Achieving Keyless CDNs with Conclaves

Stephen Herwig, *University of Maryland;* Christina Garman, *Purdue University;*
Dave Levin, *University of Maryland*

https://www.usenix.org/conference/usenixsecurity20/presentation/herwig

This paper is included in the Proceedings of the
29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

Open access to the Proceedings of the
29th USENIX Security Symposium
is sponsored by USENIX.

Achieving Keyless CDNs with Conclaves

Stephen Herwig
University of Maryland

Christina Garman
Purdue University

Dave Levin
University of Maryland

Abstract

Content Delivery Networks (CDNs) serve a large and increasing portion of today's web content. Beyond caching, CDNs provide their customers with a variety of services, including protection against DDoS and targeted attacks. As the web shifts from HTTP to HTTPS, CDNs continue to provide such services by also assuming control of their customers' private keys, thereby breaking a fundamental security principle: private keys must only be known by their owner.

We present the design and implementation of Phoenix, the first truly "keyless CDN". Phoenix uses secure enclaves (in particular Intel SGX) to host web content, store sensitive key material, apply web application firewalls, and more on otherwise untrusted machines. To support scalability and multi-tenancy, Phoenix is built around a new architectural primitive which we call *conclaves*: containers of enclaves. Conclaves make it straightforward to deploy multi-process, scalable, legacy applications. We also develop a filesystem to extend the enclave's security guarantees to untrusted storage. In its strongest configuration, Phoenix reduces the knowledge of the edge server to that of a traditional on-path HTTPS adversary. We evaluate the performance of Phoenix with a series of micro- and macro-benchmarks.

1 Introduction

Content delivery networks (CDNs), like Akamai [1] and Cloudflare [2], play a critical role in making today's web fast, resilient, and secure. CDNs deploy servers around the world, on which they host their customers' websites. Because the web's performance is largely determined by latency [3], many websites rely on the fact that CDNs have proximal servers to nearly all users on the web to ensure low-distance and therefore low-latency connections.

While CDNs have grown more popular, so too has the movement towards an HTTPS-everywhere web. The majority of all websites are offered via HTTPS, and with the advent of free HTTPS certificate issuance [4], this number has grown increasingly quickly [5].

Unfortunately, HTTPS and CDNs are, in some sense, pathologically incompatible. To accept TLS connections, CDN servers store their customers' secret keys—in many cases, the CDN actually generates the keys on behalf of their customers [6, 7]. As a result, CDNs are imbued with a *huge* amount of trust: they could impersonate, eavesdrop on, or tamper with all of their customers, including virtually all of the world's major banks, online shops, and many government sites.

The messy relationship between HTTPS and CDNs is made all the more challenging by the fact that CDNs today do far more than merely *host* the bulk of the web's content. They also use web application firewalls (WAFs) to analyze clients' requests for evidence of targeted attacks like SQL injection or cross-site scripting, and filter them before uploading to their customers [8]. CDN customers benefit from this service because it scrubs attack traffic far from their own networks. And yet, running a WAF on a CDN requires the CDN to have access to the website's unencrypted traffic.

There have been recent advances to address aspects of this problem, most notably Cloudflare's Keyless SSL [9], which is a protocol that allows CDN customers to maintain sole ownership of their private keys. However, even with Keyless SSL, the CDN learns all session keys, yielding little additional assurance against eavesdropping or impersonation. The ideal solution would allow for all requisite processing and functionality to be performed on encrypted data, so that the CDN operator is neither responsible for holding the keys nor able to see any of the data through it. However, even the state of the art in this area [10–16] is much too inefficient to be utilized at the scale and performance that would be expected of a CDN.

In this paper, we introduce the design and implementation of *Phoenix*, the first truly "Keyless CDN". Phoenix uses trusted execution environments (TEEs, in particular Intel SGX enclaves) to perform all of the quintessential tasks of today's CDNs—hosting web servers, applying web application firewalls, performing certificate management, and more—all on otherwise untrusted machines.

