Harpocrates: Giving Out Your Secrets and Keeping Them Too

Rufaida Ahmed Zirak Zaheer Richard Li Robert Ricci

University of Utah

Abstract

Content Distribution Networks (CDNs) offer websites and web services the ability to host content on servers that are near the edge of the network, close to users. Benefits of this arrangement include low latency, scalability, and resistance to Denial of Service attacks. Traditionally, CDNs have hosted primarily *static content*, but increasingly, there is an interest in pushing *active computation* to the edge as well. This active computation, which is similar in style to the "serverless" computing becoming popular in clouds, offers a wealth of new opportunities for web services to become faster and more scalable. With this opportunity, however, comes a much greater exposure to security threats. One is leakage of secret materials (such as keys, identities, etc.) that are accessed by these functions. Another is the possibility that sensitive calculations are not executed faithfully in the CDN; e.g. a modified version of the customer's code is run.

In this paper, we present the design of Harpocrates, a framework that allows active code to be pushed from an origin webserver out to workers at the edge of a CDN. Harpocrates makes use of Intel's SGX technology to keep data private, and presents an environment similar to the JavaScript `WebWorker` API to simplify the process of code that can run on either origin servers or the CDN. We use Harpocrates to design a number of interesting services, including a service that generates and checks secure cookies within the CDN, and a framework that protects against denial-of-service attacks in a way that is customized to a specific website. We show that the framework performs well enough to be deployable in practice.

1 Introduction

Content distribution networks (CDNs) are widely used to enhance user experience, reliability, and security through providing a system of distributed servers and networks that can deliver internet-based services to the user from locations at or near the network edge. Many CDNs [26, 3, 6, 22] work by "fronting" for an origin webserver. In this configuration, DNS lookups for the origin server resolve to IP addresses belonging to the CDN rather than the origin server itself. The CDN then acts as a reverse proxy or "man in the middle," processing all requests for the website and deciding whether to serve pages out of its cache, relay requests to the origin server, apply DDoS prevention methods, etc.

While this model is simple for the origin server, and provides strong protection against DDoS by hiding the true IP address of the origin, it leads to a number of security problems such as leakage of tenant data [19] and sharing of private keys between the content provider and the third-party hosting entity [14]. It has also, historically, been limited to static content, leaving the actual application code running on origin servers or in the cloud, where the latency benefits of caches at the edge are not realized.

Computing, however, is starting to become available at the edge. Cloudflare JavaScript Workers [20] are one example of "edge-computing" available in a CDN. The basic principle behind Cloudflare Workers is to allow origin servers to provide JavaScript code which will be run within the Cloudflare CDN itself, reducing latency for dynamically-generated content, and offering the origin server a simple way to scale. The high configurability offered with Cloudflare workers turns Cloudflare from just a CDN service into an edge computing platform.

Another example of this type of edge computing is AWS Lambda@Edge [9]. This service also allows the client to push code to the edge, allowing it to be closer to the end user to minimize latency. This code will typically be triggered by events from the Amazon CloudFront CDN. After pushing the code to the edge, AWS will take responsibility of code management duties such as replication, scaling and routing.

This arrangement increases security and privacy

concerns for the CDN's clients. Generally, CDN clients want to keep most of their code secured and protected against both the CDN and possible malicious entities access, and using such a facility requires the client to distribute both code and potentially sensitive data throughout the CDN's network.

In this paper, we present Harpocrates[1], a system that allows origin servers to distribute sensitive code and secret information throughout the world without having to worry about it being leaked. This enables secure computation at the edge, and will allow for faster, more responsive, and more dynamic web services. Harpocrates makes use of Intel's Software Guard Extensions (SGX). It provides an abstraction based on the JavaScript Worker model, which is widely used on both the client and server side (Node.js) of web applications; this enables ease of offloading to the CDN from either direction.

We claim that Harpocrates enables multiple interesting applications that can help improve CDNs by providing storage for more critical information on the CDN without comprising the privacy and security of data and code. To demonstrate the power of the framework, we designed four such applications. The first two use cases help keep the origin webserver available to legitimate users during Distributed Denial-of-Service (DDoS) attacks. They check secure login cookies set by the origin server to identify users who have legitimately logged in to the service; they use a secret provided by the origin server to establish the authenticity of the login cookies, and use Harpocrates to prevent compromise of that secret. The second application goes beyond simple checking of an existing cookie by adding an additional authentication step, handled entirely in the CDN and using the user's own password for the website. Our third use case allows aggregation of data from multiple sources to occur at the edge of the network, in the CDN, for scalability and latency reasons. The fourth is an adaptation of the Keyless SSL [18] protocol to allow the CDN to serve HTTPS requests for the origin without requiring access to the latter's private keys.

The rest of this paper is organized as follows: We begin by giving background on the technologies that we use in Section 2 and covering the related work. Our threat model, in particular the details of the materials that we seek to keep secret, is described in Section 3. The design of Harpocrates is covered in Section 4, followed by a description of several applications that it in enables in Section 5. Section 6 has some brief notes on the implementation, and Section 7 provides an evaluation of the overheads associated with using SGX in this setting and demonstrating the benefit to website performance of moving active code to the edge. We conclude and discuss future work in Section 8.

