Dynamic and Composable Enclave Measurement

Evgeny Pobachienko

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-102
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-102.html

May 14, 2024

Acknowledgement

I thank Dayeol Lee for his mentorship in exploring research areas, working with Keystone, design discussions, and the research process. I also want to thank Prof. Dawn Song, my research advisor for her guidance and feedback throughout my research.

Lastly, work on moving ELF loading into inside the enclave and out of the host for it was started by Catherine Lu, related to her work in \cite{keystone_sharing}. This work is, however, entirely separate from Lu's work.

Dynamic and Composable Enclave Measurement

by Evgeny Pobachienko

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Dawn Song

Professor Dawn Song
Research Advisor

4/30/24

(Date)

* * * * * * *

Chris Fletcher

Professor Christopher Fletcher
Second Reader

5/14/24

(Date)

Dynamic and Composable Enclave Measurement

Evgeny Pobachienko
University of California, Berkeley
evgenyp@berkeley.edu

***Abstract*—An enclave is an execution environment isolated from the rest of the system, including the OS, providing security and privacy guarantees. The technology is maturing and seeing more adaptation in large and security-niche products, for security and confidentiality, but is still too difficult to use for wider adoption to occur. Specifically, the trust derivation from a measurement of the loaded memory proves incompatible with the design of modern applications because applications are redeployed with different workloads, load resources gradually, can be optimized by using available dependencies, etc. This leads to workarounds, inefficiencies, and unnecessary complexity.**

We introduce Dynamic and Composable Measurement – following a design paradigm shift to the measurement securely relaying a collection of resources to be used instead of blindly capturing exact runtime state. The report takes on the abstraction of guaranteeing that only these resources can be used, independently of how and when they are delivered. This approach is especially helpful for dealing with resources that vary across instances, like dynamic libraries, inputs, and configurations; or that come from mutually distrusting providers. The measurement design is modular and implementation-agnostic, without having any side effects on trust assumptions. New use cases of enclaves become feasible thanks to new capabilities.

I. Introduction

Enclaves protect and isolate the execution of a program and its private contents from the OS, applications, and many physical attacks. Their popularity has been growing greatly with new use-cases being developed on top of enclaves, and cloud providers providing support for enclaves on both x86 platforms – Intel SGX & TDX and AMD SEV-SNP [1] [2].

However, the usability of these platforms is lagging behind and slowing adoption. Linear measurement of enclave apps used by all Enclave Platforms is the core cause for most remaining issues that make enclave app development more difficult than regular applications. The Enclave Platform produces measurements of the initial enclave state because it needs to prove that the untrusted host had set it up correctly. We define **linear measurement** as a single hash of the entire enclave memory contents from start to end; the most important implications are that any small change requires redoing the entire measurement, and the measurement is tied to the necessary runtime view of all initial resources.

Prior work has improved the application environment to closer match regular applications, but not much is being done to support actual application lifecycles and deployment needs.

The main issues caused by this (discussed in detail in Section 3) are as follows:

- Managing expected measurements is hard due to frequent library updates and many possible target configurations.
- The library selection at deployment time is inflexible, which prevents host-side optimizations to decrease cost, deployment latency, run time, and vulnerable library trust revocation recovery latency.
- The trust abstraction imposes unnecessary restrictions on implementation because it's tied to initial state and runtime factors, causing any novel trust relationships to be complex to support.

Managing expected measurements is often very difficult for many applications because any variable configurations and inputs must be provided and installed by custom code by app developers. Microservice and cloud platform trends exacerbate this issue because the number of dependencies and therefore frequency of updates continues to rise.

Both meeting required demands such as important updates, and brute-force approaches to optimizations are exponential in cost. This is because each library will have some number of updates and some number of candidates, so the average time to next important update shrinks exponentially and the number of total combinations grows exponentially. Each measurement must start over from scratch by forming and hashing the entire runtime view of the initial state, so the constants on the exponential costs are significant.

This thesis addresses these issues by designing and building Dynamic and Composable Measurement. Dynamic means that multiple different measurements can satisfy the same expected measurement and policy, and composable means that when the set of components is changed, updating the expected measurement is cheap and does not re-hash any components that have been hashed before.

The design is flexible, platform-agnostic, detached from enclave implementation and runtime view, and is able to describe arbitrary application structure fitting all developers' needs. Specifically, we make the following contributions:

- Measurement is dynamic: developers can match enclave instance variants to enclave app. Measurement is composable, simple, and flexible: it captures granular information and resources and is detached from enclave build, runtime view, and host optimizations.
- Dynamic negotiation of flexible dynamic libraries and dependencies: host can propose a complete enclave app instance, and the developer can securely decide if the proposal is acceptable.
- Exclusion of non-defining resources from enclave identity: enclaves can have the same sealing key for stable private storage.

- Private third-party resources: enclaves can be securely provided with private binaries, and the measurement prevents equivocation of provided resources.
- Open-source implementation: an implementation is available online and is being reviewed in PR's to be merged with the Keystone Enclave Platform [3]. Work on relevant features is in progress.

II. Background

A. Threat Model

This work seeks to maintain the same guarantees as current enclaves while providing new functionality and better design. The guarantee is that if the enclave is set up correctly and no denial of service occurs, it passes attestation, runs correctly, and does not leak any data.

Permitted attacks are based on a malicious host with physical access to the machine and control over the software. The host can mount physical attacks by reading, sending, and modifying signals and data not inside the SOC with the CPU securing the enclave. The host can mount software attack by manipulating the OS, user applications, and network.

Denial of service attacks are not currently possible because a malicious host can simply cut power to the computer or network hardware. Side-channel attacks are an active area of research as no comprehensive full-performance solution doesn't exist, and defenses can be done in hardware and in application software [4] [5] [6].

B. Enclave Basics

An enclave provides an execution environment fully isolated from the rest of the system, including the OS. It is accomplished by having a trusted manufacturer supply the CPU firmware with functionality that track enclave contexts and limit access to sections of memory to only the enclave context. The CPU also manages switching into and out of the enclave context, so the execution of the enclave is correct because the in-memory code cannot be accessed by the CPU when not running the owning enclave. This allows a developer to build an application and request that a host deploy it on their machine, without trusting the host.

Trust is established through a measurement of the initial enclave state. The host delivers developer resources to a section of memory and calls a CPU procedure to keep that memory for the enclave once it's done setting up. The CPU blocks off all access to that memory, creates the enclave to own that memory, and measures the entire enclave memory. It signs the measurement with a hardware-embedded key provisioned by the CPU manufacturer, and the host sends this to the developer to prove that the enclave is in the correct initial state. This procedure is called Remote Attestation, and the Attestation Report also includes measurements of the hardware platform revisions and a generated public key for the enclave instance. Because the enclave is in the correct initial state, it will also execute correctly.