Critical to the performance of any CDN is the ability to support multiple concurrent web servers and multiple tenants (customers). Unfortunately, no existing software infrastructures built off of SGX have been able to support multi-process, multi-tenant applications. We introduce a new general-purpose architectural primitive we call *conclaves*: containers of enclaves. Conclaves facilitate the deployment, configuration, and dynamic scaling-up and -down of sophisticated legacy (unmodified) applications.

Contributions We make the following contributions:

- We present the first truly "keyless CDN," which we call Phoenix. Phoenix performs all of the quintessential tasks

of today's CDNs, without requiring CDNs to gain access to sensitive key material, and without having to change legacy web applications.

- To realize our design, we introduce a new architectural primitive called *conclaves*, which creates a microkernel out of secure enclaves. Conclaves offer the abstraction of a "container of enclaves," thereby making it straightforward to deploy multi-process, scalable, legacy applications within a dynamic number of enclaves.
- 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 modest overhead (~2–3× for many configurations).

Roadmap We describe the essential features of today's CDNs and distill a set of goals and threat models in §2. We review related work in §3. We present the design of conclaves and of Phoenix in §4, and their implementation in §5. We present our evaluation in §6 and conclude in §7.

2 Problem and Goals

We distill down the fundamental features of today's CDNs, discuss the inherent security challenges, and formulate the goals and threat models that guide the rest of this paper.

2.1 Content Delivery Networks

CDNs 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 [6]. Historically, CDNs have been thought of as a massive web cache [17], but today's CDNs play a critical role in achieving the performance and security that the web relies on [8].

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 paper.

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 [5], 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 [6, 7]. 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 [6]. A recent protocol, Keyless SSL [9], has been proposed to address this; we describe this in more detail in §3.

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 [8] 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 [18]. WAFs analyze the plaintext HTTP messages, and compare the messages against a set of rules (often expressed as regular expressions [19]) that indicate an attack. Edge servers only permit benign data to pass through to the customer's origin server.

2.2 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 [6, 7]. 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 [6].

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 [8]. 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.

2.3 Our Goals

At a high level, our goal is to maintain all of the core properties of a CDN—low latency, key management, and resilience to DDoS and targeted attacks—without having to expose customers' keys or a client's sensitive information, and without requiring massive code changes from their customers. We distill our overarching goal down to five specifics:

1. **Protect private keys:** Support HTTPS, but without exposing the private keys corresponding to the certificate's public key to any edge server.
2. **Protect session keys:** Once a connection is established, do not expose the ephemeral session keys (nor the sensitive material for session resumption) to any edge server.
3. **Secure web-application firewalls:** Support edge-server-side WAFs, but without leaking plaintext messages to the server.
4. **Support multi-tenancy:** Be able to host multiple customers on a single machine (or even the same web server process), but with strong isolation between them.
5. **Support legacy customer applications:** Support all of the same web architectures of today, with minimal modifications to or impact on customer code.

These goals are a departure from today's CDNs, which store all of their customers' keys (at least the session keys), and operate on the plaintext of the client's data. Achieving these goals stands to improve websites' security, users' privacy, and also the flexibility in how edge-network services can be deployed.

2.4 Threat Models

An edge server is by definition a man-in-the-middle between the client and the origin server. Given such a privileged position, there is a wide range of potential threats. We define two threat models, the main distinction being who owns and operates the physical edge server, i.e., the level of control the CDN assumes over its hardware deployment. In both models we assume access to a trusted execution environment with the following features: isolation, trusted code execution, the ability to make calls into/out of the trusted environment, attestation, and cryptographic "sealing" of the data. This ensures strict isolation between customers' data, as well as strong protection for their keys, even in the event of node compromise, so long as the TEE remains secure. We will define these terms and expand upon the necessary TEE features in Section 3.2.

Honest but curious In the honest-but-curious model, the entity hosting the web server runs the software and protocols as specified, but tries to infer customer keys, client data, or cookies by observing traffic to and from the machine, and by inspecting any information leaked to the host operating system. This model applies when, for instance, the customer considers the CDN trustworthy and the CDN hosts its own hardware, but the customer is concerned about a rogue employee or administrator. Additionally, CDNs may adopt this threat model when hosting their own hardware so as to limit the exposure of their customers' data in the event of a software bug in the untrusted OS. Our goal would be to reduce an honest-but-curious attacker to have no more information than any on-path attacker (which HTTPS protects against).