[1]Harpocrates was the ancient Greek god of secrets and confidentiality

2 Background and Related Work

In this section, we provide background on the technologies that we use and cover related efforts.

2.1 Intel SGX

Intel Software Guard Extensions (Intel SGX) is a set of extensions to the Intel architecture that are designed to provide integrity and confidentiality for application running in ring 3 from parties running in the privileged ring, including the operating system, BIOS, Virtual Machine Manager, etc. This allows users to run trusted code on untrusted servers. It gives users a high level of confidence that private data will be as secure in a Cloud or Edge Cloud as it would be running in a local trusted environment.

An SGX application consists of two parts: an untrusted part and a trusted part. The trusted part is also referred to as an "enclave". An enclave is an execution container instantiated by an untrusted application. After being instantiated, the untrusted part can issue an Enclave Call (ECALL) to switch into the enclave and start the trusted execution. Similarly, the code inside enclave can switch to the untrusted world by making an Out Call (OCALL). An ECALL is a call made into an interface function within the enclave, triggering the enclave to execute a piece of code after necessary security checking, while and OCALL allows code in the enclave to make use of outside services like system calls or other functions that requires privileged permissions [23].

Harpocrates mainly uses two features of SGX. One is its secure execution environment: data belonging to an enclave is presented as plaintext within the CPU, but this data is encrypted with integrity protection applied when it is flushed from the cache to DRAM. Because SGX flushes the CPU cache and other memory mappings when switching in or out of an enclave, not even a privileged party, e.g. the kernel, can see the plaintext of the enclave's memory: it cannot see the previous contents of the CPU's cache, and can only

see the encrypted copy of the content in the memory [2]. Any modification to the encrypted content in physical memory will lead to general protection fault when the trusted entity get scheduled to run again.

The other SGX feature Harpocrates uses is remote attestation. Because most trusted code requires some set of secret data to operate on, there must be a way to securely communicate that data to the enclave. Furthermore, it is critical that the party sending this secret data have a way to trust that it is indeed talking to specific, known, trusted code, and that this code is running in a real SGX enclave (and not, say, an emulation of SGX that will allow the attacker to access the data). SGX provides these facilities through a feature called "remote attestation" that enables an enclave to "attest" the identity and integrity of the code to a remote party. The Harpocrates design uses this mechanism to transfer secrets from the origin server into the enclave, and to give the origin server confidence that its functions are being faithfully executed.

2.2 TLS and Keyless SSL

The traditional TLS protocol is primarily designed to secure end-to-end communication. This was appropriate for older styles of web service when communication was only between two parties: the client (browser) and the webserver. However, when the connection is terminated in an intermediate node (such as a "reverse proxy" CDN) the security guarantees of these protocols no longer apply. In the CDN world, in order to provide HTTPS service, CDN providers that "front" for their clients domains needs to impersonate the original content provider by having the private key at hand, which brings with it security problems inherent in sharing a private key with a third party. To mitigate the problems brought by key sharing, Keyless SSL [18] was introduced by Cloudflare. Keyless SSL exploits the fact that private key is only used once in each TLS handshake to split the whole TLS handshake geographically, with most of the handshake work happening at the Cloudflare's edge. The private key itself remains on the origin server, which is contacted once per SSL/TLS handshake to identify the origin server. The security property of Keyless SSL has attracted significant attention and has been successfully applied in the wild [17]. However, Keyless SSL suffers from significant performance degradation and limited scalability due to the extra round trip from the CDN to Key Server in each handshake [35]. Researchers have sought alternative solutions to this problem and one of the main directions is using SGX to enable the private-key-holding sever to run in or near the CDN [35], eliminating the additional round trip.

[2]Recently, Spectre-like attacks have been discovered against SGX [15]. Like all other SGX-enabled applications, Harpocrates' security will depend on the development of effective countermeasures against these attacks.

2.3 Related Work

There is significant other work on exploiting SGX capabilities to enable trusted computation in the Cloud and Edge. For example, Haven [10], SCONE [7] and Graphene-SGX [33] make attempts to run entire containers in a shielded environment. mbTLS [24], ShieldBox [32] and EndBox [21] use SGX to protect middleboxes. This work makes all execution and data within the enclave trusted, and essentially only uses the third party infrastructure as a computation support. However, we argue that to make full use of the benefits conferred by a CDN provider, a tenant needs to expose some information to the CDN so that it can help the tenant optimize their performance and security, e.g. caching and DDoS prevention. Harpocrates targets a different point in the design space in which we do allow the CDN to see less-critical "derived secrets" while avoiding catastrophic incidents in which "master secrets" are leaked.

Bhargavan et. al [11] explore a means to optimize Keyless SSL itself, but does not consider computation need at the edge. mcTLS [25], Blindbox [29], and Splitbox [8] explore different ways to keep secrets from untrusted parties, but they emphasize the protection of content instead of of private key protection. STYX [35] does explore the space of key management, using SGX to enhance key distribution. Harpocrates differs from it in that we focus on pushing execution of code from the origin server to the edge, a broader use case than SSL key management.

