Samba: Enabling Confidential and Collaborative Cloud Functions

Matthew Berthoud*
mwberthoud@wm.edu
College of William & Mary
Williamsburg, Virginia, USA

Stephen Herwig*
smherwig@wm.edu
College of William & Mary
Williamsburg, Virginia, USA

Abstract

Function as a Service (FaaS) is a popular cloud computing paradigm that enables customers to execute individual functions in response to events, while the cloud provider manages scaling and the underlying infrastructure. Leveraging FaaS' simplicity and loose coupling, customers can create complex computational pipelines by chaining the execution of multiple functions. However, function chains often process sensitive user input, as for financial transactions or machine learning inference, thus exposing private data to the cloud provider and cloud attackers. While secure hardware enclaves, such as Intel SGX, can protect function chains, prior enclaved FaaS systems rely on a trusted key-provisioning enclave to simplify autoscaling and authentication. Unfortunately, such key provisioning enclaves constitute a key escrow, and thus a single target for compromising the entire function chain.

In this thesis, we present our initial work on Samba, a confidential FaaS system that removes the need for a trusted key provisioning service. Samba runs each function instance in a secure enclave that locally generates its encryption keys. To allow the untrusted cloud provider to autoscale function instances without introducing trusted services, Samba leverages cryptographic schemes for proxy re-encryption, thus allowing the provider to safely re-encrypt data for any instance replica. We present our initial work in designing, implementing, and evaluating Samba's proxy re-encryption protocol, and discuss future work in integrating Samba with the popular Knative serverless platform.

CCS Concepts

• **Security and privacy → Security protocols**; **Key management**; **Distributed systems security**; • **Networks → Cloud computing**.

Keywords

Confidential Computing, Cloud Functions, Proxy Re-encryption

1 Introduction

Function-as-a-Service (FaaS) platforms like AWS Lambda are popular choices for building event-driven, cloud-based solutions because of their autoscaling capabilities and pay-per-use billing. The stateless nature and loose coupling of functions allow organizations to create complex workflows simply by linking multiple functions in a sequence, or *function chain*. A recent study found that 46% of real-world FaaS applications use function chains, with some chains containing up to 10 functions [44]. Despite the name, function chains may be input-dependent, allowing for branching and looping, and are thus more accurately graphs.

There are many use cases—particularly in highly-regulated sectors like government, finance, and healthcare—, where function chains must process privacy-sensitive data, and where the chain includes functions from multiple organizations. For instance, a chain to process a loan application could include: (1) the bank's initial function to process the application, (2) a government service's function to verify the applicant's identity, and (3) a credit bureau's function to retrieve the applicant's credit score. In such a setting, each organization must trust another's function with their data. Even with mutual trust among organizations, adversaries can exploit vulnerabilities in functions, manipulating data flows to steal sensitive data and conduct stealthy operations [5]. Moreover, despite seemingly trustworthy cloud providers, customers must still be wary of cloud infrastructure bugs [23], insider threats [43], and data disclosures to law enforcement [1–3].

Prior work in securing function chains focuses either on guaranteeing the chain's data confidentiality, or the integrity of function order. For data confidentiality, previous research systems [6, 16, 26, 33, 35, 45, 49] leverage hardware trusted execution environments (TEEs, particularly Intel SGX enclaves [19, 39]) to enable confidential function workflows. Unfortunately, to support autoscaling, these systems rely on a trusted key distribution enclave to provision each function replica with the same keying material, creating a key escrow and single point of failure. To ensure function order integrity, research systems like Kalium [30] and Valve [20] model the function chain as a call graph and enforce control-flow integrity to guarantee that runtime execution follows the expected sequence. Since a function has only a local view of the application, these systems also must rely on a trusted, global controller to track the application-wide control flow.

In this thesis, we demonstrate initial work in affirming our research statement:

> **A FaaS platform can guarantee the confidentiality and integrity of function chains without resorting to centralized trusted services.**

We present Samba, the first FaaS platform to guarantee data confidentiality and chain integrity without relying on a trusted control plane. Samba combines non-interactive cryptographic protocols with TEEs to secure function chains without centralized trust. In Samba, each function replica runs in a TEE and independently generates its own keys. Using proxy re-encryption, Samba enables any replica to decrypt messages intended for its target function, eliminating the need for a trusted key escrow. For control flow integrity, Samba's runtime cryptographically signs each function's provenance, removing the need for a global controller.

Contributions. We make the following contributions:

*This research was completed as a Master's project by advisor and advisee

- We introduce Samba, a FaaS platform for ensuring the confidentiality and integrity of complex FaaS pipelines. Samba composes trusted execution environments with a novel cryptographic protocol for proxy re-encryption to protect inter-function I/O while preserving horizontal scalability.
- We implement an initial prototype of Samba that demonstrates the feasibility of proxy re-encryption in a FaaS setting.
- We conduct a preliminary evaluation of Samba using several micro-benchmarks that measure the performance of proxy re-encryption operations relative to a naïve RSA baseline.

