UNIVERSITY OF CALIFORNIA
SANTA CRUZ

BUILDING SECURE DISTRIBUTED APPLICATIONS THE DECENT WAY

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Haofan Zheng

December 2024

The Dissertation of Haofan Zheng
is approved:

Professor Owen Arden, Chair

Professor Peter Alvaro

Professor Lindsey Kuper

Peter F. Biehl
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Building Secure Distributed Applications the Decent Way

by

Haofan Zheng

The core principles of information security — Confidentiality, Integrity, and Availability — are increasingly challenged as modern applications rely heavily on remote services to provide essential functionalities. Offloading complex computations to powerful servers and synchronizing user data across multiple devices have become common practice. This system model typically involves a large number of users, a service provider, third-party services, and cloud providers.

In this setup, users must trust both service providers and cloud providers to protect their data and deliver services as promised. Similarly, service providers rely on cloud providers to secure their data and ensure service integrity. This trust model is largely based on legal frameworks and the reputations of the service and cloud providers. However, such mechanisms can only offer compensation after a breach occurs; they do little to prevent potential damage. Thus, a verifiable and decentralized trust model that can proactively ensure application security is highly desirable.

Existing solutions, such as End-to-End Encryption, Fully Homomorphic Encryption, and Multi-Party Computation, offer only limited functionalities, are computationally expensive, or focus primarily on confidentiality and integrity while providing little to no availability guarantees.

In this dissertation, we propose the Decent Framework, a novel approach to building secure distributed applications. The Decent Framework integrates secure enclaves with blockchain technology to provide comprehensive Confidentiality, Integrity, and Availability guarantees. A secure enclave is a cryptographically protected memory region, isolated from the rest of the system, including the operating system, ensuring confidentiality and integrity of the data and code within. Meanwhile, blockchain technology achieves high availability through global replication across thousands or even millions of nodes.

However, several challenges arise in building this framework. The native remote attestation protocol introduces significant overhead for short-lived serverless components and lacks support for mutual attestation. To address this, the Decent Framework introduces self-attestation certificates, which reduce the overhead of remote attestation significantly. By embedding an AuthList within the certificate, it enables enclaves to authenticate each other without relying on a trusted third party.

Additionally, enclaves are susceptible to eclipse attacks, where an attacker controls all network connections to the enclave, making it difficult to determine the legitimacy of received blockchain data. On the blockchain side, no existing work addresses the challenge of performing remote attestation on-chain or verifying enclave RA reports and signatures. We present a novel algorithm that reliably detects eclipse attacks by monitoring fluctuations in block difficulty.

To provide message availability in the Decent Framework, we developed Decentagram, a highly available publish/subscribe system that guarantees the timely delivery of critical messages. Decentagram also implements revocation mechanisms to distribute component revo-

cation lists to enclaves within the system. Finally, we introduce AlacriTEE, a fully decentralized Function-as-a-Service (FaaS) platform built on secure enclaves and blockchain, which ensures the availability and integrity of services provided by enclave components.

To my mother, Jianshi Zhang.

Acknowledgments

I want to take this opportunity to express my appreciation to all the amazing people who supported me during my Ph.D. journey at UC Santa Cruz. This achievement would not have been possible without your guidance, encouragement, and assistance.

First and foremost, I extend my deepest thanks to my advisor, Professor Owen Arden, whose unwavering support and insightful guidance have been invaluable throughout my Ph.D. journey. I feel truly honored to be one of his first Ph.D. students. When I began my Ph.D., cloud computing was a hot topic, and I was curious about finding the ultimate solution for securing it. Owen introduced me to the concepts of Secure Enclaves and Blockchain, and I immediately felt like I had found the answer to my question. This direction became the foundation of my research for over seven years, and I have never regretted pursuing it. Owen's mentorship allowed me to focus on my studies and research without worrying about external distractions. His thoughtful feedback and constant encouragement have been instrumental to my success. Being part of Decent Lab under his guidance has been an amazing experience to me.

I would also like to thank the other members of my dissertation committee, Professors Peter Alvaro and Lindsey Kuper, for their invaluable insights and feedback on my dissertation. Taking Peter's distributed systems course early in my Ph.D. was pivotal, as it provided me with the foundational knowledge necessary to propose security solutions for distributed systems. And the LSD seminar led by Lindsey offered a refreshing perspective on programming languages, systems, and data, broadening my understanding of diverse research areas. These seminars often provided inspiration when I needed a mental break from my primary research.

I am deeply grateful to my labmates in Decent Lab: Priyanka Mondal, Roy Shadmon, and Tuan Tran. Priyanka and I joined UCSC in the same quarter and supported each other through the early, often confusing years of our Ph.D. journey. Her encouragement, especially during times of paper rejections, has been a source of motivation for everyone in the lab. Roy's guidance when I started programming in Smart Contracts was invaluable, helping me acquire critical skills early on. Tuan's deep understanding of Blockchain protocols and BFT systems has been truly impressive. Collaborating with him on various projects has been an intellectually rewarding experience.

I also want to express my gratitude to my family and friends for their unwavering support and encouragement throughout this journey. My parents, in particular, have always believed in me and encouraged me to follow my dreams. My interest in computer science began during primary school when I first played with a computer, and their belief in me has made this achievement possible. Special thanks to my cousin Dandan Hong and my friend Zhengjun Wang, whose guidance as experienced Ph.D. students helped me find my footing and get on the right track.

I am also deeply appreciative of my colleagues and mentors from the syGlass project at West Virginia University. During the last two years of my undergraduate studies, I worked on this project under the mentorship of Michael Morehead, Professor Gianfranco Doretto, and Professor George Spirou. They introduced me to the excitement of research — exploring novel ideas and bringing them to life through programming. This experience inspired me and gave me the confidence to pursue a Ph.D. I would like to especially thank Professor Raymond Morehead, who taught my File and Data Structures course at WVU. Though he is no longer with us, his

belief in me led me to the syGlass project, setting me on the path of Ph.D. study, and the world of research. His mentorship and teachings, including the invaluable skill of how to effectively Google for information, have stayed with me throughout my career.

Finally, to everyone who supported me along the way but whose names I have not include here — thank you. Each of you has contributed to this achievement in meaningful ways, and I will always remember and appreciate your help, no matter how big or small.

Chapter 1

Introduction

1.1 Building a secure distributed applications

Modern applications increasingly rely on remote services to deliver core functionalities to users. This model enables offloading complex computations to powerful servers and allows user data to be synchronized across multiple devices. However, in such systems, users must trust service providers to protect their data and deliver the promised services.

On the other side, a significant portion of service providers rely on third-party cloud providers to host their services [125, 73]. As a result, service providers must trust cloud providers to secure their data and maintain the integrity of their services. This trust model is primarily based on legal agreements and the reputation of both the service and cloud providers. In the event of a breach, the service provider can sue the cloud provider for damages, and similarly, users can take legal action against the service provider for data misuse. However, this reactive model fails to prevent harm before it occurs. It compensates victims after the fact and deters

perpetrators from repeating their mistakes, but it is far from ideal—particularly for applications handling highly sensitive data, such as financial transactions or medical records. Thus, there is a pressing need for a verifiable mechanism that can ensure application security and prevent breaches from occurring in the first place.

In practice, the trust model for popular applications is far more intricate than a straightforward relationship between users, service providers, and cloud providers. Many service providers rely on multiple cloud providers for different parts of their infrastructure, with 80% of organizations using more than one public cloud provider as part of hybrid or multi-cloud strategies [73]. They may also source third-party services to handle specific functionalities they lack the expertise or resources to build internally. On the user side, a widely used application can attract a large number of users, each with different beliefs and trust requirements. In such a complex system, it is impractical for a single entity to make all trust decisions while satisfying all trust requirements. Therefore, a decentralized trust model is more desirable — one where each entity can make its own trust decisions while the system still functions as expected.

However, building distributed applications with decentralized trust models poses significant challenges. Simple yet highly sensitive applications, like password managers, can employ end-to-end encryption to protect user data from service and cloud providers. In this approach, users encrypt data locally before sending it to the remote server, using their own encryption keys. Since encrypted data cannot be directly processed by the cloud, most computations must be offloaded to the client. To address this, there are two primary cryptographic approaches: Fully Homomorphic Encryption (FHE) and Multi-Party Computation (MPC). While FHE allows computations to be performed directly on encrypted data, it is computationally

expensive [133] and only practical for certain operations and applications [10, 82]. Similarly, MPC suffers from high computational costs due to the overhead of secret sharing [57, 161].

In this dissertation, we propose using secure enclaves, a type of Trusted Execution Environment (TEE), to build components in distributed applications. A secure enclave is a hardware-based security mechanism that isolates the execution of a program from all other software layers, including the operating system. Remote Attestation (RA) provides a means to verify that 1) the code is running inside genuine enclave hardware, and 2) the code matches the expected version. This guarantees both users and service providers that the application is running as intended and that the data is protected from untrusted parties. Prior work has demonstrated the security of remote execution within enclave environments [154]. Compared to FHE and MPC, hardware enclaves introduce significantly lower computational overhead but require trust in the CPU manufacturer to produce and authenticate the hardware reliably. Arguably, this trust in the CPU manufacturer already exists in our daily computational tasks. Most importantly, in the Decent Framework, both users and service providers have the flexibility to choose the CPU manufacturer they trust, ensuring that no single manufacturer holds a monopoly on trust within the system.

The aforementioned approaches primarily focus on confidentiality and integrity. However, information security is defined by three pillars: Confidentiality, Integrity, and Availability. An adversary can disrupt the availability of a service by shutting down servers, blocking network traffic, or launching Denial-of-Service (DoS) attacks.

To address this, we propose integrating secure enclaves with blockchain technology to ensure service availability. Blockchain provides guarantees of both integrity and availability.

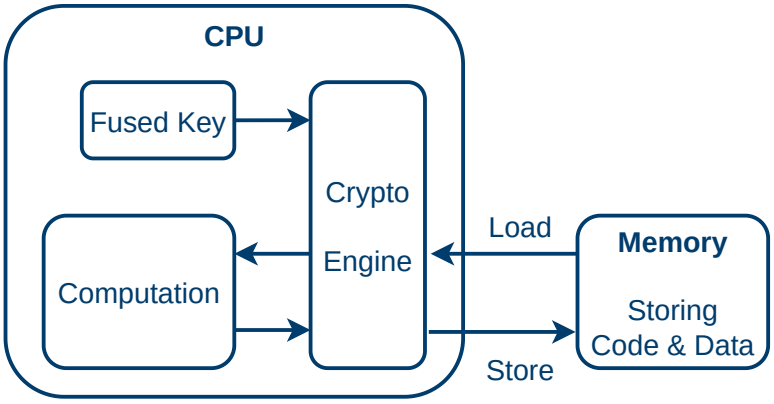


Figure 1.1: Overview of the general design of a secure enclave.

Like other availability-focused solutions, blockchain relies on data replication to ensure availability. As long as enough full nodes are operational, clients can always access at least one node with the required data. Data integrity is ensured through a consensus mechanism, which checks data hashes across peers to detect and reject tampered data. Unlike traditional replication solutions, blockchain operates with a large number of nodes [21], and data is fully replicated across all of them. In permissionless blockchains, such as Ethereum, anyone can join the network and become a full node. This eliminates the need to trust any single entity for service provision, satisfying the requirements of a decentralized trust model and offering exceptionally high availability.

1.2 Background

Secure Enclaves. Secure enclaves are hardware-based security mechanisms that isolate the code and data of a program from the rest of the system, including the operating system. As shown in Figure 1.1, the CPU is equipped with a secret key that is fused into the hardware

during manufacturing. This key is used to encrypt both the code and data of an enclave application. During execution, the CPU loads the encrypted code and data, decrypting them on-the-fly. When storing computation results, the CPU encrypts the data before writing it back to memory. Since the encryption key is exclusively known by the CPU, no other software layer can access or tamper with the actual data. This ensures the *confidentiality* and *integrity* of the data, while also protecting the code from tampering, thus ensuring *code execution integrity*.

All of these processes occur at the hardware level, making them invisible to the human eye. Therefore, without verification, an attacker could run the application in a regular environment and claim it is operating inside an enclave. To address this, it is essential to verify that the *expected* code is running inside a *genuine* enclave. This process is known as *Remote Attestation (RA)*. During RA, the enclave hardware generates a digitally signed report containing a hash of the program, which the verifier (e.g., a user) can validate with the hardware manufacturer to ensure that the report was generated by genuine enclave hardware produced by them.

Blockchain. As illustrated in Figure 1.2, a blockchain is composed of a sequence of blocks, each linked to the previous one through a hash. Every block contains a list of transactions, representing operations on the data stored within the system. Any alteration to the content in an older block results in a change to its hash, which invalidates the subsequent block in the chain. To restore the chain's validity, the hashes of all blocks following the modified block must be recomputed. This mechanism allows for easy detection of tampering through hash mismatches, ensuring the *integrity* of the data stored in the blockchain.

Furthermore, the blockchain network is decentralized, with blocks replicated across

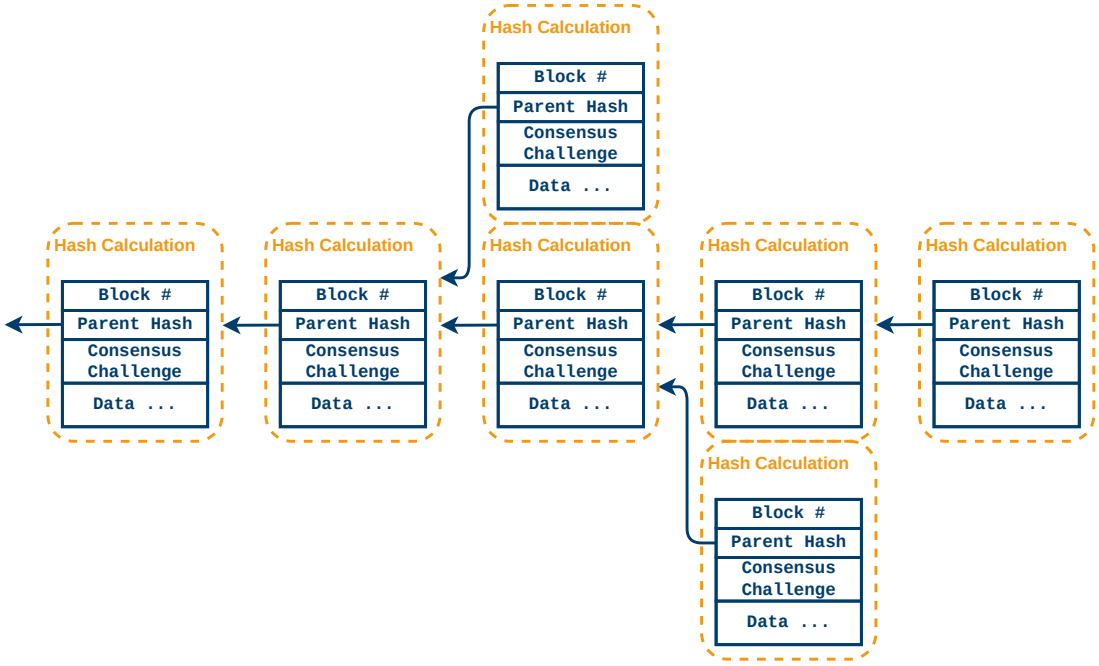


Figure 1.2: Overview of the general consensus protocol of a blockchain.

numerous nodes. In permissionless blockchains like Ethereum, the network can consist of thousands of nodes. This replication guarantees extremely high *availability* of the data. Additionally, it enhances the guarantee of integrity: by comparing the hash of the most recent block with those held by peer nodes, the integrity of the data across the entire network can be continuously verified.

1.3 Challenges

There are several challenges in building a framework that integrates secure enclaves with blockchain.

First, the native remote attestation protocol is designed for a client-server model,

where either the client runs the enclave program (as in Digital Rights Management) or the server runs the enclave program (for secure remote computation). In distributed applications, this model is useful when a client needs to authenticate a remote enclave program. However, mutual attestation is critical in distributed systems where different services, running in separate enclaves, must authenticate each other before exchanging sensitive data. To address this issue, some systems (e.g., Intel SGX [98]) allow service providers to sign their enclave programs so that programs bearing the same signature can authenticate each other freely. While this approach simplifies authentication, it introduces a single point of failure: the service provider. If the service provider or its signing key is compromised, an attacker could inject a malicious enclave program into the system, compromising the entire application. This risk undermines the decentralized trust model that this framework seeks to achieve.

Integrating secure enclaves with blockchain introduces two additional challenges: 1) How enclave programs receive blockchain data. 2) How blockchain applications (i.e., smart contracts) authenticate messages sent by secure enclaves through blockchain transactions.

For the first challenge, a regular program would typically connect to multiple peers and assume that at least one is honest. However, in our threat model, the attacker controls the operating system, which manages the network stack. This means that no matter how many peers the enclave program tries to connect with, all connections could potentially be controlled by the attacker. In contrast to HTTPS connections, where a client can authenticate a server using X.509 certificates, a permissionless blockchain network allows anyone to join as a full node, so there are no certificates to guarantee whether a peer is trustworthy. In an extreme scenario, an attacker could launch an *eclipse attack* by spawning numerous malicious nodes and rerouting all of the

enclave's connections to those compromised nodes. When the enclave receives a block from the network, it becomes challenging to determine whether to accept or reject the block.

The second challenge is that the enclave program must send authenticated messages to smart contracts via blockchain transactions. However, the Remote Attestation (RA) protocol does not natively exist within blockchain networks. Implementing RA in the blockchain requires integrating it into the smart contract itself, where the smart contract would need to verify the RA report, as well as validate signatures from the enclave hardware and the hardware manufacturer. This task is challenging because the cryptographic primitives used by the enclave program differ from those used in the blockchain network.

Another major challenge relates to ensuring availability in the system. The blockchain provides a highly available data store and smart contracts for our framework, but utilizing them to guarantee availability requires careful consideration. There are two key aspects of availability: 1) **Availability of critical messages**: For example, when a message containing a revocation list is generated, it is essential that all enclaves in the system receive this message promptly, so they can cease exchanging sensitive data with the revoked enclave. 2) **Availability of services**: When a service provider deploys enclave components across several compute nodes in the system, it is crucial to ensure that these services remain available to users when needed.

Effectively addressing these challenges is vital for the security, functionality, and the 3 information security guarantees of a decentralized framework that integrates secure enclaves with blockchain.

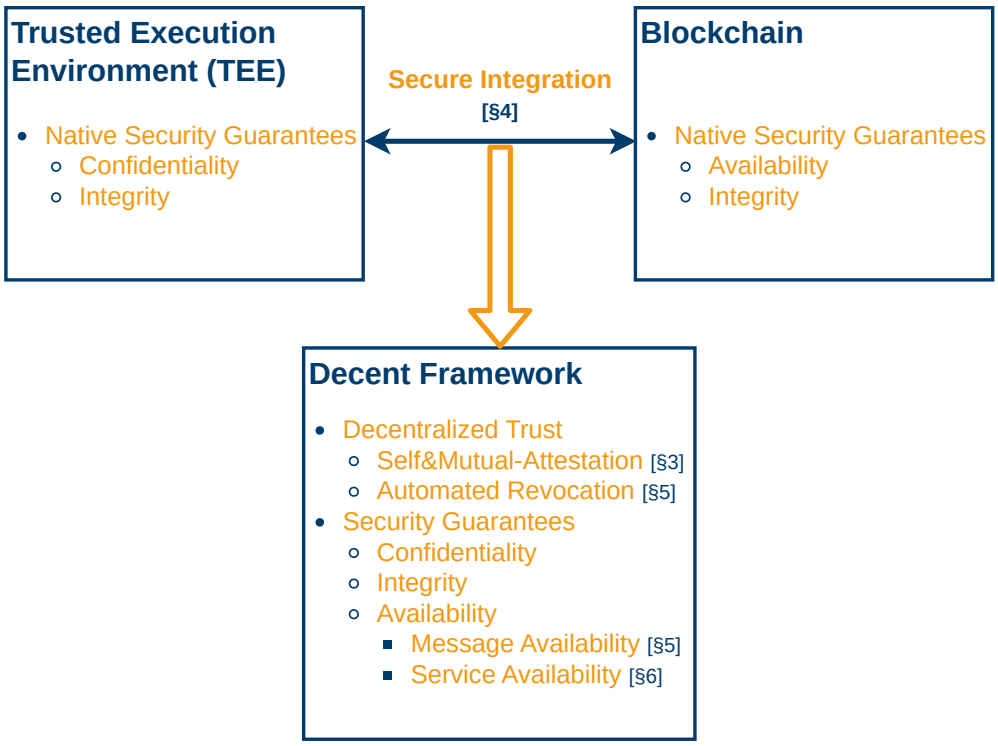


Figure 1.3: Overview of the Decent Framework.

1.4 Contributions

In this dissertation, we present:

The Decent Framework, which provides the security guarantees of Confidentiality, Integrity, and Availability, using Trusted Execution Environment and blockchain, with decentralized trust.

Figure 1.3 provides an overview of the Decent Framework, and the key contributions of this dissertation are as follows:

Core Infrastructure: Decent Framework [183]. We introduce the Decent Framework, a foundational infrastructure for building secure distributed applications using secure enclaves.

To support short-lived, serverless service components, we developed self-attestation certificates that significantly reduce the overhead of remote attestation for newly spawned enclave instances. By embedding an AuthList into the self-attestation certificates, we enable enclaves to authenticate each other without relying on a trusted third party. To demonstrate the expressiveness of the Decent Framework, we built a sample microservice-based ride-sharing platform. Additionally, we use ProVerify to formally prove the correctness of the mutual attestation protocol in the Decent Framework.

Connecting Enclaves with Blockchain: Detecting Eclipse Attacks from Inside Enclaves [186]. We present a novel algorithm to detect eclipse attacks from within enclaves by continuously monitoring changes in block difficulty. Through statistical analysis, we demonstrate that the false negative rate of our algorithm is equivalent to brute-forcing a 128-bit key. We further show that the false positive rate is negligible in practice by running the algorithm on all historical blocks of the Ethereum network.

Ensuring Message Availability: Decentagram [187]. We introduce Decentagram, a highly available publish/subscribe system built on secure enclaves and blockchain. By implementing the broker as a smart contract, we ensure timely delivery of critical messages both on-chain and off-chain. To authenticate enclave messages, we implemented the RA protocol within the smart contract and verified RA reports and signatures from both the enclave hardware and the hardware manufacturer. To showcase Decentagram's usefulness within the Decent Framework, we implemented 3 revocation mechanisms: revocation by votes, revocation by conflicting messages, and revocation by leaked keys. The smart contract verifies evidence in each scenario

and adds the hash of compromised enclave programs to the revocation list. Decentagram ensures that enclaves either receive the revocation message or detect network blocking, at which point they stop exchanging sensitive data. Our evaluation shows that the gas costs for Decentagram's blockchain transactions are reasonable and the block processing throughput is sufficient to handle message publication in new blocks.

Ensuring Service Availability: AlacriTEE [157]. We present AlacriTEE, a fully decentralized Function-as-a-Service (FaaS) platform built on secure enclaves and blockchain. AlacriTEE provides a two-way sandboxed environment, where the service provider's workload runs securely inside the enclave. This ensures the security of code execution and data protection for the user and service provider. Simultaneously, the host (e.g., the cloud provider) is assured that the workload will not damage the host system, as enclave programs are isolated from all other software layers. AlacriTEE also leverages WebAssembly and a code instrumentor to track the service provider's resource usage. The service provider and host sign a Service Level Agreement (SLA) in the form of a smart contract, and resource usage is used to calculate rewards for the host, charged from the service provider's deposit. Conversely, if the host fails to meet the agreed resource provisions, it is penalized by the smart contract.

1.5 Dissertation Outline

The remainder of this dissertation is organized as follows: Chapter 2 provides essential background information on secure enclaves and blockchain. Chapter 3 presents the core infrastructure of the Decent Framework. Chapter 4 introduces our novel solution for detecting

eclipse attacks from within enclaves. Chapter 5 describes the highly available publish/subscribe system built on secure enclaves and blockchain. Chapter 6 presents a fully decentralized FaaS platform that ensures the availability of services provided by enclave components. Finally, Chapter 7 concludes this dissertation.

Chapter 2

Background

2.1 Trusted Execution Environments

2.1.1 Code and Data Protection

Figure 1.1 provides an overview of the general design of a secure enclave. In a secure enclave, both the code and data are cryptographically encrypted and stored in memory. The CPU is equipped with a root secret key that is fused into the hardware during manufacturing, and this key is only known to and accessible by the CPU. During execution, the CPU loads the encrypted code and data into the enclave and decrypts them on-the-fly. When storing computation results, the CPU encrypts the data before writing it back to memory. Since the encryption key is exclusively known by the CPU, no other software layer, including the operating system that has Direct Memory Access (DMA) privileges, can access or tamper with the actual data. Additionally, the CPU's cryptographic engine derives unique keys for different enclave programs, ensuring that the data of one enclave remains inaccessible to others.

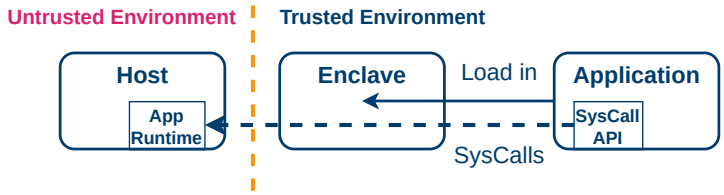


Figure 2.1: System model of an application-based secure enclave.

On top of this general design, manufacturers implement secure enclaves in various ways. For example, Figure 2.1 illustrates the system model of an application-based secure enclave used by Intel SGX. Applications are specifically built to run inside the enclave directly. When the application is launched, the enclave platform loads the application binary, computes its hash, and begins executing the code. For tasks that cannot be handled by the CPU alone, such as file I/O or network communication, the requests must be forwarded to the host operating system (OS). Since the host OS is not part of the enclave's trusted model, the platform does not provide an API for direct interaction with the OS. Instead, the enclave application packages system call requests, forwards them to the untrusted side, and manages how the OS handles them. During this forwarding process, the enclave application itself is responsible for ensuring that no secret data is leaked and that any malicious responses are detected and appropriately handled.

This model is ideal for applications with a small footprint that require high security guarantees. Since the application has no initial access to the outside world and developers must explicitly define the interface between the enclave and the untrusted side, it is easier to reason about the security properties of the enclave and maintain a small Trusted Computing Base (TCB). However, this approach has limitations. Rebuilding an existing complex application,

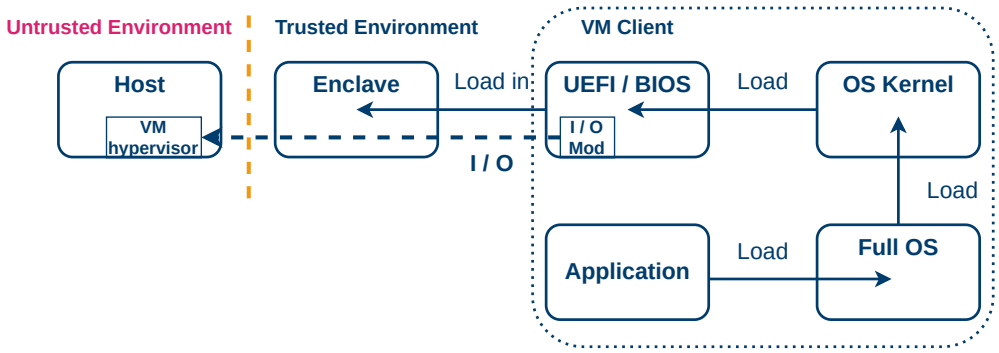


Figure 2.2: System model of a VM-based secure enclave.

which heavily depends on the host OS, into a sanitized setup with no OS dependencies is a non-trivial task. Additionally, the process of implementing system call requesters and handlers becomes tedious for the developer.

To address this, another type of secure enclave has been developed. Figure 2.2 illustrates the system model of a VM-based secure enclave, used by AMD SEV and Intel TDX. Instead of directly loading an application binary, the enclave platform loads a UEFI/BIOS image and computes its hash. The UEFI/BIOS image then boots an entire OS within the enclave environment, and the OS inside the enclave eventually executes the application. In this model, I/O modules within the UEFI image send system call requests to the untrusted side, while the hypervisor running on the untrusted side handles these requests. From the perspective of the OS and application inside the enclave, I/O and system calls are handled as usual. This allows applications to be built just as they would be on a regular machine.

The downside of this model is a significantly larger TCB compared to the application-based model. The entire OS running inside the enclave becomes part of the TCB, meaning any vulnerabilities in the OS could potentially compromise the enclave's security. Additionally,

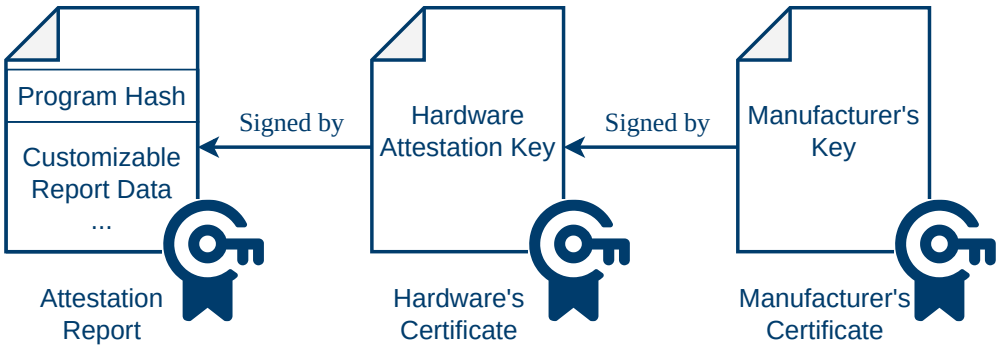


Figure 2.3: Overview of a typical chain of trust in RA.

applications designed to run on a regular OS may have different threat models compared to those running inside an enclave. For example, in a regular threat model, it may be safe to print sensitive data to the screen or write it to the disk, as the OS is trusted. However, in the threat model of a secure enclave, the OS is untrusted, and these actions could result in data leakage.

2.1.2 Remote Attestation

While runtime hardware encryption and decryption protect code and data from unauthorized access, it can be difficult for a user to verify whether code is genuinely running inside a secure enclave. An adversary could claim that the code is executing in an enclave while, in reality, it is running in an unprotected environment. Alternatively, malicious code could be executed within the enclave, masquerading as the expected program.

To address this issue, secure enclaves offer a feature called Remote Attestation (RA). The key idea behind RA is that the enclave hardware computes a hash of the program when it is loaded into the enclave and signs an attestation report containing the hash with a private key known only to the CPU. The user can then verify the signature using the hardware manu-

Figure 2.4: Overview of the EPID RA protocol.

facturer's public key, ensuring that the code is running inside the enclave. By comparing the program's hash to the expected hash, the user can also confirm that the code is authentic.

Figure 2.3 illustrates a typical chain of trust in the RA process. The hardware manufacturer maintains its own signing key pair, with the public key serving as the root of trust for the RA process. During manufacturing, an attestation signing key pair is embedded in the hardware; the private key remains exclusive to the hardware, and the public key is certified by the hardware manufacturer. When a program is loaded into the enclave, the hardware computes the program's hash and signs an attestation report with the attestation private key. The user can then verify that the attestation report was signed by the enclave's attestation key, which is in turn certified by the manufacturer's signing key.

Similar to the different system models of secure enclaves, various RA protocols are implemented across different platforms. Figure 2.4 provides an overview of the Enhanced Privacy ID (EPID) RA protocol used by Intel SGX. One unique feature of the EPID protocol is its support for anonymity, meaning the verifier cannot link the attestation report to a specific enclave hardware [98]. This feature was originally introduced in SGX, as it was initially a consumer-facing product, with the anonymity feature designed to protect user privacy.

In EPID RA, the enclave hardware generates a quote containing the enclave's hash and signs it with the EPID private key. The verifier receives the quote and forwards it to the

Intel Attestation Service (IAS), which verifies the signature and returns an attestation report containing the quote, signed by the IAS private key. The verifier can then verify the IAS signature using the IAS public key, ensuring the attestation report was generated by IAS. By reading the hash from the quote in the attestation report, the verifier can confirm that the code is running inside the enclave and matches the expected version. This protocol closely follows the general concept of RA introduced earlier: the hardware manufacturer certifies the enclave hardware, and the enclave hardware generates and signs a report containing the hash of the code running inside the enclave.

2.1.3 Threat Model

In this dissertation, we adopt common assumptions regarding the security of secure enclaves. We assume that an attacker has full control over the physical host running the enclave program and can delay, drop, or reorder messages between the enclave and the network. If network messages are in plaintext, the attacker may also read and modify them. We assume that users can verify the implementation of the enclave program and maintain a record of the hashes of trusted enclave programs. We further assume that the enclave implementation correctly enforces the security abstractions it claims to provide. These abstractions, as briefly discussed above, are fundamental to the security guarantees of the enclave. Attacks such as key extraction or side-channel attacks on specific enclave implementations (e.g., Intel SGX, AMD SEV) are outside the scope of this dissertation. Additionally, we assume that attackers cannot break the cryptographic primitives employed by the enclave hardware or this dissertation.

2.2 Blockchain

2.2.1 Consensus Mechanisms

Figure 1.2 provides an overview of the general design of a blockchain. The blockchain begins with a genesis block, and miner nodes, which are participants interested in generating new blocks and earning block rewards, package transactions and data into each subsequent block. The `Parent Hash` field within each block contains the hash of the previous block, which cryptographically links the blocks together, forming the blockchain. Any tampering with the data in an older block alters its hash, which in turn invalidates the next block in the chain. To restore validity, an attacker would need to recompute the hashes of all blocks following the tampered block. Even then, such tampering is easily detected by comparing the hash of the most recent block to the one stored in the public ledger.

In a decentralized network, multiple miners may attempt to generate blocks for the same block height, resulting in a fork. The network resolves forks using a consensus protocol, where honest miners will always extend the longest chain. This ensures that the network eventually converges to a single, accepted chain, confirming the transactions within it.

However, the presence of numerous miners creating blocks with arbitrary transactions requires a mechanism to decide which miner has the right to produce the next block. Blockchain networks are generally categorized into two types: permissioned and permissionless. In a permissioned network, only authorized nodes can generate blocks, and participants can verify block legitimacy using digital signatures. This setup implies that participants must trust these authorized nodes to produce blocks correctly.