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 web servers. Nonetheless, we assume attackers cannot violate basic assumptions of cryptography or trusted hardware, which we review next. 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 [8].

3 Prior Work

Here, we review relevant background and prior work in terms of how they have achieved the goals outlined in §2.3. There have been a variety of approaches that achieve a subset of our goals, but to the best of our knowledge, we are the first to achieve them all. See Table 1 for a comparison.

System	Protects private keys	Protects session keys	Secure WAFs	Supports multi-tenancy	Supports legacy apps	Additional deployment
Traditional CDNs				●	●	None
HTTP Solutions [17, 20]	❍					Javascript
TLS Solutions [9, 21]	●			●	●	Origin-side server
Crypto Solutions [14–16, 22, 23]	●	●	●	●		Client & server mods
TaLoS [24]	●	●			●	Trusted hardware
SGX libOSes [25–28]	●	●	●		●	Trusted hardware
TEEs and Middleboxes [29–35]	●	●	❍		●	Trusted hardware
Phoenix Conclave	●	●	●	●	●	Trusted hardware

Table 1: Prior work, grouped broadly by categories. To the best of our knowledge, the Phoenix Conclave is the first secure CDN to support multiple tenants and to provide secure web application firewalls without having to divulge customers' secret keys. ● denotes full support for a feature and ❍ denotes partial support.

3.1 TEE-less Solutions

HTTP Solutions Several systems have proposed that the origin server digitally sign their data [17, 20] or embed cryptographic hashes directly into HTML [36, 37], 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.

TLS Solutions Other approaches allow origin servers to retain ownership of their private keys by changing the server-side implementation of TLS. SSL Splitting [21] 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 [9] 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 [10–12]. 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.

Cryptographic Solutions One seemingly straightforward approach to solving this problem would appear to be fully homomorphic encryption (FHE) or functional encryption [22, 23, 38]. FHE allows one to perform arbitrary computations on *encrypted* data, without knowing any of the keys. However, even current state-of-the-art homomorphic encryption is much too slow for the performance that is required of a CDN and additionally would violate our goal of supporting legacy applications.

Various approaches [13–16] apply searchable encryption schemes to achieve functionality like deep packet inspection (DPI) while still maintaining the privacy of 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 CDN features we require.

3.2 Intel SGX (and Other TEEs)

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 will remain safely within it as well.

A wide range of TEEs are available today, with varying functionalities. We focus on Intel's Software Guard Extensions (SGX) environment, but note that any TEE with similar functionality discussed here and §2.4 would also be usable.

SGX Overview Intel's SGX provides a new mechanism for trusted hardware and software as an extension to the x86 instruction set [39, 40]. 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.

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 [41, 42]. Another feature is the ability to cryptographically *seal* data to

be used across multiple invocations of an enclave [42, 43]. SGX also provides such features as trusted time and monotonic counters [44, 45]. 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.

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 [24] simply 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. This approach protects the private keys and session keys, but does not address our goals of multi-tenancy or WAFs.

At the other extreme, SCONE [26] moves the entire C library into the enclave. Haven [25] and Graphene [27] 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 [28] extends the libOS from the SGX enclave to System Management Mode (SMM) by running device drivers in SMM memory.

CDN applications 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 and our extensions to Graphene in §5.

Other work [46] provides frameworks for developing *new* software that takes advantage of SGX, whereas our interest is in supporting *legacy* applications.

TEEs and Middleboxes A recent series of works have explored securing middleboxes by using TEEs, to provide DPI and intrusion detection [29, 30], as well as network function virtualization [31–35]. None of these systems handles the complete range of functionality required by CDNs, nor do they support multi-tenancy, to the best of our knowledge.