3 Threat Model

The first element of our model is that the origin server has a "master secret" from which other secrets are generated, and that protecting this secret is higher priority than protecting the values that are derived from it. This situation is common in modern web services. For example, a master secret is often used to generate cryptographically secure cookies. Theft of an indi-

vidual cookie is undesirable, as it allows the thief to impersonate a user for the duration of the cookie's validity or until the theft is discovered and the affected user can have her cookie re-generated. Theft of the master secret, on the other hand, is catastrophic, as it allows the thief to *generate* correct cookies, and therefore impersonate *any* user, and it can only be remedied by regenerating *all* users' cookies. Similar situations arise with TLS/HTTPS: leaking the key for a specific session is problematic, but leaking the private key used to authenticate the server is much worse. Our goal in Harpocrates is to protect the master secret, as it is not always possible to protect derived secrets and still have the CDN do its job.

Second, we assume that the CDN needs to be able to see requests and responses "in plaintext" to do its job; without this, it cannot know whether to serve responses from its cache, whether to send requests to the origin server, whether to dispatch them to a secure function, etc. All of these decisions are based on information in the HTTP request, such as the request URL and headers (particularly cookies). This is why we only seek to protect master secrets: for example, if a secure function sends a cookie to a client, that cookie will be seen by the CDN in the next request the client makes. We do assume that the connection between the client and the CDN is, or can be made, over HTTPS; techniques such as Cloudflare's Keyless SSL [18] make this possible while keeping the origin's private key secret (though they may leak session keys).

Third, we assume that the CDN may leak any information that it sees in plaintext; this includes the contents of HTTPS connections that are decrypted by the CDN through mechanisms like Keyless SSL. This may be due to a bug, such as in the case of CloudBleed [19], in which Cloudflare inadvertently leaked memory belonging to one origin in bytes of connections for other origins' clients. It could also be due to a malicious insider at the CDN or a malicious third party that has compromised the CDN.

Finally, we assume the correctness of Intel's SGX and remote attestation protocols. While there do exist attacks against SGX [27], we assume that mitigations are available or that they are closed in future revisions of SGX.

4 Design

The fundamental element of security in Harpocrates comes from leveraging Intel SGX. SGX supports code secrecy by putting sensitive code and data into CPU-hardened protected regions called enclaves. Intel also provides a remote attestation mechanism which can prove to others that they are really communicating with the specific, known, code in a real SGX enclave, and not an impostor. As part of this process, the remote party can provide data (master secrets, in our design) that can only be decrypted inside the enclave.

Harpocrates, following the SGX Developer Guide [23], splits code into two parts: *untrusted* code, which runs outside of an SGX enclave, and `trusted` code that runs inside of it. All entry points are in the untrusted code: the CDN delivers requests that it receives from clients to untrusted functions, and expects to receive a reply from an untrusted function as well. Once invoked, an untrusted function may choose to call a trusted one: for example, to perform operations involving secret data.

We offer an API based on the JavaScript `WebWorker`, an abstraction that allows a caller to start a task by *posting* an event and registering a function to receive the *completion* of that event. These two calls map well to the ECall and OCall interface offered by SGX. An ECall is made from untrusted code, and invokes code inside of an enclave; this is analogous to posting an event. An OCall is made by trusted code inside an enclave, and calls the untrusted code registered to receive the completion. Another advantage of adopting the `WebWorker` design is that it is frequently used in server-side JavaScript in the Node.js environment, and thus facilitates moving code from an origin server to a CDN. It is also used in Cloudflare's `Workers`.

There are some differences between our design and the "normal" `WebWorker`. We require the user to supply a function in the trusted code that will be called on initialization of the enclave: typically, this code will perform remote attestation with the origin server to securely transfer a secret(s) into the enclave. In addition, the untrusted code cannot call arbitrary functions when it posts an event: it can only call functions that the customer has defined for the enclave and exported to be available for ECalls. Untrusted code may still register arbitrary functions to receive completions.

Figure 1 shows a typical CDN operating in "reverse proxy" mode. Note that though we depict the CDN as a simple "stack" of servers, in practice, it is typically a set of servers distributed around the world, and clients are directed to the closest one through DNS resolution or IP anycast. The basic goal of the CDN is to use the path represented by the solid arrows for as

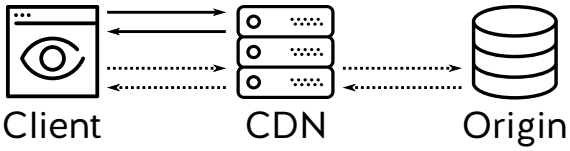

Figure 1: Typical "reverse proxy" CDN. Solid lines represent the CDN serving content out of its cache. Dotted lines represent the CDN forwarding traffic to the origin, and forwarding back the results.