In contrast, permissionless blockchains allow anyone to join the network and attempt to generate new blocks. Without restrictions on when and who can create a block, thousands of nodes could compete to generate blocks simultaneously, causing network instability. Additionally, an attacker could create a large number of fake identities, posing as different miners within the network. If these puppet miners form a majority, the attacker can easily take control of the network. This type of attack is known as a Sybil attack [60].

To reduce the possibility of forking, permissionless blockchains establish a block time, which sets the time interval between consecutive blocks. This ensures sufficient time for blocks to propagate across the network. To address the Sybil attack issue, blockchains employ a challenge mechanism to determine which miner is entitled to generate the next block. These measures help the blockchain grow at a steady pace, reduce the frequency of forks, and significantly increase the difficulty of gaining a majority in the network.

Two primary challenge mechanisms used in permissionless blockchain networks are Proof-of-Work (PoW) and Proof-of-Stake (PoS).

Proof-of-Work (PoW). In a proof-of-work (PoW) network, miners compete to solve a cryptographic puzzle at a specified difficulty level. This puzzle is tied to the block data and the hash of the previous block, ensuring that miners can only begin solving it after the content of the new block has been finalized. The first miner to solve the puzzle earns the right to generate the next block. Each miner uses computational power to solve the puzzle, meaning that, with a constant difficulty level, an increase in the number or power of miners leads to faster puzzle resolution. Consequently, miners with greater computational resources are more likely to solve the puzzle

first and produce the next block. To maintain a consistent average block time, the network dynamically adjusts the difficulty level. When blocks are being generated faster than the expected time, the difficulty increases to slow down block production; conversely, when block generation is slower than expected, the difficulty decreases.

Proof-of-Stake (PoS). Proof-of-stake (PoS) networks take a different approach by using stake rather than computational power. Nodes that wish to generate new blocks must lock up a certain amount of cryptocurrency as collateral. The more cryptocurrency a node stakes, the higher the probability it will be selected to create the next block. This selection is based on a deterministic algorithm that randomly chooses a node, weighted by the amount of cryptocurrency it has staked. Unlike PoW, PoS does not require miners to solve complex cryptographic puzzles, significantly reducing energy consumption and eliminating the computational waste associated with PoW. However, miners must possess a minimum amount of cryptocurrency to participate, which can create a barrier to entry for new or smaller participants.

2.2.2 Smart Contracts

Smart contracts are programs deployed on the blockchain that interact with transactions and data. They are triggered by function calls within transactions or other smart contracts. Each node in the blockchain network maintains the latest state of the blockchain. When a transaction includes a function call to a smart contract, all nodes independently execute the function and update the state of the blockchain accordingly. While a malicious node could attempt to alter the outcome of a smart contract execution, this would only affect that particular node.

Other benign nodes will still execute the contract correctly and achieve the expected result. This redundancy ensures that the network remains secure and tamper-resistant.

2.2.3 Transaction Cost

Blockchain transactions consume the computational resources of the network, as all nodes must execute the transaction and reach a consensus on the outcome. To prevent spam and ensure the network remains functional, blockchain networks impose a fee for each transaction. In EVM-based blockchains such as Ethereum, this fee is known as *gas*. When initiating a transaction, the sender specifies the gas price they are willing to pay and the maximum gas they are prepared to use.

During transaction execution, the Ethereum Virtual Machine (EVM) tracks the gas consumed based on the instructions executed. Once execution completes, the total gas consumed is calculated, and the corresponding fee is deducted from the sender's balance. If the gas consumed exceeds the specified gas limit, the transaction is reverted, but the sender must still pay for the gas used up to that point.

For simple transactions, such as transferring cryptocurrency between accounts, the gas cost is relatively low. However, more complex transactions, such as function calls to smart contracts, may incur higher gas costs. The gas required depends on the complexity of the function — the more instructions executed, the more gas consumed. Consequently, one of the main goals for smart contract developers is to design contracts that are as simple and efficient as possible to minimize transaction costs.

2.2.4 Layer-2 Solutions

The financial cost of a blockchain transaction depends not only on the complexity of the transaction but also on the price dynamics of the blockchain network. For example, at the time of writing, a transaction consuming 2 million gas on the Ethereum network costs approximately $81.50. Additionally, Ethereum generates blocks every 12 seconds. These limit its appeal for applications requiring low latency and high throughput.

To address these challenges, Layer-2 solutions have been developed [44]. Layer-2 blockchains are built on top of existing blockchain networks, offering faster block times and significantly lower transaction costs. With fewer miners, Layer-2 blockchains can generate blocks much faster than the main blockchain (e.g., 2 seconds on Optimism and Polygon). While Layer-2 blockchains are more susceptible to reorganization (reorgs) due to competing blocks, they still provide the same security guarantees as the main blockchain. This is achieved through roll-up technology, where transactions are ultimately verified and confirmed by the main blockchain.

Furthermore, the financial benefits of Layer-2 solutions are substantial. For instance, the same transaction consuming 2 million gas would cost significantly less on a Layer-2 blockchain (e.g., approximately $0.0002 on Optimism and $0.0387 on Polygon).

2.2.5 Threat Model

For the purposes of this dissertation, we assume standard security conditions for blockchain systems. In a PoW network, we assume that an attacker does not control more than 50% of the network's total hash power, which would prevent them from blocking transactions, reversing transactions, or taking over the network [155]. In a PoS network, we assume

that attackers control no more than the fraction of the total stake tolerated by the blockchain protocol to ensure the network's availability and integrity. In Ethereum, this means the attacker must control no more than $\frac{1}{3}$ of the staked ETH (319k validators) to avoid halting the network's progress and no more than $\frac{1}{2}$ of the staked ETH (478k validators) to prevent forking the blockchain or forcing a reorganization. Nodes controlled by attackers may deviate from the blockchain protocol arbitrarily and can send arbitrary transactions to any smart contract or blockchain address. As with the enclave threat model, we assume that attackers cannot break the cryptographic primitives used by the blockchain network.

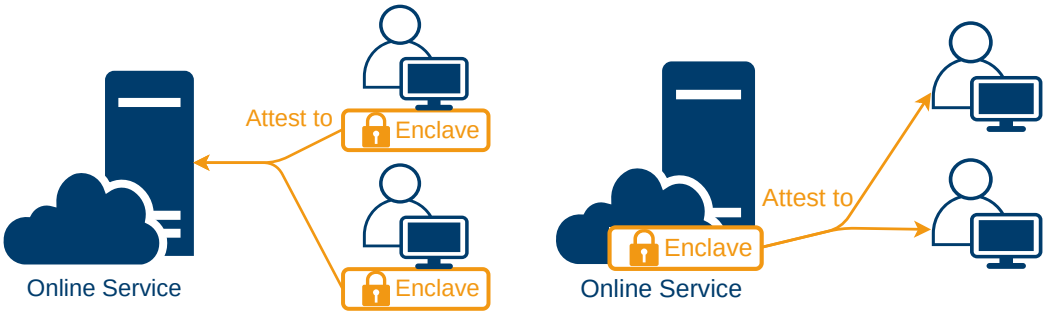
Chapter 3

The Decent Framework

3.1 Introduction

A fundamental challenge in building secure decentralized applications is that untrustworthy nodes may arbitrarily deviate from their expected behavior. A remote service may appear to execute a well-known system component when in fact it is executing a maliciously modified version. Trusted execution environments (TEEs) partially address this challenge by shifting trust from the host executing the component to the manufacturer of the host's hardware. By provisioning unique private keys to each TEE and certifying the corresponding public keys, third parties can authenticate messages produced within a genuine TEE as long as the key is kept secret.[1] Of particular interest are messages that attest to what code is executing within the TEE. These messages, called remote attestations (RAs), allow a remote node to prove it is executing an authentic system component.

[1] Doing so is not trivial: implementation errors, side channels, and physical attacks could potentially leak these keys. We assume the security of TEE platforms for this chapter, although that is demonstrably untrue for Intel SGX (e.g., [34]).



(a) Enclaves attest to a server.

(b) Enclave attests to clients.

Figure 3.1: Traditional RA authentication

Authenticating a TEE requires some degree of trust in a centralized entity such as the chip manufacturer. Once the TEE platform is authenticated, deciding whether to permit the TEE to access protected resources is determined by the entities that control those resources. A malicious application running within an authentic TEE should never be given access to secret data since the TEE does not prevent it from disclosing secrets to untrusted parties.

In a decentralized TEE application, mutually distrustful entities may wish to protect their resources within the same application, and may disagree on which entities are trusted. Conceptually, RA places *trust in code* at the center of a distributed application's security. Rather than consider whether a host will execute a component faithfully, developers can focus on the intrinsic behavior of the component to ensure their application behaves as expected.

Current TEE frameworks force programmers to work at the wrong level of abstraction where they must deal with many low-level protocol details. These frameworks work fine for simple scenarios where one host wishes to authenticate remote code running on another host, such as when a server wishes to authenticate client code, as illustrated in Figure 3.1a, or a

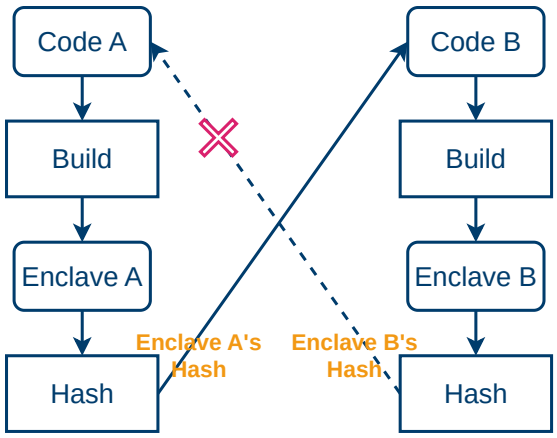


Figure 3.2: Mutual RA authentication. Enclave A and B cannot hardcode each other's identity directly in their code since it creates a circular dependency.

client wishes to authenticate server code, as illustrated in Figure 3.1b. To authenticate a server component, the server attests to a cryptographic hash of the code it is executing, signs it with its private keys and sends it to the client. The client verifies the signature to ensure the message originated from an authentic TEE and compares the hash to an expected value.

Even modest extensions of this scenario introduce challenges. Consider the scenario, illustrated in Figure 3.2, where two components wish to mutually authenticate each other. Each TEE attests to a cryptographic hash of the code it is executing and sends the signed attestation to the other host. Authenticating one component to the other is subtly different than Figure 3.1 because the verifying component must know what hash value to expect from the remote host. Otherwise, the component will be unable to distinguish authorized components from unauthorized ones. Hardcoding this value in the verifying component is not possible for both components because of a circular dependency: the hash of one component depends on the hash of the other component.

If we exclude expected code hashes when determining the hash used to identify a component, then each component must obtain the expected hashes at runtime. An honest component must therefore have a way of authenticating the hashes to prevent attackers (such as the component's host) from introducing malicious components in place of the honest component's dependencies.

Addressing this circular dependency forces many systems (e.g., [90, 137, 124, 147]) to introduce trusted third-parties to sign binaries or configurations to prevent malicious hosts from subverting applications by introducing a malicious component. We are unaware of prior work that solves the mutual authentication problem in its general form. Beekman et al. [22] propose a work-around that combines components into a single binary that is running in different modes for each component. This method clearly does not scale to applications with many components, and may not even be practical for moderately sized components if the memory available to the TEE is limited.[2]

Even if one component has hardcoded hashes, if any component it (transitively) depends on loads hashes dynamically, its security could be compromised. For example, in Figure 3.3, suppose component "A" authenticates "B" against a hash of its code. Since "B"'s hash is fixed in "A"'s code, a malicious host cannot substitute a malicious version of "B". However, suppose "B" loads the expected hash of "C" dynamically. Then if "B"'s host provides the hash of a malicious component for "C", "B" may leak "A"'s messages to "C". The core problem is that even though "A" authenticated "B"'s code, it could not authenticate which components "B" would trust.

[2]Intel SGX currently limits the size of the Enclave Page Cache (where enclave binaries are loaded) to about 90MB of usable space.

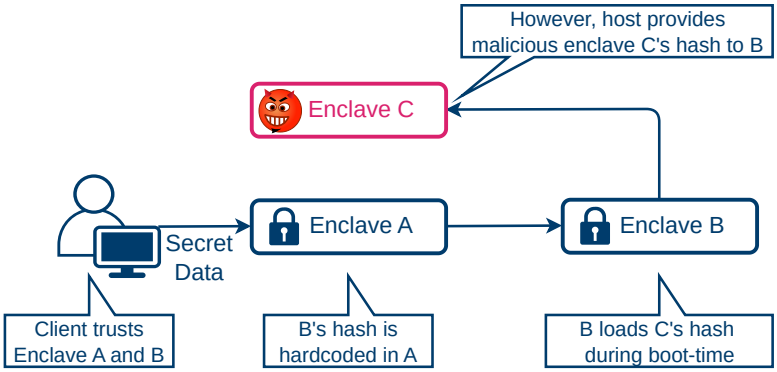


Figure 3.3: Allowing hosts to authorize enclaves independently could permit flows to malicious enclaves.

Since it is only possible to hardcode component hashes that exist at compile time, most TEE applications with multiple components face some version of the above problems. If a trusted third party exists, such as a universally-trusted developer, the problem is easily solved: components could accept expected hash values that are signed by the developer. Note that because information may propagate through multiple components as in Figure 3.3, the party must be trusted by all components in the system. Such a highly trusted entity may not always exist.[3] Even so, were this entity to be compromised, it would result in catastrophic failure of the security of the system since the entity's credentials could arbitrarily change the components of the system. To ensure the end-to-end security of distributed and decentralized applications, components need more flexible and expressive mechanisms for authorization that do not require universally-trusted entities to enforce.

[3]Strictly speaking, all entities in a system secured by TEEs must trust the TEE manufacturer. In this chapter, we assume that entities trust the TEE and its manufacturer for authentication, but not authorization. That is, entities accept an attestation as proof of what component is running in a remote TEE, but do not rely on the manufacturer to determine which components are trustworthy. A malicious manufacturer (or catastrophic flaw in its implementation) could subvert the attestation process to authenticate unauthorized TEEs as authorized ones, but we consider such attacks outside the scope of this chapter.

Another challenge for decentralized TEE applications arises when components are replicated for scalability. For example, serverless applications such as those built on AWS Lambda [145], Google Cloud Functions [52], or Azure Functions [18], reduce resource costs by factoring their program logic into stateless components that interact simply with persistent data storage. If demand for the component suddenly increases, new replicas are launched to meet the demand. If demand drops off, replicas may be killed to reclaim their resources. Therefore, to take full advantage of serverless platforms, components need to have relatively low startup costs and be able to process requests from any client, even if a different replica previously processed requests from the same client.

RA composes poorly with serverless design in part because an attestation only authenticates a specific replica. Whenever a client is presented with a new replica, the attestation protocol must be repeated and authenticated. Repeated attestations can introduce significant latency. For example, the standard Intel SGX EPID-based RA protocol requires a client and server to exchange at least five messages, and additionally requires communication with the Intel Attestation Service (IAS) to verify the attestation report [120, 8].[4]

Distributed TEE applications frequently amortize this cost by agreeing on some cheaper, ephemeral means of authenticating future communication such as message authentication codes (MACs) based on a shared key. Unfortunately, since the cost of RA is so high, the cost is fully amortized only for long sessions with many requests. Since most serverless-style applications frequently spawn new replicas and have relatively short sessions, reducing the overhead of au-

[4]Intel also supports DCAP [141], which reduces interactions with IAS by deploying a custom report generating enclave. This enclave must be authenticated by IAS via a special certification enclave, but verifiers can use the IAS root certificate to verify attestation reports without contacting IAS. We discuss DCAP and relate it to our Decent Server in Section 3.4.

thenticating new replicas could significantly improve the performance of applications that use RA.

Finally, the software development lifecycle also presents challenges for decentralized TEE applications. RA allows developers (and indirectly, users) to specify how binaries may interact, but code changes frequently over their lifetime. Updating one component should rarely result in downtime for the entire system. Therefore, developers need a mechanism to securely authorize new components and revoke old components even after a system is deployed.

To address these challenges we have developed the Decent Application Platform, a framework for building secure decentralized applications using TEEs. The major contribution of this chapter is introducing a framework that enables mutual authentication of dynamic sets of authorized enclave components in a decentralized, distributed *application instance* without requiring additional trusted third parties (other than the hardware manufacturer). Instead of forcing developers to build ad-hoc authorization mechanisms on top of RA, Decent developers refer to the components their application depends on using a high-level service name. At deployment, Decent nodes specify an *authorization list*, or AuthList, that defines the components that are authorized to implement each service. At runtime, the Decent platform authenticates each remote component and ensures it is authorized to perform the desired service.

Decent ensures that malicious hosts cannot compromise the confidentiality or integrity of a Decent application by replacing components the application depends on. Since Decent Components execute within TEEs authenticated by RA, the confidentiality and integrity of the application does not depend on the trustworthiness of the host or its operator: any host may provide the service.

We have formalized the Decent protocol in ProVerif [29] and proven it protects the secrecy and authenticity of the data it processes. We have also implemented two Decent applications to evaluate the expressiveness and performance of our design. DecentRide, a decentralized ride-sharing application, and DecentHT, a distributed hash table.

We evaluated DecentHT's on the YCSB [53] benchmark and compared the overhead of Decent's authorization mechanisms to a traditional RA approach. Our results demonstrate that using Decent improves throughput for shorter sessions by as much as 7.5x and latency by as much as 3.67x. SGX attestation technology such as Intel's DCAP extensions [141], and others [104, 163, 63] that avoid interactions with IAS using mechanisms similar to self-attestation are likely to see similar tradeoffs depending on how often authentication certificates must be refreshed with IAS. The source code of Decent SDK, DecentRide, and DecentHT including the code for benchmark have been released on GitHub [182].

The rest of this chapter is organized as follows: Section 3.2 motivates the design of Decent using our decentralized ride-sharing example. Section 3.3 gives a high-level overview and the design of the Decent Application Platform. Section 3.4 and Section 3.5 discuss the details of the Decent Authentication and Authorization. Section 3.6 outlines our implementation of Decent SDK and Decent handshake protocol. Next, Section 3.7 presents the composition and result of the formal verification for Decent. Section 3.8 introduces the two sample applications we implemented with Decent framework. Moreover, Section 3.9 evaluates the expressiveness and performance of our example Decent applications. Furthermore, Section 3.10 provides discussions on existing works that are related to Decent. Section 3.11 discusses enhancing data sealing, the extension of AuthList for stateless open service support, and our future work of

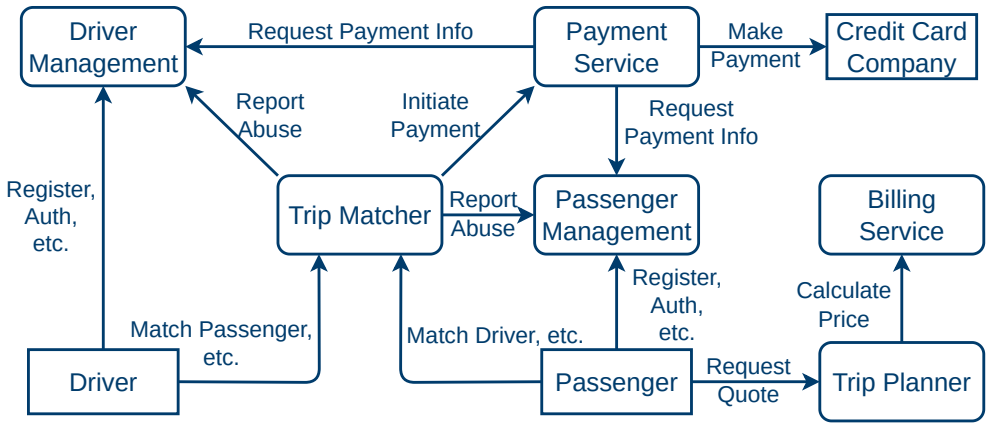


Figure 3.4: DecentRide, a decentralized ridesharing service.

automatic code verification. Finally, Section 3.12 concludes.

3.2 Motivation: DecentRide

To motivate the design of the Decent framework, we will use DecentRide, our decentralized ridesharing application, as a running example. Ridesharing services match riders to drivers who pick up one or more passengers and drive them to their desired locations.

Current ridesharing applications are highly centralized. All aspects of the system are controlled by the ridesharing company: setting prices, suggesting routes, matching riders and drivers, and processing payments. Moreover, all the data associated with these tasks is accessible to the ridesharing companies, raising concerns for passengers who may wish their travel patterns and other personal data to be kept confidential. The dominance of current ridesharing companies also gives drivers few alternatives when prices or policies are disadvantageous to the drivers' interests.

A decentralized ridesharing application could address some of these concerns by letting drivers, passengers, and service providers self-organize, but designing such an application has several security challenges. Figure 3.4 illustrates the interactions between components of DecentRide, a ridesharing service loosely based on Uber's microservice-based design [102].

In a decentralized application, these components may be hosted by multiple entities, some of which may be untrustworthy. For example, a malicious entity hosting the Trip Planner component could learn the locations and routes of drivers and passengers, and a malicious Billing Service component could manipulate prices. Trusted execution environments are a useful tool for implementing DecentRide since remote hosts can establish the authenticity of a remote component via RA, and communicate over authenticated, encrypted channels that are inaccessible to the component's host. Unfortunately, additional challenges remain.

First, since components may be hosted by untrustworthy entities, they must mutually authenticate each other, leading to the circular dependencies described in Section 3.1. Resolving these dependencies requires some authorization data, such as the hash of each component and the operations allowed for each component, to be provided by the host at load time. It is critical that this data cannot be used to subvert the security of data processed by the application. Furthermore, since some DecentRide components do not directly connect to each other, they cannot be directly authenticated and authorized by all components. Therefore, avoiding transitive trust attacks as illustrated in Figure 3.3 is also critical for security. For example, since the Billing Service only interacts with the Trip Planner, the host of the Trip Planner might attempt to introduce a malicious Billing Service to manipulate prices.

Second, distinguishing legitimate updates from malicious ones is challenging in a

decentralized application. Components that do not exist at compile time, the specifications are created, cannot be authorized based on the code they contain. Therefore, an additional mechanism must be used to authorize new components. The specific authorization process may be application specific, but any runtime process that authorizes new code (or revokes the authorization of existing code) should itself be authorized by the entities of the system whose security is at stake; otherwise, the process might be used to introduce malicious components.

Finally, because of the high cost of authenticating components using RA, most TEE applications establish an ephemeral session key during authentication, amortizing the cost over the lifetime of the session. Unfortunately, since each component in microservice-based designs like DecentRide's may be replicated in response to demand (and stopped when demand drops), the lifetime of each component may be relatively short. These shorter lifetimes make RA a significant performance bottleneck, especially for Intel SGX TEEs using EPID-based RA, where one must contact the Intel Attestation Service (IAS) [8] to determine whether an attestation is authentic.

3.3 Decent System Overview

Figure 3.5 provides an overview of Decent framework.

Building Blocks

The Decent Framework includes Decent Components and Decent Servers. Trusted code in each Decent Component and Decent Server runs in an SGX enclave. Each host runs a single Decent Server and one or more Decent Components. The Decent Server has only one

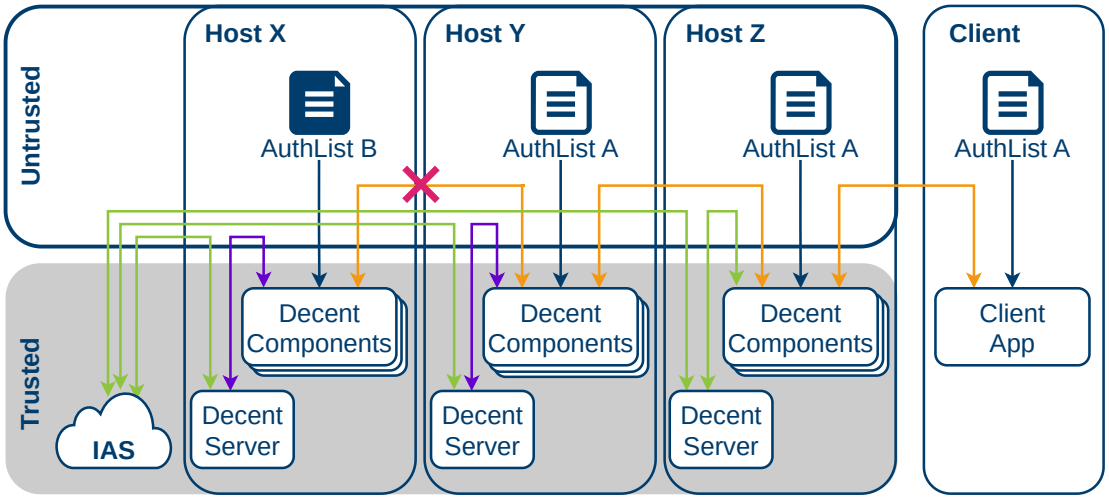


Figure 3.5: Overview of the Decent Framework; trust model shown is from the perspective of a client

purpose: to perform self-attestation and issue certificates to local Decent Components. Decent Components are enclaves that hold Decent certificates issued by the Decent Server and follow Decent protocols to verify peer certificates. There are three kinds of Decent Components: Decent Apps, Decent Verifiers, and Decent Revokers. Decent Apps contain all functionalities of the application. Decent Verifiers and Decent Revokers, discussed in Section 3.5.2 and Section 3.5.4, concern dynamic authorization and revocation.

Authentication

Decent Framework authenticates Decent Components by using the self-attestation certificate and Decent Component certificate.

The first time a Decent Server is executed, it creates a key pair and initiates the RA to bind its public key to the code running inside the Decent Server. We call this process "Self-

Attestation (SA)" because the component itself plays the role of the remote verifier in the protocol. The goal of SA is not to authenticate the server to itself but to create a certificate verifiable by a third party.

SA benefits applications like DecentRide by reducing the latency overhead of component authentication. Rather than performing RA directly during the authentication process, Decent Components authenticate each other using SA certificates created at load time. The traditional approach of establishing an ephemeral key during RA scales poorly for applications like DecentRide, where component-to-component sessions may be short-lived. Because SA certificates are reusable across sessions, even applications with short sessions scale well in Decent.

After Decent Server has created its SA certificate, the Decent Component sends its AuthList and its own public key to the server, using a secure channel established by the *Local Attestation (LA)*, which provides similar guarantees to RA but does not require verification by the IAS since the components and the server reside on the same CPU. The server then returns its SA certificate and a signed component certificate containing the component's public key, the digest of the component's code, and the component's AuthList.

Therefore, to authenticate a remote Decent Component, the verifier needs the Decent Component certificate, signed by a Decent Server's public key, as well as the Decent Server's SA certificate, so that the authenticity of the Decent Server's key can be verified. For brevity, we will use *SA certificate* or just *certificate* as a shorthand for these credentials necessary to verify a Decent Component.

Factoring out the SA code keeps the SA protocol easier to understand and audit, and

reduces the size of components. Furthermore, sharing Decent Server for components residing on the same CPU reduces the authentication overhead on hosts that run multiple components, since only one SA certificate is needed per host.

Since each component is running in a separate enclave, they cannot access the memory of the Decent Server or any other Decent Component, so even a malicious component cannot tamper with the authentication process of any other component.

Authorization

To authorize different components into the system, Decent requires all hosts and clients to provide an AuthList containing a list of Decent Components they trust. They compose the list freely on their own, but only the Decent Components and clients who hold exactly the same AuthList can talk to each other. That is because the AuthList held by one component may contain components that the other one does not trust, and vice-versa.

Decent authorization is decentralized in the sense that hosts may include arbitrary entities for the AuthList of Decent Components they host. However, the contents of the AuthList constrains which remote components will accept their connections. If a group of hosts colluded to include a malicious component, any client or legitimate component attempting to connect would see the AuthList is different and refuse the connection. As long as clients only connect to components having the same AuthList, any component outside of the client's AuthList cannot communicate with the system that the client is connecting to. For instance, as shown in Figure 3.5, the component running in Host Y received a different AuthList from the component in Host X, so the component in Host Y refuses the connection from the component in Host X.

Threat model

Figure 3.5 also shows the trust model of Decent framework from the perspective of a client. Unlike traditional applications, where everything in the figure will be trusted, Decent applications only trust the code running in the TEE environment and IAS, which is the manufacture of the TEE environment in general. Thus, the host may provide malicious inputs to Decent Components while observe outputs from them. We assume clients trust their machines, but clients that support enclaves could potentially execute the client software in enclaves to mitigate malware attacks. Decent does not prevent vulnerabilities in code but provides a mechanism for entities in a decentralized application to agree upon which components should be part of the TCB, and to verify that only authorized components receive access to protected data.

We assume that honest Decent Components follow the Decent API specifications and only communicate with peers that have been authenticated using the Decent protocol. Honest clients only communicate with components whose AuthList consists of Decent Components that the client considers trustworthy. We assume the security of the TEE mechanism: computation within the TEE is confidential, hosts can only interact with the TEE via the interfaces defined by the developer, and cannot alter the behavior of code within the TEE. We also assume that attackers cannot compromise cryptographic mechanisms such as public-key encryption or digital signatures with non-negligible probability.

Given these assumptions, the Decent system protects against adversaries that attempt to subvert security in several ways. Any number of hosts in a Decent application instance may be malicious. Malicious hosts may attempt to access and manipulate all inputs and outputs

to Decent Components and Decent Servers, including local memory and storage, messages between components, and configuration data such as AuthLists. Attackers may also develop and execute malicious Decent Components and Decent Servers that run inside or outside the TEE and partially or fully violate the Decent APIs and protocols.

We do not consider information an adversary learns by analyzing the timing or pattern of communications outside of the TEE. Complementary approaches exist for eliminating leaks from indirect or implicit flows (e.g., information flow control mechanisms [115, 150]), timing channels (e.g., predictive mitigation [14]), and access pattern analysis (e.g., oblivious computing [114, 178]).

The Decent platform currently only provides *confidentiality* and *integrity* guarantees. TEEs alone cannot provide *availability* guarantees since untrustworthy hosts can always suppress messages sent to or from the TEE, or simply shutdown the TEE altogether. In Section 3.5.4, we discuss our ongoing work to integrate enclave with blockchain to achieve confidentiality, integrity, and availability.

3.4 Authenticating with Self-Attestations

Figure 3.6 illustrates the process of creating certificates for the Decent Component Authentication, which is done in two major steps: 1) Decent Servers creating Self-Attestation certificates; 2) Local Attestation of Decent Components.

In SA process step SA1, the Decent Server first creates its key pair for authentication and sends the fingerprint of the public key to the CPU to produce a RA quote signed by the

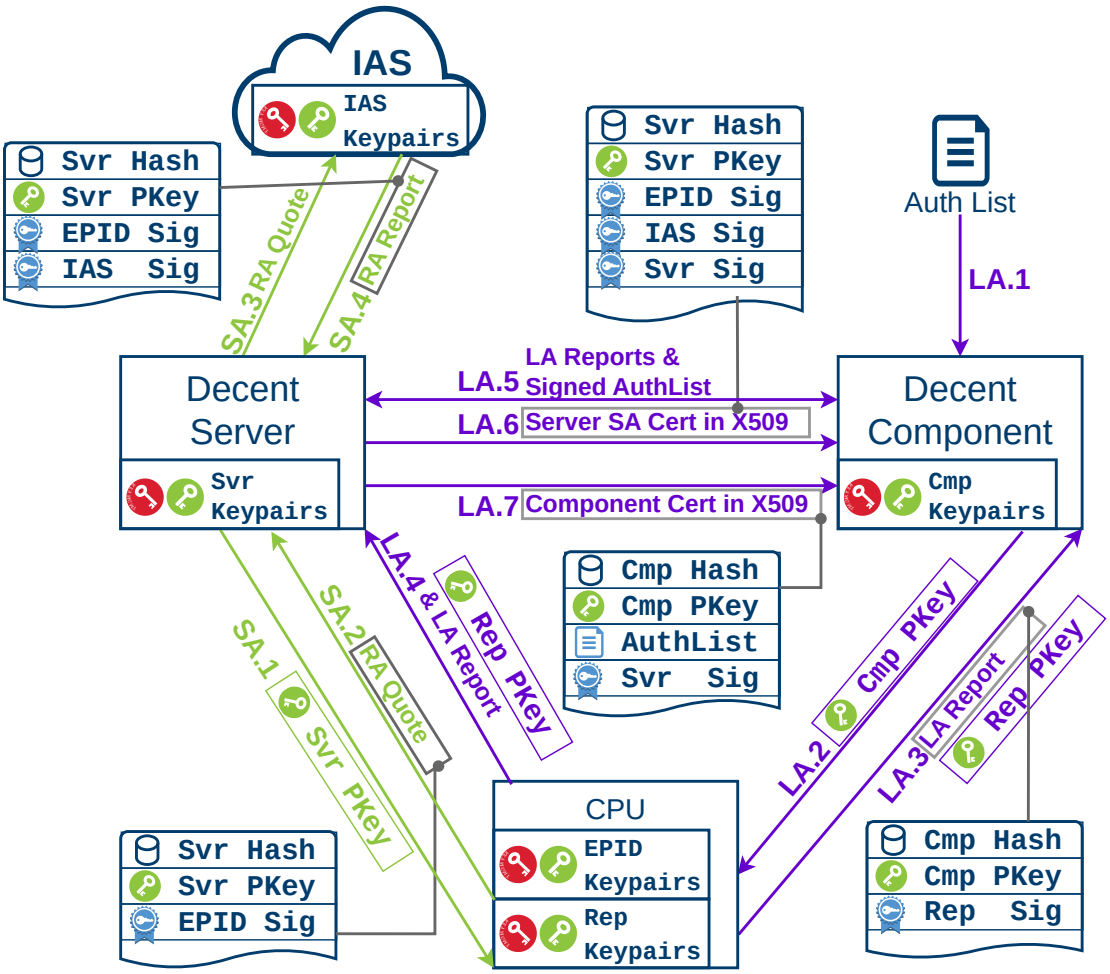


Figure 3.6: Process of creating certificates for the Decent Component authentication

EPID [33] group signing key provisioned by Intel to the CPU. In step SA3, The host forwards the signed quote, returned in step SA2, to the IAS for verification. If the signature is valid, in step SA4, the IAS returns a signed RA report, verifiable by a well-known public key. Upon receiving and verifying the IAS report, the server creates a signed X.509 certificate including the RA report and its public key.