The most relevant works combining TEEs and middleboxes are Harpocrates [47] and STYX [48]. Harpocrates builds basic CDN functionality using a TEE and alludes to performing Keyless SSL-like functionality using trusted hardware but does not provide any details. In addition, Harpocrates does not seek to protect any derived key material and instead focuses solely on protecting the long term private key. STYX improves Keyless SSL by protecting private and session keys, but does not address secure WAFs or other CDN-type functionality.

Side-Channel Attacks on SGX We must address the recent rise of side-channel attacks against SGX, including the speculative execution attack Foreshadow [49, 50]. 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 [51, 52]. Regardless, Intel has released both microcode patches and recommendations for system level code that at the current time address Foreshadow and known related attacks [50, 53, 54]. There is also ongoing research to address both speculative execution as well as other cache-based side-channel attacks on SGX and in general [54–57]. We consider protections against such side-channel attacks to be outside of the scope of this work and rely on these defenses.

4 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 [25–27] 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 Conclaves Design

The conclave design extends a libOS to support shared state abstractions among multiple processes. Recall from §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 [27] 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 [58] or Mach-US [59], in which shared resource abstractions are implemented as a set of enclaved daemons shared by all processes in the system.

4.1.1 Conclave Kernel Servers

Using the NGINX web server 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[1]:

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 §4.2.2. 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 [9], 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 [60].

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 accepting expired certificates. Unfortunately, SGX itself does not provide trusted time. Its SDK [44] provides features [45] 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 [61].

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 §5, 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 (e.g., for specialized IPC).

4.1.2 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.

4.2 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 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.

4.2.1 Bootstrapping Trust

We first address how the conclave, viewed as a distributed system, establishes the trust of each member node, whether

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

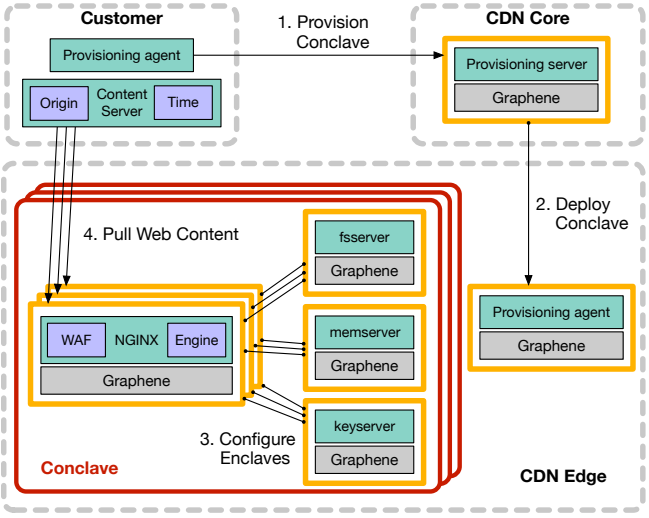

Figure 1: Architectural design of Phoenix. Multiple enclaves (yellow boxes) reside in a logical conclave (red boxes), permitting multiple processes and multi-tenant deployments. The CDN Edge and Core servers run on untrusted hosts.

kernel server or application process. This 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. [62], 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 are, mandatorily, over 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.

4.2.2 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.

4.2.3 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 [6, 7]; 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" [6], 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 [4] ACME protocol [63]—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 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.

4.3 Deployment Scenarios

Phoenix's conclave-based design permits a diverse range of deployment options to support varying threat models like those described in §2.4. 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 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 [64]) 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).

5 Implementation

We implement conclaves and Phoenix as extensions to the open-source Graphene SGX libOS [27]. 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.[2]

5.1 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 [65] 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 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 [66, 67]. 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[3]: `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 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

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

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

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 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 hardcoded 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 [68]; 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.

5.2 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 [69]. The patch is an optimization, similar to the solution proposed elsewhere [26, 70, 71], 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 [72] 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.