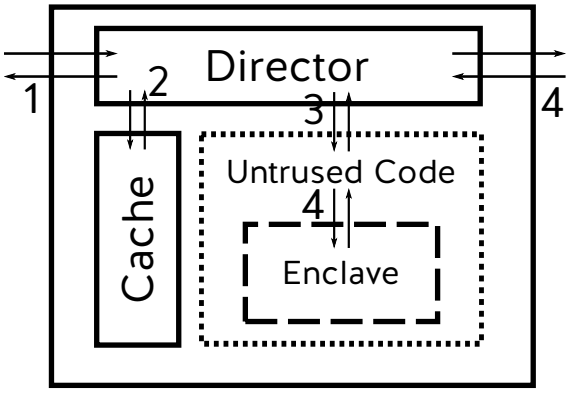

Figure 2: The design of Harpocrates. The client is to the left, and the origin server is to the right.

many requests as possible: these are the requests for which the CDN, sitting near the user at the edge of the network, can respond out of its cache. The dotted lines are essentially a fallback: when the content is dynamic or not yet cached, the CDN must forward the request to the origin server, and relay the response back to the client.

Figure 2 shows the data flow within the CDN in Harpocrates. Requests enter on the left from clients (marked #1 in the diagram) and reach the *director*. This director examines the HTTP request (acting as the endpoint of the TLS session for HTTPS connections), and selects one of several options. First, for static content, it may serve the content directly out of the CDN's cache (#2). Second, if the request URL belongs to an endpoint registered by the origin server's code, it forwards the request to the *untrusted* part of the extension's code (#3). This is the portion of the code supplied by the user that does not run inside of an SGX enclave and does not handle master secrets. In some extensions, no secret data may be needed, and this function returns data to the director (which passes it back to the client) without calling into an enclave at all; this looks similar to Cloudflare's Worker model. For computations on secret data, the extension calls into the enclave (#4) by posting a SecureWorker event. Computation on the secret data—such as secure cookie generation—occurs in the enclave, which returns the data to the untrusted code, and from there, back to the director and client. Finally, the director still has the option of sending requests that cannot be handed by either the cache or local code back to the origin server (#5).

The CDN is responsible for deciding when and where to bring up new instances of the customer's function, along with the associated enclave. Each enclave must have an initialization function; typically, this function will contact the origin server, use remote attestation to authenticate itself, and be given the master secret through this secure channel. While this bootstrapping procedure does put some additional load on the origin server, it only needs to happen once for each host in the CDN network, which is much lower than the total number of end clients.

An important point that developer should keep in mind is the fact that secrets and sensitive information should not be statically compiled into the enclave, and must only be securely transferred to the enclave at runtime during remote attestation. Code and data used to initialize an enclave are, by design, unencrypted in SGX. Any secret loaded before attestation can possibly be inspected by the unauthorized CDN or a malicious party.

5 Example Applications

Using a content delivery network (CDN) to host scripts, files, and even sensitive information that are frequently accessed can improve the overall performance of the origin website and conserve bandwidth. But unfortunately, using CDNs also comes with a risk. If an adversary gains access to these files he can inject arbitrary malicious content to change or even replace those files.

According to [14], it was reported that 76% of all organizations that use third-party hosting services such as CDNs share at least one of their private keys with this entity. Sharing such crucial data is risky, because the origin site has to trust not only the CDN's software, but also the CDN's system administrators, all those who have physical access to the provider's hardware, and any law-enfrocement body that might have authority to access the origin site's replicated data in the provider's physical location.

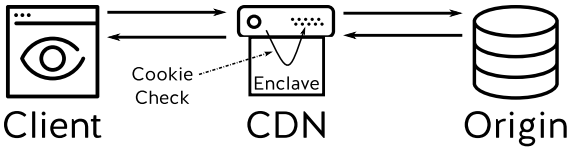

Figure 3: Simple checking of a login cookie in an enclave in the CDN

We now describe four applications that could be built using Harpocrates, the benefit they would have to users and origin sites, and the way that they fit into our security model.

5.1 Secure Cookie DoS Prevention

The most basic service that reverse proxy CDNs offer to their customers—after caching—is the ability to withstand denial of service attacks. Because end hosts—including attackers—see the CDN's IP address, rather than the origin's true location, all traffic, benign and malicious, passes through the CDN. The CDN aims to have enough capacity, in terms of bandwidth and servers, to withstand any DoS attacks that may be directed at its customers.

For static (cachable) content, the attack becomes against the CDN's resources and is up to the CDN to handle appropriately. For dynamic content, however, the problem is trickier: normally, requests for this content would be forwarded to the origin server, which would customize the response for the session, user, etc. Forwarding *all* requests for dynamic would clearly expose the origin server to the DoS attack, but the origin would often like to allow some class of requests—such as those coming from logged-in users—to be forwarded. Offloading these decisions to the CDN is problematic, as in order to make them securely, the CDN will need some way of identifying valid requests through, e.g. a cookie set in the request, and checking this cookie requires a secret.