Outline. This paper is organized as follows. In §2 we provide an overview of FaaS, TEEs, and proxy re-encryption. In §3 we specify our threat model, goals, and assumptions. We present the design of Samba in §4, our preliminary implementation in §5, and an evaluation of the overhead of proxy re-encryption in §6. We describe future work in extending our prototype in §7, and conclude in §8.

2 Background

Function as a Service Function as a Service (FaaS) is a cloud computing model that enables developers to deploy and execute individual functions or pieces of code without managing the underlying infrastructure. It is sometimes referred to as serverless computing, since these functions are event-driven and run in stateless containers, automatically scaling up or down based on demand, according to the policy of the orchestrator. FaaS eliminates the need for developers to provision and manage servers, allowing them to focus on writing code while the cloud provider handles resource allocation, availability, and execution. Popular FaaS platforms include AWS Lambda, Google Cloud Functions, and Azure Functions. Open and partially open source FaaS platforms include OpenFaaS, OpenWhisk, and Open Function, as well as Knative,[1] the open source serverless library to be used in Samba.

Confidential computing. *Confidential Computing* is executing a workload on an untrusted third-party machine (the cloud) in a way that "shields" the workload from the third-party: the third-party cannot alter, observe, or interfere with the computation or its data (though, in practice, side channels [15, 17, 34, 36, 37, 47, 48] weaken these guarantees). Confidential computing relies on hardware *trusted execution environments* (TEEs) to ensure memory isolation from the rest of the system: an application's memory is encrypted and integrity-protected when in DRAM, and decrypted only when the memory enters the trusted CPU package. When a workload is running within a TEE, we say that the workload is running within an *enclave*; as shorthand, we often simply call the workload an enclave.

Intel SGX [27, 39] was the first general approach to confidential computing. By representing enclaves at the granularity of a part of a process, Intel SGX promotes a small trusted computing base, but at the expense of incompatibility with legacy software, though several middleware solutions [9, 11, 24, 46] attempt to ease this burden. To allow for unmodified applications to run within an enclave, the current generation of TEEs, such as AMD SEV [7, 31, 32], Intel TDX [28], Arm CCA [38], and IBM's PEF [25], implements enclaves at the granularity of a virtual machine (called *confidential VMs*).

Two core properties of enclaves are attestation [8] and sealing. In *attestation*, a hardware root-of-trust signs a measurement (digest) of the initial launch state of the enclave, forming an *attestation report*; the customer can retrieve this report to verify the correct software is running in an enclave. To allow an attestation to bind some runtime value (typically a public key), the enclave can include a small amount of *user-data* in the report; the user-data is opaque to the secure hardware, but otherwise covered by the report's signature. *Sealing* enables an enclave to securely persist confidential data so that only the enclave can decrypt it later, using a hardware-derived key tied to the enclave's identity or measurement.

Proxy Re-encryption Proxy re-encryption is a cryptographic scheme that allows an untrusted proxy to convert a ciphertext encrypted under Alice's public key into a ciphertext that Bob can decrypt with his secret key, without learning the underlying plaintext. At a high-level, Alice and Bob construct a public *re-encryption key* $RK_{\text{Alice}\to\text{Bob}}$ that the proxy uses to re-encrypt the ciphertext from Alice (the *delegator*) to Bob (the *delegatee*). In 1998, Blaze, Bleumer, & Strauss [12] designed the first proxy re-encryption construction based on the ElGamal encryption system [22]. Shortly thereafter, Dodis and Ivan [29] developed a unidirectional variant, and later Ateniese et al. [10] applied bilinear maps [13] (and specifically BLS signatures [14]) to develop schemes that did not require interaction between the delegator and delegatee. Since then, numerous works have explored features like multiple re-encryptions [40] and revocation [42], and security properties like chosen-ciphertext resistance [18] and unlinkability of ciphertexts [21].

3 Goals & Assumptions

Threat model. Our Samba system has three parties: the *cloud provider*, the *function providers*, and *external attackers*. A function chain may contain functions from multiple providers. For any given function provider, the potential adversaries are the other function providers in the chain, the cloud provider, and external attackers. The goal of an adversary is to learn the inputs or outputs of a (peer) function, modify these inputs or outputs, or modify the sequence of functions in the chain (e.g., to insert a function that sends data to an adversary-controlled log). A function provider can submit any function to the chain, including a malicious function that tries to leak data or subvert the cloud provider. We assume that functions may contain bugs that unintentionally leak data or expose the function to exploitation. We assume an adversary cannot breach the security of confidential VMs; we trust the system software in the VM, and consider side-channel attacks out-of-scope.

