of a process at instruction-level granularity, without requiring access to its source code or modification to its runtime. We can think of a DBI system as an application virtual machine that interprets machine code while offering instrumentation capabilities to analyze and alter the process's architectural state. The components of a DBI system are laid out in the same address space where the program execution takes place, with the program's execution interleaved with the analysis. Many DBI systems additionally take steps to isolate the analysis from the application, such as by switching to an alternate stack before executing the analysis.

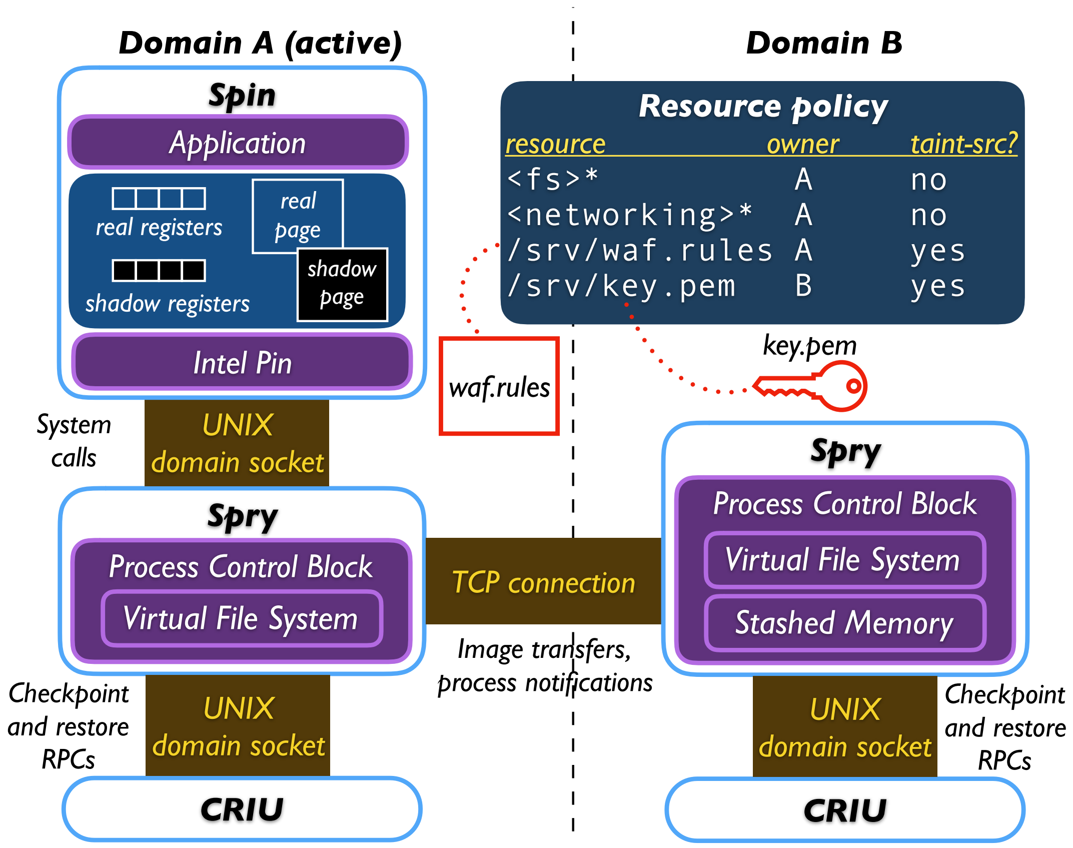
### 5.1.3 Taint tracking

In SecureMigration, the DBI system tracks the flow of information within a domain such that if the process executes a computation with a private resource, then the system marks as private any other resources of the domain that were involved in the computation. This technique is commonly called *information flow tracking*, or, more colloquially, *taint tracking*. For instance, if a domain marks a file as private, and the process reads that file, then SecureMigration marks the memory buffer into which that file is read as private to the domain; if that buffer is then copied to a destination buffer elsewhere in memory, then SecureMigration likewise marks the destination buffer as private. Taint tracking is transparent to the application process, as the application observes the same addresses and same values as it would in an uninstrumented execution. SecureMigration takes as input a policy file that lists the initial private resources (such as paths and network addresses), as well as the domain that can access the resource.

To make progress, SecureMigration also needs a mechanism for untainting data. A classic example is that, in a cryptographic signing operation, the signing key may be tainted (indicating that it belongs to a specific domain), but the key's taint should not propagate to the signature, which is public. As such, SecureMigration applies taint tracking at both the instruction-level, and, where needed to override the instruction-level behavior, at the function-level. This implies that SecureMigration can locate the symbols for such functions.

### 5.1.4 Migration

When SecureMigration detects that the *active domain*—the domain where the process is currently executing—tries to access a resource (file, memory, networking) that is tainted to a peer domain, SecureMigration checkpoints and stops the process. Checkpointing entails serializing the process's state, such as the values of its registers and its memory address space, as well as kernel-maintained state, such as the file descriptor table and UNIX credentials. As the DBI system is co-resident with the process, the DBI system is likewise included in the checkpoint. SecureMigration



| resource | owner | taint-src? |
|---|---|---|
| `<fs>*` | A | no |
| `<networking>*` | A | no |
| `/srv/waf.rules` | A | yes |
| `/srv/key.pem` | B | yes |

Figure 5.1: High-level implementation of SecureMigration. Each domain runs an instance of Spry and CRIU. CRIU dumps the application process (which includes the co-resident Spin and Intel Pin framework) and Spry transfers the dump images to the peer domain, which again invokes CRIU to reconstitute the process.

transfers the checkpoint images to the peer domain, which reconstitutes the process from the images.

## 5.2 Implementation

Our implementation consists of three components: (1) a DBI tool, Spin, that "hosts" and executes the original application, (2) a local proxy, Spry, that coordinates migration and services system calls on Spin's behalf, and (3) the CRIU (Checkpoint and Restore In Userspace) [89] Linux utility that Spry employs for taking a snapshot of a running process and later restoring the process from that snapshot. In SecureMigration, each domain is a physical machine, and each domain runs an instance of Spry, which in turn may service multiple instances of Spin. As CRIU requires root privileges, each domain runs CRIU as a daemonized service, hence separating privileges within SecureMigration. The resultant distributed system, diagrammed in Figure 5.1, is thus comprised of these three components, as well as the communication protocols between Spin and Spry, between Spry and CRIU, and between the peer instances of Spry. We have made our code publicly available so that others can continue to build off this work.[2]

Before we describe each part of the system, we note that two pragmatic decisions largely determine our implementation. Our first decision is to use CRIU unmodified, rather than adapt it to our use case. The main challenge with stock CRIU, however, is that it requires a process's resources, such as the files and network addresses, on the restoring machine to match those of the machine on which CRIU checkpointed the process, and will fail to restore if they do not. The whole point of SecureMigration, of course, is to partition a process's resources across multiple machines. This results in our decision to move system call invocation from the process to Spry, with Spin interposing on each system call: if the system call references a resource (as by name or file descriptor), Spin issues an RPC to Spry to invoke the system call on the process's behalf, and return the results; otherwise, Spin passes the system call through to the kernel.

System call forwarding from Spin to Spry allows Spry to maintain all kernel-

[2]Our code may be found at `https://github.com/smherwig/securemigration`.

level file descriptors, and to return to Spin a virtual descriptor (Spry maintains for the process the mapping from virtual to real descriptor). This achieves the invariant that, other than the local socket connection between Spin and Spry (and, optionally, descriptors attached to the terminal), the process does not reference any kernel-level file descriptors, thereby eschewing potential conflicts during CRIU's filesystem and networking consistency checks.

Our second, related decision, is to use a simple model for distributed resource management: namely, file resources are pinned to their owning domain; files do not migrate between domains and are not replicated across domains. In contrast to files, we transfer the complete memory image of a process when migrating. A different implementation could alter these choices at the expense of a more complex implementation for Spry, and of modifications to CRIU to support a distributed shared memory protocol with respect to the checkpointed memory images.

### 5.2.1 Spin

We build our DBI tool, Spin, using the Intel Pin framework [90]. Intel Pin provides a rich API for instrumenting instructions, interposing on system calls, and hooking application-level symbols. Spin has two primary functions: (1) perform taint tracking of private data within the process's address space, thereby tracking a domain's memory ownership and, (2) forward many system calls (including all calls that reference a resource, as by pathname, socket address, or file descriptor) to the local Spry daemon.

**Taint tracking.** Spin's taint tracking logic is inspired by libdft [91]. Like libdft, Spin tracks data flow by instrumenting instructions that may propagate or clear taint, and extends libdft to support 64-bit mode, as well as MMX, SSE, and SSE2 instructions. To reduce the development burden with respect to taint tracking (Intel x86_64 has over 1500 mnemonics, each of which may comprise dozens of machine instructions based on the operand types), Spin instruments the `cpuid` instruction and replaces it with an emulated version that responds to processor feature queries by returning that the SSE3, SSE4, AVX, AES, and PCLMUL extensions are unavailable. Although this does not guarantee that such instructions are never executed, in practice, binaries first query the `cpuid` before invoking a function that uses these extensions. We insert the taint analysis before the execution of each instrumented instruction due to various restrictions that Intel Pin places on post-execution analysis. Like libdft, and as with most taint tracking tools, Spin does not handle control flow taint, does not assume the dereferenced memory for a tainted pointer is tainted, and does not track the taint of control registers, such as `RFLAGS`.

Spin tracks taint at byte-granularity (tracking larger units, e.g., registers or pages, makes it impossible to detect when that larger unit transitions from tainted to untainted). Taint values are likewise one-byte in size, and store the identity value for the domain that claims the corresponding byte. A byte either belongs to one domain or is shared among all domains (indicated by taint value 0). Identity values are the powers of two representable by a byte, for a maximum of eight domains—simply OR-ing a set of shadow bytes and testing against 0 or a power of two is enough to determine whether multiple domains have taint on the set.