5.3 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

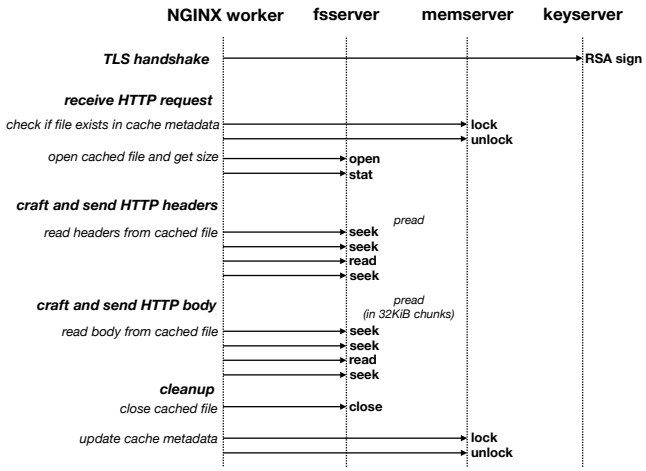

Figure 2: An NGINX worker servicing an HTTPS request for cached content, and the resultant kernel server RPCs.

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 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 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.

6 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.[4] 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). §6.2 presents a sensitivity analysis on the size of the shared memory zone.

6.1 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 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.[5]

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

[5] §6.5 shows that exitless performs better than standard ocalls for low-latency calls, but degrades for high-latency calls.

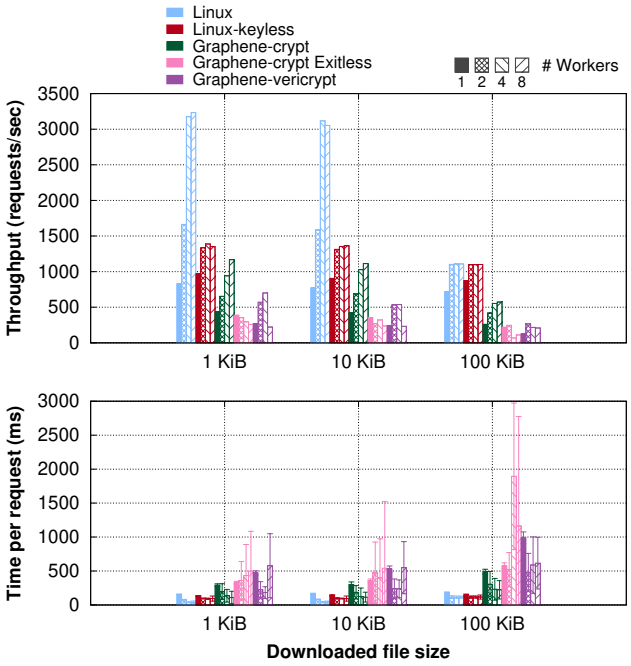

Figure 3: Throughput and latency for single-tenant configurations. The legend indicates the number of NGINX worker processes. We include the standard deviation of the latencies as error bars.

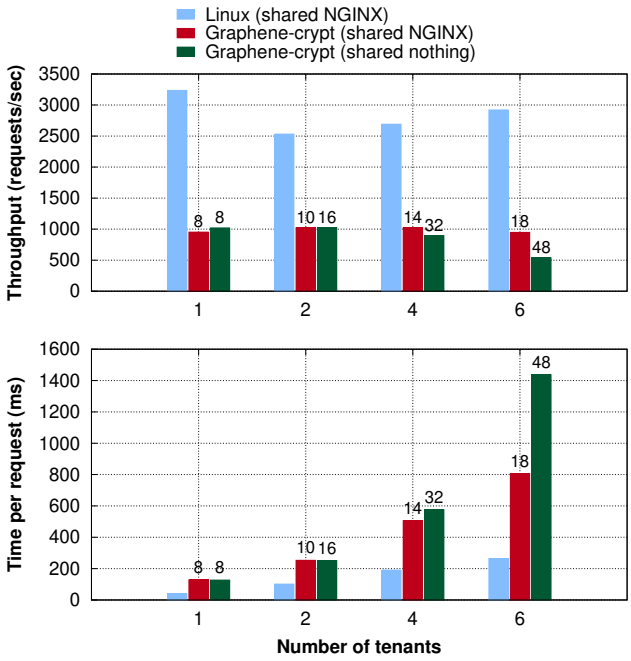

Figure 4: Multitenancy scaling. Throughputs are the aggregate throughput across all customers, and latencies are the mean latencies across customers. Above the bars, we indicate the number of enclaves in each configuration.