Goals. Our primary security goals when designing Samba are:

- **S1** *Confidentiality*: An attacker that breaches the FaaS platform or exploits a function must not be able to undermine the confidentiality of a peer function or its I/O.
- **S2** *Integrity*: An attacker that breaches the FaaS platform or exploits a function must not be able to alter the control flow of a function chain.

Additionally, we have the following functional goals:

[1] https://knative.dev

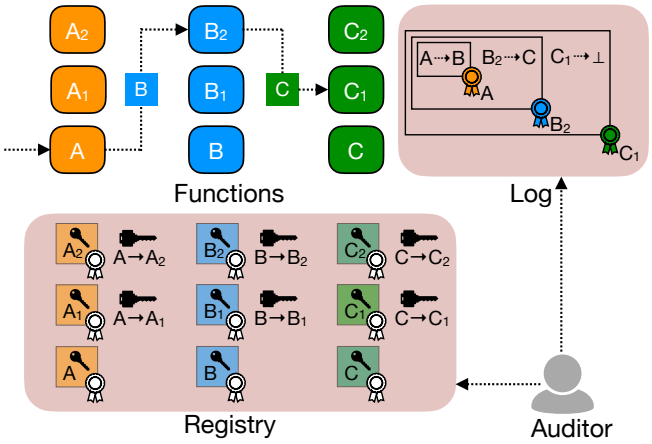

Figure 1: Samba architecture.

- **F1** *Application transparency*: Samba must support unmodified functions. In particular, a function developer must not have to modify their function to support Samba's security features.
- **F2** *Compatibility with existing FaaS platforms*: Samba must extend current FaaS platforms (our future work targets Knative). This implies supporting the autoscaling policies of such platforms (including scaling down to zero replicas).
- **F3** *Preservation of Untrusted Control Plane*: Samba must not add trusted components to the FaaS control plane.
- **F4** *Low performance overheads*: Samba must impose only modest performance overhead on the FaaS applications, both in terms of client latency and resource usage.

4 Design

Samba's future implementation will extend the open source Knative FaaS library. To facilitate this process, we designed Samba-Lite, a simplified proof-of-concept. There are three core parties involved in Samba-Lite:

(1) **Function replicas** (e.g., Alice and Bob) that run some application code.
(2) **Proxy** that manages the function replicas, re-encryption, and load balances requests.
(3) **Sender** that sends requests to the function, through the proxy.

In previous research, each function replica in a FaaS system would attest to a trusted key server to obtain a shared key pair. To eliminate this central, trusted dependency, Samba uses decentralized key management with proxy re-encryption. As Figure 1 illustrates, the first instance of a given function generates a key pair and registers the public key in this untrusted registry, managed by the proxy. Since no private keys or other sensitive information are stored here, this upholds the threat model of an untrusted proxy. In the future, instances will also register an attestation report that verifies the key was generated in a Trusted Execution Environment. For now, the proxy generates public parameters that are sent to the instances to generate their key pairs. Since this key generation is done without a TEE in Samba-Lite, a malicious cloud provider could obtain access to all function inputs by reading the private key from memory. Samba-Lite does not uphold the same threat model as Samba, but we lay the groundwork for future guarantees.

After functions are running and registered with the proxy, the sender can make a request to the function. The sender must encrypt the message under the public key of just one function instance, despite the potential for any replica to operate on the request. Therefore, the proxy maintains a record of which instance is the "leader" instance for each function. They advertise the public keys for those leader instances, and the sender encrypts their message under the public key of the leader for the function they're calling.

When the request gets to the proxy, the proxy will first check if the leader instance is available to handle the request. If so, the proxy forwards the request to that instance, which can decrypt it with its private key. If the leader instance is unavailable, the proxy will select an available replica to handle the request. The proxy will then request the leader instance to generate a re-encryption key from itself to the selected replica. Upon receiving the key, the proxy will re-encrypt the request to the selected replica, and forward it to that replica. The selected replica can then decrypt the request with its own private key, and process it.

An interesting aspect of proxy re-encryption is that decrypting an encrypted message and decrypting a re-encrypted message require different decryption method. This means the decryption performed by the leader is different than the decryption performed by the replica, so replicas need to be aware of the whether the message is re-encrypted, to call the correct subroutine.

After the message is decrypted, the replica can process the request and send a response back to the proxy, who forwards it to the sender. In Samba-Lite, the function code simply turns the string message uppercase, and sends it back unencrypted. In the eventual Knative implementation of Samba, the replica will encrypt the response back to the sender, or to the next function in the chain, depending on the details of the request.