Spin records the taint values for each byte of the CPU's registers in a per-thread virtual CPU that has shadow registers for the general purpose registers, MMX registers, and XMM registers. For main memory, Spin records taint in a conventional four-level page-table structure (a radix tree) that allocates a shadow page for each real page. To keep the shadow page tables in-sync with the application's true pages, Spin hooks the return of the `mmap`, `munmap`, and `brk` system calls, and adds or removes pages from shadow memory accordingly. As some mappings (as for the stack) exist prior to Spin transferring control to the application, Spin also pre-populates the shadow page tables based on the process's mapping information as retrieved from `/proc`. For diagnostic purposes, Spin also maintains the virtual memory area (VMA) metadata corresponding to the shadowed pages.

**Register and memory ownership.** In Spin, ownership of register memory and main memory is byte-level, and thus mimics the taint tracking. This is in contrast to alternative approaches that use a larger granularity of ownership, such as granting a domain ownership of all register memory if any register byte is tainted, or an entire page of memory whenever the domain has tainted any byte on that page. In general, expressing ownership at larger granularities potentially improves performance by allowing for less costly ownership checks, but at the expense of a greater risk for ownership deadlock and false sharing. Our early experiences running test applications on SecureMigration with register-set and page-level granularity revealed that both phenomena were an issue.

For instance, with a register-set ownership, we found that it was not uncommon for a process to need to migrate while having one or more registers tainted to the active domain. Likewise, for page-level ownership, a frequent problem we encountered is that domain $B$ would taint page $i$, and then migrate to $A$ to make a system call; however, one of the arguments for the system call resided untainted on page $i$—this is a form of deadlock. Due to such experiences, Spin applies fine-grained memory ownership.

**Checking memory access.** Since Spin expresses ownership of the registers and main memory at byte-level, Spin's instrumentation inspects each instruction to determine the operand count, the registers used, and for each operand, its type (memory, register, immediate, or address generator—a special operand type used by the `lea` instruction), size, and access mode (read, write, or read-write). Additionally, for memory and address generator operands, Spin records any registers used as a base or index (in `mov qword ptr [rsi+rax*8], rdx`, the registers `rsi` and `rax`, respectively)—such registers require read-access.

Spin maintains a tree of all unique instructions encountered in the process, indexed by the instruction's bytecode, and where the node value contains the aforementioned details regarding the instruction and its operands. During instrumentation, Spin first searches the tree for the instruction; if not found, Spin inspects the instruction and adds a new node to the tree. Spin then inserts a call to an access-check analysis routine before the real instruction (and before any taint analysis routines) for any instruction that reads from a register or memory. Additionally, Spin checks whether the instruction is an idiom for clearing a register (such as `xor` `rax, rax`) and does not insert an access check for such instructions.

The access-check routine queries the taint for every operand that is read or read-write (for a memory operand, the access routines receive as an argument the memory's virtual address). The access routines do not query the taint for operands that are strictly written, since the instruction will overwrite the real value of the operand, and our corresponding taint analysis will either clear the taint or set it to a new domain. If any byte of a readable operand is tainted to a peer, then the process must migrate; the instruction—and Spin's corresponding analysis routines—are restarted on the destination domain.

The access-checks also detect taint deadlock scenarios: specifically, a single operand with bytes tainted to two domains, or distinct readable operands tainted to different domains. If the access-check detects taint deadlock, then Spin aborts the program and logs a summary of all tainted registers and pages.

**Clearing abandoned taint.** Spin uses two techniques for clearing abandoned taint in main memory—that is, taint on a byte that the process will not again read before a write. While these techniques are not strictly necessary given our ownership semantics, they facilitate measuring the extent of the address space that is truly tainted.

The first technique removes abandoned taint from the stack. A naïve approach would simply instrument `ret` to clear (that is, zero both the shadow and real memory) from the current value of the stack pointer, `RSP`, down to the bottom of the stack. However, as the stack VMA may be large (on our system, it is initialized to 33 pages), and mostly unused, and since `ret` is a common instruction; such an approach is wasteful. Thus, Spin instead tracks a conservative measure of the lowest value of `RSP` by instrumenting three instructions commonly used to decrement the stack pointer: `push`, `sub`, and `lea`. Spin's instrumentation for `ret` retrieves the current value of `RSP`, and instead clears from `RSP` down to the lowest-tracked `RSP` value. Spin then sets the lowest-tracked value to $\texttt{RSP} - 8$ (to account for `ret`'s implicit `pop` of the return address residing on the stack into the program counter, `RIP`).

Spin's second technique removes abandoned taint from the heap by shadowing libc's memory allocator. To that end, Spin hooks the call and return of the `malloc`-`/calloc/free/realloc` family of C library functions, and maintains a tree, where each node is an extant allocation (represented as a tuple of the allocation's starting address and size). On `free` (and some calls to `realloc` that serve to free memory), Spin searches the tree by the allocation's starting address, and, if found, zeroes the real and shadow memory for that allocation. Additionally, as Spin already hooks the `brk` system call for the purpose of determining the size and location of the heap VMA, Spin also detects when `brk` shrinks the heap, and likewise clears the trimmed portion.

To remove abandoned taint from the registers, Spin further instruments `ret` to zero the shadow and real registers for any registers that are, as per the SysV AMD64 ABI [92], not preserved across a function call (the so-called *caller-saved* registers) and not designated as registers that can hold the function's return value; this is the vast majority of registers. However, to safely apply this technique, the code must not have been compiled with GCC's `-fipa-ra` option (interprocedural analysis for register allocation—enabled as part of the `-O2` optimization level), which allows the compiler to disobey ABI conventions to optimize register allocation. Thus, Spin has a configuration option where a user can list the images within the process that were compiled without this option; the instrumentation for `ret` inspects the return address on the stack; if the return address (and thus the caller) is within the memory spanned for such a loaded image, the instrumentation can safely clear the clobbered, non-return value registers.

**Function-level taint analysis.** Aside from techniques for removing abandoned taint, there are legitimate cases where taint analysis at the instruction-level does not obey the desired operational taint semantics, and for which Spin must explicitly intercede. For example, consider a case where the private data is a private TLS key, and the key is encoded as PEM—that is, Base64 encoded. A common implementation for a Base64 decoder is to use a lookup table indexed by the Base64 ASCII character, and which evaluates to the corresponding binary. Within Spin, the PEM file is a taint source and the ASCII character is therefore tainted; moreover, the pointer represented by the lookup table base and ASCII character index is likewise tainted; however, the lookup value (the dereferenced memory) is not tainted. In other words, decoding the PEM file launders the taint such that the in-memory, binary representation of the TLS key is not tainted; this is clearly undesired.

The reverse situation is also possible. For instance, assume that the TLS key is properly tainted and that the process uses the key to produce a signature; the signature is public and should therefore be untainted, but the instruction-level taint analysis may very well taint the signature when computing its value with the private key.

To address this, Spin includes a hardcoded list of function symbols within commonly used libraries (for our prototype this includes only libc and OpenSSL) which require explicit tainting or untainting of their outputs. Spin includes wrapper implementations of these functions which inspect the taint of the input arguments, call the original function, and then explicitly set or clear taint on the output values. Spin receives notification from Intel Pin whenever an image from the list of libraries—for which it needs to wrap functions—is loaded.

**System call interposition.** Recall that CRIU requires that the files and network interfaces of the restoring machine match those of the checkpointing machine; otherwise restoration fails. Our approach is therefore to move the file descriptors outside of the checkpointed process, and into the Spry proxy; the process deals only with the virtual descriptors that Spry returns, and forwards all calls that reference a pathname, network address, or file descriptor to Spry so that Spry can invoke the system call and perform the virtual-to-real file descriptor translation.

To that end, Spin hooks all system calls. For each system call, Spin either passes the call through to the local kernel without modification, or, for system calls that reference a file or network resource, forwards the call to Spry. If the active domain owns the resource, Spry issues the call locally on behalf of the application and returns the results; otherwise Spry migrates the process to the owning domain. Spin also performs taint analysis for all system call results by tainting a read buffer that Spry indicates is from a taint source (proxied system calls), or by removing taint from an untainted output buffer (proxied and non-proxied system calls).

Spin connects to Spry over a UNIX-domain socket. On connection, Spry queries the application's credentials (PID, UID, and GID) via the `SO_PEERCRED` socket option, and then uses these credentials to retrieve additional process state, such as the process's umask and working directory, from `/proc`. As such, Spry has sufficient information to issue the system calls in the context of the process making the request.

### 5.2.2 Spry

Each domain runs an instance of the Spry daemon. Spry's primary responsibilities are to handle system invocation on behalf of the application processes, and to interact with the local CRIU service and the peer Spry instances for the purpose of migrating a process.

**Resource policy.** Each instance of Spry takes as input a configuration file that specifies the initial partitioning of the file and network resources across the domains—the domains must agree on this initial division. For file pathnames, the configuration simply lists the path (allowing restricted globbing patterns), the owning domain, and a set of flags that indicate whether the file is a taint source, and the action to take when a peer domain attempts to access the file. SecureMigration only implements one action—migrating the process to the owning domain and restarting the system call upon process restoration—but one could imagine additional, optimized actions such as remote system calls to the owner peer.

For networking, the resource policy specifies the IP addresses to which each domain may `bind` or `connect`, as well as a similar set of flags that indicate whether data received from the peer end should be marked as tainted, and the action to take when a peer domain attempts to perform a system call on the resultant socket descriptor. For some connections (for instance, those to a localhost DNS resolver) it may be desirable to allow any domain to claim the resource. Thus, the configuration allows a special owner of `OWNER_ON_CREATION`.

**Exit notification.** On each domain that the process has executed, that domain's Spry maintains a process control block (PCB) for that process. The PCB references various process resources, such as the process's file descriptors that are opened on that domain, as well as the stashed private memory. Thus, when the process terminates, the active domain's Spry must notify the peer Sprys of the termination so that they can release their respective resources. In our closed-world system, where all of the domains are specified at system startup, we accomplish the distributed cleanup by simply having the active domain broadcast to all peers that the process has terminated. From an implementation viewpoint, Spin hooks the *exit* system call family to notify Spry that the application is terminating.