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 2: 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).

6.2 Single-Tenant

Figure 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 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 2 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.

6.3 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 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.

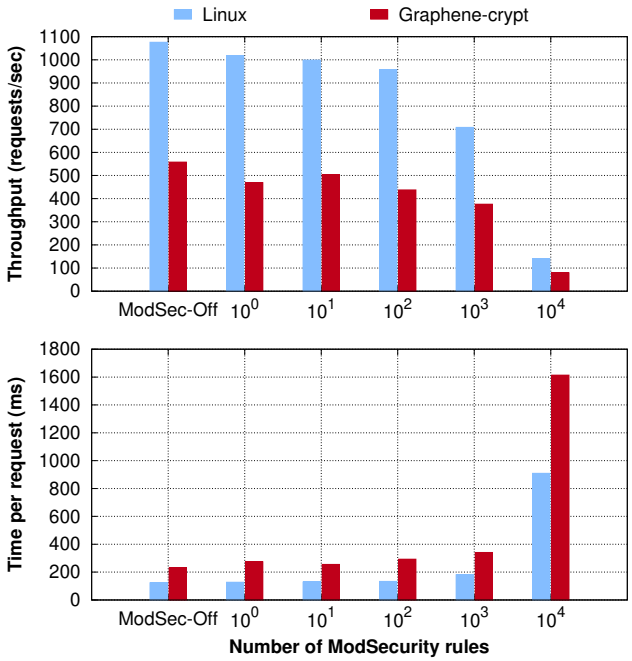

Figure 5: Effect of ModSecurity rule count on NGINX performance. NGINX runs with a single worker, and we fetch a 1 KiB file.

For the conclave deployments, we also measure the number of SGX paging events using the kprobe-based technique from Weichbrodt et al. [73, 74]. 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.

6.4 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 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.

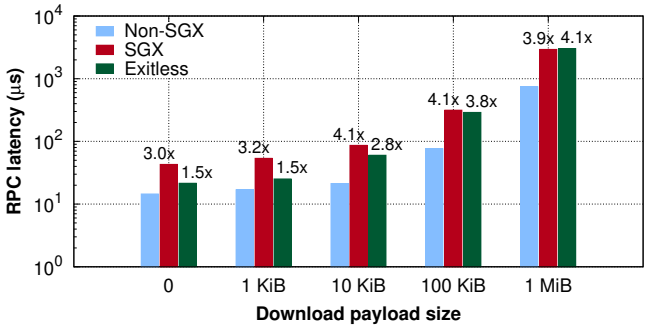

Figure 6: RPC latency versus payload size. The numbers above the bars are overheads compared to non-SGX.

6.5 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.

6.5.1 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 [6] 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 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, 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).

6.5.2 Kernel Servers

fsserver We use `fio` [75] 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`

[6] 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.

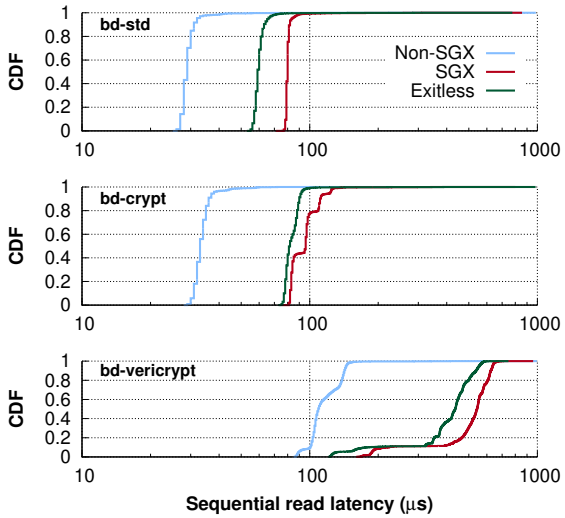

Figure 7: CDFs of read operation latency (µs) for a 10-second test that repeatedly reads 4096-bytes from a nextfs server, for each block device implementation.