A typical usage that the developer writes an application that uses only the features available on a particular Enclave and Framework combination, generates the expected measurement, sends the plain-text binary and dependencies to an untrusted host. The host sets up the enclave, and the enclave starts running and remotely attests to the verifier. The attestation report includes a public key from a public-private pair securely generated by the platform from the measurement, so the enclave and the verifier are able to establish a secure, encrypted channel. If the verifier accepts the attestation report, it facilitates the enclave getting a developer-provisioned key or proof that the enclave is trusted. Finally, the enclave requests private data payloads to work on over the secure channel with the proof attached, or decrypted by the key.

Intel SGX is the most widespread enclave technology, and the first fully isolating, but has the most limitations such as supporting only user-level privilege mode inside the enclave. The development of this is ongoing, with new features, such as allocation of memory after initial start, being bundled into SGXv2 [7]. AMD SEV-SNP and Intel TDX are VM enclaves, providing a VM-like experience that is protected through the same concepts [8] [9]. ARM Trustzone is not used much as an Enclave Platform due to very weak isolation [10].

Figure 1 shows an example architecture for an Enclave Platform, the open-source Keystone platform. Here, the Security Monitor manages enclaves, memory access, and measurement. The Runtime is responsible for serving syscalls by passing them through to the host OS.

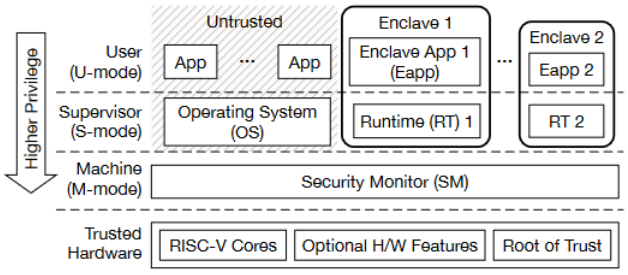

Fig. 1. Keystone Diagram [11]

C. Measurement

When developing an enclave app, the developer will form the expected measurement after finalizing its contents. The developer sends the contents to a host to deploy, the host loads the binaries and files into a memory region, and signals to the Enclave Platform to take this region to create an enclave. The Enclave Platform blocks access to this region of memory for everything except the enclave, and hashes the memory contents. It saves the enclave measurement to an attestation report in Enclave Platform's memory, which also not accessible by the OS. The attestation report includes additional information, like CPU firmware version, a generated enclave public key, etc. The enclave app can request this report and send it to a verifier to attest itself and receive private data payload through an encrypted network channel. The exact details of various platforms are summarized in [12].

The measurement implemented in all current platforms is what we will call a **linear measurement**: a single hash of the entire enclave memory contents from start to end.

There are a lot of alternatives to the core design as summarized in [13], but the problems presented in our work are not address here.

D. Sealing Key

The Sealing Key is an enclave primitive available in all Enclave Platforms. The procedure call mixes together a hardware-embedded manufacturer key with the enclave measurement to create an encryption key. This encryption key can be used for secure local storage accessible to all instances of this same enclave, since they all will measure the same. This provides easy key that persists across enclave restarts and does not need to be managed and provisioned by the developer.

A highly important use of the sealing key is when the enclave receives data from a third-party that wants to remain private to the developer. This creates secure storage with the trust vested in the code itself, not the developer. Successful use-case examples of this are Signal providing contact search without ever reading a user's contacts [14], and Opaque providing a platform where mutually distrusting (i.e. due to legal reasons) parties can compute on pooled data [6].

III. Motivation

All current Enclave Platforms use **linear measurement**: a single hash of the entire enclave memory contents from start to end, for Remote Attestation. This leads platforms to hash loaded binary files, meaning all expected measurements require re-creating the exact memory view. Furthermore, changing any part of the memory will require redoing the entire expected measurement from scratch. These properties lead to a myriad of issues, which are as follows.

Managing expected measurements is difficult due to frequent library updates and many possible target configurations. Linear measurement means we will have an exponential number of final hashes, and calculating each is expensive. The number of dependencies swells very high due to libraries directly ingested relying on other libraries themselves, and the average frequency of both mundane and critical updates will grow exponentially. A typical application can exceed 100 total dependencies easily, as the average count of dependencies within 254 popular Java packages is 14 [15]. The ubiquitous logging library Log4j alone has 141 dependencies. This is exacerbated by emerging microservice and cloud application industry trends, as functionality is split into more separate components that are still updated often. The most popular web app package manager is affected by the microservice trend, leading to the average count of dependencies climbing to 77. Cloud providers are also dividing their offerings into small components, leading the AWS Java SDK reaching 331 dependencies. Lastly, this also requires developers to supply per-run arguments such as arguments, options, and data payload after the enclave starts. This goes against typical deployment flows where the worker node will receive the application and simply modify the command to point to the per-run files, and means that developers must implement non-standard methods for retrieving and installing the options and files. Beyond run configuration, the host platform may vary a lot in OS, OS version, and CPU architecture, leading to many possible required combinations of dependencies. An example use case is if Netflix wanted to use enclaves to protect streaming of copyrighted materials from piracy. Netflix would have to manage all library builds that could possibly be needed, and the exponential number of expected measurements due to different required library combinations. Even without using enclave technology, Netflix fails to tackle the complexity today. Full HD (1080p) or better on desktop is supported only on Windows-Edge, Windows-Chrome, Mac-Safari, and Mac-Chrome clients [16]. So, no OS gets the reasonable quality on Firefox, and no browser at all gets it on Linux. Chrome is the only browser that plays Full HD Netflix content on multiple platforms, but it can't do 4K (2160p) – only Windows-Edge and Mac-Safari can.

The library selection at deployment time is inflexible, which prevents host-side optimizations to decrease cost, deployment latency, run time, and vulnerable library trust revocation recovery latency. The inflexibility is due to the exact match requirement of linear measurement. This makes it unlikely that cost is reduced by enclaves sharing libraries, deployment latency by cache avoiding large downloads, and run time by host selection of platform-optimized libraries. Downloading all libraries is a significant deployment latency because modern applications use many common libraries for many subroutines, and sizes are large: the aforementioned AWS Java SDK is over 300 MB [17], and a popular linear algebra Python library NumPy is 70 MB itself and relies on OpenBLAS which is another 30 MB [18]. Notably, NumPy can also by configured to use other, proprietary platform-optimized linear algebra backends by publishers like Intel, which can automatically be selected by hosts if available. Revocation is slow because the following needs to happen to revoke and recover: the verifier and private data serving components are updated to stop accepting the old measurement, the running enclaves are shut down, a new enclave app is built and the corresponding expected measurement is re-generated, new enclave app is downloaded to host, and enclave app is restarted and re-attested. So, the build process, host, enclave app, verifier, and data serving are all involved.