**System calls referencing multiple resource.** The vast majority of system calls reference a single resource, and thus, when Spry services a system call RPC request from Spin, Spry need only determine the owner of that resource. Nevertheless, a small set of commonly used system calls (`select`, `poll`, `epoll`[3]) act upon a set of

[3]`epoll` is slightly different in that the set of resources is specified via a helper system call,

resources.

For these multiplexed system calls, if the active domain owns all fd arguments, then Spry services the call normally. If all fd arguments have the same owner, which is not the active domain, then the process migrates to the owning domain. If the fds have a mix owners, then the active domain emulates the system call by sending a polling message to each peer Spry for the fds that the peer owns. (We note that Spry's implementation of this emulation is currently incomplete and is not used in any of our evaluations.)

### 5.2.3 Migration

A process migrates on one of two conditions: (1) Spin detects that the application attempted to access another domain's private memory or registers, or (2) Spry detects that the application attempted a system call with an argument that references a resource that belongs to another domain. Each type of access fault requires a similar set of actions. At a high-level, Spin and Spry cooperatively cloak the domain's private memory, Spin and Spry close their connection to one another, and Spry invokes CRIU to checkpoint and then kill the local copy of the application. Note that Spin, which is co-resident in the process's address space, is also dumped as part of the checkpoint, and reconstituted on restore. The source instance of Spry must then transfer the checkpoint images to the destination domain's Spry, and the destination's Spry invoke CRIU to restore the images.

**Migration conditions.** Spin's access-check analysis routines detect if the active

`epoll_ctl`, but the general idea remains the same.

domain is trying to read peer memory. On such a condition, the access-check routine arranges for the instruction to be restarted, and then commences the migration protocol with Spry.

Similarly, during a system call, Spry checks the system call arguments against the resource policy. For system calls that explicitly reference a name (such as `open` or `bind`), Spry can directly match the name again each resource policy rule. For file descriptors, Spry queries its shadowed virtual file system to resolve the descriptor to its name (that is, to a pathname or socket address), and then matches the name against the resource policy. If the match indicates that a peer owns the resource, then Spry returns a special error value of `EMIGRATE`, along with the owning domain's ID; on receiving `EMIGRATE`, Spin arranges for the system call to be restarted, and then commences the migration protocol with Spry.

**Spin-to-Spry migration protocol.** Regardless of whether a memory access or system call causes migration, Spin always initiates the migration by making a `migration_start` request to Spry that includes the ID of the domain whose resource the application is trying to access. This allows Spry to verify that the peer domain exists and is connected, and to update the state of the process. Next, Spin cloaks any registers or main memory that the active domain owns. Cloaking entails stashing the real values with Spry, and then overwriting the real values with junk. In our implementation, for every page on which the active domain has taint, Spin stashes with Spry the real page and the corresponding shadow page (the shadow page serves as a mask for the bytes that the domain actually owns). Similarly, Spin stashes the real contents of all general purpose, MMX, and XMM registers, along with their corresponding virtual CPU shadow registers. After Spin has finished swapping-out the sensitive memory, it sends Spry a `migration_end` message to indicate that it is now ready to be checkpointed.

There are two important points with respect to the checkpoint. First, CRIU will not checkpoint a single end of an established local IPC connection, and thus at least one side (Spin or Spry) must close their end of the connection prior to the checkpoint. Second, to prevent race conditions, the checkpoint itself must logically occur at a stable, defined point in the process's execution.

To that end, the pre-checkpoint protocol proceeds as follows: Spin closes its connection with Spry and then invokes `nanosleep` to sleep for a long period. When Spry receives the `migration_end` message, it closes its connection with Spin, ensures that the process is in a sleep state (as by inspecting `/proc`), and then invokes CRIU `dump`. When CRIU `dump` seizes the process with `ptrace`, CRIU sends the process a STOP signal, and dumps the process at its current state. Per POSIX specification, `nanosleep` returns early when interrupted by a signal (here the STOP signal); thus on restore, execution commences with the `nanosleep` call returning because of the interrupt.

With execution restored on the destination domain, Spin next connects to the destination's instance of Spry, and exchange a series of messages to swap-in any private data that the tool had previously stashed with that instance of Spry. Since execution on a previous domain could have cleared or overwritten the now active domain's taint, swapping-in entails ANDing the stashed shadow memory with the current shadow memory; any resultant bytes that are the active domain's ID are still tainted to the active domain, and Spin overwrites these bytes with the stashed real memory. At this point, Spin returns execution to application at the restarted instruction.

**Spry-to-Spry migration protocol.** After dumping the process, the source and destination Spry engage in a small protocol to transfer the checkpoint images. The source first makes a `migration_request`, which includes the PID of the process; this is followed by any state that the source maintains for the process—namely the shadowed virtual filesystem, including the virtual-to-real file descriptor mappings. If the process has never before executed on the destination, the destination creates a fledgling process control block (PCB) for the process; otherwise the destination retrieves the existing PCB; in either case the PCB is updated with the the virtual filesystem information. The destination ultimately responds to the `migration_-request` with the directory where the source should upload the CRIU dump images. The source then securely copies these images to the destination; when the image transfer is complete, the source sends an `images_uploaded` notification, and the destination then invokes CRIU restore.

**Multi-thread and multi-process support.** SecureMigration handles multi-threaded processes, but requires that all threads within a process migrate together. This restriction stems from the fact that CRIU checkpoints at the granularity of the process tree.

To handle concurrent migration conditions, during Spin's initialization, Spin spawns an *internal* thread—the Intel Pin term for a thread that does not run application code—that sleeps, waiting for a migration condition. The application threads (which run both the analysis code and the original program's code, interleaved), enqueue migration requests and set a migration condition variable. When the internal thread wakes on the condition, it stops all application threads. Intel Pin ensures that threads are stopped in between traces, and, in particular, that they are not stopped in the middle of analysis functions or instrumentation callbacks. Once all threads have stopped, the internal thread performs a taint deadlock check across the threads to ensure that the aggregate readable arguments of each stopped instruction would not result in a deadlock. Additionally, Spin checks that, if there is more than one enqueued migration request, that these requests are for the same peer domain. After these checks, Spin initiates the migration protocol with Spry.

Spin does not yet support multi-process applications because (1) CRIU dumps at process tree granularity and (2) the handling of shared memory would require a distributed shared memory protocol. We leave solutions to such challenges—such as virtualizing the process tree by emulating `fork`—to future work.

## 5.3 Evaluation

We evaluate SecureMigration using four configurations of the NGINX 1.18 [4] webserver, each highlighting a broad use case. As SecureMigration does not yet

[4]For configuring NGINX, we use the command `./configure --with-http_ssl_module --with-http_dav_module --add-module=../njs/nginx --add-module=../nginx-http-auth-digest-1.0.0 --add-module=../naxsi-1.3/naxsi-src`. This results in compilation commands that use NGINX's standard `gcc` flags of `-O -W -Wall -Wpointer-arith -Wno-unused-paramter -Werror -g`.

support `fork` and `exec`, each configuration runs NGINX as a single process that uses `epoll` to manage concurrent requests. Each use case involves two domains, $A$ and $B$, where $A$ owns the majority of the webserver's resources (including all communication with the clients), and $B$ owns some private data (a taint source) that it isolates from $A$. We also evaluate variations where both parties have tainted resources.

By exploring these use cases, we seek to answer two fundamental questions. First, how well does SecureMigration perform versus normal, uninstrumented NGINX, and what portion of the overhead does Intel Pin, CRIU, and the split Spin and Spry design contribute. To remove the variability of network latencies for migration, we run all evaluations with domains $A$ and $B$ residing on the same machine, and where the communication and image transfers between the peer Sprys is over TCP sockets bound to `localhost`.

Second, how "closely" does SecureMigration mimic the behavior of an idealized, distributed version of NGINX that has been modified to use RPCs to encapsulate access to the private resources, instead of process migration. Rather than modify NGINX for this purpose, we approach this question indirectly by analyzing the migration patterns, and measuring the taint propagation and code coverage of the process while executing on $B$. To that end, Spin instruments the `call` and `ret` instructions to maintain a shadow call stack, relying on Intel Pin's support for debugging symbols to resolve the call's target address to the function's name.

Our evaluation machine is a standard desktop with an 8th Generation Intel Core i3-8100T CPU at 3.10GHz, with 8GiB of memory, and running Lubuntu 18.04

with the Linux 4.15 kernel. SecureMigration uses Intel Pin 3.17 and CRIU 3.15.

### 5.3.1 Use cases

We first describe the setup for each use case, as well as any function-level taint analysis required to either preserve taint on private resources or remove taint from a public output. We then evaluate the use cases jointly in §5.3.2.

**#1: Isolate TLS private key.** Our first use case is an analog to Keyless-SSL. In this setup, a CDN operator runs NGINX on an edge server, and serves the customer's website. Rather than give the operator the website's TLS private key, the customer instead wants to retain exclusive, physical custody of the key.

NGINX accesses the private key at two places: first, when loading the PEM-encoded key file from disk as part of reading the NGINX configuration file, and during the TLS handshake that precedes each HTTPS request. As we are using the ECDE-RSA-AES256-GCM-SHA384 cipher suite, NGINX specifically accesses the in-memory private key between the initial `ClientHello` and the server's `ServerHello` to produce a digital signature to return to the client. To correct for the laundering of the private key file's initial taint during PEM decoding, we wrap the `EVP_DecodeUpdate` family of OpenSSL functions so as to propagate any taint on the input arguments to the output. To correct for the tainted private key tainting the digital signature, we wrap the `EVP_DigestUpdate` family of functions to remove any taint on the output signature.