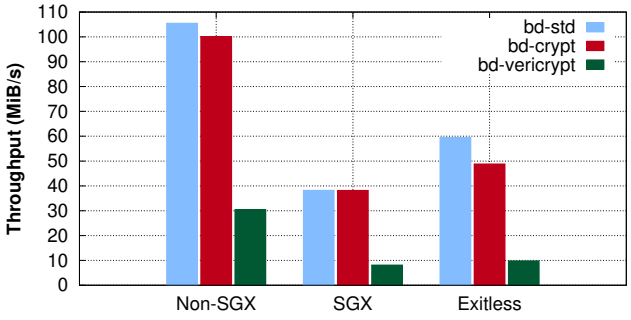

Figure 8: Total throughput for a 10-second test that repeatedly reads 4096-bytes from a nextfs server, for each block device implementation.

runs inside an enclave, uses exitless system calls, and invokes read operations from a single thread.

Figure 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 8 shows the associated throughput. For comparison, the enclaved versions of bd-crypt and bd-vericrypt have 20× and 97× less throughput, respectively, than Linux's standard ext4 filesystem (954 MiB/s, on our test machine).

memserver Figure 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

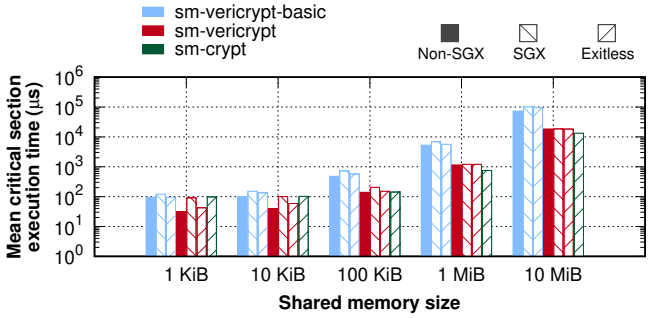

Figure 9: Mean wall clock time (µs) to process a critical section.

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

Table 3: Mean wall clock time (µs) to compute an RSA-2048 signature using default OpenSSL (left) and the keyserver. The last row is overhead compared to OpenSSL.

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.

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 3. The keyserver itself uses OpenSSL's default RSA implementation; compared to the RPC micro-benchmarks in Figure 6, we again see that the raw time overheads are consistent with the RPC latencies.

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, 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.[7]

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 [70, 71, 73] that puts this cost at 8000 cycles (3.077 µs on our test machine). The timeserver cost is dominated by the signature computation; exitless calls to the timeserver actually hurt performance, as,

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

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: Mean wall clock time (µs) 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.

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.

7 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 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 2 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, §6.5 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. To assist in future efforts, we have made our code and data publicly available at:

`https://phoenix.cs.umd.edu`

Acknowledgments

We thank the Graphene creators and maintainers, especially Chia-Che Tsai, Dmitrii Kuvaiskii, and Michał Kowalczyk, for their help in understanding Graphene's internals and debugging numerous issues. We also thank Bruce Maggs, Nick Sullivan, and the anonymous reviewers and artifact evaluators for their helpful feedback. This work was supported in part by NSF grants CNS-1816422, CNS-1816802, and CNS-1901325.

References

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

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

[3] Ilker Nadi Bozkurt, Anthony Aguirre, Balakrishnan Chandrasekaran, P. Brighten Godfrey, Gregory Laughlin, Bruce Maggs, and Ankit Singla. Why is the Internet so slow?! In *Passive and Active Network Measurement Workshop (PAM)*, 2017.

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

[5] 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.

[6] 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.

[7] 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.

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

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

[10] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego Lopez, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste. Multi-context TLS (mcTLS): Enabling secure in-network functionality in TLS. In *ACM SIGCOMM*, 2015.