In LA process step LA1, the Decent Component first loads an AuthList, which is immutable once loaded. It also creates its key pair for authentication and sends the public key fingerprint to the CPU for signing in step LA2. Next, in step LA3 and SA4, both the server and the component request the CPU to produce a LA report which is verifiable to any enclave running on the same enclave platform, including the server, with CPU's public report key. After verifying each other's LA report, a secure channel is established. Through this channel, the component sends its AuthList in step LA5. Then, in step LA6 and LA7, the server issues a Decent Component certificate containing component's hash and AuthList signed by the server's authentication key, along with the server's SA certificate.

After Decent Components have received certificates from the Decent Server, they can communicate over TLS connections that are authenticated with their certificates. The detailed procedures during the handshake process is given in Section 3.6.2.

By verifying the RA report using Intel's public key, third parties can confirm that the public key in the server's certificate was created by a specific Decent Server in an authentic SGX enclave. By verifying the signature on component's certificate, third parties can confirm the Decent Component resides in the same enclave platform as the server, and the authenticity of the AuthList.

SA certificates can also be used to verify outputs of a Decent application. For example, the DecentRide Payment service could provide digital receipts that verify a user's payment for a particular trip. By signing the receipt and attaching its SA certificate, any third party can verify the contents of the receipt was produced by a legitimate instance of the DecentRide app containing no unauthorized components, even if the instance who signs the receipt is no longer running.

It is also possible to verify attestations without contacting IAS using Intel's recently added DCAP [141] features (although it is still necessary to contact IAS for timely revocations). DCAP allows developers to use a local customized service to verify quotes, reducing the latency of obtaining quote verification reports from IAS. Extending Decent to support DCAP would be relatively simple: instead of using the IAS report as the root of the SA certificate, the DCAP report would be used instead. The primary requirement for Decent is that the authenticity of custom DCAP report is verifiable by a third party.

3.5 Authorization

3.5.1 Authorization List (AuthList)

Figure 3.7 illustrates an example AuthList for the DecentRide example. Each entry in the AuthList maps the hash of a Decent Component's code to the service name it is authorized to implement. A service may be provided by multiple enclave binaries to support multiple platforms and backwards compatibility.

Each Decent Component creates a unique key pair that is not only bound (by the

Code Digest	Service Name
dff1...8e41	BillingService
3fb5...cc46	PaymentService
6233...0f6d	TripMatcher
717e...5c1b	TripMatcher
...	

Figure 3.7: Example AuthList for DecentRide

SA certificate) to a specific, authentic instance of an Intel SGX enclave, but is also bound to a specific *immutable* AuthList which contains a list of service names and the components that are authorized to provide them. That means hosts cannot modify the AuthList without launching a new instance of the Decent Component. Malicious components may present a false AuthList, but they cannot forge the SA certificate of an authorized component. Since any authentic attestation report includes a hash of the component's code, malicious components cannot represent themselves as authorized ones.

The (untrusted) host establishes the initial network connection and forwards messages for Decent Components. The Decent Component establishes a secure communication channel using TLS on top of this connection. Components authenticate themselves by submitting a SA certificate, and a remote connection from a component is only authorized if the signatures of all certificates (including the IAS report) are valid, and the AuthLists match.

3.5.2 Dynamic Component Authorization

Requiring all AuthLists in an application instance to match prevents unauthorized Decent Components from connecting to the instance, but it also prevents new components from being authorized dynamically. Applications may wish to dynamically authorize components for a number of reasons, but a common reason is to update components to add features, improve performance, or fix bugs. Since Decent Components are authorized based on a digest of their code, these new components will not be allowed to connect to existing application instances. To authorize new components, all existing components must be restarted with a new AuthList.

Requiring full system restarts for component updates goes against the typical microservice-based design workflow where components are frequently and independently updated, and multiple versions of a component may co-exist at runtime. To avoid the downtime associated with such restarts, Decent distinguishes two special roles that enable dynamic authorization and revocation: *Decent Verifiers* and *Decent Revokers*.

A Decent Verifier is also an enclave program, which is permitted to authorize new Decent Components (including other verifiers) by an application instance. Comparing to trusted third parties, trusting a verifier is expressing trust only in the specific code running in the enclave, and the enclave platform ensures that its behavior cannot deviate from that code. In addition, when some verifier needs to be authorized by another verifier, the developers must specify the service names for each level explicitly, so there is a fixed depth for each valid chain.

Figure 3.8 provides an overview of Decent Verifier. Like any other Decent Component, in order to join an application instance, the Decent Verifier must be listed as a verifier on

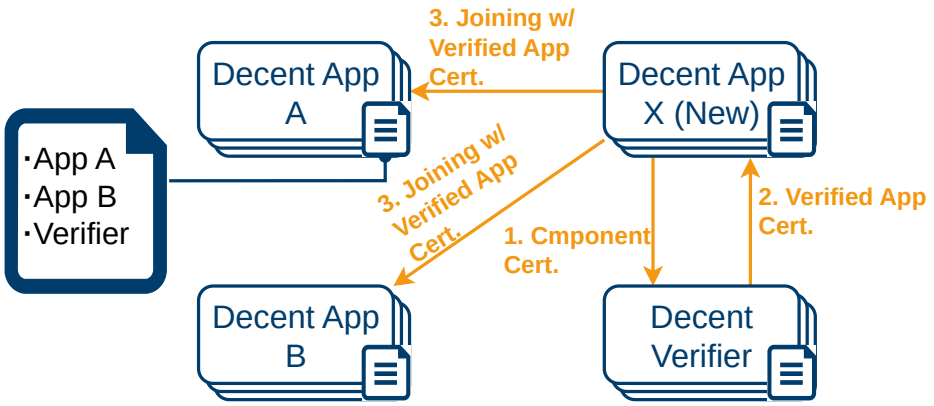


Figure 3.8: Procedures for the verifier

the AuthLists of the components in the instance. In other words, all Decent Components must agree on which verifiers are authorized to make dynamic authorization decisions. By defining multiple verifier service names, applications can designate which verifiers are authorized for which services. For instance, the DecentRide application could define a BillingServiceVerifier name for authorizing BillingService components, and a TripMatcherVerifier for authorizing TripMatcher components.

The new component first presents its SA certificate to verifier. Verifiers process new components' SA certificate through its verification mechanism defined by developers. If the certificate is successfully verified, verifiers authorize new components by signing their SA certificate. Finally, the new component can communicate with the existing components by presenting its SA certificate signed by the verifier. A component is authorized to connect to an application instance if (a) it appears in the AuthList for the expected service or (b) its SA certificate is signed by a verifier who appears in the AuthList for the expected service verifier. In both cases the AuthList of the new component must match the application instance's. In the

latter case, the verifier's SA certificate must also contain a matching AuthList.

Developers may plug in any desired verification mechanism into the Decent Verifier. One example approach is for the verifier to collect signed approvals from an application's "stakeholders," e.g., the entities whose data security may be affected by the authorization. When new Decent App is created, stakeholders that wish to authorize new app create and sign new app's code digest. New Decent App authenticates itself to the verifier using its SA certificate. If the verifier has received the necessary approvals, it responds by signing new app's certificate.

Other dynamic authorization approaches could avoid the need for external entities to explicitly authorize new components. For example, in Fabric [115] nodes download mobile code from untrusted hosts, formally verify their information flow properties, and link the compiled code into a distributed application at runtime. A similar approach could be adapted for Decent Verifier. Each new Decent Component's source code would be checked by the verifier against a formal specification associated with the service it claims to implement. More details about this approach will be discussed in Section 3.11.3.

3.5.3 Revocation

Both the Decent Component and the SGX platform can be compromised. During such a event, the private key of the Decent Component could be leaked, so that attacker can pretend to be a Decent Component and join the system, while the damage is specific to application. BFT (Byzantine Fault Tolerance) protocol could help application to tolerate compromised nodes, but it is beyond the scope of this chapter. Regardless, the ability to revoke vulnerable component is key to addressing such problem once it's detected.

Decent Components may have their SA certificates revoked in one of two ways. First, the IAS maintains a number of revocation lists it uses to determine whether an SGX platform (the chip, the credentials provisioned to it, or the platform software itself) have been revoked. Since RA reports are signed by a group signature to protect privacy, only Intel is able to distinguish revoked platforms within a group. Even with DCAP enabled, enclaves still need to acquire revocation lists from IAS A Decent Component running on a host whose platform has been revoked will be unable to refresh its SA certificate with the IAS server. Therefore it is prudent to set SA certificates to expire at reasonable intervals to force periodic refreshing.[5] Additionally, compared to the original RA protocol, saving the RA result to the certificate and refreshing it periodically does not weaken the security guarantee. According to the sample code in Intel SGX SDK, to reduce the overhead caused by RA, it is suggested that the RA state is also kept by secret provisioning for a period of time. Thus, the revocation of the a platform will also only be discovered after the RA state is refreshed.

Dynamic component revocation

If authorized Decent Components are discovered to have latent vulnerabilities, or are incompatible with new updates, these components should no longer be permitted to connect to a Decent application instance. However, dynamically revoking component authorizations presents many of the same challenges as dynamic authorization. Forcing full system restarts to modify AuthLists leads to increased downtime, and may delay when revocations could reason-

[5] While a malicious host may manipulate the operating system clock, the SGX platform provides a trusted interval timer that can be used as the basis of a mechanism to measure certificate lifetimes and force refreshes. We leave the design and implementation of such a mechanism to future work.

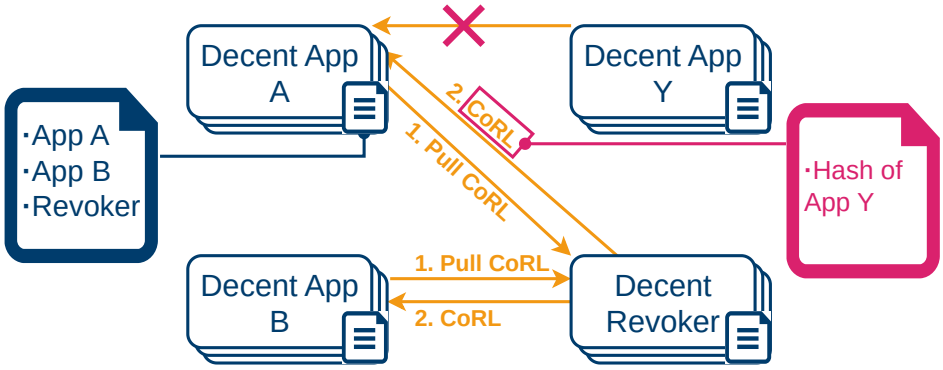


Figure 3.9: Procedures for the revoker

ably take effect. Furthermore, modifying AuthLists does not address revocation for dynamically authorized components.

Decent revokes authorized components using Component Revocation Lists (CoRLs). A CoRL is similar to a Certificate Revocation List [54], but instead of revoking a specific SA certificate, a CoRL entry is used to revoke *any* SA certificate generated for a specific Decent Component. In other words, that component may no longer connect to the instance, regardless of who is hosting it.

Decent Components called *Decent Revokers* maintain the CoRLs associated with an application instance. Like verifiers, revokers must appear on the AuthLists of all Decent Components in the application instance, or must themselves be authorized by a verifier.

Figure 3.9 illustrates a revocation workflow. The Decent Revoker must also be listed as a revoker on the AuthLists of the components in the instance. Components periodically pulls CoRL from designated revokers, to ensure they are having the latest CoRL locally. Components shuts themselves down if no responds is received from the revoker, to prevent subsequent dam-

age causing by malicious host suppressing revocation message. When a revoked component attempts to communicate with other components , it will be detected when other components consulting their local CoRL.

Similar to verifier, developers can also plug in any revocation mechanism to the revoker. For example, stakeholders submit revocation requests to one or more revoker containing the hash of the component whose authorization is to be revoked. Once a threshold of revocation requests are received, the component's hash is added to the CoRL.

In some scenarios, it may be possible to revoke components without the intervention of an external entity. If evidence that a components is compromised is mechanically verifiable, revokers can offer API that allows nodes to submit messages or private keys as evidence of compromise, so that revokers can automatically add entries to the CoRL. Our Decent prototype does not yet support revokers.

Revokers may revoke the authority of any Decent App or Decent Verifier. Revoking the authority of a Decent Server or Decent Revoker dynamically is problematic. The Decent Server is designed to be small and rarely updated. Many components in an application instance will likely share the same Decent Server, so revoking the server would invalidate the Decent certificates of many components. Moreover, Decent Server are not allowed to be dynamically authorized, so these components would be unable to rejoin unless there is a different version of authorized Decent Server that has not been revoked. Revoking a verifier can similarly cause many components' SA certificates to be rejected, but unlike the server scenario, these components may rejoin as long as they are able to locate some other authorized verifier that belongs to the instance.

Decent prohibits the revocation of Decent Revokers for two reasons. First, the desired effect of revocation is unclear. Entries on CoRL from a revoked revoker might not be replicated on the CoRLs of other revokers, so discarding those entries might allow compromised components to rejoin the application instance. Accepting entries from a revoked revoker is also inadequate, because some entries may be erroneous or malicious (hence the revocation), but we cannot identify which ones are. Second, if two different revokers place each other on their CoRL, which revocation should take precedence is unclear. Finally, revokers may be dynamically authorized by verifiers, but these verifiers must be treated specially. Since revoking the authority of a verifier of revokers would lead to the same issues that accompany revoking a revoker directly, the verifiers of revokers cannot have their authority revoked dynamically. To avoid these issues, no CoRL is consulted when authenticating revokers or their verifiers.

3.5.4 Distributed revokers

In the previous section, we introduced a simple revoker setup, where all Decent Components periodically poll for updates to the CoRL from a revoker, allowing them to automatically shutdown to prevent further damage if the attacker is trying to suppress messages from the revoker. A single revoker service creates a point-of-failure for ensuring the authority of vulnerable components can be effectively revoked. To reduce load on the revoker and tolerate revoker failures, system designers could replicate the revoker service and use a consensus protocol to ensure the CoRL is consistent across replicas. In this section we describe a design that distributes CoRLs using a Proof-of-Work (PoW) blockchain such as Ethereum.

Figure 3.10 shows the general chain structure of a blockchain. Each block contains

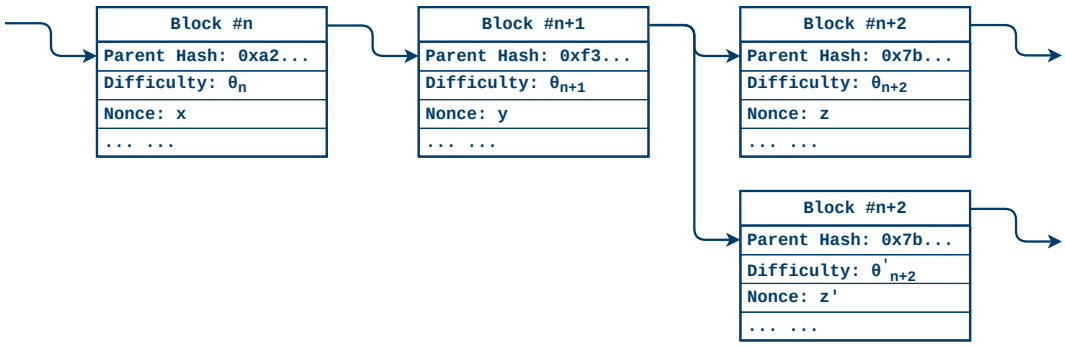


Figure 3.10: Blockchain chain structure

the hash of its parent block, a difficulty value of the puzzle that miners are going to solve, and the nonce which is the answer to the puzzle. When mining a new block, miners work concurrently to find the nonce that solves the puzzle whose difficulty is specified in the parent block, and multiple miners may find an answer before the next block is widely distributed. When miners mine different blocks pointing to the same parent, a fork (or forks) is formed. If there are multiple forks, miners choose which fork to extend, picking the longer chain if one exists. In order to ensure new blocks are generated at a relatively steady rate, the difficulty value is adjusted based on the time it took for a miner to mine and distributed the new block.

For applications with trusted stakeholders available, developers may choose to use permission-based ledger, where only authorized nodes (e.g., the stakeholders) may create new blocks. However, in case that stakeholders are unavailable, a public available blockchain network like Ethereum is needed. In Ethereum, any node can follow the protocol to mine new blocks, and there is and assumption that at least 50% of miners are honest. Otherwise, attackers could mine the longest chain and have the blocks to only include the transaction they prefer. An attacker may still want to maintain a private malicious fork and distribute it to other peers. This

is not an issue in a regular blockchain client, which can connect to as many peers as possible and hope that at least one of them is honest, so that it can always obtain the view of the honest chain. the honest chain is longer than the malicious chain because of the assumption that the majority of miners are honest. However, in enclave applications, all network connections are routed through the OS. Thus, it is possible that none of the blockchain message received by the enclave are coming from the honest node.

Zheng et al. [186] have proposed an approach to detect such an attacker by monitoring the difficulty value in the block. The attacker only has limited hash power, so they usually take longer time to mine the new block. As the block time increases, the difficulty value will drop according to DAA. Therefore, by monitoring the difficulty value, the enclave can determine if the blocks are from malicious fork or not.

By storing the CoRL on blockchain, multiple revoker instances can update/retrieve the same CoRL as well as serve multiple components. Meanwhile, by utilizing the smart contract feature of Ethereum blockchain, we can even automate the processing of adding new components into the CoRL. For instance, when the private key of a component is found outside the enclave, it can be submitted to the smart contract, which will verify the key and add the corresponding component into the CoRL. Additionally, if a compromised component is sending conflicting messages, we can submit the messages to smart contract to add it to the CoRL.

3.6 Implementation

We have implemented a prototype of our design using Intel SGX for Windows. Our prototype consists of about 20k lines of C++ (12k excluding header files) and uses Intel SGX SDK version 2.3.101.50222, and Mbed TLS version 2.16.0. While some of our design decisions are informed by the constraints of the SGX platform, our high-level design is applicable to any TEE platform that supports RA and secure memory.

3.6.1 Using the Decent SDK

The Decent SDK provides a high-level API for establishing secure channels between components that greatly simplifies authentication and authorization of remote application components while preserving fine-grained control over which components are authorized to perform specific services.

System calls such as those that handle network connections cannot be executed within an SGX enclave. The standard approach [104] for establishing a secure channel with an enclave is for untrusted code to first create (or accept) a TCP connection to (or from) the remote host, and then act as a proxy between the enclave and the network.

Figure 3.11 shows a fragment of code from the Payment Service component in DecentRide that handles incoming requests from the Trip Matcher component. The `void*` pointer `cnt_ptr` points to a TCP connection created by the untrusted code and passed into the enclave. The authorization data of this instance of component is retrieved on line 5, and a wrapper class for the TCP connection is instantiated on line 7. Lines 10-15 create an object that config-

```
1  void proc_trip_matcher_req(void* cnt_ptr)
2  {
3      //Get DECENT authorization data:
4      //key pair, certificate, auth list, etc.
5      States& state = GetStateSingleton();
6
7      EnclaveCntTranslator connection(cnt_ptr);
8
9      //Configure TLS session
10     std::shared_ptr<TlsConfigWithName> tlsCfg =
11         std::make_shared<TlsConfigWithName>(
12             state,
13             TlsConfig::Mode::ServerVerifyPeer,
14             "TripMatcher",
15             GetSessionTicketMgr());
16
17     //Create TLS channel
18     TlsCommLayer tls(
19         connection, tlsCfg, true, nullptr);
20     tls.Recv(/* ... */);
21     tls.Send(/* ... */);
22 }
```

Figure 3.11: Creating a TLS channel with AuthList

ures how the connection should be authenticated and authorized. The authorization data `state` is passed in to provide the TLS library with the component's key pair, certificate chain, and AuthList. The mode `ServerVerifyPeer` indicates that the Payment Service is expecting an incoming request (hence Server) and should request a certificate from the remote component (hence VerifyPeer). Line 14 specifies that the expected name of the remote component is "TripMatcher", thus the name "TripMatcher" must be listed under the component's hash entry in the AuthList. On line 15, "GetSessionTicketMgr()" retrieves a reference to a TLS session ticket manager that helps resume sessions to avoid re-negotiating the TLS handshake unnecessarily. Lines 18 and 19 establish the TLS channel using the connection wrapper and the configuration object, and lines 20 and 21 use the channel to receive and send data with the remote component.

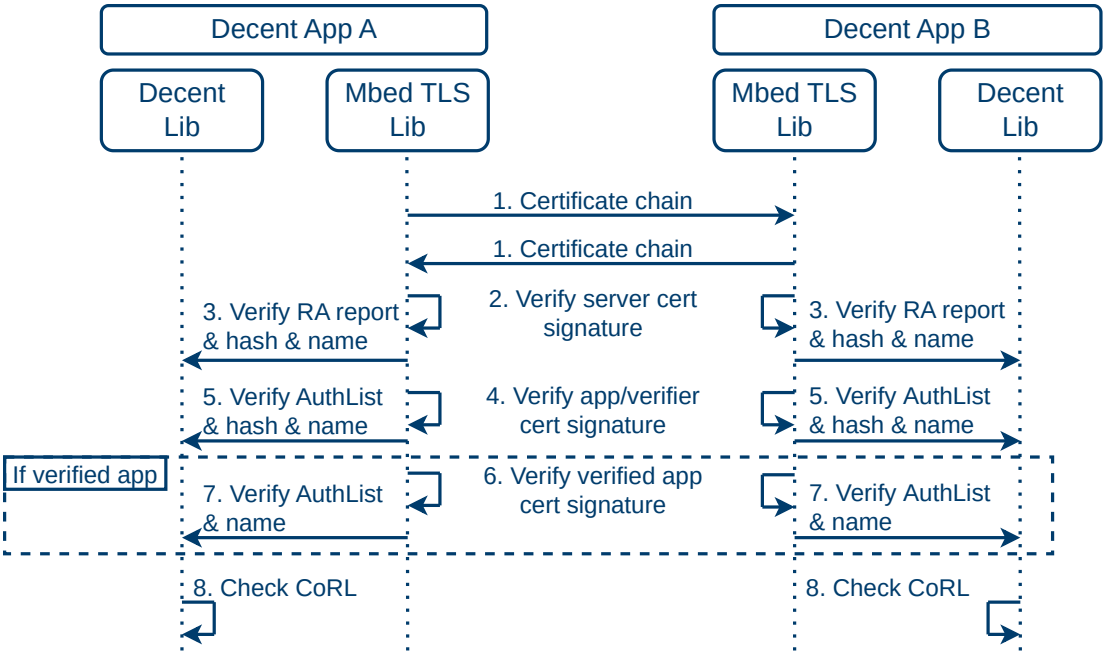


Figure 3.12: Decent handshake workflow

Permitting TripMatcher components to be authorized using a verifier only requires a few modifications to lines 10-15 of the Payment Service code above: instead of instantiating the ”TlsConfigWithName” object, we create a shared pointer to a ”TlsConfigWithVerifier” object, whose constructor accepts an additional component name, ”TripMatcherVerifier”, for the verifier’s name permitted to authorize ”TripMatcher” components dynamically. This configuration requires that any verifiers of a ”TripMatcher” component be listed under the name ”TripMatcherVerifier” in the AuthList.

3.6.2 Decent Handshake Protocol

The Decent handshake extends the usual TLS handshake where both sides authenticate with X.509 certificates. The `Mbed TLS` library code will handle most TLS handshake

process, including the certificate exchanging step. If signatures are successfully verified, a callback function implemented in Decent is executed by `Mbed TLS` to verify the customized contents in the X.509 certificate. Figure 3.12 illustrates the handshake procedure between two Decent Apps.

1. Decent App A and B exchange their X.509 certificate chains, which includes:
   - The X.509 certificate of Decent Server containing the RA report
   - The X.509 certificate of the Decent App or Decent Verifier signed by server
   - The X.509 certificate of the Decent App signed by the Decent Verifier, if the Decent App is verified by the verifier.
2. The Mbed TLS library code validates the signature on Decent Server's certificate
3. The RA report contained in Decent Server's certificate is verified with Intel's public Report Key, and the local AuthList is consulted to ensure the Decent Server's hash appears under the "`DecentServer`" service name
4. The Mbed TLS library code validates the signature on Decent App's (or verifier's) certificate
5. Our callback function will ensure the AuthList found in the certificate matches the one we loaded, and the hash of Decent App (or verifier) found in the LA report appears under the expected service name in the AuthList
6. If the remote peer is a verified app, there will also be a certificate issued to the verified

app by a verifier; The Mbed TLS library code validates the signature on Decent verified app's certificate

7. The callback function will ensures the AuthList found in the certificate extension section matches the one we loaded, and its service name matches the one defined in the code

8. The local revocation list is consulted to ensure all hashes—apps, and verifiers—are still valid and have not been revoked

If all checks are successful, the connection is accepted, otherwise it is rejected.

3.7 Formal Verification

We have formalized and verified the secrecy and authentication properties of the Decent protocol design using ProVerif [29]. ProVerif is an automated formal verification tool for cryptographic protocols, which we use to formalize our protocol's behavior and describe the scenario we want to test. The tool allows us to ask whether the secret could be revealed to the attacker or if the attacker can manipulate a message without being detected. The protocol is public to attackers, and attackers may intercept and manipulate any (raw) message transmitted within message channels.

Our ProVerif formalization consists of 1095 lines of code and comments for the implementation and 1437 lines for the verification scenarios, and it follows the threat model defined in Section 3.3. We use the ProVerif standard library for cryptographic definitions used in our implementation, which assumes that encryption and digital signatures are uncompromisable if appropriately used.

As discussed in Section 3.3, we defined five process types representing Decent's architecture building blocks: Decent Server, Decent App, Decent Revoker, Decent Verifier, and Verified Decent App (a Decent App authorized by a Decent Verifier). We also defined additional processes for modeling malicious Decent Components and Decent Servers. For simplicity, we only consider verification of components from the perspective of (honest) Decent Apps; the verification process is identical for clients of Decent services.

Each Decent App process loads an AuthList at the beginning of the process. One honest app contains secret data to protect, or will receive a computation result whose authenticity must be verified. We designate the AuthList of this app to be the "reall" AuthList, meaning it contains only trustworthy components. The AuthLists of all other processes are given by an attacker representing the untrusted host. The authorized Decent Components and Decent Servers are represented by honest processes, since a host cannot alter their behavior.

Attacker-controlled Decent Components and Decent Servers are expressed by issuing RA and LA reports in an honest process, but leaking the component's private key to the attacker. Hence, attackers will be able to authenticate as the component without being bound by the original process's behavior. Note that the IAS key, CPU EPID key, and CPU report keys discussed in Figure 3.6 remain secret. We assume revocation lists and hashes of approved Decent Apps are provided to Decent Revokers and Decent Verifiers via out of band process.

We verified the secrecy and authenticity of the data transmitted between Decent Components using two verification scenarios. In one scenario, there are two Decent Apps: one sending the secret data, the other receiving the data. The other scenario is identical, but between Verified Decent Apps. Any of these processes may be replicated an arbitrary number of times.

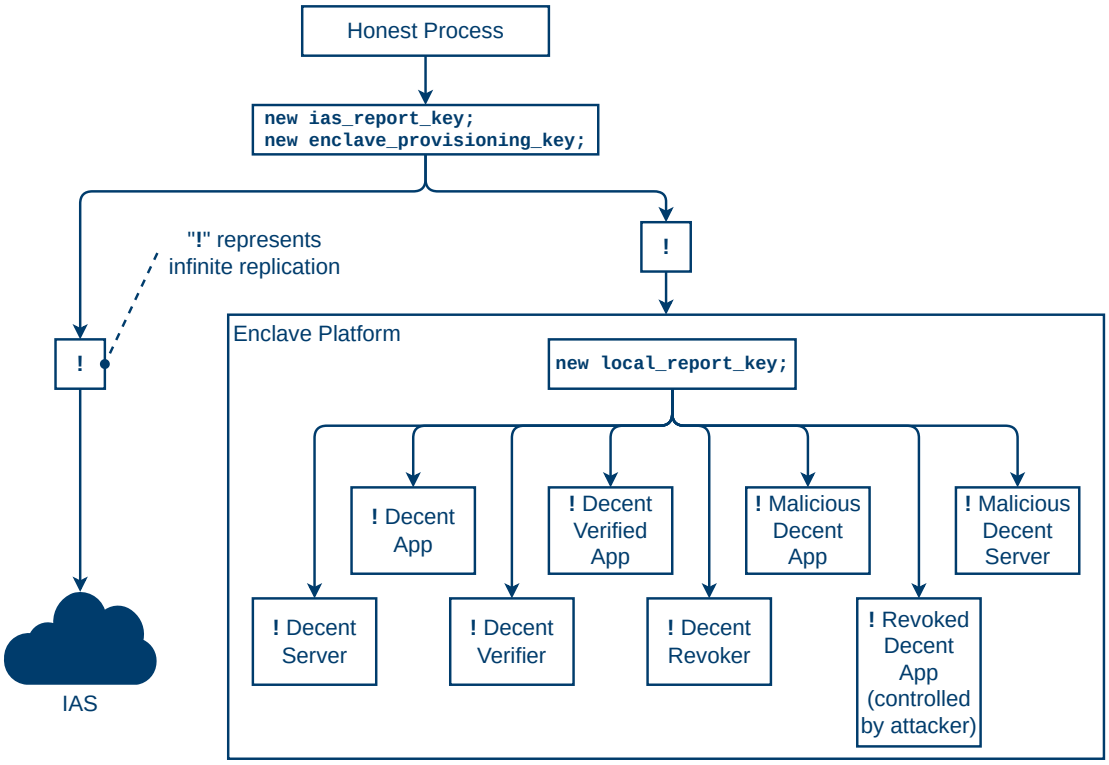


Figure 3.13: Verification Processes Overview

	Decent App	Verified Decent App
Secrecy	2 minutes	8 hours
Authenticity	4 hours	61 hours

Figure 3.14: Verification times

Figure 3.13 illustrates the process diagram for our formalization.

Table 3.14 lists the verification times for our ProVerif formalization. Verifying data secrecy for Decent Apps completed relatively quickly, which only took 2 minutes, while verifying secrecy for Verified Decent Apps required 8 hours. We also verified the authenticity of Decent Apps, which took 4 hours to complete. Verifying authenticity for Verified Apps required decomposing the verification problem into three simpler tasks. First, we prove that the AuthList stored in the certificate issued by Decent Verifiers is identical to the AuthList loaded by the Verified Decent App. Next, the verifier only issue certificates to Verified Decent Apps with the same AuthList loaded. Lastly, we prove that Verified Decent Apps only accept peers with identical AuthLists in their certificates. The entire process took 61 hours to complete. Our complete verification code and reports are available on GitHub [182].

3.8 Example Decent Applications

3.8.1 DecentRide

We used our prototype to implement two Decent applications. The first application is an implementation of DecentRide, discussed in Section 3.2. DecentRide components are hosted

by a distributed network of mutually-distrustful nodes. Because these components run inside SGX enclaves, confidential information like the location and routes of drivers and passengers is kept private, and critical computations like route and pricing calculation cannot be manipulated by the hosts.

Each DecentRide component (circles in Figure 3.4) implements a microservice—a small, targeted service that is loosely coupled with the other components by a simple application protocol. With the exception of the TripMatcher, DecentRide components maintain no local state. This design makes it easy to launch (or halt) component replicas on demand and balance request load among hosts. DecentRide hosts are incentivized to host components by compensation they receive (calculated by the Billing Service) from payments for trips (processed by the Payment Service). Passengers and drivers register and authenticate to the Passenger Management and Driver Manager services, and receive credentials that are presented to (and verified by) the DecentRide components they interact with. All user data provided to the management services is authenticated encrypted with keys derived from a component's seal key and the AuthList as described in Section 3.5.1. Thus, hosts have no access to the user data they store, even if they launch malicious DecentRide instances.

When a passenger wishes to find a ride on DecentRide, they contact a Trip Planner service which finds a path between their current location and the desired destination. The Trip Planner submits the proposed route to the Billing Service to get a total price for the trip, including fees for the hosts operating DecentRide components. This price quote is signed by the Billing Service and returned to the passenger (via the Trip Planner) along with the Billing Service's certificate chain, which the passenger and Trip Matcher uses to verify the authenticity of

the price quote.

Drivers advertise their availability by sending their location data to a Trip Matcher, which replies with a list of nearby passengers and their desired destinations. Once a driver selects a passenger, their contact information will be exchanged, and the passenger will receive the current location of the driver. At the destination, both driver and passenger must confirm to the Trip Matcher that the trip completed successfully, and only then does the Trip Matcher initiate payment for the trip with the Payment Service. The Payment Service obtains payment information from the management services and submits the transaction to the relevant financial entities.

To prevent abuse such as fake trip requests designed to probe for driver locations, or fake driver locations to probe for passenger locations and destinations, driver and passenger interactions with the Trip Matcher are logged to their respective management services. The Trip Matcher will not proceed with a driver or passenger request until it receives confirmation from the service that the logged action was successfully received, ensuring that malicious users cannot bypass the abuse detection by suppressing these logging messages.

3.8.2 DecentHT

To complement to DecentRide, we implemented a simple Distributed Hash Table (DHT) in Decent based on the Chord [153] protocol. Running each node within an enclave lets us store confidential information at untrusted hosts and control access to that information using Decent AuthLists. Sealed data stored in the DecentHT can be accessed based on a consistent hash function applied to the desired key, just as in the Chord system. A non-enclave alternative

to DecentHT could store encrypted *values* that were inaccessible to their hosts, but controlling access to the decryption keys introduces extra complexity in such a decentralized system.

DecentHT nodes encrypt their data using their own sealing keys and only provide access to authorized entities. It requires no additional key management beyond the Decent authentication mechanisms. Executing the Chord protocol logic within the enclave rules out attacks based on manipulating protocol messages or routing attacks since even malicious hosts process messages with trusted code. The host may still, of course, suppress incoming or outgoing messages, and may still learn some information from analyzing communication patterns between nodes, but at a much slower rate when there is a high ratio of keys to nodes. Moreover, an additional data replication scheme is needed since neither enclave nor Decent guarantees availability, and some versions of DecentHT nodes could be revoked at any time.

DecentHT derives each node's identifier, which determines the items it is responsible for, using the Decent seal key. This approach prevents many Sybil attacks [60] since every DecentHT component launched with the same AuthList on the same CPU will receive the same identifier. Components launched with different AuthLists will be unable to connect to other DecentHT instances that have agreed on the same AuthList.