In our initial experiments with this use case, we discovered that taint also propagates to some ancillary data that OpenSSL uses for analyzing memory usage and leaks, as well as to the SHA384 context that OpenSSL uses to hash all handshake messages (the server sends this hash as part of the `ServerFinished` message, which signals the end of the TLS handshake). While still functionally correct, the result was that the process would thrash between the two domains, migrating to $B$ for calls to `CRYPTO_malloc` and friends, and returning back to $A$ for the socket-related system calls. Additionally, the process would migrate to $B$ to note only construct the `ServerHello` message, but also the `ServerFinished`. To correct this behavior, within Spin we wrap and replace the `CRYPTO_malloc` family functions to call directly into their libc counterparts. Additionally, we add declassification to the high-level `EVP_Digest` family of hash functions, as well as the specific hash implementations, such as `SHA384_Update` and `SHA384_Final`.

**# 2: Isolate TLS session resumption ticket key.** As an optimization for re-establishing prior connections, RFC 5077 [93] allows the TLS handshake to optionally return to the client a *ticket*—the session state (including the TLS session key), encrypted with a secret, symmetric key on the server called the *ticket key*. When resuming a connection, the client can present this ticket as part of the `ClientHello` message; the webserver decrypts the ticket and resumes the session. However, if an attacker obtains the ticket key, they can recover the session key for any ticket encrypted by that key, and thus decrypt any traffic for such sessions, regardless of whether the TLS cipher suite supports forward secrecy. Our second use case assumes that a single principal owns all of the webserver's resources: the webserver services traffic on Internet-facing machine $A$, and the ticket key is isolated to non-Internet-facing machine $B$.

For ticket-based session resumption, NGINX reads the ticket key from disk into memory during the webserver's initialization and adds a reference to the key to the SSL context (`SSL_CTX`). The ticket key is 80-bytes, where the first 16 bytes are the key's name, the next 32 are the encryption key, and the last 32 are the HMAC key. While NGINX sets AES-256-CBC and SHA256 as the ticket's cipher and hash functions, respectively, OpenSSL calls these methods through the higher level `EVP_EncryptUpdate`, `EVP_DecryptUpdate`, and `HMAC_Update` family of functions. Thus, in Spin, we wrap these functions to provide explicit, function-level taint analysis that removes any leakage of the ticket key's taint into either the encrypted or decrypted ticket.

**#3: Isolate scripts for custom request handling.** NGINX allows an administrator to extend the server's request handling with custom JavaScript; such scripts can implement an array of features, such as providing access controls and security checks, manipulating response headers, or filtering the HTTP message body. Internally, NGINX implements its own, bespoke JavaScript interpreter, called `njs`. The `njs` interpreter executes the JavaScript code during distinct phases of request handling—in other words, NGINX uses `njs` as a middleware, rather than an avenue for creating an application server. Upon each request, `njs` initializes a new JavaScript VM and an associated memory pool. During VM execution, memory allocations for JavaScript objects draw from the pool; upon request completion, NGINX releases the pool and frees the VM.

For this use case, we again assume that a CDN operator runs NGINX on an edge server, and serves the customer's website. However, we assume that the NGINX configuration references the customer's private JavaScript file as part of the request handling. While the JavaScript could perform custom access or firewall controls that reference the customer's sensitive data, for our evaluation purposes, we instead mock such logic by having the JavaScript simply return a hardcoded HTTP response body. In other words, the script is a taint source, but the script's output must be declassified as non-sensitive.

NGINX exports to the JavaScript environment various native functions that a script may call; the actual C implementations of these functions may use the `njs` C-API to retrieve values from the VM. For our use case, we deal with the `return` native method of a response object, whose implementation is the C function `ngx_http_js_ext_return`. Within this C function, we apply function-level taint analysis to declassify the outputs of the `ngx_js_integer` and `ngx_js_string` C-API functions, which the C function uses to retrieve `return`'s arguments for the HTTP status code and response body, respectively.

**#4: Isolate WebDAV content and authentication.** In our final use case, we assume that NGINX runs on a third-party cloud host, and that a content owner uses NGINX to serve both public, static content as well as provide a WebDAV [94] interface for allowing authenticated clients to access and modify sensitive content. In this use case, NGINX uses HTTP digest access authentication [95] to confirm the clients' identity. The central idea of digest authentication is that the webserver returns a challenge to the client that includes a nonce when the client attempts to access a protected resource. The client then computes a hash ($HA_1$) over their username and password, a separate hash ($HA_2$) over details of the URI access, and a final *response* hash over the concatenation of $HA_1$, the nonce, and $HA_2$. The client then reissues the original request, adding their username and response hash as an HTTP header. As the webserver has a flat password file (conventionally named `htdigest`) that maps the username to the $HA_1$ value, and since the webserver knows the nonce and the URI, the webserver can verify that the client-presented response matches the expected hash.

We assume that the cloud host stores the public content, but that the content owner wants to mediate authentication of clients and modification of the sensitive content, and thus wants for these operations to occur on their own machine. In particular, the content owner wants to guard against the cloud host impersonating as an authenticated client. To that end, the content owner pins the `htdigest` file to their machine, and marks this file as a taint source; the sensitive content is likewise pinned to their machine, but not marked as a taint source, since the cloud host can view such content as part of servicing an HTTP `GET` request.

**Variation: Bilateral taint.** While all of the use cases thus far have only domain $B$ specifying a taint source, we also explore variations where both $A$ and $B$ have tainted data. For these variations, we again assume that $A$ owns the majority of the webserver's resources and communicates with the web clients, but that $A$ also uses a proprietary set of web application firewall (WAF) rules, which $A$ specifies as a taint source. For our WAF, we use the popular NAXSI [96] (NGINX Anti XSS & SQL Injection) module, which allows an administrator to specify rich filters for blocking malicious web traffic.

### 5.3.2 Analysis

We now analyze SecureMigration's performance and behavior. For the TLS-key, njs-script, and authenticated WebDAV evaluations, we use `curl` to make the HTTPS requests. As `curl` does not support TLS tickets, for the ticket-key evaluations, we instead issue HTTPS requests using OpenSSL's `s_client` tool. Each underlying HTTP request and response message is less than 1 KiB.

**Overheads.** We first investigate the overheads of SecureMigration for each use case as compared to running NGINX in a normal, uninstrumented fashion. To isolate the overheads of each component of SecureMigration, we run SecureMigration in five different configurations. The first configuration, *Pin*, tests the fixed overhead of using Intel Pin in a noop-fashion; here Spin disables taint tracking and passes all system calls through to the kernel; Spry is not used and there is no migration. The second configuration, *Taint-Tracking*, tests the overhead of taint tracking: instrumentation is not added for memory access checks, Spry is still not used, and there is no migration. Although taint tracking is enabled, there is no taint in the system. The third configuration, *Access-Checks* adds the memory access checks. The fourth configuration, *1-domain*, uses both Spin and Spry, but does not migrate. Spin performs taint tracking and makes system calls to Spry, and Spry indicates whether the system call responses are tainted. This is the first configuration where the process is tainted. The final configuration, *2-domains*, executes the complete SecureMigration system, with the process migrating between the two domains $A$ and $B$.

Figure 5.2 indicates the performance overheads for each use case under each configuration. Each bar represents the latency for a single `GET` request (averaged over 20 separate, sequential requests) and normalized against the average latency of a request for normal, uninstrumented NGINX. For the ticket-key example, all requests have the client present a previously downloaded ticket. When computing the average, we disregard the first request; this request is noticeably longer (roughly $3\times$ slower) as Intel Pin must instrument new instructions; for subsequent requests, Intel Pin directly JITs from its code cache of previously instrumented instructions.

From Figure 5.2, we observe that there is a roughly $1.6\times$ overhead by just using Intel Pin. We also observe that the reduced code complexity for session resumption lessens the overheads compared to the other use cases.

Upon adding taint tracking, we notice a roughly $525\times$ overhead. In terms of latency, this means that a request for the tls-key example goes from taking 7.3ms in the uninstrumented case, to 3.87s. Upon adding in the memory access checks, the overheads approach $600\times$, with requests taking nearly 4.4s. The 1-domain configuration has overheads that are just slightly higher than the Access-Checks configuration: while there is some overhead in making RPCs for the system calls (each request results in less than 30 system calls), the more dominant factor is that our taint setting and clearing logic depend on whether the memory page already

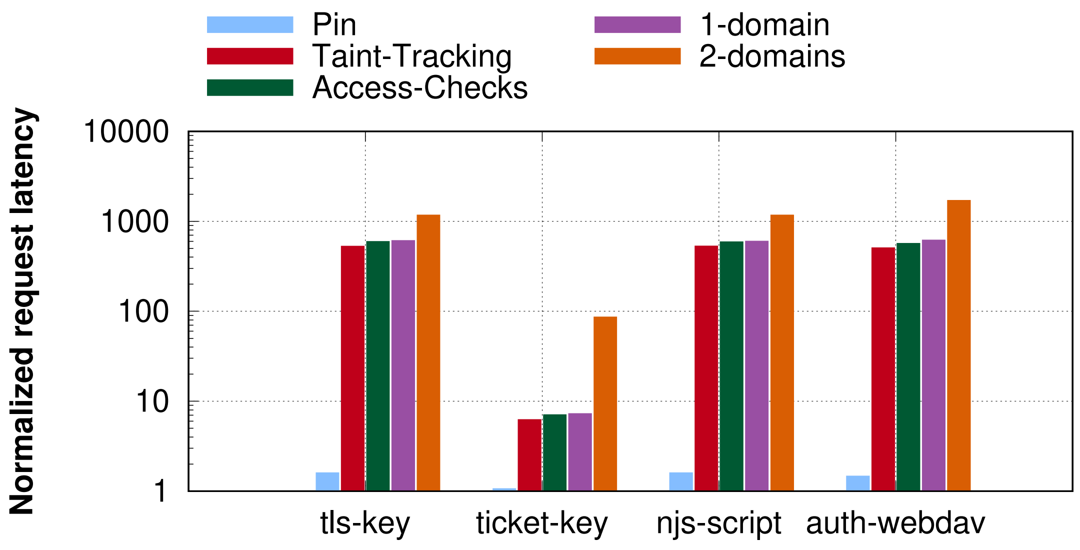


Figure 5.2: Average latency for an HTTPS request from a single client, normalized against an uninstrumented NGINX. Note the log-scale y-axis. For each use case, we benchmark SecureMigration by progressively enabling features: *Pin* (Intel Pin with a noop instrumentation), *Taint-Tracking* (taint tracking), *Access-Checks* (taint tracking, memory checks) *1-domain* (full Spin & Spry, but without migration), and *2-domains* (Spin & Spry, migrating NGINX between two domains).

has taint on it.