In Harpocrates, this tension is resolved by allowing the origin to write a function that checks login cookies. These cookies are generated by the origin server on successful login using a widely-used method of applying a cryptographic hash to a concatenation of the username, a nonce, an expiration date, and a master secret. These cookies are quick to generate and check; all information other than the secret is sent in plaintext in the request, so all that must be done to check the validity of the cookie is to concatenate the secret and check that the hash of this string matches the hash in the request. This master secret can be generated on the origin server, and shared with the enclaves running in the CDN. The untrusted code supplied by the origin to the CDN can parse the request, extract the necessary fields from headers, etc., and pass the appropriate fields to the trusted code to test. The trusted code, with access to the master secret, simply returns success or failure depending on whether the cookie validates, and the untrusted code takes the appropriate action to block or forward the request. The path of an individual request is shown in Figure 3.

5.2 Site-Specific DoS Authentication

The procedure above works so long as the adversary does not have access to a legitimate, valid cookie. If he does have one, such as by compromising a legitimate client, he can distribute this cookie to all attackers and pass the security check, allowing attack traffic to reach the origin server. Our next scheme protects against this case. When there is some indication that a particular cookie might be involved in an attack (e.g. evidenced by seeing the same cookie in many requests), we can enter a more cautious mode in which possessing a login cookie is not enough: one must also have a cookie that is tied to the user's IP address to prevent the same login cookie from being used by an entire DDoS botnet.

Login cookies are not typically tied to a particular IP address, because users are often on DHCP, behind NAT, move between networks, move between mobile and WiFi networks, etc. When one wants to verify that the user behind a particular IP address is a person, rather than a bot, a typical way to do so is is through a CAPTCHA [34]. In many cases, today's reverse-proxy CDNs issue CAPTCHA tests to clients coming from IP addresses that they consider suspicious. A major problem with this arrangement is that this CAPTCHA test is essentially conducted "outside" the website; e.g. it provides a confusing user experience by serving up a page from the CDN rather than the origin website, asks the user to input information that they are not used to providing the website, etc.

Our scheme works much more cooperatively with the original website, as shown in Figure 4. When the regular login cookie checking function at the CDN decides that a login cookie might be being abused, it serves the client a page—clearly coming from the origin website, but served from the CDN—requesting that the user re-enter their password. This is shown as interaction #1 in the figure: the request contains

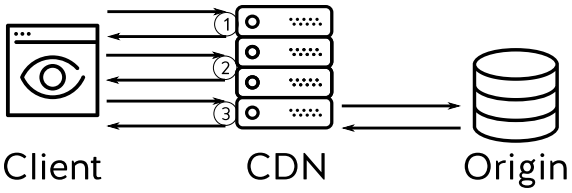

Figure 4: Exchanges when the enhanced protection mode is enabled. For visual simplicity, individual calls into the enclave are not shown.

a valid login cookie, but not an IP-binding cookie. The login cookie is checked as in Figure 3, and if it is correct, the worker in the CDN returns a page asking the user to re-enter their password, instead of forwarding the request to the origin server.

The user login database is shared from the origin to the enclaves: enclaves can easily contain tens of megabytes of memory, sufficient to hold login credentials for hundreds of thousands of users. The user's response is received by the origin's function on the CDN, where the password is checked inside the enclave (here, the password database is the master secret), and if successful, a new secure cookie, similar to the one used for normal logins is generated. This cookie, however, is tied to a particular client IP address. Thus, any bots attempting to use it from other IP addresses can be quickly and easily rejected within the CDN; requests coming from the correct IP address can be forwarded to the origin. If the user does roam to a different IP address, they will simply have the slight inconvenience of having to enter their password again. This is shown as interaction #2 in the figure: here, the request contains the user's password, the enclave checks that password, and, if the password is correct, the worker returns the IP-bound cookie and a redirect to the page the user was originally trying to access. In interaction #3, the request now contains a correct IP-bound cookie, and the function within the CDN checks this cookie and forwards the request to the origin.

Note that that this new authentication procedure takes place within the CDN: the origin server sees no additional load from the requests to re-enter passwords, whether successful or not. This solution therefore scales with the CDN, and this additional level of protection comes without additional load on the origin. The end user only sees the additional password dialog once (unless she or he changes IP address while the attack is ongoing).

5.3 Content Aggregation at the Edge

One of the documented use cases for Cloudflare's workers is aggregating information from multiple points within the CDN [16]. In this configuration, the worker grabs information from multiple sources and creates a unified page to serve back to the user. This information can come from third-party APIs such as weather, mapping, stock, and social media services.

In many cases, such services are accessed by having an "API token" that belongs, in this case, to the origin server, and which is used to authenticate to the API. We treat this as a "master secret." For some APIs—particularly, those dealing with financial or personal information—the information being handled is very sensitive, and disclosure of the key can be very dangerous. For example, consider a bank which has a relationship with a credit reporting agency, which it uses to display credit scores to the bank's customers: displaying this information within the bank's website is a valuable service, but the bank must take great care to ensure that the key that it uses to authenticate itself to the credit agency is not compromised.