The trust abstraction imposes unnecessary restrictions on implementation because it's tied to initial state and runtime factors, causing any novel trust relationships to be complex to support. All components of an enclave app are public to the app developer because the app developer needs to be able to form the expected measurement which is the hash of the enclave memory. Due to this, dependency providers cannot provide private builds for developers to use inside of enclaves, and have to instead deploy users' workloads as data instead of regular code. It could instead be possible that a dependency provider runs an enclave collocated on the same host that serves and protects the dependency, thus providing

full native performance and flexibility in how it's used. Linear measurement prevents this because there's no mechanism to extend the trust chain to this, other than custom enclave app code. There is a split of trust between what's testified to by the measurement, and the correctness of implementation for getting the correct private data retrieval. This puts the burden of correct setup on custom code developers have to write. This also undermines use cases where end users do not trust the app developer, but do trust the open-source app and the Enclave Platform. If the payload data was attested to in the measurement, and the enclave code could send an attested done message, the user would be confident that the data payload was actually processed. It is difficult for mutually-trusting enclaves to work together because sealing key changes on any change to the initial state, so the same enclave with different initialization-time arguments cannot access sealed data. Furthermore, fine-grain policies based on initial app arguments are difficult due to cost of expected measurement generation. There are many works that have had to implement lots of functionality on top Linear Measurement to solve their trust settings, including Signal Contact Discovery [14], Opaque [6], MAGE [19], and Ryoan [20].

IV. Related Work

Some works tackle some novel trust relationships arising from enclave trust design described before, but most issues are not addressed by any works.

A. Attestation

There has been work attempting to solve some of the above issues through dynamic, but still linear measurement.

In Securing Remote Policy Enforcement by a Multi-Enclave based Attestation Architecture [21], Niemi et al. tackle the problem of 2 distrusting parties providing private arguments into a public code template, but don't address any other issues, and deployment is slow due to re-creating a full linear measurement every time. The parties share a public template that has parameters variables. A deployer enclave, open-sourced, is created and is supplied with private data from one party and the values for the parameters defining what exactly to do with the data from the other. The template insures that the parameters don't cause any data to leak. The deployer enclave generates the expected measurement of the worker enclave for the request, and runs and validates the worker enclave. Our work makes the deployer enclave a lot easier.

MAGE [19] implements mutual enclave attestation. Each enclave must include trusted expected measurements, including itself, in its initial state, which is a self-reference. To avoid this, the enclave is split up into a main enclave part, and a trusted measurements set part. After startup, the enclave generates the expected measurement by extending all items in the trusted set by the trusted section. This solves only mutual attestation, but demonstrated an online-generated (but not dynamic) measurement, and somewhat composable measurement by having a checkpoint to extend from. Our work natively splits up the measurement into sections, so we support the same design, and can even extend it by including the entire verifier to allow arbitrary policy verification.

Various works change attestation to solve problems not in scope of this work. Opera [22] cuts Intel Attestation Service out of the Intel SGX attestation critical path, allowing for custom and developer-scalable attestation; notably, Keystone does not have problems addressed by Opera. Confidential Attestation: Efficient in-Enclave Verification of Privacy Policy Compliance [23] generates proofs that attest to no leaks of developer designated data in developer annotated binaries. Intel SGX Quote [24] introduces more custom fields for the developers to specify in the attestation report, but does not have an effect on the actual measurement.

B. Enclave Frameworks

Enclave Platforms generally come with SDK's, making it easier to develop. Enclave Frameworks seek to make adoption and usability easier by further simplifying the underlying mechanisms. Open Enclave aims to help create platform-agnostic code base by providing a single interface that uses different backends; it supports Linux and Windows, and Intel SGX and preview of ARM Trust-Zone [25]. Gramine is meant to allow regular applications to be built into enclave applications without modification by providing a library OS [26].

Gramine for Intel SGX doesn't solve any of the above problems, but takes a step in the composable measurement direction. It uses a customized Graphene user-space library OS to service syscalls. It includes support for file reads and dynamic libraries. These are secured by a manifest file included in the initial enclave state as well as the Gramine libOS, which then reads and loads files into memory, comparing the read hashes to expected. It has some limitations due to being user-space only and not being able to modify page table permissions. The inclusion of the manifest file ties the measurement to the exact configuration of the enclaves' dependencies. For remote attestation, Gramine requires an Intel SGX Quote Quoting enclave to be ran by the host [27].

Academic extensions to Gramine are also done, such support for devices through memory mapping and `ioctl` calls [28]. They also supply the Gramine manifest file with the expected devices, which shows the advantage of having a flexible resource collection. If there's platform support, measured items can even be enforced, such as a confidential computing enabled GPU.

Some Function as a Service (FaaS) work has been done on taking advantage of enclaves naturally fits what we will call the **Blank-Slate Worker (BSW)**. We define it as a stable worker base that is ran as the initial enclave, and which then accepts the enclave app, commands, and resources sent by the developer through a secure channel. A minimal base must have some runtime, TLS termination, and ability to load and start applications; a bigger base could include anything that is shared across many enclave instances and changes rarely, however, which is the case for FaaS.

S-FaaS [29] manages a pool of attested Blank-Slate Workers by attesting them as they come up. When a workload comes, the (regularly-attested) Key Distribution Enclave generates keys and gives them to the worker enclave and code and data providers. The providers send resources through secure channels based on the keys, and can then be confident that the worker will execute their task correctly and privately. This avoids the providers from having to deal with complexities and measurements, and provides a faster attestation and setup flow than the regular multi-round. The runtime is trusted and provides extra functionality like measuring and attesting to resource usage of the enclave. Other works had similar circumstances where there was a large, stable, shared runtime that was used as the base, with smaller apps and data being what actually changed from task to task: SCL [30] also uses a FaaS runtime, but with efficient shared communication and storage between enclaves, and Ryoan [20] uses a more general sandbox that prevents untrusted app code from leaking private input, allowing output only in a few sandbox-protected ways.

BSW eliminates the difficulty with managing expected measurements, because the base doesn't change often, and everything else including application code, dependencies, configurations, and data are supplied post-measurement; essentially, we no longer rely on the measurement for correctness, but rather the worker code correctly responding to secure communication. However, the other issues are not addressed: developer must still manage all possible required library versions, there cannot be optimizations performed by the host or host cache, and novel trust relationships are still unsupported. Some of these issues are tempered somewhat by the fact that in FaaS, the runtime being ready to go is the biggest savings to be had, but are still important.