5 Implementation

The Samba-Lite system relies on two libraries that we implemented. First, we discuss the `pre` library[2], which implements the AFGH proxy re-encryption scheme in Go. [10] Then we discuss the `samba` library[3], which calls provides a simple interface for the Samba system, relying on the `pre` library to perform the cryptographic operations. Finally, we cover the Samba-Lite system itself, which is contained as a separate entity within the Samba repository.

5.1 Proxy Re-encryption Library

We implemented a library for proxy re-encryption (PRE) in Go based on the AFGH proxy re-encryption scheme [10]. The four main functions of the library are

(1) `Encrypt`, which encrypts a message with a public key,
(2) `GenerateReEncryptionKey`, which generates a re-encryption key from a private key and a public key.
(3) `ReEncrypt`, which re-encrypts a message with a re-encryption key.
(4) `Decrypt1`, which decrypts a message with a private key,

[2]https://github.com/etclab/pre
[3]https://github.com/etclab/samba

(5) `Decrypt2`, which decrypts a re-encrypted message with a private key.

The plaintext of a proxy re-encryption scheme is a random element from the target group of a bilinear map. We defined a `KdfGtToAes256` function that converts the `pre` "plaintext" to an AES key. The true plaintext message must be encrypted under that AES key, and sent alongside group element encrypted by the `Encrypt` function.

5.2 Samba Library

We defined a `SambaProxy` struct that contains the public parameters, a map of function instances to their replicas, a map of instance keys, and a map of function leaders. Its methods perform tasks such as registering public keys, sending messages, and generating re-encryption keys. The `BootProxy` method initializes the proxy, generating the public parameters, and starting up an HTTP server both for receiving function requests from sender, and for instances to be able to register their public keys with the proxy.

We defined a `SambaInstance` struct that represents an individual instance of a function. Its methods perform tasks such as generating re-encryption keys, responding to messages from the proxy, and sending messages to the proxy. The `BootInstance` method initializes the instance, requesting the public parameters from the proxy, generating the instance a public/private key-pair, and registering its public key with the proxy. Then it starts up an HTTP server to receive messages from the proxy, which can be function messages or re-encryption key requests.

Some of the fields underlying the `pre` public and re-encryption key structs are private, so in order to send them over the network we have to serialize those fields. To do so, we created corresponding structs for the serialized versions of the keys. These structs have `Serialize` and `Deserialize` methods that convert between the serialized and de-serialized versions.

Samba messages themselves are defined by as a struct called `SambaMessage`.

```
type SambaMessage struct {
        Target          FunctionId
        WrappedKey1     Ciphertext1Serialized
        WrappedKey2     Ciphertext2Serialized
        IsReEncrypted   bool
        Ciphertext      []byte
}
```

The `Target` field is the function ID of the target function, which is used by the proxy to look up the function's leader instance. The `WrappedKey1` and `WrappedKey2` fields store the encrypted parameters that when decrypted and passed to `KdfGtToAes256` yield the AES key. The `IsReEncrypted` field is a boolean that indicates whether the message has been re-encrypted or not, and its value determines if `WrappedKey1` or `WrappedKey2` is used. The `Ciphertext` field is the actual ciphertext of the message, which is encrypted with the AES key. We used Go's `json` library to marshal `SambaMessages` into JSON, to be sent as http requests. The receiver then un-marshals them back into `SambaMessages` and processes them.

We defined a `SambaCrypto` interface to wrap the logic of encrypting, re-encrypting, and decrypting true plaintext messages.

```
type SambaCrypto interface {
        Encrypt(
                pp *pre.PublicParams,
                pk any,
                plaintext []byte,
                functionId FunctionId
        ) (*SambaMessage, error)

        Decrypt(
                pp *pre.PublicParams,
                sk any,
                m *SambaMessage
        ) ([]byte, error)

        ReEncrypt(
                pp *pre.PublicParams,
                rk *pre.ReEncryptionKey,
                m *SambaMessage
        ) (*SambaMessage, error)

        GenReEncryptionKey(
                pp *pre.PublicParams,
                sk *pre.SecretKey,
                req *ReEncryptionKeyRequest
        ) (*ReEncryptionKeyMessage, error)
}
```

This interface is implemented by the `SambaPRE` struct, which is what Samba-Lite uses to perform the cryptographic operations. The `SambaRSA` struct also implements this interface, and we discuss in §6 how we use it to compare the performance of Samba-Lite with a naïve RSA implementation.

These methods mainly wrap calls to the `pre` library, and perform the serialization tasks mentioned earlier. The `Decrypt` method also handles the case where the message has been re-encrypted, calling `Decrypt2` instead of `Decrypt1` when the `IsReEncrypted` field in the `SambaMessage` is true.

5.3 Samba-Lite

The `proxy` service boots a `SambaProxy` that runs an HTTP server exposing endpoints for registering keys, and sending messages.