In Harpocrates, the origin's API keys can be kept within the enclave; this allows the origin to benefit from the latency and scaling advantages of aggregating information at the edge, without needing to worry about compromise of the keys it uses to gather that information.

5.4 Keyless SSL

Keyless SSL [18] is a design that allows a CDN to serve HTTPS connections with the origin's SSL/TLS certificate without needing to have access to the origin's private key. It does so by having a small server that runs (in the original design) on the origin, which keeps the private key and is contacted during TLS session establishment to perform the cryptographic operations that authenticate the server. Others have shown that the server holding the private key can be run within an SGX enclave, and that there are substantial latency and scalability benefits to doing so [35]. Harpocrates can be used as an alternate way to implement the keyless server.

6 Implementation

To implement Harpocrates, we use an existing package called SecureWorker [2] that allows JavaScript

workers to run inside the trusted environment provided by an Intel SGX enclave. The execution environment outside of the enclave (for untrusted code) is provided by Node.js, a popular engine for server-side JavaScript. Node.js is much too large and has too many dependencies to be practical for running inside of an SGX enclave, so SecureWorker uses the Duktape [4] JavaScript engine to provides a familiar `Worker`-like environment inside the enclave. Duktape is designed to be embedded in other environments, and is therefore very lightweight and has few dependencies. SecureWorker allows users to write isometric code and use the same code on client, server, and inside enclaves, it presents the secure workers as just another, secure and trusted, component in the JavaScript-based architecture. The SecureWorker package provides a rich API in which we can start a new worker, terminate it, receive and send messages between the trusted and untrusted part of the system, and even perform remote attestation services.

The original SecureWorker package provided wrappers for a subset of the JavaScript webcrypto API [5]; however, this subset was small, and only covered a few symmetric key ciphers and has functions provided by early releases of the Intel SGX SDK. Because we expect cryptography to be the main use case for trusted code, we extended these wrappers to cover public key cryptography and a larger set of ciphers and hashes from the OpenSSL library.

For the sake of this prototype, we focused on the design aspect of the system and did not include the full remote attestation in our implementation. The full remote attestation process requires a license from Intel, a signed certificate from a recognized certificate authority, and a registered service provider ID [1]; this does not add to the proof-of-concept value of our implementation.

7 Evaluation

We evaluate the feasibility of our approach by showing that it does not cause undue overheads on worker functions; showing the latency benefits that are to be gained from moving these workers to the edge; and showing the utility of the system by presenting a detailed case study showing how to build a website-specific DDoS defense mechanism.

Our evaluation was done in the CloudLab [31] testbed. All hardware SGX evaluation was done using an Intel `i5-6260U` processor.

7.1 Microbenchmarks

In general, small functions that do not use large amounts of memory execute at a similar speed within an SGX enclave with outside of ones [13]. We found this to hold for the types of functions needed for many fingerprinting and secure cookie schemes: we ran a SHA-256 hash of 20 bytes of data using the JavaScript `js-sha256` library. Outside of the enclave, this function was run in Node.js; inside, it was run using the Duktape JavaScript engine. In both cases, the actual hash implementation is native code. Outside of the enclave, we found that it took, on average, 0.0017ms (averaged over 10,000 repetitions) and 0.0029ms (averaged over 100,000 repetitions), while the same hash, run entirely within an enclave took only 0.0041ms on average (averaged over 100,000 repetitions). We found that the gap between the execution speed outside and inside the enclave gets smaller, in relative terms, as we increase the input message size. From this, we can conclude that for many small functions, running inside SGX will not, itself, introduce overheads that are significant in comparison to network RTTs.

There is, however, another source of overhead that we must consider: crossing the trust boundary to enter and exit an enclave *does* have a noticeable performance effect. We assume that workers implemented in our system will have two such boundary crossings: one ECALL made when the untrusted code posts an event that causes execution of the trusted code, and one OCALL when the trusted code posts results back. Under this assumption, the effect is still within reasonable range compared to wide-area network delays: when we re-factor the function above to put the testing loop in the untrusted code and the hash itself in the trusted code, the average time is 0.9ms. This is significant, but still well under the typical network round trip time between a client and origin server.

We do note that initialization of the enclave adds additional latency: in our experiments, this amounted to an extra 16ms, and would be much higher with fully implemented remote attestation. This is a one-time cost per CDN server, and CDNs have two choices for how to handle it. One would be to implement SGX enclave initialization on-demand and to shut down an enclave after some period of inactivity. This would keep overheads low by preventing the CDN from having to maintain many enclaves, but would result in occasional higher latencies seen by clients (e.g. the first to use the origin from a particular geographi-

cal region). Another would be to proactively and predictively bring up enclaves; this could reduce the occasional latency spikes, but would require more sophisticated workload prediction and could mean more resources spent maintaining inactive enclaves.

7.2 End-to-end Benchmark

For the deployed topology configuration, we emulated link latencies for the user-to-CDN and CDN-to-origin links. In [12], the authors found that a typical round trip time between a client and the Cloudflare CDN was 16ms; we approximate this in the emulated topology using a 10ms (one-way) delay between the end user and the CDN. (We do not model the time it takes for the client to find the closest CDN server.) For the link between the CDN and the origin server, we used a 30ms (one-way) delay, approximating an intercontinental link of about 9,000 km.