BSW itself doesn't support trust settings where the data provider is a different entity from the code provider and doesn't trust the code, or when there are even more mutually-distrusting resource providers. Ryoan in particular implements another layer of measurement to support private but untrusted code with private data: it performs linear measurement on the code itself to attest to the data provider for non-equivocation, and has further protections from data leakage. Lastly, outside of FaaS, splitting resource distribution across two stages with no flexibility on moving things around makes developing deployment hard due to the complexity of managing retries and various bad system states is hard; furthermore, the deployment split is very uncharacteristic for modern deployment flows.

Gramine is the most mature and feature-rich Enclave Framework, and Blank-Slate Worker is very light-weight, so we be comparing our work against these.

C. Non-Standard Trust Settings

These arise when there are multiple entities providing resources to an enclave instance, with different entities possibly wanting different properties to be ensured about privacy and correctness. Works here generally implement sandboxes or additional measurements within the existing linear measurement, and design only for their particular trust setting.

Private data pooling: multiple data providers may want to pool data for processing, but no one entity can be trusted with it. In Opaque [6], there are multiple hospitals or banks who have critically private data, but want to train models and do data analysis on pooled data to make better predictions on health issues and for fraud detection. Opaque provides a sandboxed data analysis platform in enclaves, where the sandbox is known to all parties and performs data analysis and parameterizes requests to prevent output from being too revealing about the input. The data analysis enclaves attest to all data providers, then each data provider establishes a private channel and sends data over it, and finally all data providers can make requests. Data is encrypted at rest.

Privacy-conscious users: e2e systems can take advantage of enclaves when work must be done on plain-text data. In Signal Contact Discovery [14], the message app users send their contact list to Signal-managed enclaves that find what phone numbers are associated with a Signal account, and the communication is e2e encrypted between the enclave and the trusted Signal client. Generally, on-premise requirements from privacy-sensitive industry entities can also be replaced with enclave-attested code.

Service component authentication via measurements: distributed systems can rely on attestation reports instead of a key distribution component managing key distribution and rotation. MAGE [19] implements mutual enclave attestation.

Host as enclave user: a content or code provider may want to protect their intellectual property by serving them into enclaves on user machines. However, this means that users will want some guarantees that what runs on their machine is safe for their machine and respects user resources. This can be provided by measurements that report additional information about either the components or properties of the whole application, but even a single hash is useful for at least non-equivocation. In Ryoan [20], private consumers can rent their resources out and have the Ryoan sandbox attest to resource usage to ensure they are paid it correctly, as well as limiting what the application can do.

D. Dynamic Library Works

Dynamic libraries inherently introduce composability to the enclave contents, but existing dynamic library works only focus on getting the base functionality. Gramine supports dynamic libraries but not sharing them, motivated by modern applications not being compiled statically [26]. Multiple works explore sharing specifically, by having multiple applications in the same enclave and rely on a trusted runtime to isolate them [31], or to have a dummy enclave control the library memory and allow that memory to be mapped as execute only to untrusted other enclaves [32]. None of the works propose a negotiation procedure to avoid requiring exact version and build of libraries, which makes sharing unlikely as each enclave may require a slightly different version or build.

V. Design

A. Design Principles

The design should be able to fully describe a modern application and reflect developer-perspective semantics. The key insight is that files should be measured, and set up post-enclave start by standard tools for full interoperability. To this end, our design will resemble an in-memory filesystem.

We have some important desired properties:

- The measurement is efficiently Dynamic and Composable, and easy to construct.
- The measurement is completely detached from the implementation of enclave functionality.
- The design is flexible to fit arbitrary use cases.
- Measurement is simple and achieves a small TCB inside Enclave Platform.

To detach the measurement from implementation and initial state, we identify 2 dimensions to what type of component we are measuring: a component can be present in the initial state or only in the future, and a component can be defining for the enclave, or is unique to this instance and therefore shouldn't affect the sealing key.

B. Measurement

We seek to measure everything about an enclave app. To make the measurement composable, we will view a single enclave app as mostly a collection of files. The measurement should prove that the enclave will run correctly, however, so it needs to relay how this is going to be ran as well. The measurement will signify one of the components as the start component, the component to whose start the CPU PC will be set. This component must be resident because it needs to be present for the enclave to start and be able to accept absent resources later. Furthermore, Enclave Platforms may be able to enforce some runtime properties, so we will also record facts like amount of memory given to the enclave.

So far we have all enclave initial state aspects. Notably, we changed the initial state of the enclave from what it needs for functionality or implementation to what the correct identity representation is. However, the measurement is a superset of this; we will also allow components that are not yet present. To do this, we will include a set of allowed hashes, so the enclave application is able to enforce that it will receive further resources that match the measurement.

We will place arrays of hashes and component pointers at a designated location in the enclave, so that the filesystem can be read both for measurement, and from inside the enclave app to find files and expected hashes. Placing files inside the array would make freeing loaded components expensive. The resulting structure and memory layout is summarized in Figure 2.

All resources need to have a name so that they can be used by the application. `type` is a bitvector to be used by developers as needed, for instance for file read write execute permissions. Offset from enclave bundle base is used because various code reading this structure will be in physical and different virtual address spaces. The measurement will include the name, type, and actual contents or hash, while offset and size are excluded because the offset is a runtime property of the memory layout and the size is implicit from the file itself. The name and type are needed because the components will be presented as being that, so we need to ensure that the mapping is not tampered with maliciously. The resulting structures are shown in Figure 3.

```
struct enclave_bundle {
    uintptr_t runtime_arr, id_res_arr,
        id_abs_arr, res_arr, abs_arr,
        data_start; // point to below
    resource_value_t runtime_values[];
    resource_ptr_t identity_resident[];
    resource_hash_t identity_absent[];
    resource_ptr_t resident[];
    resource_hash_t absent[];
    byte data[];
};
```

Fig. 2. `enclave_bundle` holds arrays of hashes of absent resources and pointers to present resources.

```
struct resource_ptr_t {
    uintptr_t offset;
    uintptr_t size;
    uintptr_t type;
    char name[64];
};
struct resource_hash_t {
    hash_t hash;
    uintptr_t type;
    char name[64];
};
struct resource_value_t {
    uintptr_t value;
    uintptr_t resource;
};
```

Fig. 3. Resource Definitions

To actually construct the measurement, we iterate over each array in-order and advance a single hash by the hash of each component. An important consequence is that whenever the component set changes, we can reconstruct an expected measurement in $O(\text{count of components} \cdot \text{hash size})$ when we already know component hashes, which is very fast compared to $O(\text{total size of components})$ of linear measurement. We chose this over a Merkle Tree to make the CPU firmware measuring code execute in constant memory, and for simplicity; this would speed the expected measurement to $O(\text{count of updates} \cdot \log_2(\text{count of components}))$ which is not a worthwhile gain. Lastly, measurement from scratch is parallelizable, but doing so in CPU firmware would complicate the TCB too much, and is not significant enough to optimize for in builds.