The `alice` and `bob` services boot `SambaInstances` that request the public parameters from the proxy, generate public/private key-pairs for themselves, and register their public key back with the proxy. Then, they launch HTTP servers to which the proxy will make requests to send messages and request re-encryption keys. We arbitrarily set `alice` to be the leader, which the sender encrypts to. Samba-Lite doesn't do any real load balancing, since this is a responsibility of the eventual cloud provider. To simulate it, however, when running Samba-Lite you can toggle between Alice being able to handle requests and too busy to handle requests, to see requests send to either Alice or Bob, who decrypt and respond.

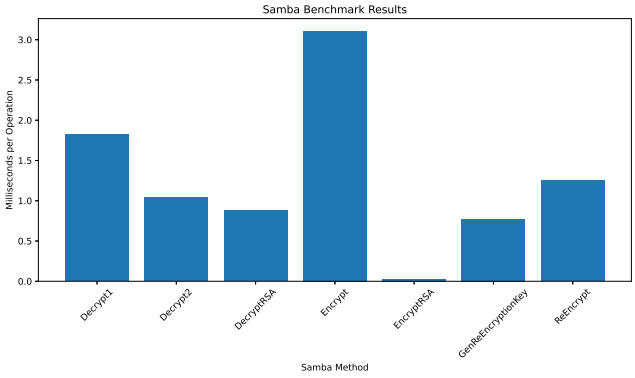

Figure 2: Benchmark results for the Samba library, with both PRE and RSA operations.

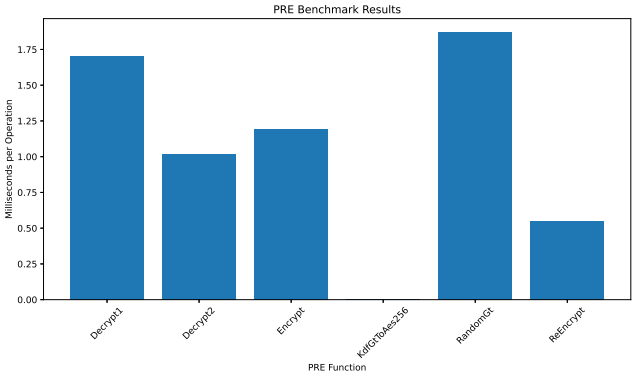

Figure 3: Benchmark results for the PRE library.

The `sender` service is a simple application that calls `samba.Encrypt` to encrypt a message, then sends it as an HTTP request to the proxy and prints the response. Running these services in the order listed achieves Samba-Lite's goal of demonstrating the control flow of the Samba system.

6 Evaluation

To evaluate the performance of the Samba library, we wrote a series of benchmark tests. Not only did we test the performance of Samba using proxy re-encryption, but we also compared it to an RSA implementation. Note that actually implementing Samba with RSA instead of PRE would defeat the purpose, as either the proxy would have to be trusted with every instance's private key, or all instances of the same function would have to share the same key pair.

For the RSA implementation, we only benchmarked the Samba functions responsible for encrypting and decrypting messages, since "re-encryption" is not a concept in RSA. For the proxy re-encryption implementation, we benchmarked the Samba functions responsible for encrypting and decrypting messages, generating re-encryption keys, and re-encrypting messages. Additionally, we benchmarked the underlying proxy re-encryption functions, from the PRE library we wrote. For example, the `samba.Encrypt` function contains the following code:

```
m := pre.RandomGt()
ct1 := pre.Encrypt(pp, m, pkPRE)
key := pre.KdfGtToAes256(m)
ct := AESGCMEncrypt(key, plaintext)
```

We benchmarked all three of the PRE library functions in this code snippet, as well as the Samba encrypt function that calls them. All benchmarks were run using the Go testing framework, which provides a simple way to measure the time taken by each function.

Our results from an 8-core M1 Mac of for the Samba benchmarks are shown in Figure 2, and the PRE benchmarks are shown in Figure 3.

The cost of RSA encryption is basically negligible, and almost invisible in Figure 2. Samba encryption, on the other hand, is our most expensive operation. This is because it calls all three of the PRE functions mentioned above. The `pre.RandomGt` function takes about 1.9 milliseconds, and the `pre.Encrypt` function takes about 1.2 ms in our benchmarking. Those two make up the bulk of the cost of Samba's `Encrypt`, which in total takes almost 3.2 ms.

RSA decryption takes much longer than RSA encryption, and both of our Samba decryption operations only incur slight additional overhead. The more expensive of the two, `Decrypt1`, is less than twice as expensive as RSA.

We believe the security guarantees of Samba justify the performance costs. Since Samba is to be used exclusively in FaaS systems, network request latency, which is usually in the hundreds of milliseconds, will dominate our cryptographic slowdowns. Additionally, the method to generate a re-encryption key doesn't have to be called on a per-request basis, but rather only the first time traffic is rerouted from the leader to an instance.