Figure 3.15 illustrates a DecentHT network used by a DecentRide instance to store passenger and driver data used by the management services. Drivers and passengers interact with the management services, which may cause them to create, fetch, or modify data stored in DecentHT. As in Chord, requests will be routed to the appropriate node. When the component responsible for the key is reached, the component loads the associated (sealed) data into the enclave and sends it to the requesting application.

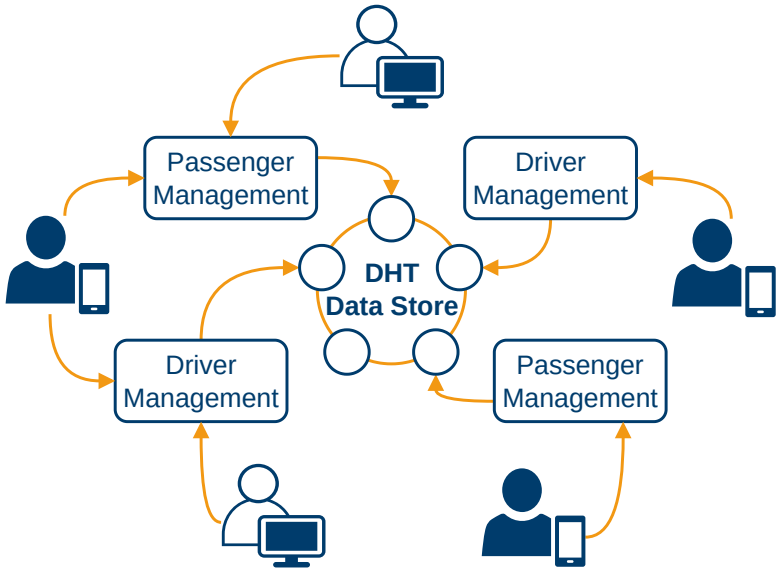


Figure 3.15: DecentHT: a Decent implementation of Chord [153].

3.9 Performance Evaluation

3.9.1 Experiment setup

We evaluate the overhead of Decent authentication and authorization by comparing the performance of DecentHT on the YCSB benchmark [53] to an implementation that uses a RA approach as suggested by the Intel SGX SDK documentation. Since the SGX RA Only approach does not support mutual authentication of the DecentHT components, we omit code for enclave identity verification. In our results, we refer to the Decent implementation as Decent RA and the SGX SDK implementation as SGX RA Only.

We also evaluated the performance of two non-enclave implementations to examine the baseline performance of the system without the overhead of SGX operations. In TLS Only group, the DecentHT code executes outside of the enclave and communicates over TLS chan-

nels without verifying certificates. In TLS+Sealing group, we additionally encrypt the stored records with a symmetric key, similar to how the enclave versions seal the records.

Our experimental setup is similar to management services accessing data stored in DecentHT, as discussed in Section 3.8.2; but instead of DecentRide components, we created a small Decent App that exposes Java bindings for the YCSB benchmark to invoke. We used Workload B containing 95% reads and 5% writes, with uniform request distributions for all our experiments below. Each read/write operation consists of one lookup request to determine the node responsible for storing the desired record, followed by a request to read/write the record. Each DecentHT node loads approximately 3,000 records. To ensure a relatively even distribution of records across nodes, we disabled DecentHT's Sybil resistance and explicitly specified node ID's that were evenly spaced. Each test measures performance for 60 seconds after a 60-second warm up phase. We repeat each test *three times* and report the median of measurements as points on the graph, with minimum and maximum values as error bars, which are *not distinguishable* at the scale of the evaluation graphs.

DecentHT nodes execute on a single 3.6 GHz Intel i7-7700 with 4 cores (8 logical) running Windows 10. The server has 16 GB of RAM, with 128MB (the maximum permitted) dedicated to SGX. Records stored at each node are sealed and stored in non-enclave memory. Each node is assigned its own logical core, with two cores reserved for the network stack and the Intel AESM service which is responsible for managing interactions between the operating system and SGX enclaves.

Clients execute on a single 3.4 GHz Intel i3-7100T with 2 cores (4 logical) running Windows 10. The client machine has 4GB of RAM and 128 MB is dedicated to SGX. The

client and server machines are connected via 1-Gigabit Ethernet.

SGX requires that all thread-local memory be pre-allocated, thus the maximum number of threads used by an SGX application is fixed at runtime and is limited by the memory available to SGX. For the DecentHT nodes, we specified 18 threads for handling incoming requests from either clients or peers, 6 threads for forwarding the finger table lookup requests to other peers, and 2 threads for replying requests received from other peers. In the SGX RA Only experiments, 14 additional threads are used for requesting quotes for the enclave itself. These threads are unnecessary for the Decent experiments, but we reserve the same amount of memory in both cases. Using the additional memory to allocate more threads for the Decent experiment would further improve Decent's performance over the SGX RA Only case.

On the client side, we limited YCSB to a maximum of 50 threads due to the enclave memory required by each thread. This limitation did not affect the throughput of the SGX-based implementations, which were fully loaded at around 40 client threads.

To test the performance of DecentHT in different session lengths, we run the experiment with various numbers of requests per session; the more requests in each session, the lengthier the session is. The DecentHT nodes perform a full TLS handshake (or RA in SGX RA only group) in each session's first request, and the following requests resume the session with TLS session tickets [140]. The RA process results in a shared secret between the enclave and the host verifying the enclave's attestation report. Thus, In SGX RA only group, we used a similar approach to the sample code provided by the Intel SGX SDK to establish a secure channel using AES-GCM encryption between the DHT and the application nodes. In addition, we give nonces and randomized Initial Vectors (IVs) to the encryption process for all messages, and

implemented a mechanism similar to TLS session tickets [140]. The purpose of this protocol is simply to approximate the security guarantees of Decent's authenticated channels using secrets shared during the attestation process and avoid the additional overhead of a TLS handshake, for a fair comparison.

Since our experiments generate many IAS requests in the SGX RA Only case, we use a simple IAS simulator to avoid violating the terms of use for our Intel Developer account. Instead of sending requests to the IAS, all requests are sent to our simulator which replays a single hardcoded response from the official IAS. Requestors follow the same protocol as they would for a real IAS request, but they ignore the nonce in our simulated response to allow us to replay the same response multiple times. The response times of our simulated IAS are *gamma-distributed* with parameters estimated from *30 IAS API response time samples* collected from the IAS portal. For retrieving the current EPID signature revocation list, we measured a mean response time of 39 ms with standard deviation 24 ms. For retrieving an attestation report we measured a mean response time of 255 ms with standard deviation 70 ms.

3.9.2 Results

Both the Decent RA implementation and the SGX Only implementation amortize the cost of authentication over the length of a session. Therefore the negative impact of authentication on system throughput will decrease as the average session length increases. To analyze the tradeoff between authentication overhead and session length, we ran each implementation with six server nodes on the YCSB benchmark while varying the number of requests each client made before establishing a new session. For the Decent RA implementation, each new session

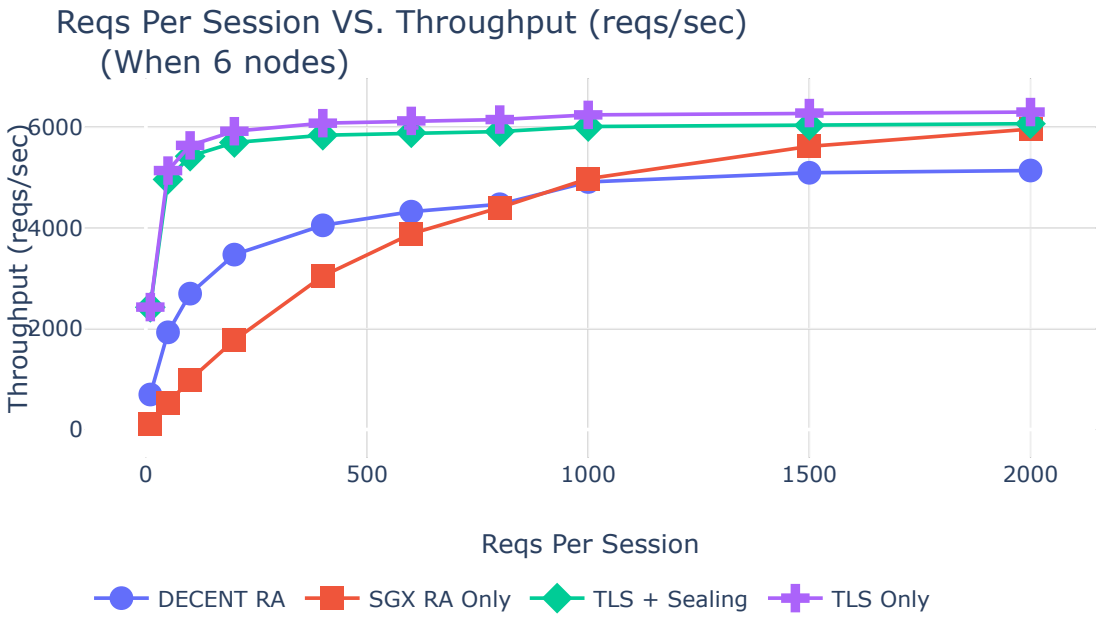
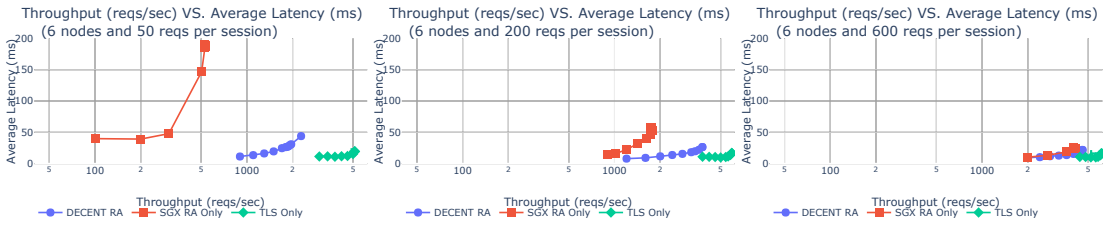


Figure 3.16: Requests per session versus throughput

involved a TLS handshake and the exchange and verification of SA certificate chains. For the SGX RA Only implementation, each new session required a new RA. The non-enclave versions required only a TLS handshake with no certificate verification.

Figure 3.16 presents the results of these experiments. The non-enclave performance improvement plateaus at about 200 requests per session, with the TLS+Sealing implementation achieving a lower throughput due to the extra overhead of encrypting and decrypting the stored records. Between 10 and 400 requests per session, Decent RA significantly outperforms the SGX RA Only implementation. For long sessions of 800 requests, Decent RA and SGX get roughly the same throughput. Beyond 800 requests, the SGX RA Only performance approaches the TLS Only implementations, which we attribute to the additional messages required to resume TLS sessions compared to our custom SGX RA session ticket scheme.



(a) Each session includes 50 requests

(b) Each session includes 200 requests

(c) Each session includes 600 requests

Figure 3.17: Average latency versus throughput

Figure 3.17 plots the tradeoff between latency and throughput for sessions of length 50, 200, and 600. We gradually increased the target throughput and measured the average response time for requests. At 50 requests per session, the difference between Decent RA and SGX RA is most pronounced, with latency rapidly increasing for SGX RA as throughput exceeds 150 requests per second. At 200 request per session, the behavior begins to converge, but Decent RA still significantly outperforms SGX RA. At 600 requests per session, however, their performance is roughly equivalent. Note that the performance of the TLS Only implementation is mostly unaffected for these session lengths.

We also measured the latency of requests as the number of nodes increases from three to six to sanity-check whether Decent RA affects scalability. In Figure 3.18, the results show that, as expected, latency in both implementations is largely unaffected by the small increase in nodes. However, the average latency for Decent RA is almost four times lower than SGX RA.

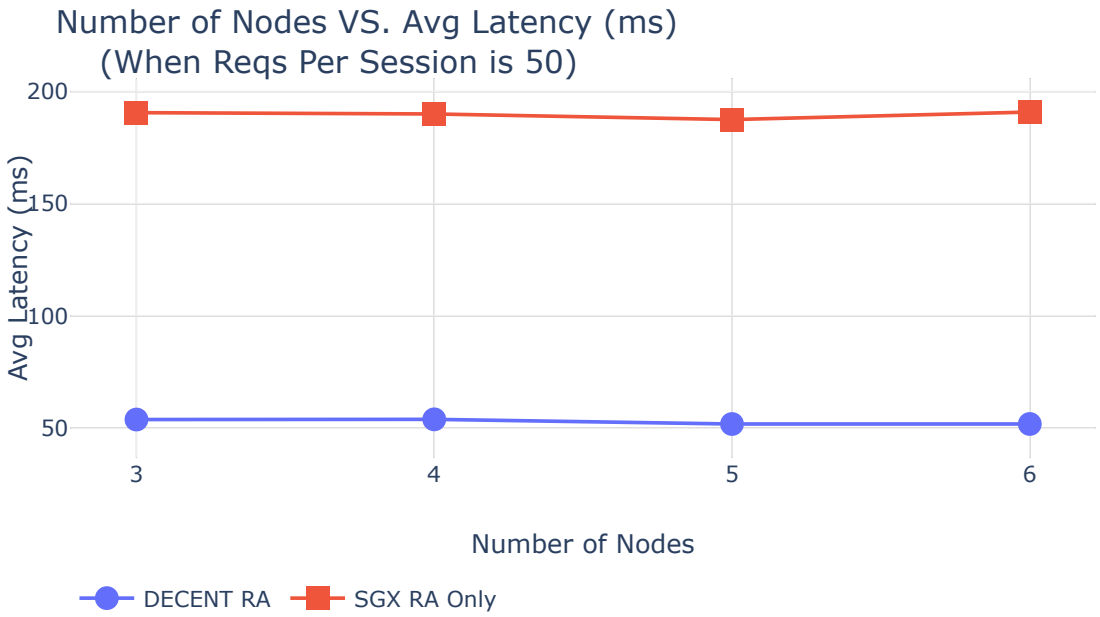


Figure 3.18: Average Latency vs. Number of Nodes

3.10 Related Work

Several recent projects use enclaves and RA to provide confidentiality and/or integrity for distributed applications, but almost none address the problem of mutual authentication. Beekman et al. [22] suggest a work-around for an application with two components by building both of them into one enclave binary and using a runtime switch to select the desired component. This approach is impractical in a distributed application with large number of components, since it results in (potentially very) large enclave binaries and is difficult to maintain. This approach also requires rebuilding the entire binary when any binary requires updating. VC3 [143] allows a user to launch MapReduce workloads using cloud-hosted SGX enclaves, ensuring the confidentiality of the processed data and the integrity of the results. VC3 jobs distribute a single

enclave binary to each host, avoiding mutual authentication issues.

Other systems deal with mutual authentication using a external party. Ryoan [90] authenticates the components using RA, but modules are identified by a public key that signs the module rather than the code itself, while the corresponding private key could be a key that is stored outside of an enclave.[6] MesaTEE [123] solves the problem of mutual authentication by relying on third-party *auditors* [124], which sign binaries that pass an audit process. Decent does not require trust in external entities (which could be compromised) to authorize enclaves, but requires hosts in an application instance to agree upon which components are authorized to implement which services, and enables clients to verify which components are included in an instance before using its services. Panoply [147] partitions applications into multiple small enclave components. It relies on a shim library to assign names to each enclave and maintain the mapping from name to enclaves' hash. However, since mutual authentication is not its main focus, details regarding how the shim library obtains the hash and enforces the mapping make it difficult to assess how or whether it supports mutual authentication similar to Decent's.

CCF [137] uses a distributed ledger to manage which TEE components are enabled. This fills a similar role to the AuthList in Decent. CCF nodes are more heavyweight than Decent nodes because they must participate in a consensus protocol to process requests. Since CCF relies on consensus protocols to protect the integrity of the list of authorized components, the security of CCF against attacks by malicious components relies on the security assumptions of these protocols (e.g., $> 2/3$ of hosts are honest, for BFT protocols). Decent offers stronger integrity: regardless of how many hosts are honest, no malicious components may be introduced

[6]The authors claim that Ryoan could also support identities based on code hashes, but it is unclear how they would address mutual authentication.

into an application instance. Decent does not offer availability guarantees, but coupling Decent with a BFT protocol could provide availability guarantees similar to CCF without sacrificing integrity.

MAGE [47], a work published a year after our Decent framework [183], also proposes a mutual authentication mechanism for enclaves that does not rely on a trusted third party. By intervening in the enclave image building process, MAGE separates the enclave image into two sections: the program logic section and a reserved section. The hash of the program logic section is then calculated and added to a list, which is stored in the reserved section of all enclave images in the network. During the remote attestation process, MAGE uses the hash of the peer's logic section to initialize a specialized hash calculator. The hash calculator is then updated with the content of the reserved section, allowing MAGE to recover the expected complete hash of the peer. However, MAGE lacks dynamic enclave authorization and revocation mechanisms, suffering from similar limitations as the workaround proposed by Beekman et al. [22]. Specifically, any update to a component in the system requires rebuilding all enclave images and redeploying them across all hosts in the network, making it impractical for large-scale distributed applications. Additionally, MAGE's reliance on intervening in the enclave image building process makes it highly dependent on specific enclave image formats and building tools, which may restrict its compatibility with different enclave platforms.

Decent's SA process is similar to a now-common approach to authenticating TLS connections with enclaves. Knauth *et al.* [104] describe the process of using RA to bind a public key to an enclave that generated it and include the report in a self-signed certificate. This certificate is used to establish the TLS connection allowing the remote host authenticate the key.

Rust SGX [163] and Open Enclave [63] offer similar support. Similarly, OPERA [48] proposes an RA service that is separated from the SGX protocol, but still requires a report generated by IAS during its preparation phase.

All the above TLS approaches support, in principle, the option of SA where the host of the enclave participates on both sides of the attestation process to aid in the creation of the certificate. None address the issue of mutual authentication of enclaves and therefore cannot support applications like DecentRide or DecentHT. Furthermore, our results in Section 3.9 quantify the performance gains of SA over on-demand attestation for short sessions.

Other work has focused on supporting more general systems programming within SGX enclaves. SCONE [12] and Graphene-SGX [46] support standard library functions not natively available to enclaves, such as filesystem and network I/O. In general, these projects focus more on the local secure systems programming aspects of using SGX enclaves, and do not directly support RA.

Intel's DCAP support permits customized SGX RA protocols without contacting the IAS (except during setup). Alternative TEE designs such as Sanctum [55] and Keystone [111] allow direct verification of a public key certificate chain signed by the manufacturer. Keystone also supports additional roots of trust beside the manufacturer. DCAP, Sanctum, and Keystone's approaches to RA could reduce the overhead of RA similar to using SA certificates. They do not however, address mutual attestation between enclaves.

Key Separation and Sharing (KSS) is a recent feature added to the SGX SDK that help differentiate multiple instances of the same enclave. For example, Decent could use KSS to derive different seal keys for each application instance by placing a hash of the AuthList in the

configuration parameters instead of using our HKDF approach (as discussed in Section 3.11.1). Furthermore, since we can specify an AuthList in terms of each component's `MRENCLAVE` identity, and bind it to the configuration id used for attestation, KSS could provide an alternate implementation path for Decent's authorization mechanism. Nevertheless, although KSS provides additional tools for mutual authentication, it doesn't provide a solution. Any KSS-based approach would need to address the same challenges as Decent, but like DCAP, KSS could provide an alternate implementation path for some of Decent's features.

3.11 Extensions and Future Research Directions

3.11.1 Binding Data Sealing Keys to AuthLists

Enclaves use cryptographic algorithm to protect code and data stored in memory. However, the space of memory is limited comparing to permanent drives, and data stored in memory is volatile. Thus, in many cases, the enclave application also needs to write data to permanent drives by using the process called data sealing: the enclave derives the data seal key from its root key and encrypts the data with authenticated encryption algorithm using the derived seal key. Intel SGX provides two key-derivation schemes for deriving *seal keys*. In one scheme, `MRSIGNER`, the enclave platform uses the *signer* (typically the author) of the enclave to derive seal keys, meaning that any enclave binary signed by the same key may decrypt sealed data. In the other scheme, `MRENCLAVE`, the enclave platform uses the enclave's code hash to derive keys, thus, when multiple enclave applications writing sealed data to the drive, they can only decrypt the data that has been sealed with their own key. `MRENCLAVE` scheme prevents one

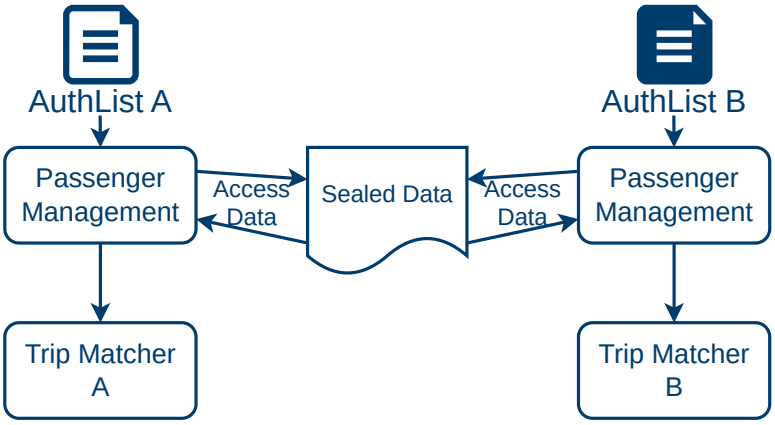


Figure 3.19: Data sealing with `MRENCLAVE` scheme only

enclave application from reading the data that has been sealed by another one. The `MRENCLAVE` scheme is useful for Decent applications since we do not want components exchanging data outside of mutually-attested channels or malicious components reading data sealed by an honest component. However, since this approach derives identical keys for each instance of an enclave, a host could use a malicious AuthList to attempt to unseal data encrypted by an application with a secure AuthList.

Figure 3.19 shows an example of the attack utilizing the traditional data sealing scheme. Two instances of passenger management service are generating sealed data and writing to the permanent drive. Since their enclave hashes are the same, they can un-seal each other's sealed data. This is generally fine for traditional single component enclave application, because both instances should have the same behavior - if the instance on the left does not violate confidentiality of user's data, the one on the right will not violate either. However, this is not true in distributed applications with multiple enclave components. For example, in Figure 3.19, the instance on the left loads an AuthList containing *TripMatcher A*, while the instance on the right

loads AuthList containing malicious *TripMatcher B*. Thus, data sealed by the instance on the left could be un-sealed and processed by the instance on the right and leaked to the malicious TripMatcher. Therefore, `MRENCLAVE` scheme is insecure for distributed enclave applications, because the hash of one enclave does not represent the behavior of the entire application as a whole.

The Decent SDK uses a HMAC-based Key-Derivation Function [105] (HKDF) to derive a key from the `MRENCLAVE`-derived key to also bind the key to *a specific AuthList*. Decent's HKDF scheme prevents malicious hosts from using legitimate components to leak sealed data to malicious components, and malicious components from directly deriving another component's seal key.

However, binding the AuthList to seal keys also implies that sealed data must be explicitly migrated from one application instance to another if the AuthList changes or a component is upgraded. One approach to data migration is to implement a migration API by which a new component can request sealed data from an existing component before it is replaced. For example, if a new component is authorized by the verifier, it can contact a specified component from which to migrate data and seal the retrieved data under its own key. In some scenarios, however, it may be unreasonable to use the existing component to migrate sealed data. For example, if the AuthList is changed, or a component is about to be revoked because of a vulnerability, delaying revocation to migrate data to a replacement component could subject the application instance to exploitation. Guarding against sudden revocation of a component with a large store of sealed data requires sealing the data under a key that can be provisioned to "recovery components" that can access and migrate data securely in the event the component's

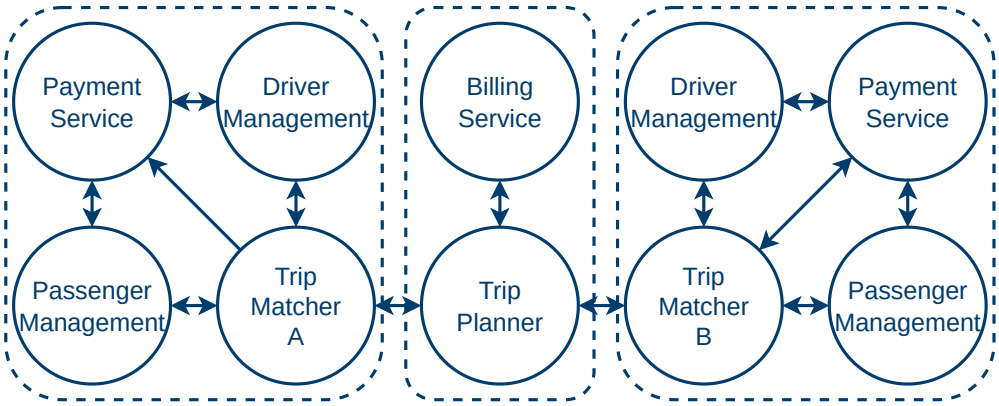


Figure 3.20: AuthList with stateless open services

authority is revoked.

3.11.2 Stateless Open Services

One disadvantage of our approach to component authorization with AuthLists presented in Section 3.5.1 is that component instances cannot be shared by multiple applications unless their nominal AuthLists are combined. For example, suppose we wanted to share the BillingService and TripPlanner components from Figure 3.4 with DecentRide instances that each use a different TripMatcher component, as illustrated in Figure 3.20. TripMatcherA does not appear on the AuthList of components in the TripMatcherB instance, and vice versa, but both application instances authorize the same BillingService and TripPlanner components.

Each instance is primarily concerned with protecting the confidentiality of the user's GPS location and the integrity of the BillingService's price quote. An authorized TripPlanner component with a matching AuthList ensures the GPS location will not be revealed to unauthorized entities and that the BillingService could only be influenced by authorized components.

However, if the BillingService and TripPlanner are *stateless* with respect to each incoming request, the TripMatcher only needs to worry about communication occurring during the processing of its request. That is, if no information from TripMatcherA's request is retained after it has been processed, then none of TripMatcherA's confidential information can be leaked to TripMatcherB, and TripMatcherB's price quote cannot be influenced by TripMatcherA's request. From the perspective of TripPlanner, it only processes the data from the requester (i.e., TripMatcher) without revealing any additional secret. Thus, even a malicious trip matcher will not be able to obtain any secret of TripMatcherA from TripPlanner, or affect the price calculation result returning to TripMatcherA. Therefore, TripPlanner does not need to be care about the identity of the requester.

We propose a simple extension to our AuthLists mechanism that permits secure sharing of stateless services like the TripPlanner and BillingService between multiple applications with otherwise incompatible AuthLists.

Figure 3.21 demonstrates the AuthLists of RideShare application (given in Figure 3.21a) and planner & billing sub-application (given in Figure 3.21b). The enclave's hash is not only mapped to the service name, but also a definition of that service's components. The TripPlanner entry given in Figure 3.21a has a nested AuthList that defines the authorized components of the TripPlanner sub-application. These are the components that each TripMatcher application expects to be in the TripPlanner's AuthList, which will be confirmed when the TripMatcher and TripPlanner components establish a secure connection.

Code Digest	Service Definitions
`3fb5...cc46`	PaymentService
`6233...0f6d`	TripMatcher
`23ed...e470`	TripPlanner: {
	`dff1...8e41: BillingService,`
	`23ed...e470: TripPlanner` }
...	...

(a) Example AuthList with sub-application

Code Digest	Service Definitions
`dff1...8e41`	BillingService
`23ed...e470`	TripPlanner

(b) Example AuthList for planner & billing application

Figure 3.21: Example nested AuthList

3.11.3 Dynamic Verification

Currently, dynamic authorization of Decent components requires stakeholders to separately audit and approve components. This introduces some degree of centralization and potential for exploitation if stakeholders' credentials are compromised or the stakeholders themselves are malicious. An alternative we are currently exploring seeks to shift trust from external auditors onto automatic verification tools running within Decent. For example, a Decent Verifier could formally verify that a new component satisfies a formal specification of the desired service. The verification could be performed on the binary directly or performed on the source and then compiled (and signed) by the verifier. While this approach adds the verification tool chain to the trusted code base, it removes the ability of external entities to subvert the application with malicious authorizations, while providing a flexible dynamic authorization mechanism.

3.12 Conclusion

In this chapter, we present the Decent Application Platform, a framework for building secure decentralized applications. Decent Components supports mutual authentication without requiring a universally-trusted entity to authorize components. AuthList ensures only authorized components can interact with the system. verifiers and revokers allow new components to be authorized or revoked dynamically. We formalized Decent in ProVerif and verified that it protects the secrecy and authenticity of application data.

We implemented DecentRide to demonstrate the expressiveness of Decent framework, while the evaluation based on DecentHT shows that, for short sessions, Decent provides

7.5x higher throughput and 3.67x lower latency comparing to the non-Decent implementation.

Chapter 4

Introducing Blockchain into TEEs

4.1 Introduction

One of the benefits of permissionless blockchains offer is *censorship resistance*: adversaries cannot suppress transactions from clients to the blockchain, nor can they suppress the publication of new transactions. In most Proof-of-Work (PoW) blockchains, this guarantee holds as long as adversaries do not control a majority of the computational power in the network. An additional, but often overlooked, requirement is that clients must be able to create reliable network connections with at least some honest blockchain nodes.

Heilman et al. [89] discuss how an adversary can monopolize these connections to perform *eclipse attacks* that suppress communication between targeted clients and blockchain nodes. Most defenses against eclipse attacks involve diversifying connections to peers in the hopes that at least one connection is not controlled by the attacker.

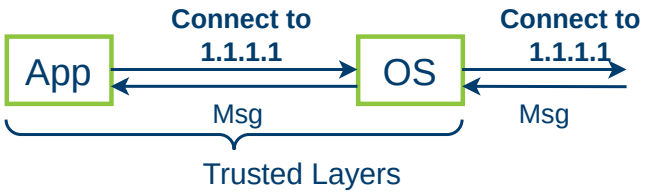
Enclave applications run in a special restricted mode where even privileged code (such

as the OS) cannot inspect or modify the enclave's code or memory, making it possible to run confidential applications on untrusted hosts that produce high-integrity outputs. The reliance on the OS for inputs and outputs also means that enclave mechanisms alone cannot provide any availability guarantees. Furthermore, since the enclave depends exclusively on the (potentially malicious) host for all network communication, mitigating eclipse attacks by creating more connections is not possible.

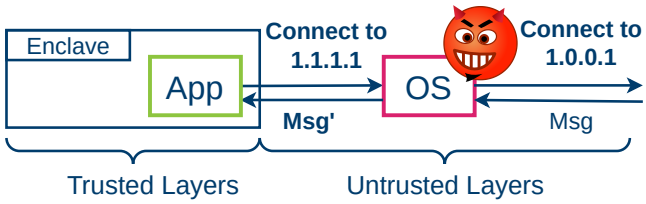
Even though an attacker can only directly access coins it possesses the keys for, eclipse attacks can have serious consequences. For example, an attacker may try to suppress incoming blocks containing time-sensitive transactions (e.g., revocation notifications, payment channel closures, etc.) published to the blockchain. Alternatively, the attacker may try to lower the difficulty to a level where it can generate malicious blocks within the expected block time. Using double spend attacks, the attacker can try to extract value from the enclave without spending coins on the real chain. In both cases, shutting down the enclave prevents it from operating on incomplete or malicious information.

Figure 4.1 compares a threat model of a traditional application with an enclave application, with both applications wanting to communicate with the remote node on IP address `1.1.1.1`. The OS layer establishes the connection and forwards the message; however, an untrustworthy OS could intercept the message and manipulate or suppress it in the enclave application threat model. Cryptographic protocols such as TLS can protect the message content, but cannot guarantee delivery.

The main contribution of this chapter is a novel algorithm that can reliably detect eclipse attacks both inside and outside of the enclave environment. Our algorithm continuously



(a) Traditional Application



(b) Enclave Application

Figure 4.1: Threat Model Comparison

monitors changes in block difficulty to detect an eclipse attack in the enclave. We evaluate this algorithm by running it against all historical blocks in Ethereum.

4.2 Related Works

4.2.1 Eclipse Attacks in General

Castro et al. [42] introduced eclipse attacks on structured peer-to-peer overlay networks. Heilman et al. [89] then realized this style of attack in Bitcoin, wherein an attacker can seize control of a sufficient number of IP addresses of peers of the victim node. Upon a successful eclipse attack, several other attacks can then be performed, such as selfish mining and double-spending transactions. The authors suggest diversifying the peer connection to mitigate this attack, but this relies on at least some of those connections to be outside the control of the

attacker. In an enclave application, all connections may be intercepted by the host.

4.2.2 Eclipse Attack with Enclaves

Ekiden [50] addresses eclipse attacks in blockchain applications built with enclaves using a Proof-of-Publication (PoP) protocol. This approach requires the host to publish an enclave-generated nonce to the blockchain within a limited time. A successful PoP shows that the block containing the nonce is fresh and valid. However, relying solely on PoP for ensuring timely block delivery is too expensive since each PoP requires a blockchain transaction fee.

Tesseract [23] defines a time threshold by which the enclave must receive a new block from the Bitcoin network; otherwise, it waits for n more confirmation blocks to ensure the block was not mined by the attacker during an eclipse attack. Tesseract does not monitor the block difficulty, so an attacker could take advantage of drops in the difficulty to continue mining blocks in a malicious chain during the confirmation period. Additionally, blockchains with faster block arrival rates (such as Ethereum) typically have higher variance in arrival times, so choosing a time threshold that minimizes both false positives and negatives may be more difficult.

4.3 System Overview

4.3.1 Threat Model

We assume the attacker *physically* and *effectively* controls no more than some percentage q of the total network hash power, Q. The biggest Ethereum mining pool at the time of writing has around $26\%Q$ [131]. To make our exposition more concrete, we choose $q = 30\%$

as an upper bound, but this percentage is configurable. A lower percentage may be sufficient for many applications; a higher percentage may be required for smaller Ethereum-based blockchains (e.g., Ethereum Classic). To simplify our model, we assume q is the attacker's effective hash rate during the eclipse attack. Withholding their hash power could potentially lower the difficulty of the network overall (rather than just the enclave), making it easier for the attacker to mine blocks on a malicious fork, but new miners joining to take advantage of the lower difficulty could undermine the effectiveness of this technique.