When producing the measurement, we contribute identity components to both the measurement and the sealing key, and non-identity components to only the measurement.

Because the sealing key is not affected by non-identity components, we must protect against malicious hosts that may

supply arbitrary data to non-identity resources in an attempt to leak data encrypted with the sealing key. Possible attacks include the host replacing a dynamic library with hooks into a script that dumps all encrypted data, and changing the configuration to trust a bad certificate, allowing the enclave app to output results to the host instead of the developer. There are 2 possible defenses: 1) enclave app must wait for verifier to approve it, and the verifier must validate that the entire setup is safe, or 2) only the most benign resources can be made non-identity, such as input data and mode selection. The first option is much better, and it resembles existing mechanisms, like Apple servers approving OS installations. The start resource is required to be identity because 1 becomes impossible without it, and by definition in 2 it must be identity.

C. Host and Developer Interactions

The Enclave Platform sends just a single hash as the measurement, and because the host can select some values like flexible libraries and inexact memory allocations, the verifier needs to be able to figure out what went into the hash. The host sends the `resource_hash_t` for all resources, and the verifier validates that all the proposed resources match what is desired and allowed. Then, the verifier constructs the expected measurement based on this proposal, and checks that the enclave measurement it receives matches the claimed and approved proposal.

Dynamic libraries can be proposed by the host. Once the untrusted host receives a request to locally deploy an enclave, the host examines all resident files given by the developer for deployment, and any ELF files it finds it ensures that dynamic libraries are present for. If some dynamic library is not provided, the host will choose an available library and add it as a resource to the enclave bundle. It can be added as either resident or absent, with absent allowing for potential lazy loading of the resources.

The host can choose to support proposing other types of dependencies, such as Python packages. This functionality is not implemented in this work, but should rely on standard dependency requirement specifications, like `requirements.txt`.

In either case, these choices will be sent through to the verifier, which will verify that the given hash matches a valid, trusted library build. This means that the developer doesn't need access to the actual library, only to hashes reported by the resource publisher.

There are TOCTOU issues with library proposals, and remote attestation in general, because an old exact enclave app version may be found to be insecure. When the verifier issues the approval, it should also securely send server time and security policy version, which is a strictly increasing number, incrementing whenever at least one library version becomes insecure. The enclave app should communicate periodically with the verifier to shut down if necessary to protect data that is already retrieved, and include the security policy version on any data retrieval to prevent new data from entering insecure enclaves.

At this point, we can build a deployment flow of an enclave application, shown in Figure 4.

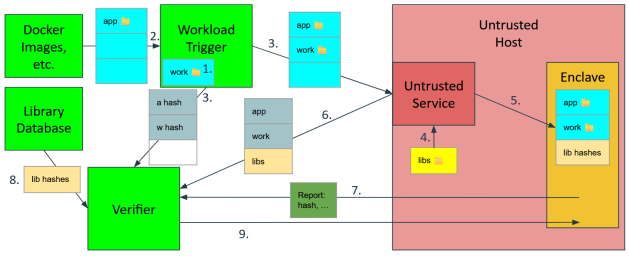

Fig. 4. Deployment. 1. New work or run is needed. 2. Application to service the work is retrieved. 3. The work is copied in and added to the `enclave_bundle`, and sent to a cloud service. The request is kept track of for validation and retry. 4. The host selects libraries that fit the requirements. 5. The enclave is built with the updated `enclave_bundle`. 6. The host sends response with library choice proposal. 7. The enclave starts attesting itself by sending the attestation report provided by the Enclave Platform. 8. The verifier gets publicly available library hashes for the libraries chosen by the host and rebuilds the measurement using expected values. The measurement should match, and the chosen libraries should be considered secure by the developer's policy. 9. If all checks pass, the verifier responds with proof of approval to be used for retrieving private data.

The last consideration is when the host does not have an acceptable library available. For this to work well, the developer needs to evaluate which libraries are available on the target service and package the libraries that will not be available in the initial request. The developer should test their enclave on the service they expect to deploy to, and the host can return errors if it is unable to fulfill all missing libraries. This will cover most use cases, but alternatively the developer can manage a fallback that re-makes the request with extra libraries supplied.

D. On Top of Linear Measurement

Ideally, the measurement is implemented on the Enclave Platform, but this design fits very easily on top of linear measurement. We will start by examining on how Gramine on Intel SGX can be modified to implement this design, and then present the generic solution. Our work does not do the implementation.

Gramine has a blank-slate libOS that is loaded by the host into the enclave environment along with a manifest. The manifest specifies all resources and exact hashes allowed for it. The Gramine starts executing, receives the resources from the untrusted host, verifies that they match the hashes, and install them appropriately. This is similar to having only the bare minimum bootstrap as resident resources in this work, and all other resources are absent.

Our modified Gramine will thus mimic having only absent resources; for now, only absent identity resources that affect the sealing key. When we send a deployment request, the untrusted host will be able to make proposals by editing the Gramine manifest file. The host will send the final manifest file instead of the enclave bundle of this work; alternatively, the enclave can send it instead if for some systems this is more convenient. Notably, we will require the manifest to be the

last thing added to enclave memory; this allows the verifier to reconstruct the expected measurement quickly by taking a frozen Gramine blank-slate hash and advance it with the new manifest (converting the manifest to the memory view). This is not as efficient as this work, but is on the same order of complexity. The rest of the remote attestation process will remain the same.

To support non-identity absent resources, we will have the enclave retrieve the array from the host after it starts. It will send this array along with the attestation report, and the verifier will service it as usual. This means that the correctness of the reported non-identity absent resources relies on Gramine being correct, as opposed to the Intel SGX measurement being correct, which is worse in terms of TCB size for this part of functionality.

The same can be done on an arbitrary linear measurement platform. The developer will elect a blank-slate base that will simply enforce a manifest file when accepting files from the host, and set those files up. The blank-slate can either include an entire library OS or runtime as in the Gramine case, or be more minimalist. The main limitation is that this blank-slate should be secure and change infrequently. Lastly, the manifest file should be placed inside the enclave such that it is the last thing that the hash is extended on, so that the verifier can cheaply continue from just before the manifest for calculating expected measurements.

VI. Implementation

Our implementation is build on Keystone, a RISC-V Enclave Platform. We chose this platform because it is open source and fully extensible.

The core TCB is the new measurement code that traverses the `enclave_bundle`, which is 100 LoC in the SM. The code mainly simply traverses the arrays and follows pointers. It also does bounds checks to make sure anything measured is in the enclave-owned memory, and checks for the resident resource whose address is equal to the start Program Counter for filling in that field. It also validates `runtime_values` provided by the host – the total memory allocated and untrusted memory allocated to the enclave. The host fills in arrays and files, so it's easier for it to create the resource values.