7 Future Work

While Samba-lite enables experimentation with the proxy re-encryption protocol, several key tasks remain: *(1)* integrating the runtime into a mainstream FaaS platform, *(2)* enforcing control-flow integrity at the network I/O level, and *(3)* extending the function runtime to support trusted execution environments. We now outline our initial designs for each of these concerns, leaving a full-fledged design and implementation for future work.

7.1 Integration with Knative

Knative architecture. Knative is a platform-agnostic framework that brings serverless capabilities to Kubernetes. As Figure 4 illustrates, the Knative control plane consists of an HTTP ingress router, an activator, and an autoscaler.

Under normal traffic,the ingress router routes requests to the activator, which buffers incoming requests and notifies the autoscaler if additional capacity is needed; the autoscaler, in turn, requests more pods from Kubernetes. Once new pods are ready—or existing ones become available—the activator forwards the buffered requests. Under high traffic, the ingress router bypasses the activator and routes requests directly to the active function pods. Regardless of the traffic pattern, all requests pass through the queue-proxy, a sidecar container in each function pod. The queue-proxy collects

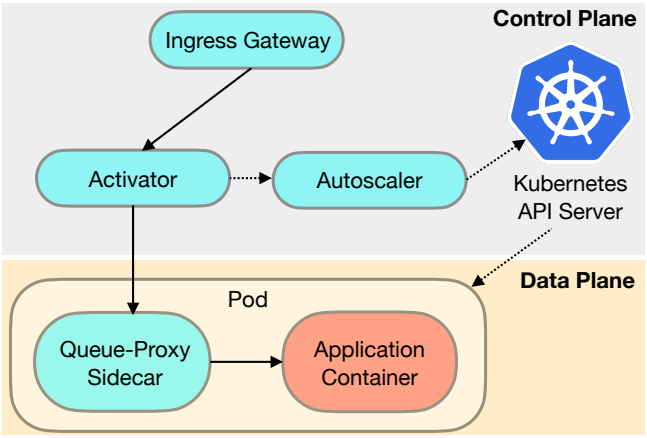

Figure 4: Knative architecture. The Knative components are in light blue. The solid arrows are the request flow, and the dashed arrows the autoscaling flow.

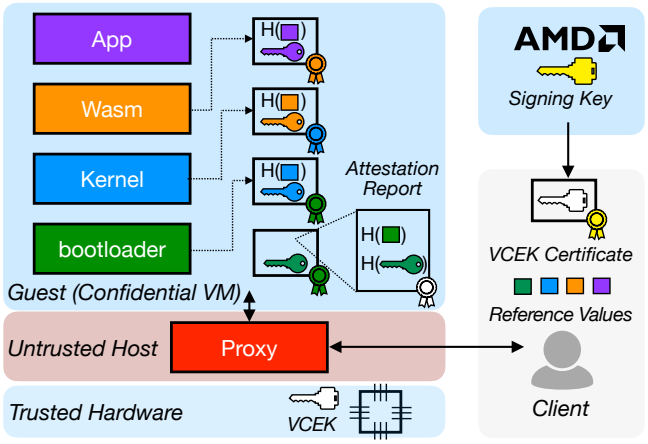

Figure 5: Oak and DICE architecture. In the DICE model, each software layer loads and measures the next layer, generates an ephemeral key pair for the layer, and issues the layer a certificate endorsing its measurement and public key. Oak uses AMD's trusted hardware as the root of trust.

metrics, enforces concurrency limits, and can also buffer requests when needed, similar to the activator.

To support function chaining, Knative provides an event mesh—an abstraction of a publish-subscribe system—that enables asynchronous, store-and-forward message delivery, decoupling senders and recipients in time.

Samba Knative support. As the queue-proxy sidecar interposes on the function's I/O, we plan to augment the sidecar with the functionality for proxy re-encryption and control-flow enforcement. In this way, the function container itself remains unchanged.

7.2 Control Flow Integrity

Beyond encrypting function I/O, Samba must ensure the integrity of the sequence of functions. Samba approaches this challenge in two ways. First, we will extend the queue-proxy to include a cryptographically binding record of provenance: each function extends a path signature by signing over its current link. The last function then posts this provenance record to an untrusted log that any organization can monitor and audit. Additionally, if an organization knows a flow graph for the function chain (or some subset thereof), the queue-proxy will locally verify that the received event and the function's subsequent output conform to the graph.

Samba relies on certificate chains both for attestation and provenance. To reduce bandwidth and storage costs, we will investigate compressing the chains with an aggregate signature scheme [14]. In an *aggregate signature*, each private key sk_i signs a *distinct* message m_i to form signature σ_i, and any party can compress the σ_i into a single, small aggregate signature σ^*. The aggregate signature convinces a verifier that each signer signed their respective message.