Finally, migration usually doubles the total latency, with the latency for the tls-key example reaching 8.6s. Note that, for the tls-key case, the average time for a CRIU dump is 0.190s, and for a restore 0.107s. In other words, the vast majority of the request overhead is in transferring the checkpoint images (which total 342 MiB) once from domain A to B, and then again from B back to A. For the tls-case, each one-way transfer takes roughly 1.759s; normalized to the average uninstrumented NGINX request latency, this represents an additional 240× of overhead.

**Migration patterns.** In Figure 5.3 we examine the process migrations for each use case during the webserver's initialization, and then over four sequential HTTPS requests. For the ticket-key example, the requests alternate between the client starting a new TLS session and resuming the prior session. For the auth-webdav example, the sequence of HTTPS operations is: creating a directory (`MKCOL`), creating a file in that directory (`PUT`), renaming the file (`MOVE`), and then deleting the file (`DELETE`). Additionally, we examine the waf/tls-key and waf/njs-script examples, whereby domain $A$ runs a WAF module and taints the WAF rules, and domain $B$ taints the TLS key and njs script, respectively.

The dashed vertical lines in Figure 5.3's subplots indicate NGINX's receipt of the client message that immediately precedes when the server accesses domain $B$'s tainted resources. For tls-key, the vertical lines are the `ClientHello`. For ticket-key, the vertical lines are the `ClientKeyExchange` when starting a new session and the `ClientHello` when resuming a session. For the njs-script subplot, the vertical lines mark receipt of the HTTP GET request; NGINX will subsequently access the private njs script to compose the HTTP response. Finally, for auth-webdav, the lines mark the client's second HTTP request for a given URL (the client's first request receives the server's nonce; the second request includes the HTTP header with the client's authentication).

During server initialization, the tls-key and ticket-key example make one expected round-trip to load their respective key into memory. In contrast, the njs-script makes three round-trip migrations to load the script: the first round-trip to load the script; the second to call `lstat` on the script to form the list of module paths, and the third due to an execution of `mov r10, qword ptr [rsp]`, where the source memory operand was gratuitously tainted.

For the four client requests, we see that that the migrations for all examples occur at the expected HTTPS phase. At a functional-level, the migrations also occur within the expected routines. For instance, the ticket-key session resumption migrations occur in the `ngx_ssl_session_ticket_callback` function, and the njs-script the migrations occur within the the `njs_vmcode_interpreter` function—the JavaScript interpreter's entry point. We note that the auth-digest `PUT` request (the second client request) results in two migrations to $B$: the first reads the `htdigest` file (during which the uploaded file is written to a temporary file); next the process migrates back to $A$ to send the HTTPS response; finally the process migrates back to $B$ to delete the temporary file as part of the request's cleanup.

The migration patterns for waf/njs-script is identical to the njs-script example; the additional overhead of the WAF module shifts the time series right. The waf/tls-key migration pattern only differs from the tls-key example during server initialization. Instead of migrating once to $B$ to load the private key PEM file, the waf/tls-key example migrates four times; each migration to $B$ is for a step in loading the private key (calls to `openat`, `fstat`, `close` for reading the key, and a final call to a subfunction of `SSL_CTX_use_PrivateKey` for decoding the key); each migration to $A$ is due to $A$ having tainted part of the stack, with $B$ attempting to `pop` a tainted value.

For the four requests of Figure 5.3, Table 5.1 additionally indicates the average time the process spends in each domain (as well as migrating) for a single request, and the percentage of the request that is spent in each domain. For all examples, the time spent executing on $B$ is less than 10% of the total requests time; this is important, as in most of the use cases, $B$ represents the cloud customer, rather than the cloud provider. The time spent executing on $B$ is naturally higher during the tls-key example since the private key operation dominates the total time for the request. Table 5.1 additionally motivates the case for optimizing migration, as roughly 40% of the time spent servicing a request is for migration.

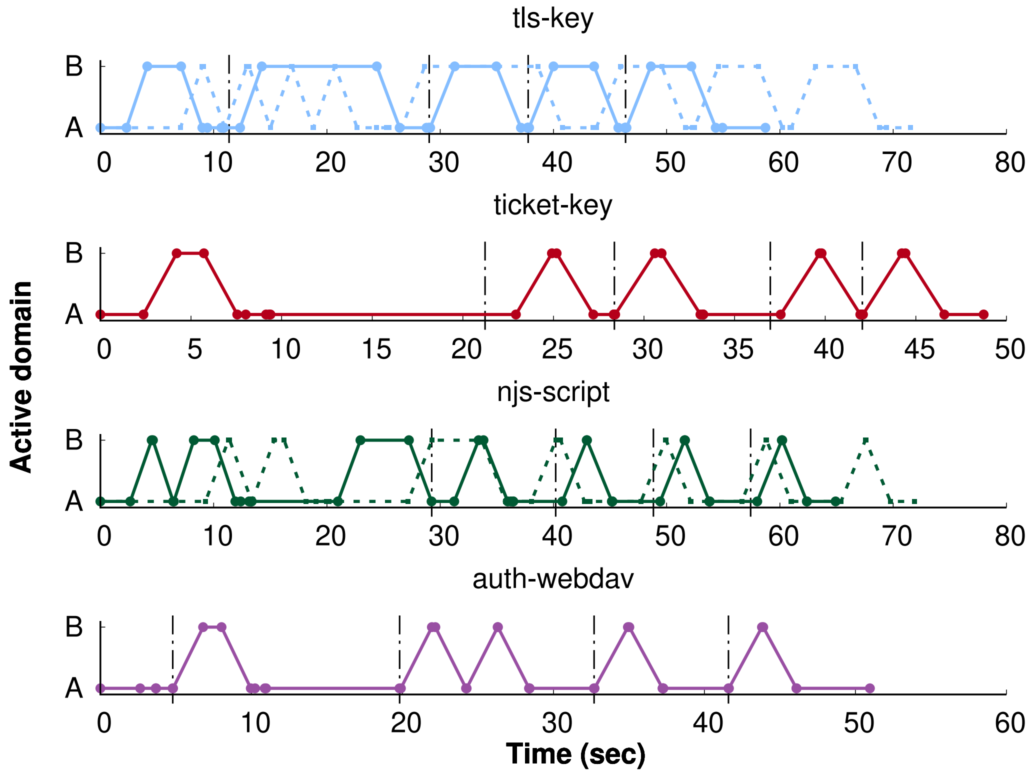


Figure 5.3: The migration pattern and process execution times on each domain over the course of four HTTPS requests. The dashed vertical lines indicate NGINX's receipt of the client message that immediately precedes accessing $B$'s tainted resource. The dashed plots in the tls-key and njs-script subplots are the waf/tls-key and waf/njs-script examples, respectively. For these use cases, domain $A$ runs a WAF module and taints the WAF rules.

| | $t_A$ | $t_B$ | $t_{migration}$ |
|---|---|---|---|
| **tls-key** | 2.506 (21.0%) | 5.242 (43.8%) | 4.208 (35.2%) |
| **ticket-key** | 5.394 (54.7%) | 0.235 (2.4%) | 4.230 (42.9%) |
| **njs-script** | 6.383 (49.4%) | 1.207 (9.4%) | 5.316 (41.2%) |
| **auth-webdav** | 6.180 (52.3%) | 0.412 (3.5%) | 5.228 (44.2%) |

Table 5.1: The time in seconds that the process spends on domain $A$ ($t_A$), $B$ ($t_B$) and in migration ($t_{migration}$) for a single request, averaged over the four HTTPS requests of Figure 5.3 (excludes server initialization). The numbers in parentheses are the percentage of the entire request.

**Taint propagation.** We ask whether the total number of tainted bytes in the process remains stable, as a concern with taint tracking is that over-tainting may lead to the entire process address space becoming tainted. To that end, we instrument the `ret` instruction to log the number of bytes tainted to each domain. In Figure 5.4, we plot the number of bytes tainted to domain $B$ over the course of the webserver's initialization, and then for 20 sequential HTTPS requests (we initially set the total to the size of taint source, even before the process loads the tainted file). To give an idea of the dispersion of taint, we also plot on the opposite y-axis the number of pages on which $B$ has tainted a byte. As expected, the amount of taint increases when loading the resource and servicing a request (the spikes in the plot) before returning to a stable band.

Table 5.2 additionally shows the initial taint, max taint, and average taint at the start of the request (NGINX's call to the `accept4` system call) for the four examples, as well as the two WAF variants. Generally speaking, the max taint can be upwards of 7× the initial taint size, whereas the number of bytes tainted at the start of a request is 2–3× the initial taint size.