We moved binary loading from host to inside enclave, allowing us to supply the enclave with files rather than memory-view loaded binaries. The bootloader is directly an executable code section in assembly and loads the runtime ELF into virtual memory. Runtime runs the enclave application. The bootloader is the start resource, and all current examples use these 3 files for enclave apps. These programs look for files in the `enclave_bundle`.

The SDK provides a function used for declaring one resource at a time, in which the SDK keeps track of the set of resources we want in the enclave. Once done, it creates the `enclave_bundle` based on it, automatically arranging everything in memory.

Dynamic library support is not implemented yet, but there a design is done. Runtime will allow a modified `ld` subroutine to discover shared objects that correspond to entries in `enclave_bundle`. If absent, the hash must match, otherwise the file can be used directly.

Notably, enforcement of any other absent resources is up to the application and framework developers.

The best way to combine this work with library sharing works is to combine it with Lu's work in RISC-V Dynamic Linking [32]. There should be one simple enclave that manages libraries – simply loads them into memory and tracks what hash value the ELF file was. An application enclave can check the attestation report of the library enclave, then get the library that corresponds to the specific hash. It may need to do symbol resolution, but the library itself does not need to be loaded again as the memory can just be mapped into the application enclave. The shared library can be protected by the SM mapping it only as executable and neither readable nor writable by non-hosting enclave.

VII. Evaluation

We evaluate by examining how well the new system solves the presented problems, ease of use, and new use-cases.

A. Achieved Properties

The design has some key properties that allow it to address the presented problems, summarized in Table I. We compare against Gramine because it's the most developed Enclave Framework and is strictly additive to Intel SGX, and Blank-State Worker because this approach has seen some practical adoption in unrelated projects to reduce measurement management burden. For BSW, equivalent features are achieved by not having to do verification on the actual enclave application, not literal verifier and measurements. The comparison of the designs themselves is in Figure 5.

Property	DCM	Gramine	BSW
One verifier policy matches different input args & lib versions.	●	○	●
One verifier policy matches host proposals for libraries.	●	○	○
Fast revocation recovery.	●	○	◐
Data providers can verify app.	●	●	○
Mutually private code within app.	●	○	○
Sealing key features.	●	◐	○
Fast generation of expected measurements.	●	○	◐
Deployment flexibility.	●	○	○

TABLE I
Overview of Properties achieved by Dynamic and Composable Measurement (this work), Gramine, and Blank-State Worker, respectively. ●, ◐, ○ are full support, partial, and none.

One verifier policy refers to a policy where no manual work needed from developer to specify additional exact matches that are allowed except the first one. For both of the rows, we achieve this by having Dynamic and Composable measurement that works with file hashes. This also leads to fast generation of expected measurements. Fast revocation recovery is enabled by the fact that the verifier can seamlessly

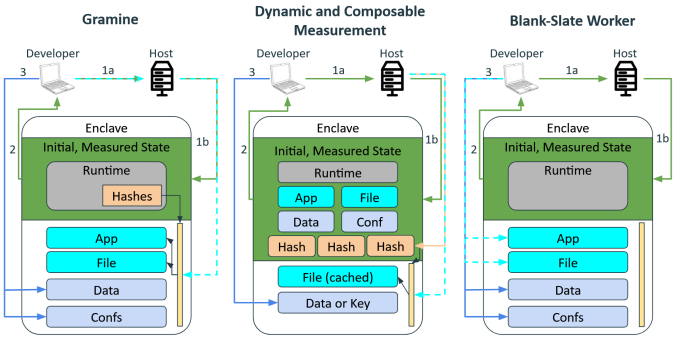

Fig. 5. Comparison of Gramine, DCM (this work), and BSW. Green is initial enclave state, gray is runtime, cyan is semi-stable resources, dark blue is per-run resources, light orange is for hashes, yellow checks untrusted resources against trusted hashes. Arrow colors indicate what resource is being carried. 1a is developer requesting an app to be ran, 1b is host relaying that to the enclave, 2 is attestation report, 3 is additional resources over secure channel. In DCM, the host has one file cached, so it adds one of the hashes and provides the file, without getting it from the developer.

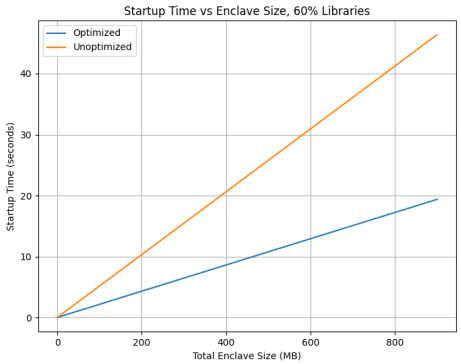

Fig. 6. Startup Time vs Enclave Size. Optimized is libraries cached on host instead of downloaded. Simulated and timed in QEMU.

switch to requiring a higher version of a library, so the build doesn't have to update the verifier and propagation is less synchronous; a high quality host will also be able to propose the new secure version fast, in which case the developer doesn't need to update their build to bring the app back up. Data providers are able to verify the app because the measurement actually reflects the code that will be ran, as is with Gramine. Mutually private code is enabled by Dynamic and Composable Measurement because private code providers can send the hash and not the actual code to allow the developer to conveniently verify that the code is as expected. Sealing key features come from being able to make it not depend on the entire initial state. Deployment flexibility comes from being able to supply resources at any time, and still include them in the measurement as desired through resident and absent options.

Managing expected measurements is now easy because there is only one measurement of the actual app binary file to keep track of, and all libraries and initial arguments are composed automatically to produce the expected measurement as needed.

The library selection is now aided by the host's proposal that uses the flexible selection to find the version that will gain some benefit. Trust revocation recovery latency and deployment latency are often more critical than the other potential benefits, of which we measured the deployment latency. The bottleneck in deployment is the download of resources, which is often optimized via caching of cloud providers, who each recommend using specific libraries and resources available by default. We graph the results in Figure 6. This latency is especially important due to the industry trend of having on-demand availability of many services, meaning resources like databases and stateless functions are kept off and come on only when a user makes a request to avoid renting a machine 24/7.

The testing methodology for the Startup Time is as follows. QEMU gives a very rough idea of performance, so we used microbenchmarks of different code sections to minimize simulation inaccuracies. We found that QEMU limits disk reads and writes to just 5 MBps as opposed to a typical NVME SSD achieving 1000-4000 MBps. We standardized network speed to 20 MBps, which is reasonable for unoptimized internet traffic that happens when source and destination are not within the same organization. We found that enclave setup has a small constant of 49ms, after which the cost is dominated by the measurement hashing the files, which came out to 2000 MBps, with both numbers averaged over 5 trials. We modeled an application with 60% of its size being dynamic libraries replaceable by host caching. At 300MB, we get the startup time down to 6.5s from 15.5s, and at 900MB, to 19.4s from 46.4s. This is a performance gain of 58%, approaching the 60% limit; this shows that image download is by far the biggest bottleneck in deployments.