The host executing the enclave may be malicious, and controls all other software layers including the network stack. Thus, a malicious OS could hijack the network connection or suppress messages.

4.3.2 Difficulty Monitoring

Our primary mechanism for detecting eclipse attacks is based on monitoring the current difficulty level, θ, set by the PoW protocol. During normal operation, θ is typically around $13 \cdot Q$, which targets an expected arrival time of 13 seconds.

One subtle point is that this arrival time accounts both for the hash power spent on mining the next primary block as well as *uncle blocks*, which are valid blocks that arrived later than the previous primary block. Based on Ethereum's Difficulty Adjustment Algorithm (DAA) formula [35, 76], primary blocks that have uncle blocks may arrive within 18 seconds, instead of 9 seconds, to avoid lowering the difficulty value. However, splitting the hash power in half between uncle blocks and primary blocks to get twice the time does not result in strategic advantage. Therefore, we assume the attacker spends its hash power generating blocks on a

single malicious fork and produces no uncle blocks.

During an eclipse attack, the attacker must generate blocks every 9 seconds to avoid a difficulty drop.[1] Initially, this is difficult for the attacker, but each time the attacker fails, Ethereum's DAA lowers the difficulty by a maximum of 5%. For an attacker with $30\% \cdot Q$ hash power, θ will eventually approach $2.7 \cdot Q$. At this point, the attacker can expect to mine blocks fast enough to maintain the difficulty level indefinitely. We define the difficulty at which the attacker can maintain a block arrival rate of 9 seconds as $\theta_{min} = 9 \cdot (q \cdot Q)$.

While θ_{min} is the boundary at which the attacker has complete control over the chain, applications may wish to set a lower difficulty drop as a cutoff point. The lower the θ_{min}, the more confirmations are necessary to have high assurance that the most recent blocks were not mined by the attacker. Setting a more conservative bound on the drop will require fewer confirmations, at the price of potential false positives: blocks that trigger our algorithm while the system is not under an eclipse attack. We evaluate this tradeoff in detail in Section 4.7. Once the difficulty nears θ_{min}, however, no number of confirmations can eliminate the possibility of an attack.

Ethereum's DAA adjusts the difficulty value from block to block,[2] so we must monitor difficulty drops over time to prevent attackers from incrementally dropping the difficulty value. The difficulty level adjusts naturally as miners leave or join the network, so we need to establish a stable difficulty level for a given time span, and adjust this level over time.

Beginning at the genesis block, the chain is split into fixed checkpoint ranges of N_c

[1] Given Ethereum's DAA [76], if the current block arrived within 9 seconds of its parent, then the next block's difficulty will not decrease

[2] In contrast, Bitcoin adjusts the difficulty every 2016 blocks.

blocks. We set the current difficulty level as the median of the blocks in the previous (complete) checkpoint range. N_c should be large enough to make it impossible for an attacker to maintain a θ higher than θ_{min} after N_c blocks but small enough to avoid false positives.

Current enclave platforms do not provide a trusted world clock time, but some (e.g., Intel SGX [119, 8]) provide trusted elapsed time, which is sufficient for our purposes. We only require that the block time, calculated based on the block timestamps, is reasonably close to the elapsed time. Even if the timestamp is malicious, Ethereum bounds the drift of timestamps, so the influence on difficulty values is small.

4.3.3 Accepting transactions in the enclave

When clients and enclaves communicate, they compare the hash of the most recent checkpoint to ensure both parties are following the same fork of the blockchain. Applications may also use N_c as an upper bound on the number of confirmations needed to consider a block finalized. This upper bound is conservative since our monitor prevents *sustainable* forks of the blockchain during an eclipse attack.

Within N_c blocks, attackers may have a non-negligible chance of mining a sequence of malicious blocks, but the chance of sustaining that sequence for N_c blocks without detection is negligible. It is possible that a smaller number of confirmations would be sufficient to guard against temporary forks, but we leave this question for future work.

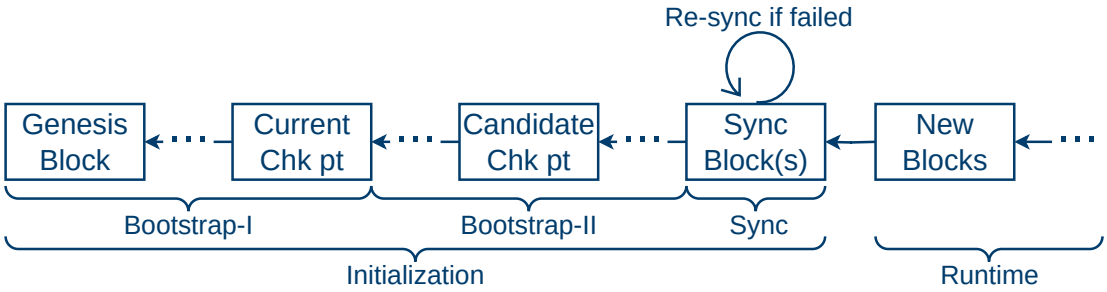


Figure 4.2: Difficulty Monitor initialization

4.4 Difficulty Monitor

4.4.1 Initialization

Our difficulty monitor has three initialization phases, shown in Figure 4.2, which ensure the legitimacy of the existing blocks.

Bootstrap-I Phase The monitor loads and processes all blocks starting from the *genesis block* to the latest *checkpoint block*. Blocks associated with well-known false positives (such as hard forks), are classified as *confirmed benign blocks* and included when determining the difficulty of the checkpoint range they appear in.

Bootstrap-II Phase Next, the monitor loads all blocks published after the checkpoint block. Difficulty changes will be checked with the same process used at runtime, but without restrictions on the elapsed time.

Sync Phase Finally, the monitor synchronizes with the blockchain by publishing a sync message, similar to the PoP used by Ekiden. Unlike Ekiden, we only require a sync message at

initialization. This prevents attackers from feeding a pre-mined malicious fork to the monitor. Thus, it ensures the freshness of both the *Sync block* and following blocks. Sync messages contain a nonce generated by the monitor, and a block containing the sync message with the matching nonce must be received by the enclave within the expected block time, which is around 13 seconds in Ethereum. If the sync message fails to be published within the restricted time limit, the host will need to perform a re-sync by publishing a new nonce until a sync block is accepted.

4.4.2 Runtime Difficulty Monitoring

New blocks received by the monitor are first validated as normal. Then the monitor determines the nominal block time based on the block timestamps. If the difference between the trusted elapsed time and the nominal block time is greater than 15 seconds, the enclave rejects the block to prevent the attacker from either lowering the difficulty in primary chain by manipulating timestamps, or maintaining the difficulty in malicious chain with pre-dated timestamps. Temporary forks in the chain require each branch to be monitored simultaneously. Our documentation discusses further details [184].

Once the new block has passed the initial validation, the monitor compares θ_{new} presented in the new block with the θ_{min} calculated based on the formulas shown below.

$$\theta_{\text{ref},i} = \text{Median}([\theta_{(i-1)\times N_c}, \cdots \theta_{i\times N_c}]) \tag{4.1}$$

$$\theta_{\text{min},i} = \theta_{\text{ref},i} \times (1-p) \tag{4.2}$$

The monitor first gets the median of difficulty values among the blocks in the previous checkpoint window, which have been checked and are finalized.[3] By using the median as a reference

[3]Finalized blocks are blocks having large number of successors and negligible probability of being reorganized.

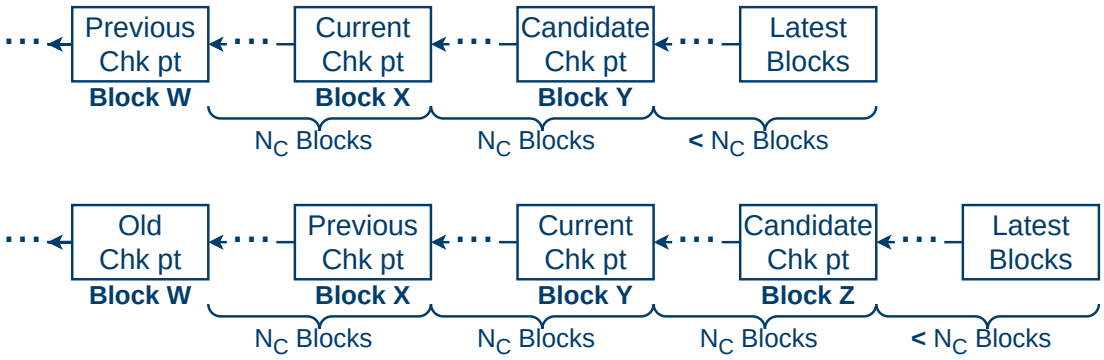


Figure 4.3: Checkpoint Update

value, we minimize false positives and false negatives caused by outliers. Next, the monitor uses θ_{ref} to calculate the θ_{min}, where p is the percentage drop allowed (set by the application). The new block is accepted if θ_{new} is higher than θ_{min}.

The attacker may choose to provide no blocks at all, preventing the monitor from determining how much the difficulty value has dropped. To address this issue, the monitor will periodically estimate the current difficulty value, θ_{est}, by using the trusted elapsed time, and compares θ_{est} with θ_{min}. Additionally, the maximum allowable block time in Ethereum's DAA (around 900 seconds) prevents the difficulty value from dropping more than 5% in each block. This value also serves as the maximum wait time ($T_{\Delta Max}$), and blocks arriving later than this time will trigger a shutdown.

4.4.3 Checkpoint Update

The monitor will automatically update the checkpoint every N_c blocks. As shown in Figure 4.3, the candidate checkpoint is the block that has less than N_c blocks ahead and is too new to be considered finalized. Once there are N_c blocks found ahead of the candidate

checkpoint, the candidate will become the new checkpoint, and the N_cth block ahead of the new checkpoint will become the next candidate checkpoint.

4.5 Forked Chains

A blockchain may fork if miners find multiple blocks having the same predecessor. Most blockchain protocols address this by requesting honest miners to mine on the longest fork [28, 35]. Because our approach relies on recording arrival times, if a chain becomes (even temporarily) forked, we must either monitor each fork simultaneously to record block arrival times, or repeat the Bootstrap-II and Sync phases if a chain is discovered that is longer than the chain (or chains) we are currently monitoring.

Our monitor also supports monitoring multiple forks simultaneously. When the chain is forked, the monitor also forks into identical sub-monitors, each monitoring their own fork. Any sub-monitor that has detected an eclipse attack will terminate, and when there are no active sub-monitors, meaning the entire chain is under attack, the enclave will shutdown to prevent further attacks.

In case a fork is found that is longer than all monitored forks, it is necessary to add this fork to the monitor to start monitoring. However, we cannot directly add this fork because its difficulty value has not been monitored; we can only consider it if it satisfies two conditions. First, the predecessor of the new fork must not be any block before the checkpoint, since all blocks before the checkpoint should be finalized and any modification made to blocks before the checkpoint is uncommon and suspicious. Second, the new fork must be longer than all

monitored forks, because adding a shorter fork is meaningless since honest miners mine on the longest fork. Once these conditions are satisfied, the new fork can be added by using the same steps as those in Bootstrap-II phase and Sync phase.

4.6 Equations for Possibility of Mounting a Successful Attack within a Checkpoint

The possibility of a miner mining a block within a given time follows the exponential distribution [23], where the parameter λ is the inverse of the expected block time.

$$T_{\text{a\_exp}} = \frac{\theta}{Q_a}$$
$$\lambda_{\text{a}} = \frac{1}{T_{\text{a\_exp}}} = \frac{Q_a}{\theta}$$

$$Pr[T_{a\Delta} = T_{\text{block}}] = \text{CDF}(T_{a\Delta}, \lambda_{\text{a}}) - \text{CDF}(T_{a\Delta} - 1, \lambda_{\text{a}})$$
$$= e^{\lambda_{\text{a}} - \lambda_{\text{a}} T_{a\Delta}} - e^{-\lambda_{\text{a}} T_{a\Delta}}$$

Thus, as shown in the formula above, to find the exponential distribution for an attacker mining a block, we need to calculate the expected block time ($T_{\text{a\_exp}}$) in terms of the attacker. $T_{\text{a\_exp}}$ can be obtained by dividing the current difficulty value (θ) with the hash power/hash rate (Q_a) controlled by the attacker. Next, we inverse the expected block time to get the λ_a of the exponential distribution. Nonetheless, the timestamps recorded in the block header is an integer value in seconds. Therefore, we can get the possibility of the attacker mining a

new block with a given block time ($Pr[T_{a\Delta} = T_{\text{block}}]$) by subtracting the result of two Cumulative Distribution Functions (CDFs), given two consecutive block times ($T_{a\Delta}$ and $T_{a\Delta} - 1$).

In Ethereum, the difficulty value changes from block to block. Hence, based on the formula we have, we can see that the distribution will also change from block to block. Thus, we need the transition function for difficulty value (θ) that can be plugged into the formula above.

$$\begin{aligned}\theta_{\Delta\%}(T_\Delta) &= \frac{\text{MAX}(1 - \frac{T_\Delta}{9}, -99)}{2048}, \text{where no uncle block} \\ &= \frac{1 - \frac{T_\Delta}{9}}{2048}, \text{where } 0 < T_\Delta \leq 900 \\ &= \frac{9 - T_\Delta}{18432}\end{aligned}$$

$$\theta_{N+1} = \theta_N \times (1 + \theta_{\Delta\%}(T_{\Delta N}))$$

As shown above, we first get the difficulty transition function ($\theta_{\Delta\%}(T_\Delta)$) from Ethereum's DAA [35, 76]. This function accepts the block time (T_Δ), and calculates the change of difficulty in percentage ($\theta_{\Delta\%}$). We can then obtain the next block's difficulty value (θ_{N+1}) given the current difficulty value (θ_N) by using this function.

In addition, we assume there is no uncle block on the malicious fork; otherwise, attackers have to spend part of their hash power to mine for uncle blocks.

$$Pr[\theta_N > \theta_{\min}] = \begin{cases} 0, & \text{if } \theta_N < \theta_{\min} \\ 1, & \text{if } N = 0 \\ \sum_{T_{a\Delta}=1}^{900} (Pr[T_{a\Delta} = T_{\text{block}}] \times Pr[\theta_{N-1} > \theta_{\min}]), & \text{otherwise} \end{cases}$$

Finally, we want to find the possibility of the attacker finding the next N blocks without having the difficulty value drop lower than the minimum value ($Pr[\theta_N > \theta_{\min}]$). This formula shown above is a recursive function. In the first base case, if the difficulty value of the N-th block is lower than the minimum, the possibility will be zero, since the condition fails the assertion that $\theta_N > \theta_{\min}$. In the second base case, if there is no block that the attacker needs to mine ($N = 0$), the possibility will be 1, meaning that after the attacker has mined n blocks (from $N = n$ to $N = 0$), since the attack is successful as long as $\theta_N > \theta_{\min}$. For all other cases, we multiply the possibility of mining a block with block time i seconds by the possibility of attacker mining the next $N-1$ blocks with the updated difficulty value, then we sum them from $i = 0$ to 900 seconds.

Using the given formula to calculate on a commodity PC, we use the Dynamic Programming (DP) technique. To fit the DP table into memory and get the result within a reasonable amount of time, we reduce the DP table's size by keeping the highest 6 or 7 digits of the difficulty value and assume the rest are zeros.

4.7 Evaluation Against Historical Blocks

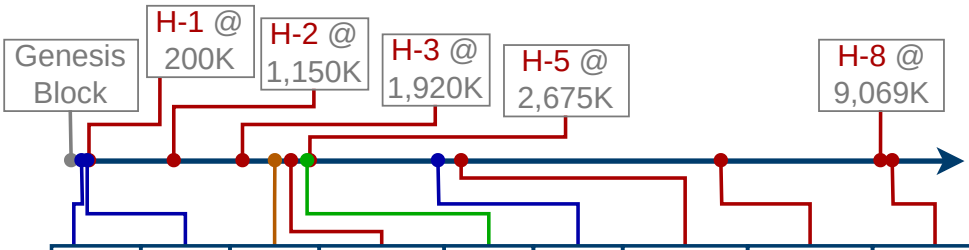
We implemented our methodology on top of Geth [66], the official Go implementation of the Ethereum client, since it is the most mature and widely used client. To evaluate the false positive rate under different parameters, we ran the monitor's algorithm against 10,900,373 historical blocks in Ethereum. We released our custom Geth database connector, experiment code and more technical details at [185, 184].

We have found that querying such a large amount of blocks headers through a third-party such as Infura [92] is impossible due to rate limits, and accessing blocks via Geth's API is extremely slow due to performance fluctuations when new blocks are received. Even batching requests only reduce the latency to tens of seconds per checkpoint. Therefore, the best practice to efficiently retrieve all block headers is to directly access the client's backend storage.

Geth divides the block storage into two parts: LevelDB, a key-value store for blocks mined in the most recent 15 days, and an ancient database for all older blocks. To get the block header from LevelDB, we encode the block number to get the header hash and encode the header hash to get the RLP-encoded header [71]. The ancient database is in a Geth customized append-only structure. For this, we wrote a DB connector in our experiment code based on the Geth's source code that allows lookups for the data location in the index file, and returns the data (i.e., block header).

Test Case Index	$\theta_{\min}$	N_c	Num of False Detections	False Negative Rate
1	$50\% \cdot \theta$	1270	2	$< 2^{-128}$
2	$60\% \cdot \theta$	880	2	$< 2^{-128}$
3	$70\% \cdot \theta$	620	3	$< 2^{-128}$
4	$80\% \cdot \theta$	430	5	$< 2^{-128}$
5	$80\% \cdot \theta$	610	7	$< 2^{-256}$
6	$90\% \cdot \theta$	275	17	$< 2^{-128}$
7	$90\% \cdot \theta$	420	23	$< 2^{-256}$

Table 4.1: Evaluating the difficulty detection algorithm under various checkpoints sizes against the entire Ethereum database.



	B-1	B-2	A-1	H-4 @ 2,463K	U-1	B-3	H-6 @ 4,370K	H-7 @ 7,280K	H-9 @ 9,200K
False Pos. By Test Case:	1 to 7	7	3 to 7	6, 7	6, 7	7	1 to 7	5, 6, 7	7

Table Legends:

H: Hard Fork | B: Bugfix | A: Attack | U: Update

Figure 4.4: Timeline of major events that cause false detections. The hardforks are [159], [171], [37], [93], [94], [68], [95], [96], and [97], respectively. The bugs are [151], [152], and [69], respectively. The attack is [170]. The major API update is [67].

4.7.1 False Negative Rate

The false negative rate, as shown in Table 4.1, represents the possibility of an attacker maintaining the difficulty value at the level higher than θ_{min} within N_c blocks. It is calculated based on the exponential distribution [23] established with attacker's hash power and network difficulty [184].

All combinations of N_c and θ_{min} that we have examined result in extremely low false negative rates: equivalent to brute-forcing a 128 or 256 bit key.

4.7.2 Difficulty Monitoring and Checkpoint Sizes

Table 4.1 also shows false positives that triggered our difficulty monitoring algorithm under various $\theta_{\min}$ and N_c. For all configurations, the algorithm resulted in multiple false detections. We have investigated all of these false positives and found that, as shown in Figure 4.4, Most of these events correspond to hard forks. During hard forks, network hash power fluctuates significantly due to miners upgrading their nodes. There are also instances where the detections were caused by DoS attacks [170] addressed with software upgrades. In both cases, enclave components would need to be shutdown and replaced by one built with the latest protocol anyway, so the monitor's behavior is not as concerning.

When $\theta_{\min}$ is set to a high value, such as $90\%\theta$, the monitor becomes more sensitive to smaller events, such as bugfixes and API updates. In the 7th test case, there is one false positive caused by a 10% difficulty drop that we could not tie protocol-related events but could be due to miners leaving for Ethereum Classic, which was increasing in value at the time.

When $\theta_{\min}$ is set to a higher value, the monitor needs a smaller checkpoint size (i.e., N_c) to confirm that the recent N_c blocks have a negligible chance to be mined by the attacker. As N_c increases, so does the number of false positives since the checkpoint captures a larger time interval surrounding hard forks. Thus there is a tradeoff between N_c and $\theta_{\min}$. The application may set a larger $\theta_{\min}$ or N_c to make malicious forks more difficult, but may incur more false detections since difficulty drops accumulate over a longer period.

Therefore, setting a reasonable $\theta_{\min}$ is a key to avoid false positives, and then configuring a reasonable N_c to reach the desired false negative rate level. For instance, with $\theta_{\min}$

set to 80%θ, all those false positives can be avoided if the enclave operator times the upgrade correctly; a lower N_c with a 70%θ or lower will also have the same sensitivity to difficulty drops.

4.7.3 Block times

Currently, the average block time in Ethereum is around 13 seconds. To find a proper value for the $T_{\Delta\text{Max}}$, which is used to prevent attackers from remaining silent forever, we record the top 100 block times from historical blocks in Ethereum. Our results show that while the majority of the blocks do arrive within the expected time, all of the top 100 blocks took over 200s to arrive, with 91 blocks taking less than 300 seconds, 7 blocks less than 400 seconds, 1 block less than 500 seconds, and one outlier that took 13013 seconds. Upon closer inspection, almost all of these blocks were mined very close to the Byzantium hardfork [68]. The reason that outlier block took almost four days to be mined is because of a consensus issue (which is B-1 in Figure 4.4) in Geth that caused miners to have to switch to alternative clients[151]. Given these results, choosing the maximum time given in Ethereum's DAA (around 900s) should give no false positives during normal operation; for applications that are very time-sensitive, a reasonable $T_{\Delta\text{Max}}$ would be 400 seconds.

4.8 Conclusion

Blockchain applications built with an enclave are vulnerable to eclipse attacks. In this chapter, we present a novel solution to this problem using a PoW difficulty monitor that is inexpensive, immune to long-range attacks, and provides the maximum reaction time. Based on

the evaluation against historical blocks, choices for θ_{min} and N_c with negligible false negative rates are unlikely to trigger false positives.

Chapter 5

Availability for Message Delivery

5.1 Introduction

The goal of publish/subscribe (pub/sub) systems is the dissemination of information from *publishers* to interested *subscribers* quickly and efficiently. Several production systems have been developed (e.g., [106, 84, 135, 88, 7]), and pub/sub systems have been applied in a variety of contexts, from financial applications [117, 45] to health monitoring [72], as well as real-time vehicle detection [107, 108]. To declare interest in a category of published events, subscribers register with a *broker*. Publishers send their events to the broker, possibly including metadata indicating relevant categories, and the broker notifies registered subscribers of the new event.

Some recent systems [136, 188] integrate pub/sub systems with blockchains to provide Byzantine fault tolerance and auditability of published events. However, these systems rely on Byzantine fault tolerant (BFT) protocols for consensus among off-chain replicas before

publishing events to the blockchain. Unfortunately, the trust assumptions between blockchain protocols and these off-chain BFT protocols are mis-matched. In addition to the blockchain protocol's trust assumptions (e.g., attackers control less than 51% of staked ETH [148]), subscribers must also trust an additional entity to manage membership in the BFT protocol. The membership management mechanism (MMM) is implicitly trusted to authenticate replicas, select the system parameter f that specifies the maximum number of Byzantine faults to tolerate, and maintain the required number of member replicas needed. Note that even if the MMM is distributed among the replicas in some fault-tolerant way, the subscriber must still trust that the parameter f is sufficient to prevent malicious events from being published, and that the non-faulty hosts—as viewed by the protocol—are trustworthy from the subscriber's perspective.

Publishing events to the blockchain means that the data will be visible to anyone, not just the subscribers. Some pub/sub systems, such as PUBSUB-SGX [11], use *secure enclaves* to protect sensitive data. The enclave is a secure hardware component that provides an application-level Trusted Execution Environment (TEE). Enclave programs, including their data, are cryptographically protected from all other components, including privileged OS components. However, PUBSUB-SGX is unable to ensure availability for event publication since it relies on direct TLS connections between publishers and subscribers, which are controlled by the host OS.

This chapter presents Decentagram, a secure and decentralized pub/sub system that combines the core capabilities of both blockchain and TEE-based mechanisms to address the above issues of availability and confidentiality. Open-membership blockchains are only capa-

ble of providing *integrity* and *availability* guarantees[1] for smart contracts and their data. Since block proposers and validators must know the contents of each block, they cannot provide confidentiality. TEEs are only capable of providing *integrity* and *confidentiality* for the code and data they contain. Since the TEE's host controls all channels to and from the TEE, no availability guarantees are possible. By carefully integrating these two technologies, Decentagram provides the trifecta of information security guarantees, as well as extends the capabilities of the system. Decentagram is the first pub/sub system that supports decentralization of *data oracle*, *publishers*, *brokers*, and *subscribers*, and is also the first blockchain-based pub/sub system that can deliver events on- and off-chain simultaneously. Decentagram's on-chain smart contract framework allows TEEs running on untrusted hosts to authenticate with attestation credentials. Authenticating TEEs on-chain provides Decentagram with three significant advantages over previous systems:

- First, Decentagram enables *permissionless publishing*, allowing any host to run a data oracle without requiring direct attestations to subscribers.

- Second, Decentagram supports *incentivized event availability*, which compensates publishers for timely event publication and enables nullification of any (rational) benefit gained by delayed or suppressed events.

- Third, Decentagram replaces trust in third-parties (such as a quorum of replicas) with much weaker trust assumptions: other than standard TEE and blockchain assumptions, subscribers only need to trust the enclave code, which can be audited (or even verified)

[1] In theory, confidentiality could be protected with advanced cryptographic protocols such as secure multi-party computation or fully-homomorphic encryption, but these are unlikely to sufficiently scale to blockchain settings.

beforehand.

Decentagram also supports on-chain subscribers: third-party contracts that receive published events. Propagating events to third-party contracts in previous systems [188, 173, 136] requires off-chain subscribers to monitor blocks for new events and submit a transaction with the event to the third-party contract. This Monitor-and-React (M&R) approach introduces a delay between when an event is published and when the client's smart contract can react, making it challenging to process events consistently across on-chain and off-chain subscribers. During periods of network congestion, the time between on-chain and off-chain reactions could increase if the relayed events are not included in the next block.

Enabling timely authenticated on-chain notifications is especially useful for applications with non-deterministic smart contracts that require data from an external party to make decisions [5]. For example, in Decentralized trading markets [64] in which clients monitor an on-chain marketplace contract for the latest price updates to make bids, an on-chain client can react immediately (within the same block) and make time-sensitive bids much faster than M&R clients (at least 1 block away). As another example, applications that utilize smart contracts to manage the membership of a group of off-chain compute nodes [31] also benefit from on-chain notifications. In this case, if any of the compute nodes are compromised, its credentials need to be revoked, and immediate notification of the on-chain membership contract will prevent the compromised node from any further participation in on-chain activities. We present a case study in Section 5.6 that demonstrates the effectiveness of decentralized revocation using Decentagram.

To the best of our knowledge, Decentagram is the first pub/sub system that delivers

on-chain events directly. This functionality uses gas limits on cross-contract calls to safely execute callbacks to third-party contracts, preventing broken or malicious contracts from launching gas-exhaustion attacks [15]. Furthermore, using Decentagram's incentivized event availability, the increased cost of publish transactions incurred by these callbacks is paid with subscriber fees, not by data oracles.

Finally, we present a design for confidential event publishing in Decentagram, where events are encrypted before publication and authorized subscribers decrypt events off-chain. Unlike other TEE-based pub/sub systems, Decentagram supports an on-chain key-exchange protocol where a subscriber can obtain a subscription key without requiring a direct connection to the publishing oracle. We provide a design for this protocol that uses the Decentagram framework to distribute keys.

In summary, this chapter makes the following contributions.

- We present the design and implementation of Decentagram, the first decentralized pub/sub framework with on-chain TEE attestation and authentication.
- Decentagram is resilient to Byzantine failures and is highly available, allowing any host to run as a data oracle as long as its enclave instance can authenticate itself to the on-chain broker contract.
- Decentagram's smart contracts support on-chain subscribers for clients, and provides atomic on-chain event notifications for subscribers to instantly react to events.
- An evaluation of the on-chain smart contracts shows that the gas cost to publish and subscribe is reasonable, and that the throughput of our off-chain components is more

than sufficient to handle events in new blocks.

The rest of this chapter is organized as follows. In Section 5.2, we discuss the related work and how Decentagram compares against state-of-the-art pub/sub systems. Section 5.4 contains the design of the on-chain and off-chain components in Decentagram, followed by the implementation details in Section 5.5. Next, in Section 5.6, we present a case study that utilizes all the features provided by Decentagram. Then, we provide the evaluation of Decentagram in Section 5.7, and finally, we conclude the chapter in Section 5.8.

5.2 Related Work

Table 5.1 shows the features of representative Pub/sub systems and how they compare to Decentagram.

Early work in pub/sub systems such as Linda [40] and SIENA [41] described how, in loosely-coupled distributed systems, events can be generated and consumed by a set of processes, but did not consider the possibility of machine failures. To provide availability in the presence of failures, data can be replicated across a set of servers, as is done in ISIS [26, 103] and modern industrial pub/sub systems like Kafka [106] and RabbitMQ [135], where a subset of the servers can fail by crashing. One common feature between these crash fault tolerant Pub/sub systems is the ability to provide causal delivery of events to subscribers. Causal ordering of events ensures that events sent by multiple publishers to a subscriber are delivered in the same order that they were sent. Though sufficient for some applications, a stronger reliability

[2]Most Chainlink clients access oracle data off-chain, but some on-chain processing for special *aggregator* contracts is possible.

Systems	Fault Tolerance	Fault Threshold	Auditability	Confidentiality	On-chain Notification	Publishing Incentive
Linda [40] SIENA [41]	✗	✗	✗	✗	✗	✗
PUBSUB-SGX [11]	✗	✗	✗	✓	✗	✗
ISIS [27, 26] Kafka [106] RabbitMQ [135]	Crash	$\lfloor \frac{n-1}{2} \rfloor$	✗	✗	✗	✗
HyperPubSub [188]	Crash	$\lfloor \frac{n-1}{2} \rfloor$	✓	✗	✗	✗
Trinity [136]	Byzantine	$\lfloor \frac{n-1}{3} \rfloor$	✓	✗	✗	✗
Chios [62]	Byzantine	$\lfloor \frac{n-1}{3} \rfloor$	✗	✓	✗	✗
Chainlink [31]	Byzantine	$\lfloor \frac{n-1}{3} \rfloor$	✓	✓	✗[2]	✓
Decentagram	Byzantine	Blockchain	✓	✓	✓	✓

Table 5.1: Representative Pub/sub systems.

guarantee such as publication total order [62], which ensures that all subscribers receive events in the same order, may be required for many applications (e.g., stock market data).

Some recent systems make use of blockchain technology to eliminate centralization of the event broker, tolerate Byzantine failures, and provide auditability for publications [136,

188]. HyperPubSub [188] keeps a record for each pub/sub operation on the blockchain, but the system does not fully tolerate Byzantine faults because the broker is implemented using Kafka which, as mentioned earlier, can only provide fault tolerance against crashes. Trinity [136] solves the issue of Byzantine brokers by using a Byzantine fault tolerant consensus protocol, and a Byzantine quorum of brokers (i.e., more than two thirds) need to agree on an operation before a transaction is sent to the blockchain to record the operation. While Byzantine fault tolerant consensus improves resilience against malicious brokers, they are limited by the fault threshold $f = \lfloor \frac{n-1}{3} \rfloor$, where n is the number of brokers. This means that if there are n brokers, at least $n - f$ brokers must be operational for the system to function, and the system will be unavailable if the number of accumulated faults surpasses f. Chainlink [31] describes a system with an on-chain oracle contract that can validate reports generated by a set of off-chain oracles. A designated off-chain leader oracle collects attestations to form a report and submits it to the oracle contract, which then determines the oracles that contributed to the report and compensates them. Each client can then monitor the oracle contracts for events that are emitted, and then send transactions to update their contract. The system, however, requires that the oracle contract belongs to an administrator who can not only add or remove oracles, but also set the compensation amount for the oracles. Clients can also interact with the oracles directly, essentially becoming their own on-chain broker, but this requires paying oracles per request.

In Decentagram, the broker is implemented as a smart contract, so the system inherits the blockchain's availability guarantees. For public blockchains, these are typically much stronger than what is possible to achieve with traditional BFT protocols. In Ethereum, there are currently over 974K validators and 24M ether staked in the Ethereum mainnet [21], meaning

that at least one-third [148] of them must be compromised to prevent the blockchain from proceeding. The system remains available as long as the blockchain is available and at least one oracle can provide authenticated publication. BFT protocols have been demonstrated to scale only up to the low hundreds of nodes [24], implying blockchain availability guarantees are at least three orders of magnitude higher. Thus Decentagram remains available under significantly more faults than systems such as Chainlink, Trinity, and Chios. Decentagram also improves on monitor-and-check approaches such as Trinity [136] by delivering events to subscribing contracts immediately. In Section 5.7.4 we compare the average, minimum, and maximum latency experienced by an application based on Decentagram and one based on the M&R approach.

Several prior pub/sub systems works have considered security and, more specifically, confidentiality for event subscriptions and notifications [20, 130, 181, 11]. One approach to providing confidentiality is through using access control policies for publishers and subscribers [20, 130, 181]. PUBSUB-SGX [11], like Decentagram, uses TEEs to provide confidentiality and integrity for subscriptions and to enforce subscription policies. Subscribers directly connect to event brokers (called matchers) and authenticate them via remote attestation. Unlike Decentagram, publishers do not execute in TEEs and are not incentivized to publish events, meaning that dishonest publishers may choose to delay or drop events to their own advantage. PUBSUB-SGX does not offer fault tolerance for event brokers or publishers, although the authors propose a replication scheme for brokers at the cost of duplicated event delivery. Unfortunately, this scheme does not eliminate the possibility of message loss [11].

Some works have presented smart contracts that run inside TEEs, such as enclaves. However, to authenticate the enclave, FastKitten [58] and LucidiTEE [74] require the partici-

pants to communicate with the enclave directly to perform remote attestation (RA). This prevents other smart contracts from verifying the results generated by the enclave. Ekiden [51] and PDO [30] require a customized consensus layer with the ability to verify the enclave's certificate chain, making it incompatible with existing blockchain networks, and expensive to deploy TEE-based contracts. To the best of our knowledge, Decentagram is the first system that supports on-chain RA using EVM (Ethereum Virtual Machine) smart contracts. Therefore, TEE-generated events can be verified on-chain, and Decentagram can be deployed on any EVM-compatible blockchains.