The function we used for this evaluation is a simple one which calculates a secure one-time user cookie for the end user. The primary computation done by this function is the hash described above. The request made for this experiment is minimal, as is the response from the server.

From the client to the origin, passing through the CDN, the time to retrieve the result averaged 118ms. (This includes the time to set up the TCP connection, send the request, compute the secure cookie, etc.) When the function is moved to the CDN, the time is only 38ms, a reduction of almost 2/3. The exact benefit in practice will, of course depend on how close the CDN is to the end user (the closer, the higher the benefit), how close the origin server is to the CDN (the farther, the greater the benefit), and the amount of time the function takes to execute (long functions may dominate network RTT).

7.3 Case Study

To illustrate how Harpocrates can be used to implement one of the applications from Section 5, we show snippets of code from our implementation of the cookie-checking DOS protection.

Listing 1 gives the general sense of what the untrusted portion of the code for this function does. Lines 1 and 2 set up the SecureWorker package, loading in the binary (`enclave.so`) and Javescript (`cookies.js`) that implement the trusted portion of the cookie checker; recall that the binary run inside of the enclave is the lightweight Duktape JavaScript engine. Lines 4–10 are the function that will receive the completion event from the untrusted code; we omit the body of this function, which simply forwards the request to the client or drops it depending on the message sent by the trusted code. Line 13 is where the message is posted to the secure worker; here, the username, nonce (to prevent replays), and expiration are available in plaintext in the request, as is the cookie.

Listing 2 shows the trusted code that runs in the SGX enclave in response to the `postMessage` from the untrusted code. Line 2 performs the concatenation of the data passed in with the secret In a full implementation, this secret would be obtained (just once) from the origin on enclave initialization via SGX's remote attestation; our prototype does not implement remote attestation. Line 3 hashes this concatenated value, and line 4 checks this concatenation against the cookie sent by the client and sends a completion event to the untrusted code via `postMessage`.

8 Conclusion and Future Work

In this paper, we have presented Harpocrates, a system for moving computation from origin webservers to nodes in a CDN network, which typically reside at or near the network edge. Using SGX enclaves, our system allows the origin webserver to place "master secrets" in these enclaves, giving them strong assurance that these secrets will not be leaked even if the CDN needs to be able to "see" data derived from them in order to do its job. By using JavaScript and an API similar to the WebWorker API, we make it easy for the CDN's customer to move computation from the server to the CDN. We demonstrate that the overheads inherent in this arrangement are not high, and that there is significant benefit to end users in the origin being able to offload secure computations to the CDN servers near them.

Harpocrates is designed for applications with small amounts of trusted code and small master secrets. This is due to the limited enclave size provided by SGX and the costs associated with crossing the enclave boundary. These have performance implications for bigger applications, due to page swapping or the need to build multiple enclaves with secure communication channels between them. We look to other work that has focused on running larger, more intensive applications in SGX [28, 7] for ways to alleviate this limitation in the future.

Listing 1: Partial code listing for untrusted portion of cookie checking function

```
var SecureWorker = require('./lib/real.js');
const worker = new SecureWorker('enclave.so', 'cookies.js');

worker.onMessage(function (message) {
    if (message.result) {
        // Forward request to origin
    } else {
        // Drop request
    }
});

// Parse username, nonce, expiration, and cookie from request
worker.postMessage({name: username, n: nonce, exp: expiration, c: cookie});
```

Listing 2: Partial code listing for trusted portion of cookie checking function

```
SecureWorker.onMessage(function (message) {
    var hashstring = message.name.concat(message.n,message.exp,SECRET);
    var hashvalue = crypto.subtle.digest({name: 'SHA-256'}, hashstring);
    SecureWorker.postMessage({result: hashvalue == meassage.c});
});
```

In the future, we propose to further improve performance by leveraging Intel QuickAssist Technology (QAT) [30]. QAT-based accelerators can speed HTTPS connection services by securely offloading TLS operations on private keys to the NIC.

Although our proposed design improves overall performance and scalability by eliminating the need for the long distance connection to the origin server, the scalability in the current prototype can present a problem because of the need for a separate SGX enclave for each origin server. This problem can be mitigated with a method to guarantee clean separation between different CDN's clients data.

The use cases we present and possible applications are to demonstrate the power of our proposed prototype, but this prototype is certainly not limited to those use cases. We also expect to implement additional cloud-related applications that require both secrecy and high performance.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Numbers CNS-1419199 and CNS-1314945, and a fellowship from the University of Utah School of Computing.

References

[1] Intel software guard extensions remote attestation end-to-end example. https://software.intel.com/en-us/articles/intel-software-guard-extensions-remote-attestation-end-to-end-example, 2016.

[2] SecureWorker. https://github.com/luckychain/node-secureworker, 2017.

[3] Cloudflare - the web performance & security company. https://www.cloudflare.com/, 2018.