The trust abstraction is now that the measurement asserts what is and what is going to be in the enclave. It does so through hashes of present components, so the expected measurement can be constructed based on provided expected hashes without the actual resource. The initial state must include everything to ensure that the correct exact execution will happen. This is detached from the sealing key, that is responsible for allowing the same enclave to store data conveniently, regardless of execution.

B. *Ease of Use*

The measurement is very flexible, providing developers with a type field they can set for privilege and permission management, and other annotations. Files can absent, so only declared at measurement time, or resident. Files can affect the sealing key, or not. This allows the developer to fully select at file granularity what should be in each measurement, fitting arbitrary system designs. The resources are simply files, including raw binary files, so anything can be included.

The Enclave Platform simply iterates through the arrays of resources. This means that bootstrapping, which files do what, and the file types are fully configurable by developers

without any changes to the Enclave Platform. In Keystone, the bootloader and runtime are open source, so developers can use these as a highly modular base for their particular needs; the same on other platforms may be more difficult.

Individual Enclave Platforms can choose what resource values they are able to enforce and attest to, so future work on for instance CPU time guarantees can take advantage of this.

The hashes used in the measurement are file hashes, so standard tooling can be used at all points for subroutines involving the measurement.

The enforcement and delivery of absent resources is fully up to the choice or implementation of the developer, and the host is free to offer optimizations on the delivery in any design that a developer will agree and support. Resource distribution can be split across the phases in any way as needed. For example, some dependencies may be fulfilled by shared libraries, or shared objects pre-loaded and handed off to the enclave. Either way, it is the enclave runtime's responsibility to correctly enforce the measurement attested to. Alternatively, all optimizations can be turned off for simplicity, or when there are software supply chain concerns.

Revocation and recovery is a lot easier, as developers simply update the policy on the verifier to require a higher version number, meaning the verifier is able to validate new reports with virtually no work. As before this work, private data serving components should also be updated to start rejecting all new request on old attestations, and enclaves app should discover through polling that they should shut down to minimize attack surface on data that was already served.

Forming the expected measurement no longer requires complex building. So for re-deployment with an updated resource, the developer simply creates an image, then builds it as a part of testing, and finally deploys the image, without having to build it for the measurement as well. This means the deployment process is broken up into independent steps that are similar to regular build and deployment workflows.

Sealing key migration can be done a lot easier if an enclave app has a low-TCB short-circuit that can be activated to run a developer-signed migration script. This means that migration doesn't have to be planned across version changes, and there is no danger of having to do two version changes in case the already-running version did not correctly account for possible desired migration target.

C. Use Case Examples

New functionality and simplicity achieved by Dynamic and Composable Measurement paves the way to many use cases becoming much easier to implement, with near no additional work on top of the measurement.

Private LLM serving. An entity such as OpenAI likely has both the Large Language Model and the hardware to run it for private use-cases. However, a developer and the model provider can be mutually distrusting. They can agree on a public runtime component that loads a PyTorch checkpoint and runs private data on it. The enclave can run on model provider hardware, where the model provider can propose the checkpoint model into the Enclave Bundle. The developer can still re-create the final measurement by using just the model hash without knowing the model itself. This gives the developer a non-equivocation guarantee, and that private data will stay private. The model provider is guaranteed that the model will remain private. In Linear Measurement, any time the data provider wanted to use a different model tier, or any time OpenAI wanted to make a small update to the model, they would have to coordinate to guarantee a different model hash (or expand the allowlist), recompute the measurement, and re-deploy from the new image, in contrast to simply swapping out the hash in the configuration that the image is a part of.

Serving private data to clients. For example, a content hosting company may want to prevent piracy of intellectual property. They can launch an enclave on the client machine that is protected from the OS and any software on the computer, which in turn can establish a secure connection to the display monitor, encrypting the signal until then. This is now feasible because the startup time is very short for arbitrary enclaves that rely on popular libraries, and most customers will generally have roughly the same libraries available. With the recent deployment of WebGL in WebAssembly, browsers are on the path to efficiently exposing more hardware to websites. In Linear Measurement, this is not feasible because there would be many different libraries and builds on various clients; Netflix would have to manage possibly builds and at minimum expected measurements for all possible combinations of resource builds, and various versions of each for the same platform to not force excessive updates.

Blockchain orchestrator for a distributed application. Because all functionality is a lot simpler than before due to just file hashes being needed, application lifecycle can be hosted as a Smart Contract on the blockchain. End-to-end encrypted services like Signal already base trust in privacy on application code, not the organization; with this, the availability and transactions are also guaranteed by code instead of organizations. The contract could respond to democratic votes on application updates, and issue migration requests to enclaves, where old-version and new-version enclaves can establish secure channels if they observe the blockchain committing to the command. This allows for arbitrary functionality without an application secret. Furthermore, the smart contract could also pay in cryptocurrency for the cloud services running the enclaves, and application users can pay for features or donate to the contract. Only the verifier needs to run on the blockchain, and code development and build can be handled by the community. This means that a company providing the service cannot mount a DoS or data deletion attacks, and the community can always stay on current version or rollback to restore functionality. In Linear Measurement, producing expected measurements would be too expensive to host on blockchain as resources would have to be downloaded and the enclave built.

VIII. Conclusion

We presented Dynamic and Composable Measurement of enclave applications, which brings features that greatly simplify usability of enclaves. New optimizations and use cases are also enabled by the flexibility. The design's modularity allows for high interoperability and customizability to the particular needs of an application or a framework. It also provides a powerful abstraction barrier where the enclave platform can measure contents in any way an application may require, but it does not do any more or less. This removes the need to build workaround solutions for measuring on top of enclaves, but leaves the more complex and rapidly developing enforcement and provisioning work out of the enclave platform. This will encourage further adoption and investment in enclave technologies, and bring new capabilities to users.

Acknowledgment

I thank Dayeol Lee for his mentorship in exploring research areas, working with Keystone, design discussions, and the research process. I also want to thank Prof. Dawn Song, my research advisor for her guidance and feedback throughout my research.

Lastly, work on moving ELF loading into inside the enclave and out of the host for it was started by Catherine Lu, related to her work in [32]. This work is, however, entirely separate from Lu's work.

References

[1] *Announcing the public preview of azure confidential vms with intel tdx*, 2023. [Online]. Available: https://azure.microsoft.com/en-us/updates/confidential-vms-with-intel-tdx-dcesv5-ecesv5-public-preview/.