Like other large-scale pub/sub systems such as Kafka [106] and RabbitMQ [135], Decentagram provides a topic-based subscription model. Adapting Decentagram to other models, such as content-based subscriptions [146], may be possible, but resolving the usual tension between scalability and content filtering in this new context is beyond the scope of this chapter, though we expect that minimizing on-chain filter processing would be a key to reducing publishing costs.

5.3 System and Threat Model

System Model. Decentagram integrates with the smart contract framework of a permissionless blockchain to benefit from its *integrity* and *availability* properties. Our implementation targets the Ethereum Virtual Machine (EVM), so Decentagram's smart contracts can operate on any EVM-compatible blockchain. Any node can join the network as a blockchain participant or as a publisher, oracle, broker, or client in Decentagram. Decentagram uses TEEs

to provide *confidentiality* and *integrity* for event data. TEE message authentication is based on Remote Attestation (RA), where a key provisioned to the TEE by the hardware manufacturer is used to sign a new public key whose private key is only known to the TEE. Specifically, TEEs create self-attestation certificates [104, 183], which contain third-party verifiable credentials generated by remotely attesting the TEE to itself. These certificates are used to create authenticated channels between TEEs and for smart contracts to verify transactions from TEEs.

Threat Model. We assume attackers control at most the fraction of the blockchain network tolerated by the underlying protocol for its availability and integrity. In Ethereum, this is no more than $\frac{1}{3}$ of staked ETH (319k validators) to prevent halting progress, and no more than $\frac{1}{2}$ of staked ETH (478k validators) to fork the blockchain or force a reorganization. Nodes controlled by attackers may participate in or deviate from the blockchain protocol arbitrarily, and may send arbitrary transactions to any smart contract or blockchain address.

We also make the usual assumptions regarding TEE platforms: the attacker has complete control of the physical host executing the TEE and may delay, drop, or reorder messages to and from the TEE. Furthermore, the code and data contained in a TEE are cryptographically protected by a key provisioned by the manufacturer and (subsequently) known only to the TEE, and the TEE implementation is correct in the sense that it successfully enforces the security abstraction it claims to. Key-extraction and other side-channel attacks against a specific realization of TEE hardware (e.g., Intel SGX, AMD SEV, etc.) are beyond the scope of this chapter. Attackers are also assumed incapable of breaking cryptographic algorithms employed by Decentagram, the blockchain protocol, or the TEE platform.

Hosts of data oracles are assumed to be rational in that they will transmit new events

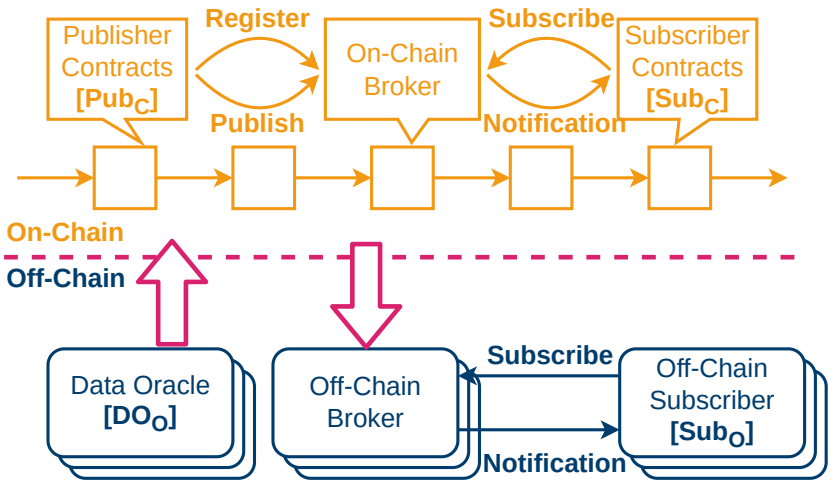


Figure 5.1: Overview of Decentagram

via transactions when profitable, and that compensation from publication is a sufficient incentive. For any TEE component, fraudulent (local) blockchain forks are detectable through eclipse attack detection [186] schemes, but we do not explicitly adapt and implement such techniques.

For encrypted event messages protected by a shared key, we assume protecting the message and its key is in the interest of both the data oracle and the subscribers. This is an appropriate assumption, for example, when the message contains the private information of the subscriber, but is *not* appropriate for applications such as digital rights management.

5.4 Decentagram Design

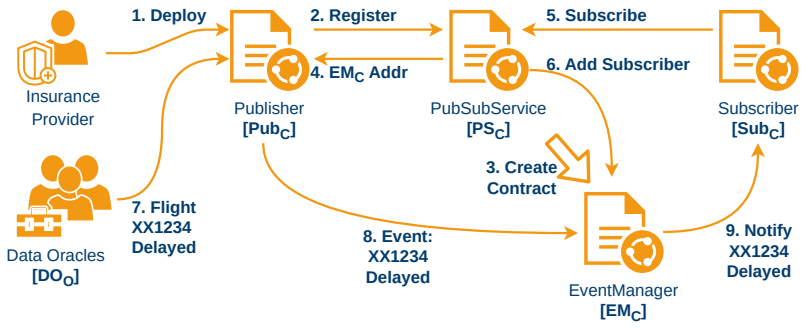
Figure 5.1 shows the architecture of the Decentagram framework, which consists of off-chain data oracles (DO_O)[3] and on-chain publisher (Pub_C), on-chain brokers, off-chain bro-

[3]In the following, we subscript off-chain Decentagram components with O for *off-chain*, and on-chain components with C, for *contract*.

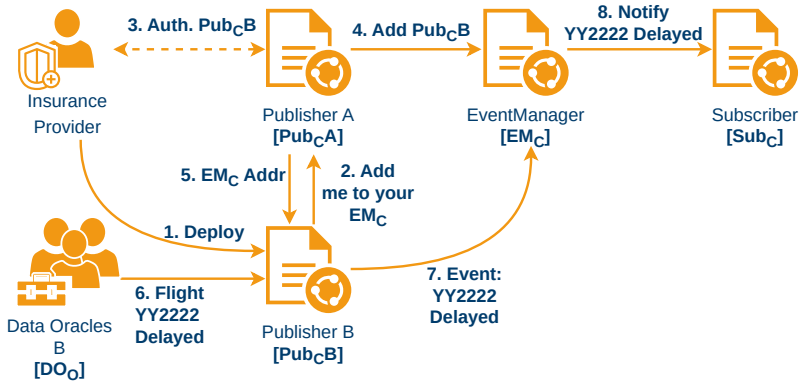
kers, on-chain subscribers (Sub_C), and off-chain subscribers (Sub_O). DO_O is the source of event data, so it knows what and how to collect data needed by the application. DO_O publishes a new event by making transactions, containing the event data, to the blockchain. The on-chain broker is implemented by a set of smart contracts that incentivize the publication of events and disseminate these events to Sub_C. Sub_O uses the off-chain broker to subscribe to events and receive notifications. Upon receiving a new event, the on-chain broker emits an EVM event, which includes the DO_O's event data in the current block. The off-chain broker then filters the EVM events in each block for those from the on-chain broker contract address.

Decentagram is made up of four types of smart contracts: PubSubService (PS_C), EventManager (EM_C), Publisher (Pub_C), and Subscriber (Sub_C). With respect to traditional Pub/sub systems, the PS_C and EM_C together form the on-chain broker, which indirectly connects publishers and subscribers. PS_C is the first contract deployed on the blockchain, after which all publishers and subscribers can use it by referencing its address. For each new event type that it registers, PS_C will deploy a new EM_C that handles the stream of events for that type. Each Pub_C is the entry point for off-chain data oracles (DO_O). When the DO_O sends transactions with data to its designated Pub_C, the Pub_C is responsible for authenticating the DO_O's data. If DO_O is a trusted source or is providing digitally signed data from a trusted source, the Pub_C can verify the digital signature on the data directly. Otherwise, when DO_O is operated by an untrusted host, Pub_C authenticates the DO_O's TEE to verify it is running known and trusted code, ensuring its content is trustworthy.

Among these components, DO_O, Pub_C, Sub_C, and Sub_O are application dependent, and implemented by application developers. We assume these known and trusted implemen-



(a) Registering a new Publisher and adding a Subscriber.



(b) Registering a new Publisher to another Publisher's EventManager.

Figure 5.2: Decentagram's on-chain workflow.

tations correctly use Decentagram libraries, so all authenticated TEEs contain correctly implemented Decentagram components.

Figure 5.2a illustrates the interactions between these contracts for the travel insurance example. At deployment, (1) the Pub_C registers itself with the PS_C (2), and receives the address of the new EM_C created by PS_C (4). To subscribe to these on-chain events, a Sub_C calls PS_C (5), specifying the address of the Pub_C it wishes to receive events from and a deposit to compensate the transaction cost of their event notifications. PS_C calls the relevant EM_C (6) to add the new Sub_C's address and callback function to its list of on-chain subscribers. For each update received and *verified* by the Pub_C (7), the Pub_C calls into the EM_C (8) for distribution to the subscribers, who are notified by executing Sub_C's callback function (9).

5.4.1 Publisher Contracts for the Same Event

Allowing more than one Pub_C for the same event type is dangerous since the new Publisher can accept data from a DO_O that the original Publisher would reject. To add an additional Pub_C safely, the initial Pub_C mediates access to its EM_C by other Pub_C. This approach provides a mechanism for expanding DO_O authentication mechanisms or policies and prevents the Pub_C from anticipating such mechanisms at deployment. For example, Figure 5.2b illustrates Publisher B's contract being added to Publisher A's EM_C. At deployment (1), instead of calling PS_C to create a new EM_C, Publisher B calls into Publisher A to request to be added to its EM_C (2). If the insurance company has authorized Publisher B (3), Publisher A adds the new publisher (4) and returns the address of its EM_C for Publisher B to use (5). Subsequent updates from Publisher B's DO_O (6) are sent to A's event manager (7), and distributed to the same list

of subscribers (8).

5.4.2 Incentivizing Publications

In Decentagram, Sub_C rely on the EM_C to notify them of events by invoking their callback function, and the EM_C in turn relies on Pub_C to notify it when these events happen. This series of cross-contract calls is initiated by a transaction from a DO_O, which invokes the Pub_C. Because these cross-contract calls happen within the same transaction, the transaction sender must include the cost of executing the target functions in the Pub_C, EM_C, and each Sub_C registered with the EM_C.

To ensure fair delivery of notifications, the EM_C requires each Pub_C call to include sufficient gas to execute all registered Sub_C callbacks. Without this check, the Pub_C could intentionally under-fund the call causing the transaction to be reverted or only a prefix of Sub_C to be notified. In either case, the Sub_O would be able to observe the transaction included in the block,[4] potentially allowing them to react to the event without Sub_C being notified.

The DO_O initiating the Publisher call is compensated for the cost of executing the transaction and successfully notifying the subscribing contracts by transferring the funds deposited by Sub_C at registration to the DO_O. The fee amount is specified by Pub_C during registration. If a Subscriber's deposit is insufficient, the Subscriber is removed from the list of subscribers before the callback is executed. The oracle may also be rewarded for each valid event submitted to the Pub_C, as specified by the Pub_C at deployment. For example, an application developer could offer a bug bounty program via Pub_C, where the first oracle to report

[4] A reverted transaction does not emit events, but the transaction (including the event data) would still be present in the committed block.

evidence of a compromised component is rewarded. We discuss such an example in more detail in Section 5.6. Our incentivization mechanism does not guarantee a particular fee schedule results in an incentive-compatible system since externalities may impact whether triggering an event is in the best interest of a DO_O. It does however provide system designers with a tool to compensate for such considerations.

5.4.3 Publications Authentication with TEEs

To our knowledge, Decentagram is the only decentralized pub/sub system that supports permissionless publishing: any entity can act as a DO_O as long as they meet the data verification requirements of Pub_C. When the publication source is independently verifiable, such as a message digitally signed by the original data source, any DO_O can send a transaction containing the signed message. To incentivize timely publication, Pub_C may choose to reward the first successfully committed transaction with the proceeds of notifying the subscribers; remaining successfully committed transactions are refunded minus transaction fees without notifying subscribers.

Some publication data may not be independently verifiable. For example, since data transmitted over a TLS connection is protected by an ephemeral shared key, participating parties can authenticate the data, but a third party is unable to guarantee after the fact which party generated the data. For these cases, Decentagram supports DO_O running in TEEs. Running an oracle in a TEE was first used by Town Crier [179], which ensures a webpage is retrieved properly via trusted code running in a TEE. Town Crier nodes are authenticated by off-chain subscribers directly using an interactive remote attestation protocol. Unfortunately, this design

scales poorly since the operators of both on-chain and off-chain subscribers would need to independently execute this protocol for each data oracle that may publish events.

In contrast, Decentagram's TEE oracles are verified by Pub_C using self-attestation certificates [104, 183], which authenticate TEEs to independent third parties without requiring an additional attestation round. Since the code authorized to produce updates is specified by the Publisher, Subscribers can assess the trustworthiness of the data sources before subscribing without connecting to specific instances.

5.4.4 Supporting Off-Chain Subscribers

The lower right portion of Figure 5.1 shows the overview of the off-chain portion of Decentagram. All nodes in a blockchain network have access to the blockchain data and the same view of confirmed blocks. The off-chain portion of Decentagram builds on this decentralized design by replicating the off-chain broker: each replica has the same copy of confirmed blockchain data.

Any application can subscribe to the events emitted by a Pub_C by authenticating the closest off-chain broker and subscribing to the events, specifying the address of the Publisher contract. The corresponding off-chain broker will be responsible for notifying the application when the relevant Pub_C emits an event.

5.4.5 Encrypted Event Notifications

Here we describe our design for confidential events in Decentagram. To protect event data, a DO_O will encrypt the data using a symmetric key and publish the ciphertext as an event.

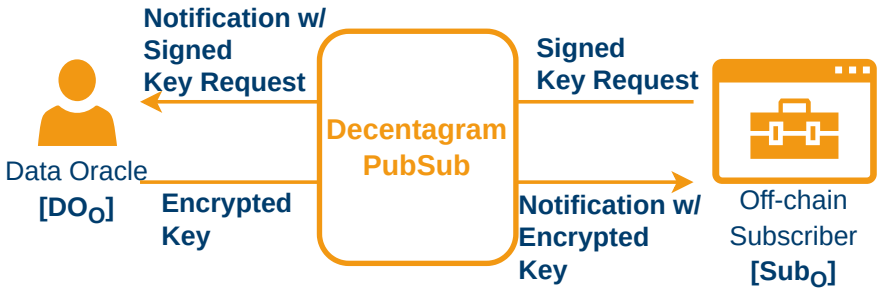


Figure 5.3: Key Exchange for Encrypted Event Notifications

For highly-available and scalable access to keys, we present a key-exchange protocol for authorized subscribers implemented on top of Decentagram.

Figure 5.3 shows the overview of this key exchange protocol. Sub_O sends a signed key request, including a public encryption key, to a pre-deployed key-exchange contract, triggering a notification to the DO_O. Based on the signed request, the DO_O will authenticate the requesting Sub_O, and encrypt the current key using the Sub_O's public key. The encrypted key is then sent to the key-exchange contract, which notifies the Sub_O. To mitigate *unintentionally* leaked keys by subscribers, keys should be refreshed periodically and stored in a secure hardware keychain hosted by a TEE or TPM [85]. As discussed in Section 5.3 distributing symmetric keys is appropriate when the subscriber can be expected to protect the confidentiality of the data. Encrypted events are only supported for off-chain subscribers since all contract inputs, intermediate values, and outputs are public to all nodes.

5.4.6 Dealing with Chain Reorganizations

A chain reorganization happens when two (or more) groups of miners in a blockchain network disagree on the sequence of blocks that form the canonical chain. Eventually, one

sequence is extended enough to be accepted by the entire network, but the nodes that initially selected the abandoned chain must rollback the effects of the abandoned blocks. Recent blockchain protocol designs such as proof-of-stake (PoS) are significantly less susceptible to chain reorganizations [70] than proof-of-work (PoW) protocols, but they are nevertheless possible under the right adversarial and network conditions [128]. Chain reorganizations could cause Sub_O in a pub/sub system to see and react to events in a sequence of blocks, only for those blocks to be rolled back and their transactions committed in a different order on the reorganized chain. Sub_C are unaffected by reorganizations since, like all contracts, they are reverted and re-executed on the new blocks.

An application may be indifferent to the order of events as long as they are eventually delivered (and not duplicated), but some may require a stricter ordering. While individual transactions cannot be replayed due to Ethereum's built-in per-address nonce for replay protection, enforcing a linear ordering of events from multiple oracles requires a nonce per event type, maintained by the contract and incremented for each event. The Publisher contract can require data oracles to include the current nonce. Since the publisher only accepts new events from an authenticated TEE, malicious hosts cannot replay previous events by changing the nonce.

In the event of a reorganization, transactions from different oracles may be reordered in a way that causes a nonce to be invalid and dropped by the Publisher contract. In this case, the oracle should revert its own local state and retry the transaction when the nonce is once again valid. A consequence of the initial transaction being rejected is that a competing oracle may successfully publish an event with that nonce before the original oracle's second transaction succeeds. Once a block is finalized (currently after at most 64 blocks), it can no longer be

reorganized, and is considered safe after at most 32 blocks. Luckily, block reorganizations are infrequent (a couple dozen out of over 7000 blocks per day), and almost always have a depth of only one block [70].

5.5 Implementation

We implemented the Decentagram smart contracts described in Section 5.4 in Solidity, a language for the Ethereum blockchain. Solidity was chosen because of its maturity and rich set of features. Specifically, we used the setting of gas limits on cross-contract calls, function access controls, access to the transaction cost and value, and digital signature verifications in our on-chain Pub/sub service implementation. Off-chain, we relied on the logging mechanism provided by smart contract event emissions together with bloom filters, transaction receipts, and merkle trees supported by an Ethereum client to filter blocks for events to notify subscribers. All implementations for on-chain and off-chain components are available on GitHub at `https://github.com/lsd-ucsc/Decentagram`, and on Zenodo at `https://doi.org/10.5281/zenodo.10224298`.

5.5.1 Decentagram Smart Contracts

The core on-chain components of Decentagram are PS_C and EM_C, which create new event streams for Pub_C and register new Sub_C. Publisher and Subscriber contracts are application-dependent, but we present examples in Section 5.6.

5.5.1.1 Securing against Problematic Subscribers

Subscribing contracts are untrusted, so we must isolate the execution of on-chain subscriber callbacks to prevent a buggy or malicious callback from causing an entire event notification to fail. All callback invocations are wrapped in *try-catch* blocks (exceptions are ignored), and the gas usage of the call is calculated after the call returns or an exception is caught.

To prevent gas exhaustion attacks [15], the EM_C limits gas usage by callback functions. If gas usage exceeds the remaining gas in the Sub_C's balance or a predefined gas limit set by the Publisher contract, the function is interrupted, and the EM_C continues notifying other Sub_C.

5.5.1.2 On-chain Subscribers and the Gas Limit

Blocks in Ethereum have a size limit that is defined by the maximum amount of computation required to execute the transactions contained in the block. Currently, the gas limit is 30 million [175]; a single transaction using as much as 30 million gas would take up the entire block. Our EM_C contract limits the amount of gas used by callback functions, but the overall transaction gas limit constrains the number of on-chain subscribers a specific EM_C can have.

The gas limit for callback functions is set by the (initial) Pub_C when a new EM_C is created. For reference, the cost to make cross-contract calls (such as to subscriber callback functions) is around 3347 gas, the cost to set a persistent (stored in the contract) boolean flag is 3050 gas, and the cost to add an element to a hash map is 22,200 gas. Assuming Pub_C consumes 2 million gas (the worst case when authenticating a new data source), this leaves

28 million gas remaining as an upper bound for executing subscriber callbacks. Balancing the tradeoff between the number of Sub_C and the callback gas limit is left to Pub_C. Our default gas limit per subscriber is set to 200,000, which allows for about 140 Sub_C.

One way of surpassing the maximum number of Sub_C would be to split subscriber notifications over multiple transactions. The publisher could register and deploy an additional EM_C for each new Sub_C, and notify each EM_C in separate transactions, each with a separate gas limit. The drawback of this approach is that some Sub_C will be notified later than others, and it may be necessary to create additional incentives to ensure the transaction sender notifies all Sub_C (e.g., requiring an up-front deposit that covers the total notification cost).

Only Sub_C actions triggered by a Pub_C transaction through a callback function are subject to these gas limits. For M&R style workflows, a user can use Sub_O to monitor the on-chain events. When a new event is received, the Sub_O relays that event to the desired smart contract in a subsequent transaction. Even more effective is a hybrid approach where a minimal Sub_C performs critical updates to the Sub_C's state, such as revoking access or setting flags. Following this, the Sub_O can send a transaction to complete any remaining tasks related to the event. For example, invoking a callback that just sets a boolean flag costs about 6500 gas. An EM_C with a subscriber gas limit of 6500 can support up to around 4300 Sub_C.

5.5.1.3 Authenticating TEEs On-Chain

Decentagram oracles and off-chain brokers are implemented as decentralized components in the Decent Application Framework [183]. Decent enclaves authenticate themselves to on-chain contracts using a self-attestation certificate: an X.509 certificate chain generated by the

secure enclave during the self-attestation process [104]. The root of trust for any TEE remote attestation protocol is the hardware manufacturer's certificate, in this case the Intel Attestation Service (IAS) certificate for Intel SGX enclaves. The certificate chain (see [183] for details) is used to authenticate the Decent enclave's certificate, which contains the enclave component's hash and public key. The hash is used to verify the intended code is running in the enclave, and will be the same for any host executing that Decent component. On startup, each component generates new public/private key pairs and initiates the self-attestation process to generate the certificate it uses to authenticate itself to on-chain contracts, clients, or other Decent components. Note that only the enclave has access to the private keys and mediates all uses of the keys in cryptographic processes. We provide a Solidity library function and pre-deployed smart contracts to verify the validity of this certificate chain. The verification process involves both RSA and ECDSA signature verification, validity period checking, and enclave attestation report parsing, but the Pub_C interface is just a single function call with the self-attestation certificate and the expected code hash. As far as we are aware, this is the first mechanism that verifies the authenticity of a secure enclave from within a smart contract.

Permissionless blockchains are Sybil-resistant [61] by design, but Decentagram's on-chain message authentication and verification prevents Sybils from being reimbursed for publishing spurious events. While any Ethereum client may send transactions to Decentagram contracts, only those produced by authenticated Decentagram components are accepted and compensated with subscriber fees. Thus only messages produced by authentic Decentagram data oracles, which are assumed to be valued by subscribers, will be accepted for publication.

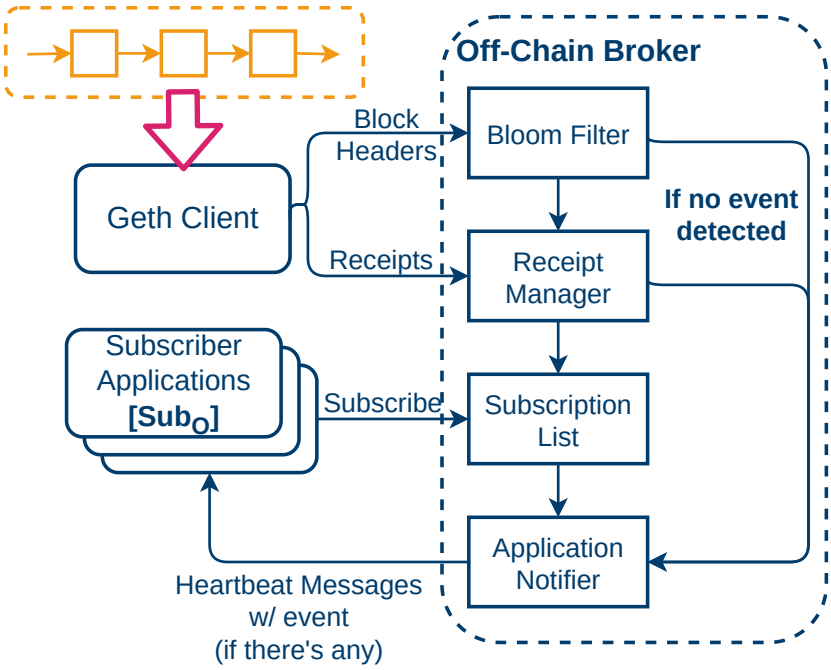


Figure 5.4: Overview of the Off-Chain Broker Implementation

5.5.2 Sending Events to Off-Chain Subscribers

Figure 5.4 shows an overview of the off-chain broker's workflow. The Geth Client [65] communicates with other Ethereum nodes to maintain the blockchain data locally. The off-chain broker constantly polls the Geth client for new block headers and checks for new events emitted by the addresses of EM_C contracts it is monitoring. When a new event is received, the off-chain broker consults its list of Sub_O for that event type and sends each Sub_O the new event.

5.5.2.1 Source of Blockdata

The blockchain data is usually generated and stored by the Ethereum nodes in the network. Different nodes in the network communicate to synchronize their view of the chain.

Instead of implementing these logics in the off-chain broker, we directly retrieve the blockchain data from the Geth client using RPC calls via TCP connections. Therefore, the Geth client is responsible for maintaining the blockchain data and synchronizing with other nodes in the network. Alternatively, developers may choose to use other Ethereum clients that follow the standard Ethereum protocol and provide compatible RPC interfaces.

To support applications using secure enclaves, our off-chain broker can also run on the Intel SGX secure enclave platform, but additional protection is needed to ensure the authenticity of the blockchain data received from the Geth client running outside the enclave. Being isolated from the OS means that enclave programs have no control over the OS resources, such as the network connections. They have to rely on the programs running outside of the enclave to feed in the blockchain data. An adversary may perform eclipse attacks [89] by crafting fake blockchain data and passing it to the enclave in order to suppress the revocation events. To prevent such attacks, in a PoW blockchain, an eclipse attack detection scheme [186] can be applied. This is less of a concern for PoS blockchains, since recent work suggests that it requires at least 2.4 years to complete the attack [180]. Therefore, proper verification of the blockchain data inside of the enclave is sufficient to prevent eclipse attacks in PoS blockchains like Ethereum.

5.5.2.2 On-Chain Events

After receiving blockchain data, the off-chain broker checks for monitored contract events. Ethereum contracts may emit named tuples called (helpfully) *events*, which are included in the transaction receipt. Decentagram uses these events to communicate with off-chain bro-

kers.

The constructor of PS_C emits a `Deploy` event marking the initialization of the on-chain broker. The off-chain broker then begins to monitor for `Register` events. Each time a publisher registers, a `Register` event announces the address of its corresponding EM_C. With this information, the off-chain broker can maintain a mapping of publisher addresses to event manager addresses, and start to monitor for `Publish` events, in order to know the occurrence of events emitted by that event manager. By knowing the address of PS_C, the off-chain broker can determine the event manager addresses it wants to monitor for `Publish` events.

These events emitted by the on-chain contract will then be visible in the log data of transaction receipts. A naive approach to monitor these events will be retrieving all the receipts of every block and checking their logs to see if any of them contain the events of interest. However, parsing and reading all the receipts incurs a significant amount of overhead, especially when validating the receipts root hash is necessary. We discuss more about this overhead in our evaluation of the off-chain broker in Section 5.7.3. Instead, we can look at the bloom filter contained in the block header to probabilistically determine if a block contains events we are interested in.

In Ethereum [172], each block header contains a bloom filter that is constructed using the address of contracts that have emitted event(s), the event's signature, and indexed event's arguments. To efficiently filter events, the Broker only has to check the header bloom against the contract addresses and the event signatures. If the filter signals a positive result, the rlp-encoded receipts will be fetched and processed. False positives can occur, but the false positive rate on Ethereum's main chain is promisingly low (about 0.5% [138]).

5.5.2.3 Subscriber Services

The off-chain broker could start at any point of time. Some instances may be started before PS_C is deployed, while other instances may be started after deployment and some events have already been published by EM_C. Similarly, the Sub_O may start subscribing to the Pub_C before or after the Pub_C has published any events. For instance, a Sub_O wants to subscribe to a Pub_C that emits events when a new item is added to a revocation list. If the Sub_O starts subscribing after Pub_C has already published some events, the Sub_O will also want to know what items are already in the revocation list. Because of this, the off-chain broker must maintain a record of all the events that have been published since the deployment of PS_C.

As described above, the off-chain broker knows the beginning of the on-chain Pub/sub services and all the addresses of the existing event manager contracts. During the boot phase of the off-chain broker, it will retrieve history blocks from the Geth client, and record all the past events. So, in addition to notifying the Sub_O of the new events, the off-chain broker will also log these events in storage. At runtime, the off-chain broker accepts event requests from Sub_O, which specify the address of Pub_C at the beginning of a TCP connection. The Broker replies with all the past events and corresponding block numbers from that Pub_C. The Sub_O receives new events over the TCP connection as they occur.

To notify the Sub_O of the new events, the off-chain broker can simply send the event data via the opened TCP connection. However, this approach is not secure enough in case the network connection between the Broker and the Sub_O is untrusted. For instance, an adversary may suppress the TCP connection between them to pretend that the publisher has not published

any new events. A subscriber that is listening to the flight delayed events from the off-chain broker may be misled to believe that the publisher has not published any delayed flight since the subscription.

To address this issue, the Broker will send heartbeat messages through the opened TCP connection periodically. Each heartbeat message contains the latest block number of the blockchain. In case a new event is published, the event data will be sent along with the heartbeat message. If the Sub_O does not receive any heartbeat message for a certain period of time, it may assume that the connection is broken, and take appropriate actions to prevent possible losses.

5.6 Case Study: Revocation of Enclaves

Decentagram is useful for more than just web oracles that import data from off-chain sources. In the flight insurance example discussed in Section 5.4, consider an oracle that uses login credentials to an airline's booking system to automatically re-book a client's travel when a cancellation or delay occurs. These credentials are secure from a malicious host if the oracle is running within a TEE, but a vulnerability in the enclave code could cause the client's credentials to be leaked or fraudulent claims disbursed. Once discovered, quickly revoking the credentials of all oracles running the vulnerable component is critical to mitigate exploitation.

Timely revocation is a challenging problem: instantaneous revocation is not possible in almost any distributed system, but is even more challenging in a decentralized system. We present a decentralized solution for timely revocation as a Decentagram service. The insurer's contract receives immediate on-chain notification when a valid revocation is published, prevent-

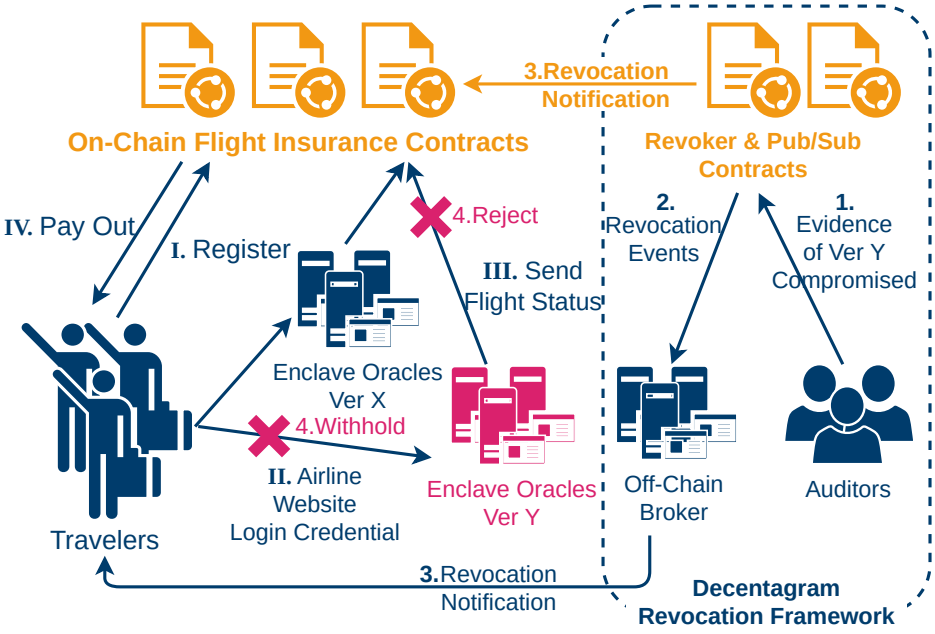


Figure 5.5: One implementation of an enclave oracle in the Figure 5.2 workflow is proven vulnerable by evidence from the auditors.

ing fraudulent claim payouts, and off-chain subscribers receive notification when the relevant block is received.

5.6.1 State of the Art

A few existing works have discussed using smart contracts to govern the membership of a system. SCPKI [3] is a PKI system that uses smart contracts to manage attributes of participants in the system. IKP [118] provides a smart contract that can parse X.509 certificates to check whether it is verified for a domain. Unlike Decentagram, neither IKP nor Proofchain [139] provide a way for other contracts that depend on certificate revocations to be notified.

In a native enclave model, such as Intel SGX, a platform revocation mechanism is provided by the hardware manufacturer [98, 142], to revoke a compromised hardware platform. However, security vulnerabilities are also commonly found in the software implementations.

Several existing works, such as SCONE [12], Graphene-SGX [46], MAGE [47], Decent [183], and CCF [137], proposed distributed enclave application frameworks, where different enclave components can be deployed on different machines and communicate with each other. SCONE, Graphene-SGX, and MAGE have not explicitly discussed the means to revoke problematic components. In the framework proposed by Decent, replicas of a specialized enclave component distribute the revocation list to other enclave components; but it is unclear how those replicas reach the consensus on what components to revoke. CCF utilizes a permission-based ledger to manage the membership of enclave components. Participants in the system can propose and vote to add or remove enclave components. This approach may not be suitable

for applications used by the general public. Additionally, participant interactions may cause unnecessary delays in the revocation decision for certain simple but critical situations, such as when the private key of an enclave application is leaked. Therefore, a more *effective* and *flexible* revocation framework is needed.

5.6.2 Decentagram Revocation Framework

We present a revocation framework built on top of Decentagram, with three types of revokers in the role of on-chain publishers as examples: the voting revoker, the conflicting message revoker, and the compromised key revoker. It should be noted that the revocation mechanism is not limited to these three types, and applications can define their own revokers based on their needs. These revokers act as publishers (as in Figure 5.2a), so each registers with PS_C to obtain an event manager. The Sub_C uses these revokers for timely revocation events. Meanwhile, the off-chain broker listens to the on-chain events emitted by EM_C, and notifies Sub_O.