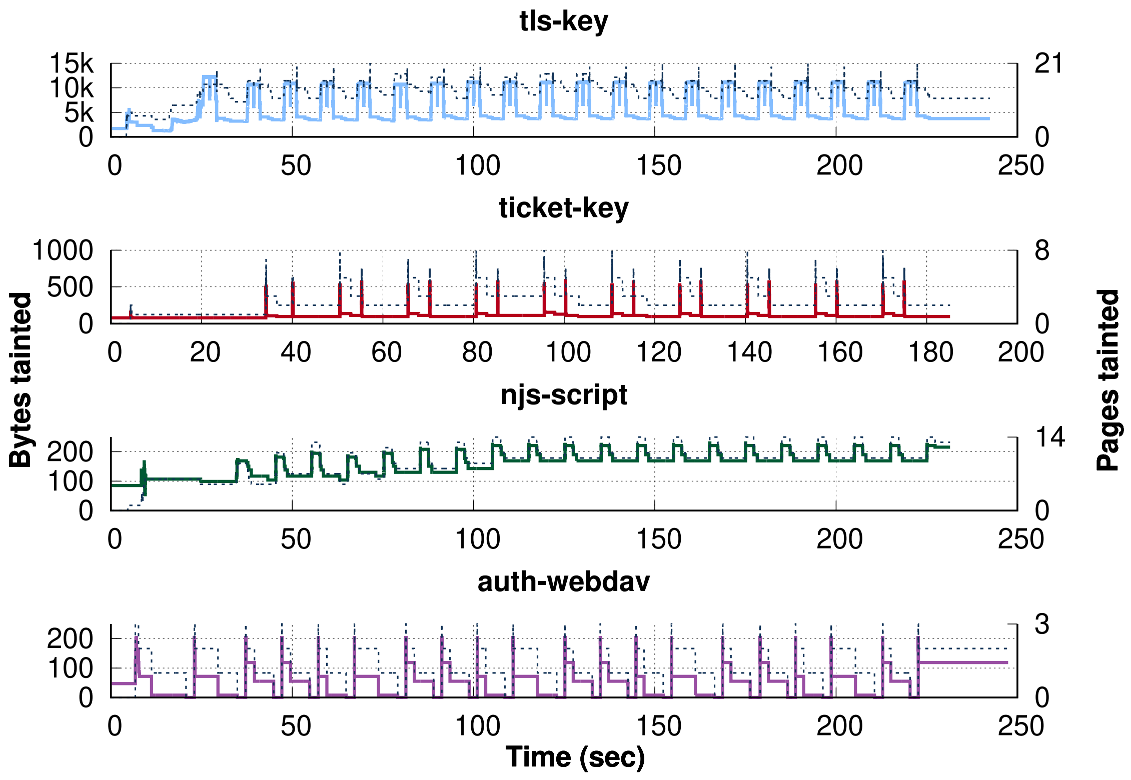


Figure 5.4: Total data tainted to domain $B$ over the course of the webserver's initialization and twenty HTTPS requests. The dashed lines are the number of pages tainted (the y2-axis).

| | Domain | $T_{init}$ | $T_{max}$ | $\overline{T}_{reqstart}$ |
|---|---|---|---|---|
| **tls-key** | B | 1704 | 12,324 (7.2×) | 3560 (2.1×) |
| **waf/tls-key** | A | 6333 | 23,982 (3.8×) | 20,671 (3.3×) |
| | B | 1704 | 12,294 (7.2×) | 3560 (2.1×) |
| **ticket-key** | B | 80 | 604 (7.6×) | 106 (1.3×) |
| **njs-script** | B | 85 | 230 (2.7×) | 200 (2.4×) |
| **waf/njs-script** | A | 6333 | 23,723 (3.7×) | 20,273 (3.2×) |
| | B | 85 | 230 (2.7×) | 191 (2.2×) |
| **auth-webdav** | B | 47 | 209 (4.4×) | 92 (2.0×) |

Table 5.2: The size in bytes of a domain's initial taint source ($T_{init}$), the maximum number of bytes tainted to the domain ($T_{max}$), and the average bytes tainted to the domain ($\overline{T}_{reqstart}$) at the start of a request, for the requests of Figure 5.4.

**Whence bad performance?** Our extensive evaluation shows that the overheads are extensive: over 1000× that of normal, uninstrumented NGINX. However, looking at the break-down of the amount of time spent in each component of our system paints an optimistic path forward.

Nearly half of the overhead is from migration, which is dominated by the time CRIU takes to transfer large checkpoint images. This is not fundamental to our system. When migrating for a quick operation like performing a digital signature, we typically only need a small fraction of memory pages to migrate, not the entire image, and then migrate back. CRIU could and should be extended to support a distributed shared memory protocol, whereby a domain transfers only the process's working set of memory, and then transfers the other required pages via remote page faults. This one change alone would decrease our overheads by about half.

The overheads for instrumentation are dominated by the taint tracking logic, rather than the byte-granular memory access checks. The reason for this is likely two-fold. First, for each new trace, Spin must perform a complex case analysis on each instruction to determine which taint tracking function to insert, even if the process has already previously seen the instruction. This is different from the access-check instrumentation, which memoizes its case analysis. Second, setting taint in shadow memory is more costly than retrieving taint. For safety, setting taint uses a two-pass algorithm that first checks that the memory range is valid (that is, that the real access would not generate a page fault in the application), and then determines whether a page transitions from tainted to untainted (or vice versa), so as to facilitate swapping private memory with Spry during migrations. Both the taint tracking instrumentation and the shadow memory implementation are prime for optimization. More generally, the shadow memory page table walks are completely unoptimized, and would likely benefit from a software analog to a translation lookaside buffer (TLB).

There are several more exploratory directions for performance improvement. For instance, we mention in §5.2.1 that taint tracking at larger granularities, such as page-level, would obviate the access-check instrumentation, since Spin could instead rely on kernel-level page protections (i.e., a sigsegv+mprotect approach). Such an approach, however, has the downside of increasing the risk for taint deadlock and false sharing. An interesting idea would be to dynamically switch between page-level and byte-level granularities based on the process's state. Finally, to reduce the costs of dumps, Spin could replace libC's allocator so as to maintain a persistent heap [97],

In summary, although our performance numbers may fail to impress, they are largely attributable to optimizations that are feasible but beyond the scope of this dissertation. Recall, however, that a central goal of this dissertation is to demonstrate the possibility of turning a legacy monolithic application into a secure distributed one, and in that regard, our evaluation demonstrates success.

## 5.4 Conclusion

In this chapter, we have taken a first step towards turning legacy, monolithic programs into secure distributed systems without modifications to the software. Combining several existing building blocks (taint tracking and process migration), we constructed a new abstraction that moves processes to the data they need, and leaves data at the principals trusted to house it. In other words, we have shown that we can achieve new security properties not by changing *what* a program executes, but *where* it executes throughout its lifetime.

Our evaluation shows that this is possible, but that there is still significant room for future work. With regards to performance, we have identified two critical areas for improvement: faster taint tracking (which would benefit many applications) and lighter-weight process migration. Another important area of future work involves what happens when a process must perform an operation on two pieces of data, each bound to a separate domain. In our current implementation, this would deadlock, but we believe there may be a path forward even then, by employing secure multiparty computation.

# Chapter 6: Conclusion and Future Work

In this thesis, I demonstrated that it is possible to run legacy application binaries with confidentiality and integrity guarantees that reflect a multi-party trust setting. My approach to running *old* software in *new* trust settings was to develop application-level virtualization layers to handle the trust concerns on behalf of the application, with the user specifying the trust policy to the virtualization layer. Both conclaves and SecureMigration are in the tradition of exokernel/libOS design [98,99], whereby the application is bundled with a custom execution environment, with conclaves implementing the execution environment as a microkernel within secure hardware enclaves, and SecureMigration as a distributed shim operating system.

## 6.1 Immediate steps

Before discussing the grander challenges that remain, I note several immediate engineering steps for both systems. Ultimately, applying these systems to a more diverse set of applications, beyond web servers, (such as DNS servers, mail servers, machine learning, join computations of randomness, databases, and remote shells) should drive these features.

### 6.1.1 Conclaves

For conclaves, immediate tasks include supporting a larger portion of the POSIX and Linux kernel interface, hardening the system against side-channel attacks (including adding oblivious filesystems [100]), and improving the performance of the shared memory implementation. Additionally, from a usability standpoint, the creation, configuration, and attestation of conclaves would benefit from more mature tooling.

### 6.1.2 SecureMigration

For SecureMigration, an interesting avenue of research is optimizing migration, as currently a significant portion of the process's time is spent migrating between domains. Such work may entail extending CRIU with a distributed shared memory (DSM) protocol, such that a domain only transfers the working set of memory pages during migration, and then lazily transfers the remaining pages via remote page faults. Additionally, Spin may benefit from implementing a custom memory allocator that maintains a persistent heap [97]; such an approach may reduce the overhead of each dump operation by amortizing the cost of dumping memory over the process's execution. Other optimizations may explore ways to avoid migration altogether, such as through the implementation of more sophisticated resource management (as by replicating non-sensitive files across domains), or forwarding system calls that reference non-sensitive resources across domains, rather than migrating for such calls.

Further research efforts could explore supporting multi-process applications, and allowing processes and threads to simultaneously run on the different domains. Some use cases may require more complex taint semantics, such as allowing for domain groups, with taint shared among the group members. Part and parcel with this feature would be provisions for sending taint across a network or persisting taint in the filesystem.

## 6.2 Grander challenges

While the aforementioned engineering efforts are by no means trivial, deeper lines of research should focus not merely on supporting *more* applications, but rather application settings that reflect a *diverse* set of trust assumptions. To that end, I next describe a number of infelicities that arise when applying my two systems (and, in particular, conclaves) to distributed and mobile settings.

### 6.2.1 Distributed, federated, systems

One line of research is how to use conclaves—or, more generally, secure hardware enclaves—in environments that are inherently distributed: that is, environments that involve the cooperation of multiple processes run by distrusting principals, where the client(s) does not have knowledge of all of the processes and principals that comprise the system. Such systems are sometimes called *federated* to emphasize that the system is distributed across multiple principals.

Note that Phoenix is an idealized notion of such a distributed system: I assumed the customer knows the exact build of the webserver and that the webserver is non-buggy (and thus does not leak secrets). Additionally, in Phoenix, I assume the webserver only handles connections from the web clients and the customer's origin server.

In a real world scenario, all of these assumptions are invalidated: the customer would not have knowledge of (and thus be unable to audit) the CDN's software stack. The CDN's software stack would contain numerous bugs [101, 102]. Moreover, the CDN would frequently update and modify the software stack (perhaps daily), with the stack comprising a sophisticated hierarchy of caches and backend services, rather than a standalone webserver.