[2] M. Russinovich, *Azure and amd announce landmark in confidential computing evolution*, 2021. [Online]. Available: https://azure.microsoft.com/en-us/blog/azure-and-amd-enable-lift-and-shift-confidential-computing/.

[3] *Keystone website*. [Online]. Available: https://keystone-enclave.org/.

[4] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting SGX enclaves from practical Side-Channel attacks," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, Boston, MA: USENIX Association, Jul. 2018, pp. 227–240, ISBN: ISBN 978-1-939133-01-4. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/oleksenko.

[5] A. Nilsson, P. N. Bideh, and J. Brorsson, *A survey of published attacks on intel sgx*, 2020. arXiv: 2006.13598 [cs.CR].

[6] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An oblivious and encrypted distributed analytics platform," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, Boston, MA: USENIX Association, 2017, pp. 283–298, ISBN: 978-1-931971-37-9. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zheng.

[7] F. McKeen, I. Alexandrovich, I. Anati, *et al.*, "Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave," in *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, ser. HASP '16, Seoul, Republic of Korea: Association for Computing Machinery, 2016, ISBN: 9781450347693. DOI: 10.1145/2948618.2954331. [Online]. Available: https://doi.org/10.1145/2948618.2954331.

[8] *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. AMD, 2020. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.

[9] *Intel Trust Domain Extensions*. Intel, 2022. [Online]. Available: https://cdrdv2-public.intel.com/690419/TDX-Whitepaper-February2022.pdf.

[10] *ARM Security Technology: Building a Secure System using TrustZone Technology*. ARM, 2009. [Online]. Available: https://documentation-service.arm.com/static/5f212796500e883ab8e74531.

[11] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20, Heraklion, Greece: Association for Computing Machinery, 2020, ISBN: 9781450368827. DOI: 10.1145/3342195.3387532. [Online]. Available: https://doi.org/10.1145/3342195.3387532.

[12] J. Ménétrey, C. Göttel, A. Khurshid, *et al.*, "Attestation mechanisms for trusted execution environments demystified," in *Lecture Notes in Computer Science*. Springer International Publishing, 2022, 95–113, ISBN: 9783031160929. DOI: 10.1007/978-3-031-16092-9_7. [Online]. Available: http://dx.doi.org/10.1007/978-3-031-16092-9_7.

[13] A. Niemi, S. Sovio, and J.-E. Ekberg, "Towards interoperable enclave attestation: Learnings from decades of academic work," in *2022 31st Conference of Open Innovations Association (FRUCT)*, 2022, pp. 189–200. DOI: 10.23919/FRUCT54823.2022.9770907.

[14] M. Marlinspike, *Technology preview: Private contact discovery for Signal*. Signal, 2017. [Online]. Available: https://signal.org/blog/private-contact-discovery/.

[15] *The State of Dependency Management*. Endor Labs, 2022. [Online]. Available: https://www.endorlabs.com/state-of-dependency-management.

[16] *Netflix supported browsers and system requirements*. Netflix, 2024. [Online]. Available: https://help.netflix.com/en/node/30081.

[17] *Ways to get the AWS SDK for Java*. AWS, 2018. [Online]. Available: https://docs.aws.amazon.com/sdk-for-java/v1/developer-guide/setup-install.html.

[18] *Installing NumPy*. NumPy. [Online]. Available: https://numpy.org/install/.

[19] G. Chen and Y. Zhang, "MAGE: Mutual attestation for a group of enclaves without trusted third parties," in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA: USENIX Association, Aug. 2022, pp. 4095–4110, ISBN: 978-1-939133-31-1. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/chen-guoxing.

[20] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA: USENIX Association, Nov. 2016, pp. 533–549, ISBN: 978-1-931971-33-1. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/hunt.

[21] H. M. Zum Felde, M. Morbitzer, and J. Schütte, "Securing remote policy enforcement by a multi-enclave based attestation architecture," in *2021 IEEE 19th International Conference on Embedded and Ubiquitous Computing (EUC)*, 2021, pp. 102–108. DOI: 10.1109/EUC53437.2021.00023.

[22] G. Chen, Y. Zhang, and T.-H. Lai, "Opera: Open remote attestation for intel's secure enclaves," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, London, United Kingdom: Association for Computing Machinery, 2019, 2317–2331, ISBN: 9781450367479. DOI: 10.1145/3319535.3354220. [Online]. Available: https://doi.org/10.1145/3319535.3354220.

[23] W. Liu, W. Wang, X. Wang, *et al.*, *Confidential attestation: Efficient in-enclave verification of privacy policy compliance*, 2020. arXiv: 2007.10513 [cs.CR].

[24] V. Scarlata, S. Johnson, J. Beaney, and P. Zmijewski, *Supporting Third Party Attestation for Intel SGX with Intel Data Center Attestation Primitives*. Intel, 2018. [Online]. Available: https://cdrdv2-public.intel.com/671314/intel-sgx-support-for-third-party-attestation.pdf.

[25] *Open enclave website*. [Online]. Available: https://openenclave.io/sdk/.

[26] C. che Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A practical library OS for unmodified applications on SGX," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, Santa Clara, CA: USENIX Association, 2017, pp. 645–658, ISBN: 978-1-931971-38-6. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai.

[27] *Gramine Documentation*. Intel. [Online]. Available: https://gramine.readthedocs.io/en/stable/attestation.html.

[28] D. Kuvaiskii, G. Kumar, and M. Vij, *Computation offloading to hardware accelerators in intel sgx and gramine library os*, 2022. arXiv: 2203.01813 [cs.CR].

[29] F. Alder, N. Asokan, A. Kurnikov, A. Paverd, and M. Steiner, "S-faas: Trustworthy and accountable function-as-a-service using intel sgx," ser. CCSW'19, London, United Kingdom: Association for Computing Machinery, 2019, 185–199, ISBN: 9781450368261. DOI: 10.1145/3338466.3358916. [Online]. Available: https://doi.org/10.1145/3338466.3358916.

[30] K. Chen, A. Thomas, H. Lu, *et al.*, *Scl: A secure concurrency layer for paranoid stateful lambdas*, Oct. 2022.

[31] Y. Shen, Y. Chen, K. Chen, H. Tian, and S. Yan, "To isolate, or to share? that is a question for intel sgx," in *Proceedings of the 9th Asia-Pacific Workshop on Systems*, ser. APSys '18, Jeju Island, Republic of Korea: Association for Computing Machinery, 2018, ISBN: 9781450360067. DOI: 10.1145/3265723.3265727. [Online]. Available: https://doi.org/10.1145/3265723.3265727.

[32] C. Lu, *Dynamic Linking in Trusted Execution Environments in RISC-V*. UC Berkeley, 2022. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-142.pdf.