The conflicting-message and compromised-key revokers require messages to be signed with a key only known to the to-be-revoked enclave in order to prove it has been compromised. To achieve this, we use the same mechanism described in Section 5.5.1.3 for authenticating DO_O.

Voting Revoker Contract. The voting revoker is similar to the propose-and-vote scheme used in CCF. During contract construction, the transaction sender needs to specify a list of stakeholders. Then, during runtime, any stakeholder can vote to add a component to the revocation list. If the number of votes reaches a specified threshold (e.g., 2/3 of the stakeholders),

the contract adds the component to the revocation list, and publishes a corresponding revocation event using EM_C. Stakeholders can also vote to add more stakeholders, or remove existing stakeholders.

Conflicting Message Revoker Contract. In a distributed application, it is often the case that components need to communicate with each other or with the users. For example, in replicated systems, components may need to vote to elect a leader or decide on a value. If replicas can be Byzantine, these systems can be vulnerable to equivocation from malicious replicas that send conflicting votes [1, 101, 43]. For this type of malicious behavior, the voting revoker becomes inefficient. Instead, we can use a conflicting message revoker to determine the presence of such behavior and quickly revoke the malicious component, without waiting for votes from stakeholders.

Unlike the regular message signing scheme, where the sender signs on one hash calculated based on the entire message, the message sender here is required to compute a hash of a session ID (e.g., leader selection session X), and a hash of the message content (e.g., vote for node C); then, the sender signs on a single hash that is calculated from both of these hashes. An outside auditor can monitor the session ID hash, the message content hash, and the signature. When two messages have the same session ID hash but different message content hashes, it indicates that the sender has sent conflicting messages; and the auditor can report the corresponding hashes and signatures to the revoker contract. The contract will then check if the hashes and signatures are valid, and add the corresponding component into the revocation list in case of a conflict. By calculating two separate hashes for session ID and message content, neither the auditor nor the blockchain participants can know the actual message content.

Compromised Key Revoker Contract. Another revocation scheme is to revoke a component when its private key is compromised. The private key of an enclave application is stored inside the enclave memory, and is only accessible by the enclave component. Therefore, the revoker contract can quickly revoke a component when it receives a signature over a well-known revocation message (e.g., `"REVOKE THIS KEY"`), signed by a private key that should have been kept secret by the enclave. By verifying the signature of a revocation message, the contract can revoke an enclave component not only when its private key is completely exposed, but also when the private key is partially exposed, such that it is sufficient for an adversary to forge a signature.

Non-Enclave Version. Often times, these revocation schemes also make sense for regular applications that do not use secure enclaves. For example, the voting revoker can be used to propose and vote on a public key, of which the corresponding private key is compromised. Or the conflicting message revoker can be used to detect conflicting messages signed by the same private key. Therefore, we also provide a set of revokers using the same logic as described above, but are used to revoke general EC keys.

After the revoker contracts have verified the evidence and added the corresponding components into the revocation list, an event will be emitted to the transaction receipt by their EM_C. The events included in the receipts will be monitored by the off-chain broker as described in Section 5.5.2. Once an event is found in the receipt, Sub_O will be notified. The Sub_O could be enclave components from a distributed enclave application or an application that communicates with enclave components. After receiving the notification, these subscribers can take appropriate actions to protect themselves from the compromised components such as updating

access control lists or purging potentially corrupted data.

By utilizing public blockchain and smart contracts, our approach not only provides a means to distribute the component revocation list, but also provides a decentralized mechanism to add components into the list. The ability to define different types of revokers shows the *flexibility* of the framework, and the timely revocation events delivery provided by Decentagram highlights the *effectiveness* of the framework.

5.7 Evaluation

5.7.1 Publisher, Subscriber, and On-Chain Broker Contracts

We evaluate the gas cost of our implementation of the on-chain broker with minimal publisher and subscriber contracts. Pub_C registers with PS_C and exposes a function to publish a fixed-payload event. Sub_C subscribes to Pub_C events and verifies it receives the expected payload for each event. To evaluate gas cost, we deployed our contracts on Ganache, a local Ethereum blockchain testing environment [158].

Based on the transaction receipt, the gas cost of deploying PS_C is 623,330. In addition, we evaluated the gas costs of *using* the on-chain broker, including three major operations - registering, subscribing, and publishing. In this experiment, we measured the cost of each operation, and repeated the process with the number of publishers and subscribers increasing from 1 to 20. For event registration, we calculated the average gas used per publisher. As the number of on-chain publishers increases, the average gas used per publisher stays relatively constant, at around 570 thousand gas, with negligible fluctuations (less than 50 gas). Ethereum's mapping

data structure allows data to be fetched and stored with constant gas cost, so the gas cost of managing the address of Pub_C and their EM_C does not increase with the map size.

A similar process is used to evaluate the gas cost of subscribing. We deployed the Sub_C, with each of them subscribing to a different Pub_C, and recorded the gas used. As the number of Sub_C increases, the average gas used per subscriber also stays constant, at around 160 thousand gas, with negligible fluctuations (less than 10 gas). Like the `register` operation, the `subscribe` operation uses the mapping data structure to look up the address of the event manager, which results in a constant gas cost.

Next, we evaluated the cost of publishing by deploying multiple Sub_C that subscribe to a single Pub_C. With one subscriber, the publishing cost is 134,768 gas. As the number of Sub_C increases, publishing cost increases linearly, with the marginal cost around 76,000 gas for each additional subscriber. That is because the `publish` operation iterates through the list of Sub_C to invoke their callback functions.

Today, the cost per gas in the Ethereum Mainnet is around 0.001 cents. Thus, the base cost to publish a message is \$1.35 increasing by \$0.76 for each Sub_C. Our contracts are also deployable on EVM-compatible chains. We have deployed and tested Decentagram contracts on Avalanche as well as "Layer 2" (L2) chains Polygon, BNB, and Optimism. L2 chains execute contracts and transactions on a local chain and commit checkpoints to the main Ethereum chain. For L2 chains, off-chain components in Figure 5.1 would monitor and interact directly with the L2 chain.

Deploying Decentagram to such a chain greatly reduces the cost to publish messages while still benefiting from the security of the Ethereum mainnet. For example, the cost per

Operations	Enclaves	secp256k1 Keys
Revocation by Voting		
Deploy	840,815	852,279
Vote (average)	81,386	81,267
Revocation by Conflicting Messages		
Deploy	1,874,373	814,601
Report	1,515,330	70,047
Revocation by Compromised Keys		
Deploy	1,882,836	762,302
Report	1,507,982	66,015

Table 5.2: Gas Costs for Revocation Contracts

gas in Polygon network is $4 \cdot 10^{-6}$ cents, and $6 \cdot 10^{-5}$ cents in BNB network. At these prices, the (base, per-subscriber) cost is ($0.0053, $0.0030) and ($0.086, $0.049), respectively. Note that subscriber fees are expected to reimburse publishing costs, and these prices represent the "break-even" cost for publishers. Higher transaction fees require higher subscriber fees to incentivize publisher participation, so reducing these fees enables a wider range of Decentagram applications.

5.7.2 Revocation Contracts

The evaluation of the on-chain revocation contracts consists of two sets of revokers: one is for revoking enclave components, as described in Section 5.6, and the other one is for revoking general secp256k1 Elliptic Curve (EC) keys; and the results are shown in Table 5.2.

We initialized the revocation-by-voting contracts with three stakeholders, with two votes being sufficient to revoke an enclave image or key. The gas cost of the `vote` operation is calculated by averaging the gas used in these two votes. As shown in the table, both of the voting revokers have similar gas costs since counting votes is similar for each situation.

The gas cost of the conflicting messages revoker, as well as the compromised key revoker, are significantly higher for enclaves than for general EC keys. This is because the two enclave revokers need to verify the remote attestation reports from the running enclave component, since the conflicting messages revoker needs to ensure that the messages are generated by the same instance of an enclave, and the compromised key revoker needs to verify that the proposed keys are held by an instance of the enclave component. The verification of the remote attestation reports includes one verification of RSA-signed X.509 certificates, one verification of RSA signatures, and one JSON message decoding, which are expensive in terms of gas costs. High gas cost is typical for frameworks that parse and manage X.509 certificates on-chain (e.g., [2]). Since revocations occur infrequently and are critical to security, these gas costs are justifiable. A vulnerable Decent component only needs to be revoked once: even if a new host loads the component and generates a new signing key, the attestation certificate will be rejected due to the revoked component identity. Enclave application developers could provide bug boun-

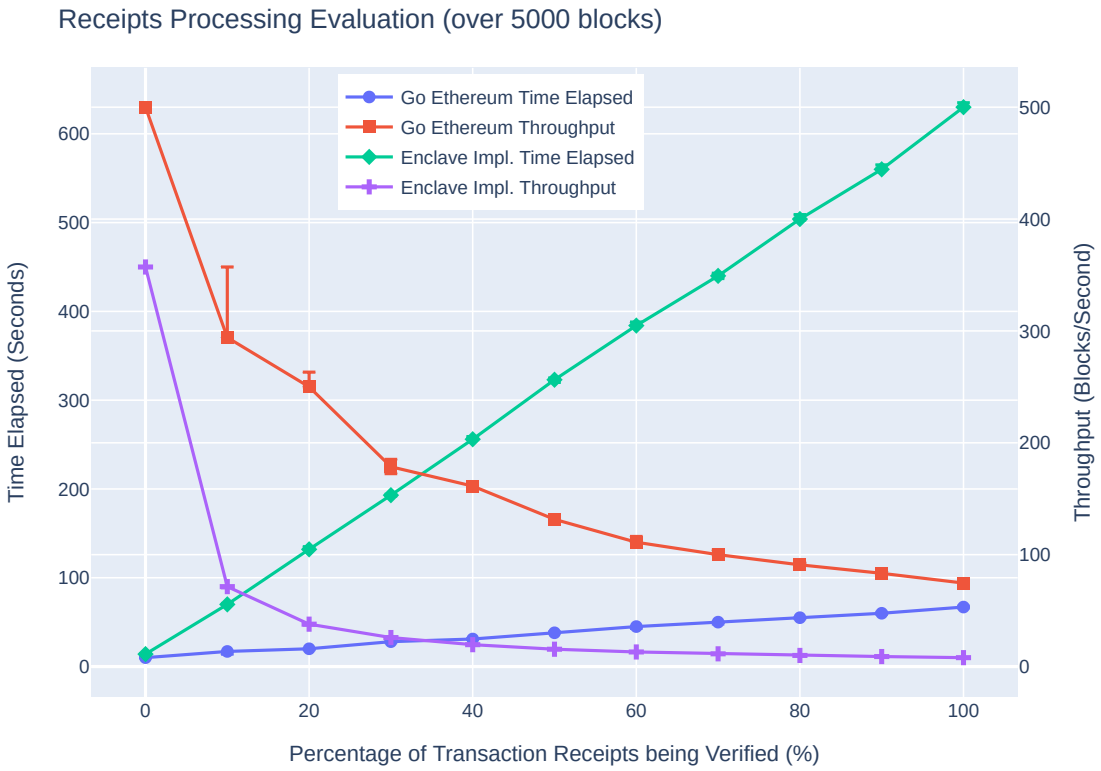


Figure 5.6: Off-chain Receipts Processing Evaluation. Error bars (where visible) represent the max and min of three runs.

ties to incentivize the disclosure of vulnerable components and cover the cost of the revocation process.

5.7.3 Off-chain Block Processing

In the off-chain evaluation, we focus on block processing efficiency, so that the Sub_O can be notified of the events in these blocks in a timely manner. Among those steps described in Section 5.4, we have identified the major overhead in block processing as being the parsing and validating of transaction receipts. We evaluated two versions of receipt processing implementa-

tions: one is our implementation built and run inside of the enclave environment, required by the framework described in Section 5.6, and the other version uses Geth under the normal execution environment. The experiment is conducted on a PC running the Ubuntu operating system, equipped with an i3-7100T CPU, and 32G of RAM. The results are shown in Figure 5.6.

The off-chain broker uses the bloom filter in the block header to skip processing of blocks that return a negative result. In our experiment, we simulate positive bloom filter results for 0% to 100% of 5000 blocks in 10% increments. These blocks were selected from the range 8,875,000 to 8,880,000, in the Ethereum Goerli testnet. We can see that the time taken by both implementations increases linearly as the possibility increases. Even when receipts from every block have to be parsed and verified, the enclave implementation shows a throughput of 7.94 blocks/second, which is around 95 times faster than the Ethereum block arrival rate of 12 seconds/block. In case of Geth, the throughput is 74.63 blocks/second, which is around 896 times faster than the block arrival rate. Hence, regardless of the off-chain broker version the Sub_O uses, there should not be any backlog of new blocks, and Sub_O will be notified of new events in a timely manner.

5.7.4 End-to-End Latency Comparison

Decentagram only requires that a candidate event be signed by an authenticated component, avoiding the need for an off-chain consensus round prior to publication as in Chainlink and Chios. Therefore, we evaluate the latency of Decentagram by measuring the time from when events are published to when they are delivered. The M&R approach serves as a good baseline for comparison since it also does not require consensus prior to publication. To com-

	$DO_O \rightarrow Pub_C$	$Pub_C \rightarrow Sub_C$	$DO_O \rightarrow Sub_C$
Decentagram	12 s (Max:35, Min:5)	0 s (Max:0, Min:0)	12 s (Max:35, Min:5)
M&R	12 s (Max:46, Min:10)	12.5 s (Max:35, Min:9)	25 s (Max:58,Min:22)

Table 5.3: Median End-to-End Latency Comparison.

pare the efficiency of Decentagram with the traditional M&R approach, we conducted an experiment to measure the end-to-end latency of notification delivery. In both test cases, a DO_O publishes new data to a Pub_C, and a Sub_C waits to be notified of the new data. Additionally, a Sub_O monitors new blocks to be notified of the new data as well as the confirmation from the Sub_C showing that it has processed the data. The Ethereum Goerli testnet, which has a block arrival rate of 12 seconds/block, was used in this experiment; and all the test cases were repeated 20 times, with the median, minimum, and maximum values reported in Table 5.3. The first column shows the time elapsed from DO_O publishes data until Sub_O receives it via the event emitted by Pub_C. The result between the two approaches are almost the same, since both of them require the DO_O to make a transaction to Pub_C, and then Sub_O will be notified when they receive the event. The second column shows the time required for the Sub_O to receive the subsequent confirmation from Sub_C after it has been notified of the event from Pub_C.

As expected, the difference in latency between the two approaches is significant. With Decentagram, the Pub_C was able to notify the Sub_C via cross-contract call within the same transaction. Thus, when Sub_O received the event, it also received the confirmation that Sub_C has

finished processing the data. While in the M&R approach, the Sub_O has to react to the event by making another transaction to Sub_C, which results in a longer latency. The third column shows the total time elapsed from event publication to the confirmation from Sub_C. Decentagram is able to complete the entire process using only one transaction, reducing the latency by half.

During the experiment, we encountered network fluctuations, with some time slots being skipped causing the new block to arrive later than expected, and some blocks being empty causing our transactions to be delayed until the next block. The time it takes for data to propagate from DO_O to Sub_C took even longer when these two situations occurred simultaneously. Such situations are common on the testnet but rare on the mainnet. However, the mainnet is also more congested, which would also cause similar effects. In both cases, Decentagram is less affected by these fluctuations since it only requires one transaction to notify subscriber contracts compared to two transactions required by the M&R approach.

5.7.5 Application Domain

Topics in Decentagram are identified by the Pub_C address, so the number of channels is not limited in any practical way (there are 2^{160} distinct contract addresses). The number of Sub_C addresses per channel is capped by the block gas limit (see Section 5.5.1.2). In our default configuration, each channel can support up to 140 on-chain subscribers. Note this limit only applies to *on-chain* subscribers—there are no limitations on the number of *off-chain* subscribers. In fact, since the off-chain brokers are decentralized, the number of off-chain subscribers scales indefinitely.

Message delivery latency relies on the block arrival rate of the underlying blockchain

network. The 12 seconds/block rate is the result of our use of the Ethereum blockchain. EVM-compatible L2 networks such as Polygon or BNB have faster block rates (2 or 3 seconds, respectively). Taking these aspects into account, Decentagram is most suitable for applications that have event-generation rates on the order of seconds, need to scale to massive numbers of off-chain subscribers, with on-chain subscribers reasonably distributed over many topics.

5.8 Conclusion

We presented Decentagram, a framework for instant delivery of on-chain events and timely delivery of off-chain events using the pub/sub messaging model. Our Publisher, Subscriber, and Broker contracts are resilient to Byzantine failures and provide incentives for event publications. We motivate the framework with a decentralized revocation case study, demonstrating its benefits over the state-of-the-art in revocation speed for on-chain subscribers. We evaluated Decentagram's gas cost for contract deployment and execution, and demonstrated receipts processing throughput of the off-chain broker is more than sufficient for processing new blocks at the rate they arrive.

Chapter 6

Availability for Compute Services

6.1 Introduction

In Function-as-a-Service (FaaS) platforms developers decompose complex web applications into "serverless" ones made up of smaller, mostly stateless functions hosted by a provider. Many cloud providers offer FaaS services including AWS Lambda, Microsoft Azure Functions, and Google Cloud Functions. These smaller components are easy to replicate and result in higher utilization of resources than their monolithic counterparts. FaaS clients are charged per invocation for the resources used to execute their workloads, whereas infrastructure-as-a-service (IaaS) clients are charged by the *duration* of resource reservations, whether the resources are idle or not.

There are two major drawbacks to current FaaS platforms. The first drawback is that realizing high availability for FaaS-based applications is challenging. When widespread cloud outages occur, intra-cloud redundancies are often ineffective since some outages affect multiple

regions operated by a vendor and may take hours to resolve [49]. Moreover, using multi-cloud approaches to replicate FaaS components on multiple clouds is somewhat more complex than replicating IaaS components. Since IaaS components control the full stack used by their application, clients only need to identify a host and communicate with it over standardized network protocols. While configuration and provisioning may differ between cloud vendors, offering a consistent interface to authenticate clients and access services can be straightforward. FaaS components reside higher in the application stack, which has the advantage of reducing resource allocation and overhead, but also results in more vendor-specific interfaces [19].

A second drawback is the vulnerability of cloud services to censorship, allowing countries to restrict access to these services in favor of ones controlled and monitored by their own government. Due to the significant centralization of cloud services across a small number of vendors, blocking access is almost trivial—much easier than maintaining blocklists for the domestic websites that might make use of them.

While not a panacea, blockchain technology excels at providing high availability and censorship resistance. Whereas 15 to 20 "severe" or "serious" service outages occur per year across industry according to Uptime Institute [110], neither the Bitcoin nor the Ethereum networks have experienced outages of similar severity in over a decade.[1] Recent work has explored adapting the FaaS model to blockchain-based applications [13, 78, 77, 75], but there are several challenges to building decentralized FaaS services.

A decentralized FaaS service must provide a secure execution environment that pro-

[1] Bitcoin last experienced downtime in 2013, according to `https://bitcoinuptime.com/`. Ethereum has not experienced any downtime since its inception. However, both networks occasionally experience congestion issues resulting in reduced availability of network services for some clients.

tects both the host and the client from malicious interference [134]. When client workloads are executed on untrusted hosts, a malicious host can tamper with the code and data, affecting the correctness of the results. Therefore, the execution environment must provide confidentiality and integrity for the client workloads to prevent unwanted access from the host. Conversely, an honest host should also be protected from malicious workloads from clients that attempt to exploit the host's resources or interfere with workloads from other clients.

Additionally, a decentralized FaaS service requires fine-grained resource accounting for client workloads that is mutually trusted by both hosts and clients. In particular, neither the host nor client should be able to manipulate the accounting mechanism to bill more or less than the actual charge for resources used.

Unfortunately, current decentralized FaaS systems do not address all of these issues, limiting their use cases. As an example, consider adapting decentralized healthcare data management [17, 174, 16] applications to a FaaS-based implementation. Integration with blockchain technology allows for designs of systems that are more reliable and can function without the need for a trusted third party to manage healthcare data. Data confidentiality and integrity is critical due to the sensitive nature of the healthcare data, but providing adequate guarantees without a universally-trusted third party is not possible with current systems. For example, suppose researchers studying the side effects of a particular vaccine want to know how many patients in a region have been treated for severe side effects, as well as aggregated statistics on their pre-existing conditions. Compiling this data requires access to vaccination records from multiple sources (e.g., hospitals, clinics, and pharmacies), and could require approval from and contractual agreements with many different entities [17]. The source institutions must be

able to trust that the researchers will not misuse the data, and must anonymize the data before release to protect the privacy of the patients. Certifying the security of the researchers' systems and security protocols could further delay access to the data.

In this chapter we present AlacriTEE, a framework for building highly-available, secure, accountable, and fully-decentralized FaaS services. AlacriTEE provides a two-way sandboxing scheme for client workloads that protects both the client and the host from each other. Client code is first compiled into WebAssembly (WASM) bytecode [87], and executed within the WebAssembly Micro Runtime (WAMR) [38] that compartmentalizes the client workloads to protect them from each other as well as the host. Additionally, the Wasm runtime is executed within a secure enclave, which provides a trusted execution environment (TEE) that protects client workloads from malicious hosts. AlacriTEE comes with a Wasm bytecode instrumenter for fine-grained resource accounting in client workloads. The instrumentation is done inside the enclave and is protected from both the host as well as client workloads, so the results can be trusted by both parties. Our two-way sandboxing scheme together with the Wasm bytecode instrumenter enables hosts to provide a secure execution environment for client workloads, and to keep track of the amount of computational resources used by the workloads. Unlike previous works that instruments Wasm bytecode and track resource usage using a static weight map for instructions, our instrumenter introduces a dynamic weight calculation that calculates the weight based on the type of workload.

AlacriTEE provides a smart contract framework for establishing service contracts between clients and hosts, and it supports on-chain TEE authentication. To the best of our knowledge, AlacriTEE is the first serverless computing system that can perform on-chain remote attes-

tation (RA) of TEEs using EVM smart contracts. This allows anyone with a machine that has a valid TEE hardware to become a possible host, and clients can trust this host to securely execute their workloads via on-chain TEE authentication and off-chain TLS handshake for computation requests. Supporting on-chain TEE authentication provides AlacriTEE with a high availability guarantee, because hosts are incentivized to become providers, form contracts with clients and to correctly execute their workloads as long as they are properly compensated for the resources they use. Since workloads can be of various sizes, payments are process in the way similar to off-chain payment channels, where clients can pay for each successfully processed request instead of paying through an on-chain transaction each time, which can be expensive. Lastly, the AlacriTEE smart contract contains an automated dispute-resolution mechanism that is used by clients to claim possible SLA violations and by providers to challenge erroneous disputes.

AlacriTEE's reputation system tracks the service history of its providers and the usage history of its clients. Both clients and providers can make better decisions on who to form contracts with based on the reputation score of the other party. This reputation system encourages both clients and providers to act honestly in the long run, since participants with higher reputation scores are more likely to form contracts. Importantly, AlacriTEE only relies on reputation scores as a heuristic to help select potentially well-behaved participants: acquiring a good reputation cannot be used to violate security guarantees.

In summary, this chapter makes the following contributions:

- We present the design, implementation, and evaluation of AlacriTEE, the first fully-decentralized FaaS system.

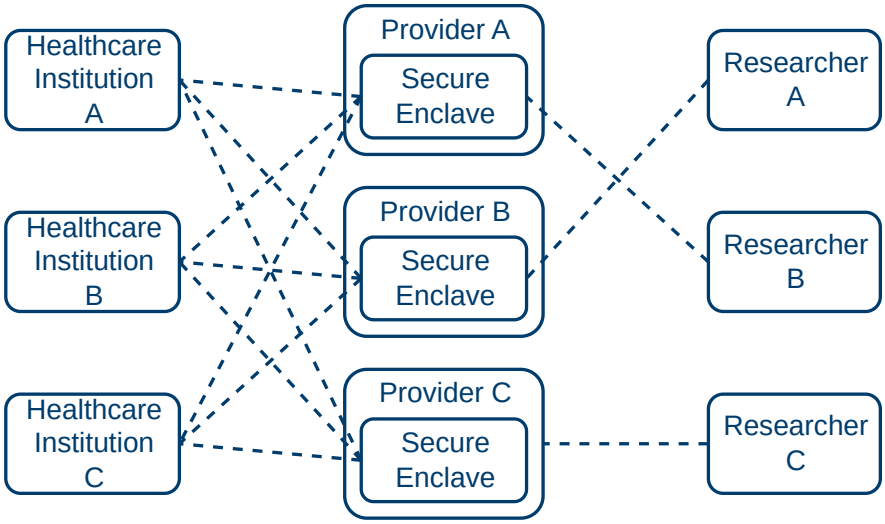


Figure 6.1: A researcher can ask a provider to execute a workload which will retrieve health records from multiple sources to perform data analysis, and anonymize them before returning the results.

- AlacriTEE provides a two-way sandboxed execution environment using TEEs and a Wasm runtime for hosts to securely execute client workloads, and a Wasm bytecode instrumenter for trusted fine-grained resource accounting.

- AlacriTEE's smart contract framework can authenticate TEE instances on-chain, allowing any host to become a provider as long as it has a valid TEE instance that can authenticate itself to the smart contract, making the service highly available and resilient to Byzantine failures.

6.2 Motivating Example: Anonymization for Healthcare Data

Figure 6.1 describes a decentralized healthcare data application based on AlacriTEE. In this system, parties need only accept the trust assumptions of the public blockchain and the secure enclave platform–neither the researchers nor the healthcare institutions need to trust anyone else. The only additional trust assumption for healthcare providers is that the code running inside the secure enclave protects the confidentiality of their contributed data, and the only additional assumption for researchers is that this code protects the integrity of the results (i.e., validates the authenticity of the inputs and correctly calculates the aggregate results). By auditing the code, entities can verify its trustworthiness to justify this assumption for a particular version of the code.[2] Using remote attestation, they can verify that version is being executed before forming service contracts that provide the enclave with access to confidential data and generate the resulting outputs.

For our example, the code might contain functions that provide query access to patient records based on specific (possibly restricted) parameters such as date range, vaccine type, patient pre-existing conditions, and any treatment outcomes from a side effect. The code is instrumented to run inside the provider's enclave, after which the researchers can begin sending requests to conduct their study.

When the researchers send a query for, say, patients treated for symptoms relating to the MMR vaccine, the enclave instance will fetch that data from the different healthcare institutions. These institutions need not anonymize the patient data before sending them, only

[2]Flawed auditing mechanisms or bugs in the compilation from source to binary could still could still introduce vulnerabilities, but each level of auditing shifts trust from application-specific code to more general, infrastructual code such as the compiler or code verifier.

sign and encrypt them to the enclave so that the provider cannot access or tamper with the data. Once the data is received by the enclave, it collects the desired statistics and performs any application-specific anonymization of the workload output that is necessary before returning the results to the researchers. The researchers can continue issuing requests to the provider until they have obtained all the necessary information that they need to complete their study.

6.3 Related Work

Blockchain-Based FaaS. Prior works on blockchain-based FaaS can be categorized into two groups: systems that integrates with a permissioned (private) blockchain and systems that integrates with a permissionless (public) blockchain. ChainFaaS [77] is a permissioned blockchain-based FaaS system in which users can go through a serverless controller to register as a client or a resource provider. When clients submit a request, the controller will wrap the function corresponding to the request in a docker container and assign it to a provider that can meet the resource requirements. The controller then records the job to the blockchain, which will also be used to transfer funds to the provider when it completes the job and sends the completion time to the blockchain. Arutyunyan et al. [13] propose a stateful serverless computing system, the Internet Computer (IC), where user code is compiled into Wasm modules called canisters and executed on the host machine. Each host is assigned to a subset of nodes in the IC peer-to-peer network of nodes that makes up the blockchain, and nodes in the subnet replicate the execution of the client workload to provide fault tolerance. Both ChainFaaS and IC rely on permissioned, leader-based BFT protocols [39, 43, 25] for replication, and thus provide limited availability

since traditional BFT protocols only scale up to the low hundreds of nodes [24].

Golem [79], iExec [91], and SONM [149] are decentralized computing services that integrates with Ethereum [172], a permissionless blockchain. These systems allow users to run their workloads on provider machines in Docker containers or virtual machines, and use the Ethereum blockchain as a medium to form contracts or facilitate payments. However, the use of these virtualization technologies as the only means to sandbox computation only protects the host from malicious workloads and does not provide the user with any confidentiality or integrity guarantees for their code and data. Additionally, iExec and Golem replicate each workload across multiple machines and rely on oracles to collect and verify these results to ensure correctness [160]. This approach not only requires redundant computation, but also introduces a new trusted entity and single point of failure.

Two-way Sandboxed Execution. Sandboxing applications and running them inside TEEs to protect clients from malicious hosts and hosts from malicious workloads has been introduced in recent works [12, 90, 81, 121, 134]. Scone [12] runs applications inside an SGX enclave which is contained inside a linux container, and provides C standard library for applications to make system calls from within the enclave. Compared to WebAssembly, containers do not provide the same level of portability and requires recompilation when moving between different platforms [122]. Ryoan [90] uses a user-level sandbox based on Google's Native Client (NaCl) [144] and hardware enclaves to provide the same protection. However, due to Wasm having wider adoption by the web community, NaCl support has been discontinued in favored of Wasm [83].

Se-Lambda [134] and AccTee [81] are two platforms that combines Wasm and TEEs for two-way sandboxed execution. These platforms use the V8 Javascript Engine as the runtime for executing Wasm bytecode inside the TEE, but V8 has a large TCB and has been the source of various security vulnerabilities [56]. Similar to AlacriTEE, Twine [121] uses the WAMR as a more portable Wasm runtime; however, it does not provide a mechanism for resource accounting. Most importantly, all of these works focus on providing a secure execution environment, and none addresses the problem of providing any availability guarantees for the user. AlacriTEE aims to provide not only a highly-available service, but one that also ensures confidentiality, integrity, and resource accounting for workload execution.

Trusted Resource Accounting. One approach to provide trusted resource accounting for a FaaS workload is by using TEEs to monitor the workload's compute time and memory consumption. T-Counter [59] instruments a client program, then runs the instrumented version outside the enclave and the uninstrumented version inside the enclave. When the instrumented version needs to increment the resource counter, it calls inside the enclave, which will check against a sequence table to determine if the host has tampered with the execution of the program. This approach does not provide confidentiality for client workloads, because the instrumented program runs outside the enclave. VeriCount [156] compiles client applications using a custom compiler that inserts an API that can interact with their resource accounting engine. However, the resource accounting engine uses the trusted time function provided by SGX that can be abused by a malicious provider to overcharge the client [4]. Recent work by Park et al. [129] proposes an extension of the SGX model to include a protected memory region where CPU and

memory usage logs can be written, but this limits instrumentation to only work on the SGX platform.

The idea of instrumenting Wasm bytecode was introduced by Wang et al. [164] to detect cryptomining activities in web browsers. Wasabi [112] built upon this idea by instrumenting Wasm in text format instead of binary format as part of a general-purpose framework for analyzing Wasm, and this approach was used by AccTee [81] which instruments WAT files inside TEEs to provide trusted resource accounting. Both AccTEE and Wasabi rely on WAT files for instrumentation, with their instrumenters containing not only hardcoded WAT instructions, but also comments automatically generated by a Wasm toolkit [169], so any changes to these instructions and comments will break their instrumenters, something that has happened multiple times in the past [166, 168, 167, 165]. Furthermore, AccTEE and Wasabi uses the V8 Javascript Engine as the Wasm runtime, which has a larger TCB and memory footprint than WAMR [32].

6.4 System and Threat Model

Blockchain. AlacriTEE relies on a permissionless blockchain to provide decentralized provider discovery, contract establishment, payments, dispute resolution, and reputation management. Any node can join the system as a client as long as they have the necessary funds to pay for their workloads, and as a provider if they have the resources to execute client workloads. Our implementation targets the Ethereum blockchain, and we assume that attackers do not control more than $\frac{1}{3}$ of the total network power (319k validators) to affect the availability

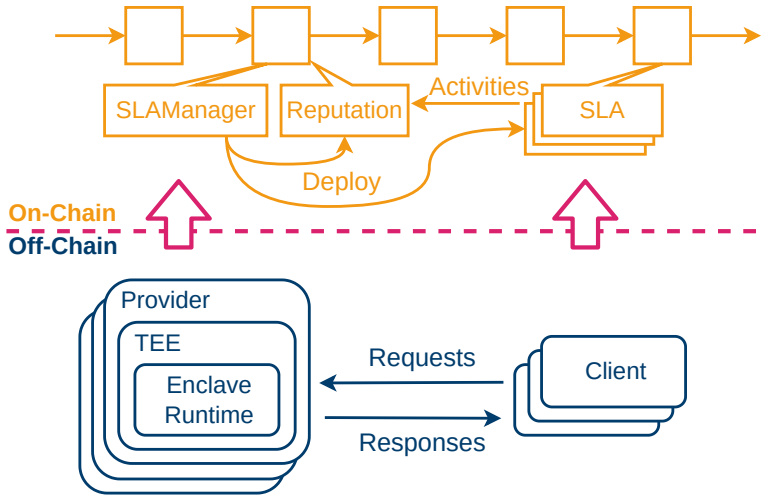


Figure 6.2: AlacriTEE Architecture

guarantee provided by the blockchain.

Trusted Execution Environments. Providers in AlacriTEE executes client workloads inside TEEs to preserve their confidentiality and integrity. We do not make any assumptions about TEEs other than the typical assumptions of TEE platforms, that is, an attacker may attain complete control of the host, but that the code and data inside the TEE are cryptographically protected even in the event of a physical compromise. In order for nodes to register themselves as providers in our platform, they must be running a valid TEE instance that can generate a self-attestation certificate [104, 183] that is verifiable on-chain to authenticate the TEE instance. Clients and providers are rational in that they want to have their workloads completed, and to be paid for the resources they use, respectively.

6.5 AlacriTEE Design

In this section, we describe AlacriTEE, a framework that enables highly-available, secure, accountable, and completely decentralized serverless computing. Our goal in AlacriTEE is to provide a serverless computing platform in which clients can request services from providers without the need for a trusted intermediary, to ensure confidentiality and integrity guarantees for client workloads that are executed on untrusted hosts, and to incentivize honest providers to actively participate in the network as well as disincentivize malicious providers who violate the terms of the service agreement.