These assumptions break down not only for a real-world web server deployment, but for nearly all major Internet services. For instance, with DNS, a noble privacy goal is make the domain name system *oblivious* with respect to a client's query, such that only the client observes the plaintext of both the DNS request and response. The solution of "just place each DNS server in a conclave" and use DNS-over-TLS [103] or DNS-over-HTTPS [104] as the transport mechanism is not quite sufficient to ensure the servers do not learn the client's query. Rather, the client needs proof that the path the query took was always shielded by secure hardware, and by TLS connections that terminate in an enclaved process. (Note that end-to-end shielding for other applications, such as email or messaging, where only the sender and recipient observe the plaintext message, requires a similar set of guarantees over their respective application-level servers.)

Unfortunately, providing such a guarantee represents serious usability issues:

returning to the DNS example, it is unreasonable for the client to know every DNS server implementation, and every build thereof. In other words, even if the client received an attestation from each resolver on the query's path, the client has no way to verify that the attestation is over a trustworthy piece of software, barring some trusted third party periodically publishing a list of all known trustworthy DNS resolver builds. This is an untenable combinatoric problem, as the server's attestation will change each time the DNS software is recompiled, the configuration changed, or a supporting shared library upgraded.

To tame this combinatoric (and usability) problem, an alternative approach is to assume a small number of enclave-based trusted execution environments, and for the client to verify attestations over properties of these environments, rather than over a specific application running within this environment. The idea is not novel, as some prior work, notably Ryoan [105], explores this very design, albeit in a restricted setting. Under such a design, the client receives an attestation that the execution runtime enforces strong guarantees of not leaking data, even for an application that (intentionally or inadvertently) may export the data.

Such designs provide a natural setting for composing conclaves with the taint tracking component of SecureMigration. Namely, the execution environment not only shields the user's data within a secure hardware enclave, but the execution environment instruments the application so as to ensure the user's data remains sandboxed within the enclave.

### 6.2.2 Mobile

For both conclaves and SecureMigration, the predominate use case was a customer using a third party to run a webserver, with the customer maintaining the confidentiality of some resource, such as the website's private key. For mobile applications, the situation is somewhat reversed, with the customer owning the machine (the smartphone) but running third-party applications that use the customer's data. The customer wants strong guarantees on what data is shared with the third party. As many apps have a cloud storage component, the user wants to ensure that the content (such as photos or messages) they upload to the cloud is not observable by the app company. Additionally, the app itself may require strong guarantees that it is actually interacting with a human user.

The first set of problems is essentially a special form of sandboxing, and there exists some prior work [59, 106] in applying information flow control to smartphone apps. The taint tracking techniques of SecureMigration may be appropriate in this setting as well, though it is unclear if the app-specific entities and data flow constraints are immediately expressible.

The second set of problems reprises the same sorts of issues as with the distributed and federated systems—the app user would like to interact with the opaque cloud-side portion of the app for content storage or processing purposes, but without revealing their actual content. In contrast to the DNS use case, and as an even more extreme version of the email use case, the app may entail a complex policy for the allowable ways in which the user's content may be shared, rather than a binary policy of “do not leak.”

Finally, as in the case of a mobile banking app that accepts a user’s pin, the banking app may want a guarantee that the human user entered the pin, and that some malicious software has not stolen the pin and is attempting a withdrawal. For such a use case, running the banking app within an enclave is not sufficient, as the app needs to further attest to the pin’s provenance from the physical input device. To that end, an interesting future avenue may involve composing conclaves with work in secured IO [107, 108].

## 6.3 Decoupling security from applications

Both conclaves and SecureMigration represent an approach to software design where security concerns are decoupled from the application logic and placed under the responsibility of the execution environment. This position of applying security solutions to an application *post-hoc* stands in stark contrast to the oft-repeated refrain of *building security in* from day one.

It is not my intent to defend one position over the other: in the end, both approaches have value and are needed. Rather, with this thesis, I hope to have demonstrated that some security concerns are impossible to predict at the time of development, as software tends to be used in ways that the developer could never have imagined. Moreover, as I demonstrated with the various evaluations, the sheer number of software configurations required to span the set of all security postures may make “building security in” prohibitive even when the requirements are known *a priori*. For instance, SecureMigration expresses not only Keyless SSL, but also a variety of other nonexistent—but potentially useful—protocols, such as "Keyless Ticket Key," "Scriptless JavaScript," and "HTDigest-less Authentication"—each of which may require significant development effort if pursued as separate features.

An obvious set of open-ended questions is: how far can we take this idea of applying security policies after the fact, and, how should we develop software *now* to make it more amenable to security refinements *later*.

For the former question, we already employ diverse kernel and compiler-level techniques to provide "seat belts" and "guardrails" for applications, such as ASLR, system call filters, stack canaries, and control flow integrity guards. Conclaves and SecureMigration differ from these techniques by moving the constraints from low-level process constructs to application-level entities, such as files and principals. Can we move the needle further? Should protocols like TLS be removed from applications and moved to the execution environment? Can, and should, private browsing be enforced at the execution environment level, rather than exposed as an application feature? Can protocols for differential privacy [109] be applied as a form of information flow control within the execution environment, rather than within a database engine? Can dynamic binary instrumentation (DBI) be composed with either fully homomorphic encryption (FHE) or secure multi-party (SMC) program, such that the DBI system dynamically translates the legacy program to an equivalent FHE or SMC one?

To assist these efforts, is it enough to merely develop software in modular (and thus, decoupled) fashion, so as to facilitate instrumentation? Do we need a new generation of developer tools, to, for instance, measure information flows and data leakage as part of the development process?

I wish the best of luck to the future researchers that take on these challenging problems.

# Bibliography

[1] Stephen Herwig, Bobby Bhattacharjee, Dave Levin, and Neil Spring. DNSql: Processing massive DNS collections. 2016.

[2] Stephen Herwig, Katura Harvey, George Hughey, Richard Roberts, and Dave Levin. Measurement and analysis of Hajime, a peer-to-peer IoT botnet. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[3] Stephen Herwig. secmodel_sandbox: An application sandbox for NetBSD. In *Proceedings of the BSDCan Conference*, 2017.

[4] Zhihao Li, Stephen Herwig, and Dave Levin. DeTor: Provably avoiding geographic regions in Tor. In *USENIX Security Symposium*, 2017.

[5] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *USENIX Security Symposium*, 2004.

[6] Michael Reininger, Arushi Arora, Stephen Herwig, Nicholas Francino, Jayson Hurst, Christina Garman, and Dave Levin. Bento: Safely bringing network function virtualization to Tor. In *ACM SIGCOMM*, 2021.

[7] Stephen Herwig, Christina Garman, and Dave Levin. Achieving Keyless CDNs with Conclaves. In *USENIX Security Symposium*, 2020.

[8] Graphene-sgx (project website). `https://grapheneproject.io`.

[9] Stephen Herwig, James Larisch, Deepak Garg, and Dave Levin. Secure process migration: Dynamic, intra-machine, privilege separation for legacy software. Under Submission, 2021.

[10] Frank Cangialosi, Taejoong Chung, David Choffnes, Dave Levin, Bruce M. Maggs, Alan Mislove, and Christo Wilson. Measurement and analysis of private key sharing in the HTTPS ecosystem. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[11] Nick Sullivan. Keyless SSL: The Nitty Gritty Technical Details. Cloudflare Blog, September 2014. `https://blog.cloudflare.com/keyless-ssl-the-nitty-gritty-technical-details/`.

[12] Liang Zhang, David Choffnes, Tudor Dumitras, Dave Levin, Alan Mislove, Aaron Schulman, and Christo Wilson. Analysis of SSL certificate reissues and revocations in the wake of Heartbleed. In *ACM Internet Measurement Conference (IMC)*, 2014.

[13] Ioana Boureanu, Daniel Migault, Stere Preda, Hyame Assem Alamedine, Sanjay Mishra, Frederic Fieau, and Mohammad Mannan. LURK: Server-Controlled TLS Delegation. In *IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2020.

[14] Karthikeyan Bhargavan, Ioana Boureanu, Pierre-Alain Fouque, Cristina Onete, and Benjamin Richard. Content delivery over TLS: a cryptographic analysis of keyless SSL. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017.

[15] Craig Hunt. *TCP/IP Network Configuration*. O'Reilly, 2002.

[16] Chia-Che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *USENIX Annual Technical Conference*, 2017.

[17] Pierre-Louis Aublin, Florian Kelbert, Dan O'Keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter R. Pietzuch. TaLoS : Secure and transparent TLS termination inside SGX enclaves. Technical Report, 2017.

[18] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure linux containers with Intel SGX. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[19] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with Haven. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[20] 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 *USENIX Security Symposium*, 2018.

[21] Akamai. `https://www.akamai.com/`.

[22] Cloudflare. `https://www.cloudflare.com/`.

[23] Yossi Gilad, Amir Herzberg, Michael Sudkovitch, and Michael Goberman. CDN-on-demand: An affordable DDoS defense via untrusted clouds. In *Network and Distributed System Security Symposium (NDSS)*, 2016.

[24] David Gillman, Yin Lin, Bruce Maggs, and Ramesh K. Sitaraman. Protecting websites from attack with secure delivery networks. *IEEE Computer*, 48(4), April 2015.

[25] Adrienne Porter Felt, Richard Barnes, April King, Chris Palmer, Chris Bentzel, and Parisa Tabriz. Measuring HTTPS adoption on the Web. In *USENIX Security Symposium*, 2017.

[26] Jinjin Liang, Jian Jiang, Haixin Duan, Kang Li, Tao Wan, and Jianping Wu. When HTTPS meets CDN: A case of authentication in delegated service. In *IEEE Symposium on Security and Privacy*, 2014.

[27] ModSecurity: Open Source Web Application Firewall. `https://modsecurity.org/`.

[28] OWASP: The Open Web Application Security Project. `https://www.owasp.org`.

[29] Intel Software Guard Extensions (Intel SGX). `https://software.intel.com/en-us/sgx`.