7.3 Running Functions in TEEs

In Samba-lite, the function instances run on conventional hardware, thus allowing the cloud provider to access each function's data, including its keys. To properly address our threat model (which allows for a malicious cloud provider and cloud-side adversaries), functions must run in secure hardware enclaves. For future work, we plan to extend *Project Oak*,[4] an open-source framework for processing private data within a TEE. The central component of Project Oak is the *Oak Functions* platform (see Figure 5), which executes each function in a confidential VM with a custom, minimal operating system kernel designed to execute only a single process. The function runs in a secure WebAssembly (Wasm) runtime sandbox, preventing the process from leaking any sensitive client data. Additionally, Oak uses the DICE [4] architecture for measured boot to extend AMD SEV-SNP's attestation from the initial state of the VM to the entire VM workload.

We will modify the Oak kernel to expose critical TEE services—namely, attestation and sealing—as system calls for the queue-proxy. In this way, the queue-proxy can register its public key with the untrusted proxy, along with a attestation that binds that key to the function instance. Additionally, sealing enables a leader replica to non-interactively migrate its key pair to a new leader replica, as during autoscaling operations.

8 Conclusion

As Mark Russinovich, CTO of Microsoft Azure, famously stated, in the future, "confidential computing will just be computing" [41]. Realizing this vision requires rethinking today's software systems to make confidential computing practical and accessible. This thesis takes a first step by bringing confidentiality to a widely used model—Function as a Service—while maintaining compatibility with existing applications through our system, Samba. Samba demonstrates that trusted hardware alone is not enough to satisfy the complex requirements of modern software systems. Instead, combining trusted hardware with cryptographic techniques—specifically, proxy re-encryption in Samba's case—enables secure and scalable solutions.

[4] https://github.com/project-oak/oak

To support future work, we have made our Samba-lite implementation publicly available at https://github.com/etclab/samba, along with our proxy re-encryption library at https://github.com/etclab/pre.

References

[1] 2014. US court forces Microsoft to hand over personal data from Irish server. https://www.theguardian.com/technology/2014/apr/29/us-court-microsoft-personal-data-emails-irish-server.

[2] 2020. The police want your phone data. Here's what they can get — and what they can't. https://www.vox.com/recode/2020/2/24/21133600/police-fbi-phone-search-protests-password-rights.

[3] 2022. Amazon gave Ring videos to police without owners' permission. https://www.politico.com/news/2022/07/13/amazon-gave-ring-videos-to-police-without-owners-permission-00045513.

[4] 2024. DICE Attestation Architecture. https://trustedcomputinggroup.org/resource/dice-attestation-architecture/. Version 1.1.

[5] Mohammad M. Ahmadpanah, Daniel Hedin, Musard Balliu, Lars Eric Olsson, and Andrei Sabelfeld. 2021. SandTrap: Securing JavaScript-driven Trigger-Action Platforms. In *USENIX Security Symposium*.

[6] Fritz Alder, N. Asokan, Arseny Kurnikov, Andrew Paverd, and Michael Steiner. 2019. S-FaaS: Trustworthy and Accountable Function-as-a-Service using Intel SGX. In *ACM Workshop on Cloud Computing Security (CCSW)*.

[7] AMD. 2020. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. Technical Report. AMD.

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

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

[10] Giuseppe Ateniese, Kevin Fu, Matthew Green, and Susan Hohenberger. 2005. Improved Proxy Re-Encryption Schemes with Applications to Secure Distributed Storage. In *Network and Distributed System Security Symposium (NDSS)*.

[11] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *Symposium on Operating Systems Design and Implementation (OSDI)*.

[12] Matt Blaze, Gerrit Bleumer, and Martin Strauss. 1998. Divertible Protocols and Atomic Proxy Cryptography. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*.

[13] Dan Boneh and Matthew K. Franklin. 2001. Identity-Based Encryption from the Weil Pairing. In *International Cryptology Conference (CRYPTO)*.

[14] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. 2003. Aggregate and Verifiably Encrypted Signatures from Bilinear Maps. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*.

[15] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In *USENIX Security Symposium*.

[16] Stefan Brenner and Rüdiger Kapitza. 2019. Trust more, serverless. In *ACM International Systems and Storage Conference (SYSTOR)*.

[17] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. 2021. One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization. In *ACM Conference on Computer and Communications Security (CCS)*.

[18] Ran Canetti and Susan Hohenberger. 2007. Chosen-ciphertext secure proxy re-encryption. In *ACM Conference on Computer and Communications Security (CCS)*.

[19] Victor Costan and Srinivas Devadas. 2016. *Intel SGX Explained*. Technical Report 2016/086. Cryptology ePrint Archive.