[4] Duktape. http://duktape.org/, 2018.

[5] Web crypto API. https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API, 2018.

[6] Amazon. Amazon CloudFront. https://aws.amazon.com/cloudfront/, 2018.

[7] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *OSDI'16*, pages 689–703, GA, 2016. USENIX Association.

[8] H. J. Asghar, L. Melis, C. Soldani, E. De Cristofaro, M. A. Kaafar, and L. Mathy. Splitbox: Toward efficient private network function virtualization. In *Proceedings of the 2016 workshop on Hot topics in Mid-*

dleboxes and Network Function Virtualization, pages 7–13. ACM, 2016.

[9] AWS. Lambda@edge. https://aws.amazon.com/lambda/edge/, 2018.

[10] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with Haven. In *OSDI '14*. USENIX – Advanced Computing Systems Association, Oct. 2014.

[11] K. Bhargavan, I. Boureanu, P.-A. Fouque, C. Onete, and B. Richard. Content delivery over TLS: A cryptographic analysis of Keyless SSL. In *Security and Privacy (EuroS&P), 2017 IEEE European Symposium on*, pages 1–16. IEEE, 2017.

[12] E. Bos. Analyzing the performance of Cloudflare's anycast CDN, a case study. In *27th Twente Student Conference on IT*, July 2017.

[13] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza. Securekeeper: confidential zookeeper using intel sgx. In *Proceedings of the 17th International Middleware Conference*, page 14. ACM, 2016.

[14] F. Cangialosi, T. Chung, D. Choffnes, D. Levin, B. M. Maggs, A. Mislove, and C. Wilson. Measurement and analysis of private key sharing in the https ecosystem. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 628–640. ACM, 2016.

[15] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. Sgxpectre attacks: Leaking enclave secrets via speculative execution. *CoRR*, abs/1802.09085, 2018.

[16] Cloudflare. Aggregating multiple requests. https://developers.cloudflare.com/workers/recipes/aggregating-multiple-requests/.

[17] Cloudflare. Keyless SSL: The nitty gritty technical details. https://blog.cloudflare.com/keyless-ssl-the-nitty-gritty-technical-details/, 2014.

[18] Cloudflare. Overview of Keyless SSL. https://www.cloudflare.com/ssl/keyless-ssl/, 2014.

[19] Cloudflare. Incident report on memory leak caused by Cloudflare parser bug. https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/, 2017.

[20] Cloudflare. Cloudflare worker. https://www.cloudflare.com/products/cloudflare-workers/, 2018.

[21] D. Goltzsche, S. Rüsch, M. Nieke, S. Vaucher, N. Weichbrodt, V. Schiavoni, P.-L. Aublin, P. Costa, C. Fetzer, P. Felber, et al. Endbox: Scalable middlebox functions using client-side trusted execution. In *Proceedings of the 48th International Conference on Dependable Systems and Networks. DSN*, volume 18, 2018.

[22] Google. Google Cloud CDN - low latency content delivery. https://cloud.google.com/cdn/, 2018.

[23] Intel. Intel software guard extensions developer guide. https://download.01.org/intel-sgx/linux-1.7/docs/Intel_SGX_Developer_Guide.pdf, 2016.

[24] D. Naylor, R. Li, C. Gkantsidis, T. Karagiannis, and P. 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.

[25] D. Naylor, K. Schomp, M. Varvello, I. Leontiadis, J. Blackburn, D. R. López, K. Papagiannaki, P. Rodriguez Rodriguez, and P. 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.

[26] E. Nygren, R. K. Sitaraman, and J. Sun. The Akamai network: a platform for high-performance Internet applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19, 2010.

[27] D. O'Keefe and J. Tian. SGXSpectre: Sample code demonstrating a Spectre-like attack against an Intel SGX enclave. https://github.com/lsds/spectre-attack-sgx, Jan. 2018.

[28] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein. Eleos: Exitless OS services for SGX enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 238–253, New York, NY, USA, 2017. ACM.

[29] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. *ACM SIGCOMM Computer Communication Review*, 45(4):213–226, 2015.

[30] X. Shuai, L. Yao, and Z. Wang. QAT: Evaluation of a dedicated hardware accelerator for high performance web service. In *Advanced Communication Technology (ICACT), 2018 20th International Conference on*, pages 277–280. IEEE, 2018.

[31] The CloudLab Team. The CloudLab website. https://cloudlab.us.

[32] B. Trach, A. Krohmer, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer. Shieldbox: Secure middleboxes using shielded execution. In *Proceedings of the Symposium on SDN Research*, page 2. ACM, 2018.

[33] C.-C. Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, page 8, 2017.

[34] L. von Ahn, M. Blum, N. J. Hopper, and J. Langford. CAPTCHA: Using hard AI problems for security. In *Proceedings of The International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, May 2013.

[35] C. Weng, J. Li, W. Li, P. Yu, and H. Guan. STYX: A trusted and accelerated hierarchical SSL key management and distribution system for cloud based CDN application. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, Sept. 2017.