[11] David Naylor, Richard Li, Christos Gkantsidis, Thomas Karagiannis, and Peter Steenkiste. And then there were more: Secure communication for more than two parties. In *ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2017.

[12] Hyunwoo Lee, Zach Smith, Junghwan Lim, and Gyeongjae Choi. maTLS: How to make TLS middlebox-aware? In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[13] 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 (ASIACRYPT)*, 2018.

[14] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *ACM SIGCOMM*, 2015.

[15] Sébastien Canard, Aïda Diop, Nizar Kheir, Marie Paindavoine, and Mohamed Sabt. BlindIDS: Market-compliant and privacy-friendly intrusion detection system over encrypted traffic. In *ACM Asia Conference on Computer & Communications Security (ASIACCS)*, 2017.

[16] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. Embark: Securely outsourcing middleboxes to the cloud. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[17] 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.

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

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

[20] 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.

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

[22] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *ACM Symposium on Theory of Computing (STOC)*, 2009.

[23] 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), 2016.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] Juhyeng Han, Seongmin Kim, Jaehyeong Ha, and Dongsu Han. SGX-Box: Enabling visibility on encrypted traffic using a secure middlebox module. In *Proceedings of the First Asia-Pacific Workshop on Networking*, 2017.

[30] Dmitrii Kuvaiskii, Somnath Chakrabarti, and Mona Vij. Snort intrusion detection system with Intel Software Guard Extension (Intel SGX). *CoRR*, 2018.

[31] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Safebricks: Shielding network functions in the cloud. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[32] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. Shieldbox: Secure middleboxes using shielded execution. In *Symposium on SDR Research (SOSR)*, 2018.

[33] David Goltzsche, Signe Rüsch, Manuel Nieke, Sébastien Vaucher, Nico Weichbrodt, Valerio Schiavoni, Pierre-Louis Aublin, Paolo Cosa, Christof Fetzer, Pascal Felber, et al. Endbox: scalable middlebox functions using client-side trusted execution. 2018.

[34] Huayi Duan, Xingliang Yuan, and Cong Wang. Lightbox: SGX-assisted secure network functions at near-native speed. *CoRR*, abs/1706.06261, 2017.

[35] Ketan Bhardwaj, Ming-Wei Shih, Ada Gavrilovska, Taesoo Kim, and Chengyu Song. SPX: Preserving end-to-end security for edge computing. *arXiv preprint arXiv:1809.09038*, 2018.

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

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

[38] Craig Gentry. Computing arbitrary functions of encrypted data. *Communications of the ACM*, 53(3), 2010.

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

[40] 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 *International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[41] 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.

[42] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based Attestation and Sealing. In *International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[43] 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.

[44] 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.

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

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

[47] Rufaida Ahmed, Zirak Zaheer, Richard Li, and Robert Ricci. Harpocrates: Giving out your secrets and keeping them too. In *IEEE/ACM Symposium on Edge Computing (SEC)*, 2018.

[48] Changzheng Wei, Jian Li, Weigang Li, Ping Yu, and Haibing Guan. Styx: a trusted and accelerated hierarchical SSL key management and distribution system for cloud based CDN application. In *ACM Symposium on Cloud Computing (SoCC)*, 2017.

[49] 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.

[50] 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, 2018.

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

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

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

[54] 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.

[55] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the

cache hierarchy. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[56] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In *USENIX Annual Technical Conference*, 2018.

[57] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

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

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

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

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

[62] 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.

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

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

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

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

[67] 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.

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

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

[70] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: ExitLess OS services for SGX enclaves. In *European Conference on Computer Systems (EuroSys)*, 2017.

[71] Ofir Weisse, Valeria Bertacco, and Todd Austin. Regaining lost cycles with HotCalls: A fast interface for SGX secure enclaves. In *International Symposium on Computer Architecture (ISCA)*, 2017.

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

[73] Nico Weichbrodt, Pierre-Louis Aublin, and Rüdiger Kapitza. sgx-perf: A performance analysis tool for Intel SGX enclaves. In *ACM/IFIP International Middleware Conference (Middleware)*, 2018.

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

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