[20] Pubali Datta, Prabuddha Kumar, Tristan Morris, Michael Grace, Amir Rahmati, and Adam Bates. 2020. Valve: Securing Function Workflows on Serverless Computing Platforms. In *The Web Conference (WWW)*.

[21] Alex Davidson, Amit Deo, Ela Lee, and Keith Martin. 2019. Strong Post-Compromise Secure Proxy Re-Encryption. In *Australasian Conference on Information Security and Privacy (ACISP)*.

[22] T. Elgamal. 1985. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory* 31, 4 (July 1985).

[23] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *ACM Symposium on Cloud Computing (SOCC)*.

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

[25] Guerney D. H. Hunt, Ramachandra Pai, Michael V. Le, Hani Jamjoom, Sukadev Bhattiprolu, Rick Boivie, Laurent Dufour, Brad Frey, Mohit Kapur, Kenneth A. Goldman, Ryan Grimm, Janani Janakirman, John M. Ludden, Paul Mackerras, Cathy May, Elaine R. Palmer, Bharata Bhasker Rao, Lawrence Roy, William A. Starke, Jeff Stuecheli, Enriquillo Valdez, and Wendel Voigt. 2021. Confidential computing for OpenPOWER. In *European Conference on Computer Systems (EuroSys)*.

[26] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. 2016. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *Symposium on Operating Systems Design and Implementation (OSDI)*.

[27] Intel 2014. *Intel Software Guard Extensions Programming Reference*. Intel.

[28] Intel. 2021. *Intel Trust Domain Extensions White Paper*. Technical Report. Intel.

[29] Anca-Andreea Ivan and Yevgeniy Dodis. 2003. Proxy Cryptography Revisited. In *Network and Distributed System Security Symposium (NDSS)*.

[30] Deepak Sirone Jegan, Liang Wang, Siddhant Bhagat, and Michael Swift. 2023. Guarding Serverless Applications with Kalium. In *USENIX Security Symposium*.

[31] David Kaplan. 2017. *Protecting VM Register State with SEV-ES*. Technical Report. AMD.

[32] David Kaplan, Jeremy Powell, and Tom Woller. 2021. *AMD Memory Encryption*. Technical Report. AMD.

[33] Seong-Joong Kim, Myoungsung You, Byung Joon Kim, and Seungwon Shin. 2023. Cryonics: Trustworthy Function-as-a-Service using Snapshot-based Enclaves. In *ACM Symposium on Cloud Computing (SOCC)*.

[34] Mengyuan Li. 2022. *Understanding and Exploiting Design Flaws of AMD Secure Encrypted Virtualization*. Ph. D. Dissertation. https://etd.ohiolink.edu/.

[35] Mingyu Li, Yubin Xia, and Haibo Chen. 2021. Confidential Serverless Made Efficient with Plug-In Enclaves. In *International Symposium on Computer Architecture (ISCA)*.

[36] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. 2019. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *USENIX Security Symposium*.

[37] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *USENIX Security Symposium*.

[38] Arm Limited. 2021. *Arm Confidential Computer Architecture*. Technical Report. Arm Limited.

[39] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*.

[40] Yuriy Polyakov, Kurt Rohloff, Gyana Sahu, and Vinod Vaikuntanathan. 2017. Fast Proxy Re-Encryption for Publish/Subscribe Systems. *ACM Transactions on Privacy and Security* 20, 4 (sep 2017).

[41] Mark Russinovich. 2023. Confidential Computing: Elevating Cloud Security and Privacy. *ACM Queue* 21, 4 (Sept. 2023).

[42] Amit Sahai, Hakan Seyalioglu, and Brent Waters. 2012. Dynamic Credentials and Ciphertext Delegation for Attribute-Based Encryption. In *International Cryptology Conference (CRYPTO)*.

[43] Nuno Santos, Krishna P. Gummadi, and Rodrigo Rodrigues. 2009. Towards Trusted Cloud Computing. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*.

[44] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *USENIX Annual Technical Conference (ATC)*.

[45] Bohdan Trach, Oleksii Oleksenko, Franz Gregor, Pramod Bhatotia, and Christof Fetzer. 2019. Clemmys: Towards Secure Remote Execution in FaaS. In *ACM International Systems and Storage Conference (SYSTOR)*.

[46] Chia-Che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX Annual Technical Conference (ATC)*.

[47] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*.

[48] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *IEEE Symposium on Security and Privacy*.

[49] Shixuan Zhao, Pinshen Xu, Guoxing Chen, Mengya Zhang, Yinqian Zhang, and Zhiqiang Lin. 2023. Reusable Enclaves for Confidential Serverless Computing. In *USENIX Security Symposium*.

Received 2 May 2025