[30] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP@ ISCA*, page 10, 2013.

[31] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. 2013.

[32] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel Software Guard Extensions: EPID Provisioning and Attestation Services. Available from `https://software.intel.com/sites/default/files/managed/ac/40/2016%20WW10%20sgx%20provisioning%20and%20attesatation%20final.pdf`, 2016.

[33] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, volume 13, 2013.

[34] Alexander B. Introduction to Intel SGX Sealing. Available at `https://software.intel.com/en-us/blogs/2016/05/04/introduction-to-intel-sgx-sealing`, 2016.

[35] Intel Corporation. Intel Software Guard Extensions SDK for Linux OS. Available from `https://01.org/sites/default/files/documentation/intel_sgx_sdk_developer_reference_for_linux_os_pdf.pdf`, 2016.

[36] Shanwei Cen and Bo Zhang. Trusted Time and Monotonic Counters with Intel® Software Guard Extensions Platform Services. Technical report, Intel Corporation, 2017.

[37] Hongliang Liang, Mingyu Li, Qiong Zhang, Yue Yu, Lin Jiang, and Yixiu Chen. Aurora: Providing trusted system services for enclaves on an untrusted system. *arXiv preprint arXiv:1802.03530*, 2018.

[38] Jethro Gideon Beekman. *Improving cloud security using secure enclaves*. PhD thesis, UC Berkeley, 2016.

[39] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution. Technical report, Technical report, 2018.

[40] Intel SGX and Side-Channels. `https://software.intel.com/en-us/articles/intel-sgx-and-side-channels`.

[41] Intel Software Guard Extensions (Intel SGX) Developers Guide. `https://software.intel.com/en-us/download/intel-software-guard-extensions-intel-sgx-developer-guide`.

[42] L1 Terminal Fault. `https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault`.

[43] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. *arXiv preprint arXiv:1811.05441*, 2018.

[44] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 428–441. IEEE, 2018.

[45] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 227–240, 2018.

[46] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2017.

[47] Amit A. Levy, Henry Corrigan-Gibbs, and Dan Boneh. Stickler: Defending against malicious content distribution networks in an unmodified browser. In *IEEE Symposium on Security and Privacy*, 2016.

[48] Devdatta Akhawe, Frederik Braun, François Marier, and Joel Weinberger. Subresource integrity, 2016. `https://www.w3.org/TR/SRI/`.

[49] Mike West. Content security policy level 3, 2018. `https://www.w3.org/TR/CSP3/`.

[50] Chris Lesniewski-Laas and M. Frans Kaashoek. Ssl splitting: Securely serving data from untrusted caches. In *USENIX Annual Technical Conference*, 2003.

[51] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego R López, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste. Multi-context tls (mctls): Enabling secure in-network functionality in tls. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 199–212. ACM, 2015.

[52] David Naylor, Richard Li, Christos Gkantsidis, Thomas Karagiannis, and Peter Steenkiste. And then there were more: Secure communication for more than two parties. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 88–100. ACM, 2017.

[53] Hyunwoo Lee, Zach Smith, Junghwan Lim, and Gyeongjae Choi. matls: How to make tls middlebox-aware? In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2019.

[54] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*, 2004.

[55] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[56] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[57] 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 *USENIX Security Symposium*, 2020.

[58] Yubin Xia, Yutao Liu, Cheng Tan, Mingyang Ma, Haibing Guan, Binyu Zang, and Haibo Chen. Tinman: Eliminating confidential mobile data exposure with security oriented offloading. In *European Conference on Computer Systems (EuroSys)*, 2015.

[59] William Enck, Peter Gilbert, 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. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[60] Mark S. Gordon, D. Anoushe Jamshidi, Scott Mahlke, Z. Morley Mao, and Xu Chen. COMET: Code Offload by Migrating Execution Transparently. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[61] Craig Gentry. Fully homomorphic encryption using ideal lattices. *ACM Symposium on Theory of Computing*, 9, 2009.

[62] Craig Gentry. Computing arbitrary functions of encrypted data. *Communications of the ACM*, 53(3):97–105, 2010.

[63] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. *SIAM Journal on Computing*, 45(3):882–929, 2016.

[64] Andrew C. Yao. Protocols for secure computations. In *Proceedings of the Annual Symposium on Foundations of Computer Science*, 1982.

[65] Nicolas Desmoulins, Pierre-Alain Fouque, Cristina Onete, and Olivier Sanders. Pattern matching on encrypted streams. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 121–148. Springer, 2018.

[66] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. *ACM SIGCOMM Computer Communication Review*, 45(4):213–226, 2015.

[67] Sébastien Canard, Aïda Diop, Nizar Kheir, Marie Paindavoine, and Mohamed Sabt. Blindids: Market-compliant and privacy-friendly intrusion detection system over encrypted traffic. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 561–574. ACM, 2017.

[68] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. Embark: Securely outsourcing middleboxes to the cloud. In *NSDI*, volume 16, pages 255–273, 2016.

[69] Free Software Foundation. GNU Hurd. `http://www.gnu.org/software/hurd/hurd.html`.

[70] J. Mark Stevenson and Daniel P. Julin. Mach-US: UNIX on generic OS object servers. In *USENIX Technical Conference*, 1995.

[71] Igor Sysoev and Nginx Inc. NGINX. https://www.nginx.com.

[72] Available from MITRE, CVE-ID CVE-2014-0160, 2014. CVE-2014-0160 (Heartbleed bug).

[73] Lars Lühr. ayeks' SGX Hardware github repository. `https://github.com/ayeks/SGX-hardware`.

[74] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. Integrating remote attestation with transport layer security. *CoRR*, abs/1801.05863, 2018.

[75] Let's Encrypt. `https://letsencrypt.org/`.

[76] R. Barnes et al. Automatic certificate management environment (ACME). daft-ietf-acme-acme-18, December 2018.

[77] HAProxy: The Reliable, High Performance TCP/HTTP Load Balancer. `https://www.haproxy.org/`.

[78] Grzegorz Kostka. lwext4. `https://github.com/gkostka/lwext4`.

[79] The XTS-AES Tweakable Block Cipher. IEEE Std 1619-2007, 2008.

[80] Morris J. Dworkin. Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices. NIST Special Publication 800-38E, 2010.

[81] Roughtime protocol. `https://roughtime.googlesource.com/roughtime/+/HEAD/PROTOCOL.md`.

[82] Dmitrii Kuvaiskii. Graphene-SGX Exitless. `https://github.com/dimakuv/graphene/tree/exitless`.

[83] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless os services for sgx enclaves. In *European Conference on Computer Systems (EuroSys)*, 2017.

[84] Ofir Weisse, Valeria Bertacco, and Todd Austin. Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves. 2017.

[85] BearSSL: A Smaller SSL/TLS Library. `https://bearssl.org/`.

[86] Nico Weichbrodt, Pierre-Louis Aublin, and Rüdiger Kapitza. Sgx-perf: A Performance Analysis Tool for Intel SGX Enclaves. In *Proceedings of the 19th International Middleware Conference*, Middleware '18, 2018.

[87] Intel Corporation. Linux SGX Kernel Driver. `https://github.com/intel/linux-sgx-driver`.

[88] Jens Axboe. Fio 3.13. `git:git.kernel.dk/fio.git`.

[89] Linux Checkpoint/Restore In Userspace. `http://criu.org`.

[90] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN's Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[91] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *ACM International Conference on Virtual Execution Environments (VEE)*, 2012.

[92] Michael Matz, Jan Hubička, Andreas Jaeger, and Mark Mitchell. System v application binary interface: AMD64 architecture processor supplement, draft version 0.99.7, November 2014.

[93] J. Salowey, H. Zhou, Cisco Systems, P. Eronen, Nokia, H. Tschofenig, and Nokia Siemens Networks. Transport layer security (TLS) session resumption without server-side state. RFC 5077, January 2008.

[94] L. Dusseault and CommerceNet. HTTP extensions for Web Distrubuted Authoring and Versioning (WebDav). RFC 4918, 2007.

[95] R. Sheck-Yusef, Avaya, D. Ahrens, Independent, S. Bremer, and Netzkonform. HTTP diegest access authentication. RFC 7616, 2015.

[96] NBS System. NAXSI (NGINX Anti XSS  SQL Injection Module), November 2020. `https://github.com/nbs-system/naxsi`.

[97] Terence Kelly. Persistent memory programming on conventional hardware. *ACM Queue*, 17(4), August 2019.

[98] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[99] D. R. Engler and M. F. Kaashoek. Exterminate all operating system abstractions. In *Workshop on Hot Topics in Operating Systems (HotOS)*, 1995.

[100] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. Obliviate: A data oblivious filesystem for intel sgx. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[101] Matthew Prince. Quantifying the Impact of "Cloudbleed". Cloudflare Blog, 2017. `https://blog.cloudflare.com/quantifying-the-impact-of-cloudbleed/`.

[102] John Graham-Cumming. Incident report on memory leak caused by Cloudflare parser bug. Cloudflare Blog, 2017. `https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/`.

[103] Z. Hu, L. Zhu, J. Heidemann, A. Mankin, D. Wessels, and P. Hoffman. Specification for DNS over Transport Level Security (TLS). RFC 7858, May 2016.

[104] P. Hoffman and P. McManus. DNS queries over HTTPS, (DoH).

[105] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, Symposium on Operating Systems Design and Implementation (OSDI), 2016.

[106] Raul Herbster, Scott DellaTorre, Peter Druschel, and Bobby Bhattacharjee. Privacy capsules: Preventing information leaks by mobile apps. In *ACM Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2016.

[107] Matthew Lentz. *Assurance and Control over Sensitive Data on Personal Devices*. PhD thesis, University of Maryland, 2020.

[108] Samel Weiser and Mario Werner. SGXIO: Generic trusted I/P path for Intel SGX. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2017.

[109] Cynthia Dwork. Differential privacy. In *Proceedings of the International Colloquium on Automata, Languages and Programming*, Venice, Italy, July 2006.