The architecture for AlacriTEE is shown in Figure 6.2. Our framework utilizes a set of four smart contracts to facilitate service establishment, payment and dispute resolution, TEE authentication, and reputation management. AlacriTEE on-chain services consists of 3 smart contracts: the `SLAManager` contract, which is deployed once and works together as the initiator of all `SLA` contracts; the `Reputation` contract, which is deployed along with the `SLAManager` contract and it keeps track of the service history of providers and the usage history of clients; and the `SLA` (Service Level Agreement) contract, which represents and manages each individual service agreement. Service providers register themselves through the `SLAManager` contract, but before they can be considered as valid and active providers, the identity of their TEE hardware will be authenticated first, which will be discussed in Section 6.5.2. Once a client and a provider has agreed to the terms of a service contract, the `SLAManager` contract will spawn a new `SLA` contract, encoding the terms of the service agreement. The `SLA` contract handles payments that clients make for the resources that providers spend executing their workloads, and it is the

mediator for any service disputes that may arise between clients and providers. By enabling on-chain TEE authentication and incentivizing providers through payments, AlacriTEE attains a high degree of availability since any user with a machine that can run a valid TEE instance will be motivated to become a provider and get paid for their idle resources. As a provider forms contracts with more clients over time, its service history is encoded in the `Reputation` contract, which future clients can query to make better provider selections; similarly, the service provider can query the `Reputation` contract to decide whether they want to accept a client's proposal based on the client's usage history.

Providing security guarantees for client workloads requires that a provider's machine has a TEE instance that can create self-attestation certificates that can be validated on-chain using our smart contract framework. Within this TEE instance are two components: the **WebAssembly Micro Runtime (WAMR)** and the **Instrumenter**. WAMR is a lightweight runtime for executing Wasm bytecode. The Instrumenter will take the Wasm bytecode provided by the user, and insert the necessary instructions to accurately account for the amount of resources used, and produce the instrumented Wasm bytecode that will be executed by WAMR.

6.5.1 Service Establishment

In order for a client and a provider to establish a service contract, the provider must first make itself known to the network, and the client must be able to find the provider. As shown in Figure 6.3, providers and clients register themselves with the `SLAManager` contract. During the provider registration process, the enclave runtime will generate a self-attestation certificate that will be passed to the `SLAManager`, along with the rate at which the provider

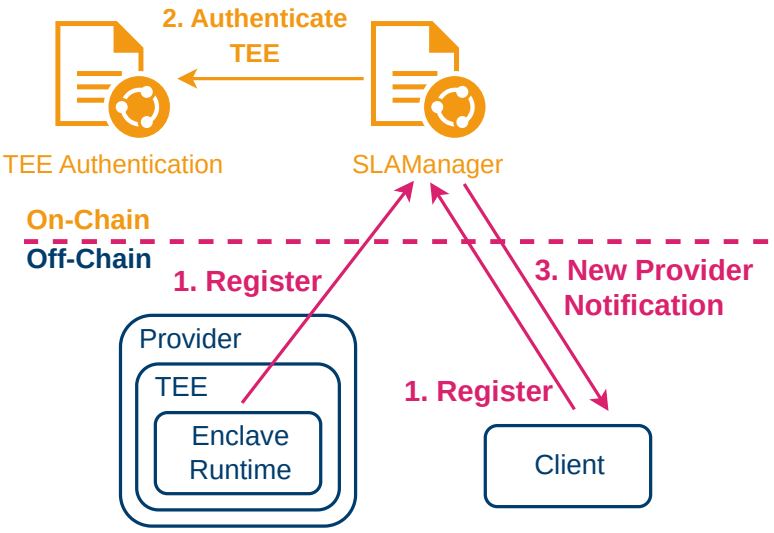


Figure 6.3: Client and provider Registration.

charges per compute unit used. The self-attestation certificate will then be forwarded to the TEE authentication smart contract to ensure that the enclave runtime is running on a genuine TEE hardware platform, for which we will discuss in Section 6.5.2. After the enclave runtime is authenticated, its public key and hardware ID will be extracted from the certificate, stored in the `SLAManager` contract, and an event containing the provider's information will be emitted to inform clients that a new provider is now available.

Clients monitor these registration events and collect a list of active providers, they can choose the provider that best suits their requirements. After a client has selected a potential provider, it will send a proposal to the `SLAManager` contract by calling the `propose` function, including in its argument the provider's hardware ID and a deposit for the amount of resources that the client expects to use, as shown in Figure 6.4. The `SLAManager` will first verify that the provider is active in the network before emitting an event detailing the proposal made from the

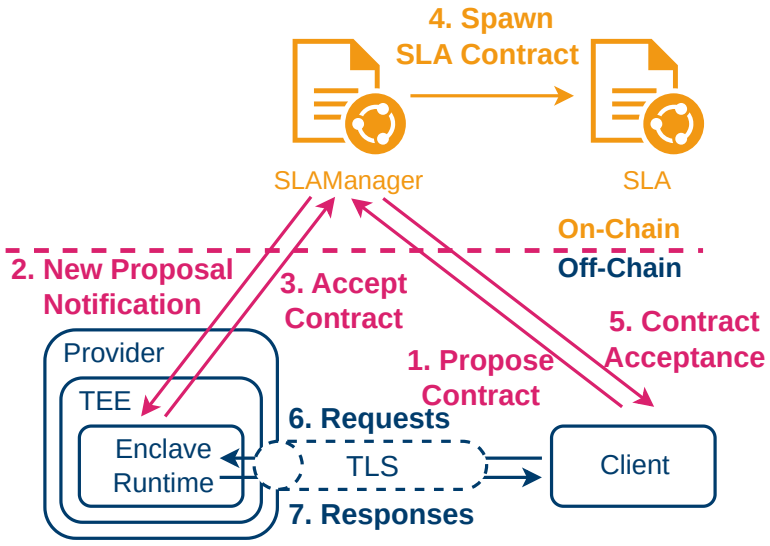


Figure 6.4: Client and provider Form a SLA contract.

client to this provider. Providers listen for these events, and can choose to accept the client's proposal by calling the `acceptProposal` function in the `SLAManager` contract, passing in a message encrypted using the client's public key that contains the IP address and port number of the provider's machine. The `SLAManager` contract will then spawn a new `SLA` contract that will encode the terms of the service agreement between the client and the provider, after which it will emit an event detailing the acceptance for the client to listen to.

We note that although the process to establish a service contract involves multiple on-chain interactions, it is completely decentralized unlike existing decentralized FaaS systems that rely on a trusted intermediary to perform the same task. For example, in Golem [79], clients and providers must go through the `Application Registry`, a service that is provided and maintained by the Golem team, in order for providers to register themselves to the network and for clients to register their workload and find providers. This reliance on a centralized service is

not ideal, because if the service becomes unavailable, then the entire network grinds to a halt and no new service contracts can be established until service is restored. In contrast, AlacriTEE does not suffer from this problem as clients and providers can continue forming service contracts as long as the underlying blockchain network remains functional. Furthermore, the location of the provider machine is protected and only known to its clients since the message containing the IP address and port number of the host machine, though sent on-chain, is encrypted using the client's public key. The details of the client's workload are also protected, since the workload is sent through an off-chain communication channel after contract establishment, as opposed to being made public, as is done in Golem's `Application Registry`.

6.5.2 Enclave Authentication

Off-chain applications usually rely on Remote Attestation (RA) protocol to authenticate the identity of a remote enclave. This ensures that the enclave is running on a genuine hardware platform and is running the desired enclave code. Existing works [104, 183] have proposed using enclave program to perform RA protocol to itself to generate a self-attestation report that can be verified by a remote party independently. By embedding the self-attestation report in the X.509 certificate, the enclave can establish TLS connection with other entity. AlacriTEE uses the same X.509 certificates to authenticate the enclave instance running inside the provider's machine. The smart contract is responsible for parsing the certificate, verifying the self-attestation report contained inside, and extracting the public key from the certificate. This approach of on-chain enclave authentication is similar to one used in Decentagram [187] to enable trusted on-chain event publication in pub/sub systems. In addition, AlacriTEE enclave

runtime will also put a hardware ID in the certificate, which is a unique identifier for the hardware platform that the enclave is running on. AlacriTEE on-chain services use the hardware ID to identify the provider so that a provider with bad reputation cannot simply create a new identity to continue providing services.

6.5.3 Two-way Sandboxed Execution

In a decentralized system where goods and services are exchanged, there is an inherent distrust between all parties involved. This is especially true for public blockchain-based services where peers have a high degree of anonymity and incentives can be in the form of currencies with real monetary value. Unlike centralized systems such as AWS Lambda in which clients can trust that the service provider will not tamper with their workloads or try to cheat them out of their money by manipulating resource usage, such a trust assumption cannot be made in a system like AlacriTEE. Conversely, service providers in AlacriTEE cannot assume that clients are honest and will not send them malicious workloads that can compromise their machine or workloads from other clients [109]. Therefore, two major problems that decentralized serverless computing platforms must address are: protecting client workloads from untrusted providers, and protecting providers from potentially malicious workloads from untrusted clients.

AlacriTEE addresses these problems by providing a two-way sandboxed execution environment for client workloads using Wasm and TEEs. We build upon the work of Ménétrey et al. [121] that utilizes Intel SGX enclaves to provide a tamper-proof execution environment that shields the client's workload from unauthorized access by the provider, and using WAMR

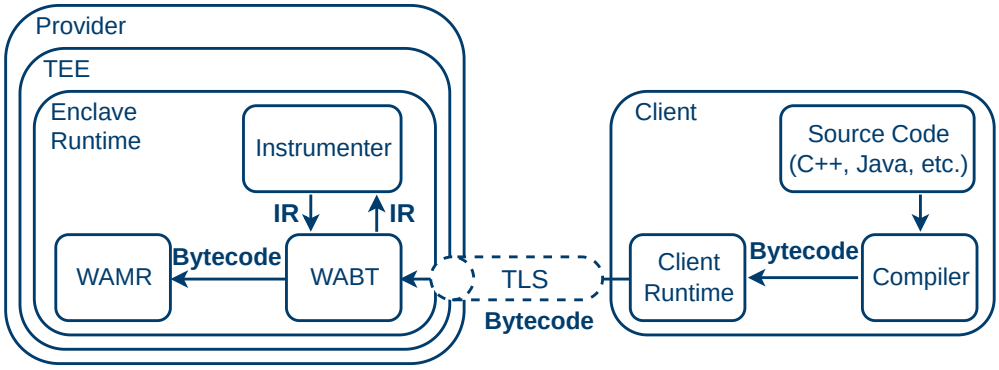


Figure 6.5: Client compiles the source code into Wasm bytecode, which is sent to the enclave runtime for instrumentation.

as the Wasm runtime for executing Wasm bytecode inside the enclave which will compartmentalize the workload to prevent it from access outside of its provisioned memory space.

As shown in Figure 6.5, A client develops their workloads in the language of their choice, then compiles the workload into Wasm bytecode using a compiler with Wasm backend support (e.g., LLVM compiler toolchain). After the client has formed a service contract with a provider, the client can use the information and credentials collected during the process to establish a TLS connection with the enclave runtime running on provider's machine. The client then sends the Wasm bytecode to the enclave runtime, which will pass the bytecode to our bytecode instrumenter for resource accounting. The instrumenter will insert the necessary instructions to account for the amount of resources used by the workload, and produce the instrumented Wasm bytecode. The instrumented Wasm bytecode is then passed to the WAMR runtime contained inside of AlacriTEE enclave runtime for execution.

WAMR provides several different execution modes for Wasm bytecode, including interpreted, Ahead-of-Time (AoT), and Just-in-Time (JIT). In order to provide near-native perfor-

mance for client workloads, either AoT or JIT execution mode must be used. We ported the JIT compiler provided by WAMR to run inside the enclave, so that the Wasm bytecode will be optimized during runtime. This approach is different than the one taken by TWINE, which instead relies on AoT execution [121]. Our reasoning behind embedding a JIT compiler is twofold. First, we would like to simplify the steps a client takes in establishing a service contract with a provider. The client does not know the exact hardware specifications of the provider's machine, so it would be challenging for them to produce a bytecode that is optimal for the targeted environment, even if they have access to a AoT compiler. Second, if the AlacriTEE's instrumenter receives a AoT-compiled Wasm program, it would be challenging to insert the necessary instructions for resource accounting, since the instrumentation has to be done on the native code level (AoT-compiled Wasm program is in native code) and we would have to provide different instrumenters for different CPU architectures. We will show in the evaluation section that the JIT compiler can provide near-native performance for client workloads, thus, there is no benefit in using AoT compilation over JIT compilation.

6.5.4 Trusted Resource Accounting

Resource accounting is a crucial aspect of any FaaS system, and in a decentralized setting, it must be done in a way that both clients and providers can trust that the results are accurate and fair. In addition to providing fast, portable, and sandboxed execution, another benefit of running client workloads as Wasm bytecode is that it enables fine-grained resource accounting at the instruction level. The general idea in instrumenting Wasm bytecode is to introduce a counter that increments each time a Wasm instruction (e.g., `load`, `store`) is executed, and pay-

ing specific attention to any branches in control flow that can affect the number of instructions executed. Additionally, a Wasm bytecode instrumenter has to also take into consideration the different types of instructions and their weights, as some instructions are more computationally expensive than others. The result of the instrumentation is a Wasm bytecode that is functionally equivalent to the original bytecode, but with the counter correctly reflecting the total amount of resources that was used once execution is complete. When done inside the enclave, the integrity of the instrumentation process and the resulting accounting of a workload's resource usage can be trusted.

Instrumentation using Wasm IR. Currently, the state-of-the-art approach to instrumenting Wasm bytecode is to rely on the WebAssembly Text Format (WAT) representation of the Wasm bytecode [81, 112]. Compared to the bytecode representation, the WAT format is designed to help developers understand the Wasm bytecode, and is more human-readable. However, one drawback to this approach is that the WAT format is constantly evolving, with several major changes done in the past few years to its naming conventions for types and instructions to improve readability and consistency [166, 168]. This in turns affects all Wasm-to-WAT translator tools as different Wasm bytecode could have different WAT representation with each new iteration on the instruction format. Even worse, existing WAT instrumenter [81], instead of using tokenizers and parsers to convert the WAT format to internal representation (IR) and manipulate the IR, directly works on the text string of the WAT and heavily relies on the comments and space indentation generated by the Wasm toolkit [169]. Thus, minor changes to autogenerated inline comments or space indentation will not break the semantics of the WAT file, but will break the instrumenter.

AlacriTEE takes a different approach to instrumentation, which is shown in Figure 6.5. Instead of asking the client to compile the workload to Wasm bytecode and then convert it to WAT format using a specific version of the Wasm toolkit, our instrumenter directly works on the Wasm bytecode. AlacriTEE uses WABT as a library to load the Wasm bytecode into an IR, and then we analyze the IR to insert the necessary instructions for resource accounting. There are two main benefits to this approach. First, the amount of steps needed to arrive at the instrumented bytecode is greatly reduced, as there are no translations needed to convert the bytecode to its WAT representation and back. Second, and more importantly, the IR format is stable since it is strictly following the Wasm specifications, so that the conversion to the AST remains functional as long as there are no breaking changes to the Wasm specifications. Therefore, unlike current approaches that rely on the WAT format, our instrumenter is more robust to changes in the Wasm ecosystem.

Halting Resource Counter. Another difference in our approach towards instrumentation that makes AlacriTEE better-suited as a decentralized FaaS service is the addition of a halting mechanism that will stop the execution of a workload once a resource limit is reached. For prior works such as AccTee [81], instrumentation results in a counter and a method to retrieve the counter's value after execution. This means that a provider will have to request payments from the client once they have determined the amount of resources used by retrieving this counter. In an environment where the client is untrusted and identities are anonymized, the provider does not have a means to enforce payment from the client outside of the deposit. A malicious client can even have the provider execute a resource-intensive workload of which the cost is well above the deposit the client have made. Or, a client may give a workload consisting

of an infinite loop, which will cause the provider to execute the workload indefinitely. Such a situation cannot happen in AlacriTEE, where clients are required to make a deposit up front, and this deposit is used to pay for resources as their workloads are executed. AlacriTEE enclave runtime will calculate the resource units available to each request based on the deposit of the client and the rate of the contract. The amount of units will then be set as the threshold for the resource counter. During the execution of the workload, once the resource counter reaches the threshold, the enclave will halt the execution and respond to the client with an error message. The halting mechanism not only protect the provider from potentially malicious clients, as discussed above, but also ensures that providers are properly compensated for their resources. However, a halted workload does not necessarily mean that the client is malicious, especially if the contract is long-lasting, and the client may simply need to top up their balance to extend their contract. We discuss the details of payments and how clients can address this issue in the next section.

Dynamic Instruction Weight Mapping. The purpose of code instrumentation is to ensure that clients are billed accurately for the resources that providers spent on executing workloads. Wasm instructions vary in complexities, with some instructions requiring much more computational resources than others, so billing accuracy depends on whether these instructions are weighted correctly. Prior works [81] simply uses a static weight map that maps a Wasm opcode to a fixed weight. Since AlacriTEE instruments the Wasm bytecode at the IR level, we can provide a dynamic weight calculation based on the instruction's context. For example, a native function call to `heavy_computation()` will result in a higher weight than `light_computation()`.

6.5.5 Request Execution and Client Response

Once the provider has instrumented the client's workload, the client can begin issuing requests to the provider to execute the workload. When a client sends a request to the provider, it will construct a message $\tau_{pc}(n, P)$, where n is a monotonically increasing nonce used as request ID, P is a list of parameters for the workload, and they are sent via τ_{pc}, a TLS connection established between the provider p and client c. The provider will then execute the workload with specified parameters P, before constructing a response message $\tau_{pc}(n, \varepsilon(k_n, R))$ to send to the client, where $\varepsilon(k_n, R)$ is the result, R, of the workload, encrypted using an ephemeral key k_n generated for request n. Upon receiving the response from the provider for the request, the client must send an acknowledgement message $\tau_{pc}(n, \texttt{ACK})$ to the provider to confirm that the encrypted results for the request have been received. After the provider receives the acknowledgement, it will then send the client the decryption key k_n so that the client can decrypt the results of the request.

The two-step request-response process above serves to mitigate potentially malicious behaviors from both clients and providers. An acknowledgement message from a client prevents it from denying that it has received the results, and a client withholding the acknowledgement will not receive the decryption key for the results, so it does not gain any advantage. Similarly, a client receiving a response with the encrypted results can be sure that the provider has executed the workload, so it can send the acknowledgement to the provider. It could also be the case that the provider refuses to send the decryption key k_n, preventing the client from decrypting the results, but we will show in Section 6.5.7 that the smart contract framework can be used to

resolve these situations.

6.5.6 Resource Compensation

Payments in blockchain-based applications can be facilitated through the use of the underlying cryptocurrency supported by the blockchain network. The inherent cost of transactions on these networks, however, means that it could be very cost-inefficient for AlacriTEE to support payments in the form of on-chain transactions for client requests. For example, the price that a client pays to a provider for executing a small workload could be much lower than the cost to send the blockchain transaction itself. Fortunately, various off-chain scaling solutions have been proposed in recent years to address the scalability issues, allowing users of blockchain-based applications to transact with one another at a significantly lower cost. One of the leading scaling solutions are payment channels [126], an instantiation of state channels where users can make micropayments between each other, and on-chain transactions are only needed to establish and close the channel.

AlacriTEE utilizes one-way payment channels to facilitate payments from clients to providers. During service establishment, the spawned `SLA` contract receives the client's deposit through the `SLAManager` contract, which it will record as the user's balance in the payment channel with the provider. After the provider finish executing the request n, the provider records the total amount of units μ_n used for n by checking the resource counter. Essentially, the client has now paid the provider for its resources, but the provider has not yet claimed the payment.

At every checkpoint interval, determined by the number of blocks in the network, a provider will submit a *Service Report* to the `SLA` contract. This checkpoint report serves two

purposes: to prove that the provider is still active, and to claim the payments for the requests that it has executed. In this service report, the provider will include μ_n for each request that has been completed since the last checkpoint. The blockchain transaction containing the service report will be directly signed by the AlacriTEE enclave runtime, which serves as a proof that the report is generated by the enclave. By computing $\mu_n \times$ rate, the `SLA` contract will know the amount of payment, P_n that the provider should receive for request n. The provider's on-chain balance will then correctly reflect the off-chain balance that the client has with the provider at the time the report was submitted. Either party can choose to close the service contract at any time to claim their on-chain balance by initiating a closing transaction, which will trigger the `SLA` contract to emit the corresponding event to notify both parties. A grace period is given to allow the provider to submit a final checkpoint report that includes any payments that it has not yet claimed, after which either party can send a transaction to finalize the contract closure and have the funds deposited back to their respective wallets.

6.5.7 Providing High Availability

AlacriTEE takes a different approach to providing high resilience to faulty providers. Instead of relying on a replication protocol and be bound by its fault threshold, our system inherits the much higher availability guarantee provided by the blockchain network. Recall that in AlacriTEE, any user can become a provider as long as their machine has a valid TEE instance that can authenticate itself to the smart contract. Hence, any number of providers can fail as long as there remains at least one active provider in the network with whom clients can enlist to execute their workloads. A client can, at any time, decide to stop using a particular

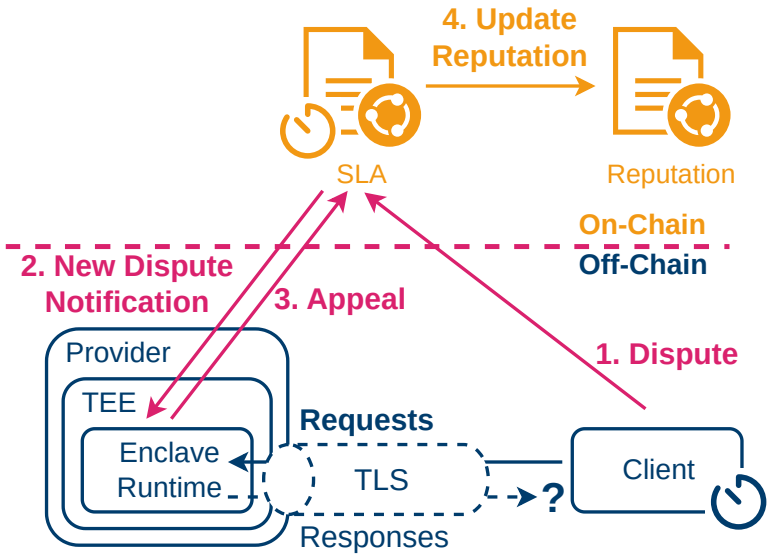


Figure 6.6: A service dispute between a client and provider.

provider if they have shown to be unreliable or unresponsive, close out the current contract that they have with that provider, and establish a new contract with a different provider. The process of reestablishing a service contract may not be the best option if service interruption is intermittent, so a client can first issue a service dispute with the `SLA` contract.

Figure 6.6 shows the process of a service dispute between a client and a provider. When a client sends a request to execute their workload, they will initiate a timer ΔT_n, which is the maximum amount of time that they expect the provider to complete the request n. If the client does not receive the encrypted result for request n by the time ΔT_n expires, they will send a $\texttt{dispute-noResp}_n$ to the `SLA` contract. If the client receives the encrypted result and replied with an ACK message, but does not receive the decryption key for the results by the time ΔT_n expires, they will send a $\texttt{dispute-noKey}_n$ to the `SLA` contract. The `SLA` contract validates these disputes to check that it has not already been initiated or resolved for the request n, to prevent

clients from requiring responses from providers for frivolous disputes. Once verified, the `SLA` contract will set its own timer in the number of blocks ΔB_n, and emit an event to notify both parties that a dispute has been initiated. A dispute will automatically trigger a refund back to the client for an amount that is half of the payment made to the provider for that request. The provider will have to respond to this dispute to claim this refund back (if the dispute is invalid), otherwise it will lose the refund and risk a decrease in its reputation score.

As mentioned in Section 6.5.5, the system must provide a way to mitigate potentially malicious behaviors from clients who may issue wrongful disputes and from providers who attempt to cheat their clients by not sending the results for requests and their corresponding decryption keys. The `SLA` contract provides three ways for providers to appeal to disputes: `appeal-rejection`, `appeal-withAck`, and `appeal-withKey`.

If a client initiates a dispute for a request which it had already received a rejection from the provider due to some error (e.g., insufficient balance), the AlacriTEE enclave runtime can respond by invoking the `appeal-rejection` function in the contract.

In cases where the client falsely claims that it had not received a response containing the encrypted results, after the client have acknowledged the result and received the decryption key, the enclave runtime can respond by invoking the `appeal-withAck` function, showing that the client did in fact acknowledge the receipt of the results.

It is also possible that the enclave runtime has finished executing the request, generated the encrypted result, and claimed the payment via checkpoint report, but the provider withholds the encrypted result from sending to the client by blocking the network traffic. Or, conversely, a malicious client receives the encrypted results but refuses to send an ACK message

to the provider, and then issues a dispute. In both of these cases, no one can win the dispute, so that the provider will lose half of the payment for the computation that has been done; and the client will still be charged for half of the payment for the computation but with a useless result (since it is encrypted). Since both sides lose in this scenario, it is in the best interest of both parties to cooperate and act honestly.

Finally, if the client claims that it has received the encrypted result, but has not received the decryption key for the results, the enclave runtime can respond by invoking the `appeal-withKey` function and include the decryption key. In the event that the provider is the one at fault (e.g., it has become unresponsive), it will not be able to respond to the dispute within ΔB_n. After ΔB_n has expired without the provider's response, the dispute will be considered unresolved, and the client will be able to claim a full refund for the request. Regardless of the outcome of a valid dispute, the `SLA` contract will call the `Reputation` contract, which will adjust the reputation scores of the client and/or provider depending on the result of the dispute.

It is important to note that since these appeal function calls to the `SLA` contract are made and signed by the AlacriTEE enclave runtime directly, we can trust the integrity of these appeals; and it is also in the interest of the provider host to properly forward the blockchain transactions containing these function calls to the network, because these transactions will help the provider to reclaim their payments.

We argue that this approach to relying on smart contracts to enforce SLAs is more effective than those used by centralized FaaS systems. For example, in AWS, service credits can be issued in the event the service fails to meet its service commitment, but only if the user submits a claim, and the credit takes one billing cycle to be applied to the user's account [6].

Additionally, service credits are only issued if the user's monthly bill exceeds one dollar, which could take at least one million requests for small workloads. The long duration to settle service disputes along with the minimum amount required to be considered for compensation is unappealing. In contrast, service disputes in AlacriTEE are settled on-chain within ΔB_n, which is a much shorter timeframe, and compensations to clients for successful disputes are done immediately.

6.5.8 Stronger Trust using Reputation

The notion of trust is crucial in a system where peers are anonymous and incentives drive participation [100]. Although our use of a two-way sandboxed execution environment prevents a client and a provider from harming one another, and our compensation mechanism ensures fairness between the two parties, the possibility of either party being malicious still exists. A client cannot be certain that a provider which has successfully executed its workload for a certain period of time will continue to do so for the remaining duration of their contract. Likewise, a provider cannot trust that a client who has not had any problems with the service provided will not suddenly start issuing frivolous disputes and force them to submit transactions to resolve them. In fact, it is impossible to guarantee these things in a decentralized system. However, we can provide confidence that they won't happen using a reputation system that builds upon experiences that clients and providers have with each other [99].

AlacriTEE's reputation system builds upon the work of CoopEdge [176, 177], where a provider's reputation reflects its history of service, and the reputation score is managed on chain by the `Reputation` contract. Unlike CoopEdge, however, our reputation calculation is

done entirely on-chain through the smart contract and does not require consensus among a set of peers before a reputation score can be updated. The use of a smart contract to perform the reputation allows AlacriTEE to retain the high availability guarantee provided by the blockchain network. The `Reputation` contract keeps track of the reputation of both clients and providers, so when a client forms a service contract with a provider that has a good reputation score can have high confidence that the provider will respond promptly to requests, and a provider accepts a service contract with a client that has a good reputation score can have high confidence that the client will not issue frivolous disputes.

In AlacriTEE, the reputation score is implemented as a unsigned 64-bit integer value, with a initial reputation score R_I of $10,000,000$ assigned to both clients R_c and providers R_p during their registration process. After that, there are three factors that can affect the reputation score of a provider or client.

Service Report. In Section 6.5.6, we discussed about the checkpoint report generated by enclave runtime will be sent to the `SLA` contract periodically. In addition to calculate payments, the information in this report is also used to calculate the amount of reputation to add to both the provider and the client using the following formula:

$$\omega = \frac{R_p + R_c}{2 * R_I} \tag{6.1}$$

$$\Delta R = \omega \times \sum_{i=1}^{n} P_i \tag{6.2}$$

$$R_p = R_p + \Delta R \tag{6.3}$$

$$R_c = R_c + \Delta R \tag{6.4}$$

This reputation score is used to calculate the weight (ω) value by adding the current reputation scores of the provider (R_p) and the client (R_c), then dividing by twice the initial reputation value R_I. The purpose of the weight is to control the rate at which reputation is added to both parties while benefiting those with higher reputation scores. For each request i in the provider's checkpoint report, its payment amount P_i is added to a sum that is multiplied with the weight to determine the reputation amount (ΔR) to add to both parties.

Service Disputes. A service dispute can signify potentially faulty behavior from either the provider or the client, as discussed in Section 6.5.7. We consider a dispute to be resolved if the provider submits a valid appeal to the `SLA` contract within the dispute period ΔB_n. In this case, the `Reputation` contract will decrease both parties' reputation scores using the value from the following formula:

$$\Delta R = \frac{max(R_p, R_c)}{R_I} \tag{6.5}$$

The maximum reputation score between the provider and the client is divided by the initial reputation score R_I. By using the maximum reputation score between either party, we dissuade clients and providers from behaviors that will lead to disputes. The reasoning behind deducting both parties' reputation for a resolved dispute is that either party can be at fault, since a faulty provider could have withheld the response to the client, or a malicious client could have issued a dispute for an already completed request. It is also important to note that the initial reputation score is at the 10 million level, so by dividing the current reputation score by the initial reputation score, the amount of reputation deducted for a resolved dispute is only a

small fraction of the current reputation score. Therefore, a malicious client who wants to harm a provider's reputation will have to issue a large number of frivolous disputes to have a significant impact on the provider's reputation score. Meanwhile, a dispute from a reputable client, or the appeal from a reputable provider, will have a larger impact on the reputation score; this follows the intuition that we usually believe the dispute (or appeal) from a reputable party is more likely to be legit.

For a successful dispute, one in which the provider does not respond within ΔB_n, the provider can be the one considered faulty, and the following reputation formulas are used:

$$\Delta R_{c+} = \frac{R_c}{R_I} \tag{6.6}$$

$$\Delta R_{p-} = \Delta R_{c+} \times \Psi \tag{6.7}$$

Since the client's dispute was successful, its reputation score will adjust upwards by an amount (ΔR_{c+}) proportional to its current reputation score. This amount is then multiplied by a penalty factor Ψ, to determine the amount (ΔR_{p-}) to deduct from the provider's reputation score. The penalty factor, which is 100 in default configuration, *magnifies* the amount of reputation that providers lose so that they are incentivized to provide better service to their clients and respond to disputes in a timely manner.

We note that although our reputation system improves on the existing one used by CoopEdge in that it is entirely done on chain, and that it addresses the possibilities of both malicious clients and providers, it may not be ideal for every decentralized system. Reputation management is an active area of research in blockchain-based systems, with various reputation

schemes proposed in recent works [86, 113, 162], and the choice of a reputation scheme to use will depend on the application. The main contribution of our reputation system is that it presents a way to encode a reputation scheme on-chain, without the need for a consensus protocol to update reputation scores, which can be adapted to other decentralized systems accordingly.

6.6 Evaluation

6.6.1 Instrumentation Overhead

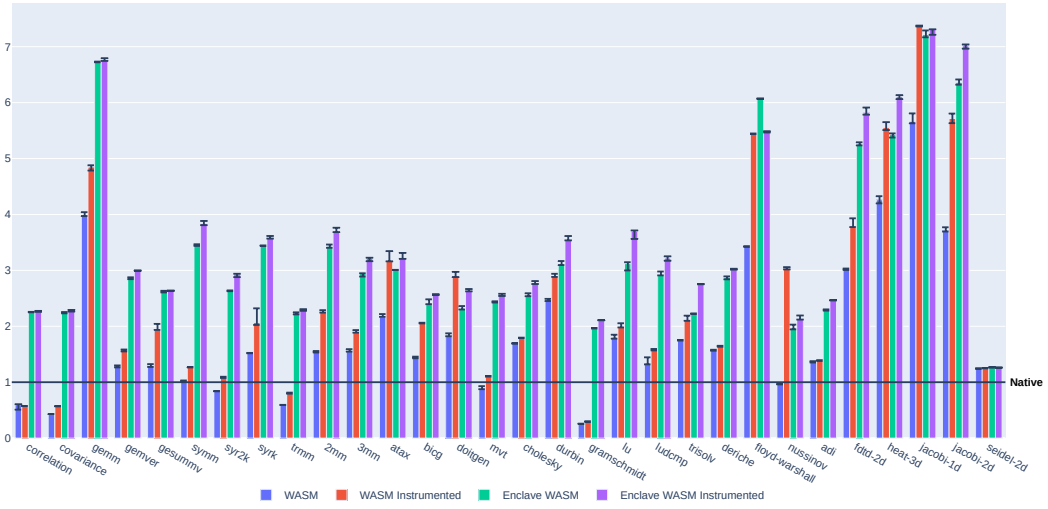


Figure 6.7: Runtime overhead of uninstrumented and instrumented Wasm code running outside and inside the enclave, normalized to native execution time for microbenchmarks in the Polybench benchmark suite.

In this first evaluation, we investigate the overhead of running instrumented Wasm code inside the enclave. We use the Polybench [132], a popular benchmark suite that con-

tains a variety of compute-intensive microbenchmarks written in C. Each benchmark is executed under five different configurations: *native x86-64* executable running outside the enclave, *uninstrumented Wasm* and *instrumented Wasm* bytecode running outside the enclave, and *uninstrumented Wasm* and *instrumented Wasm* bytecode running inside the enclave. We run each benchmark 5 times, where the first 2 times are used as warmup runs, and the measurements of the last 3 runs are recorded in the graph. Among these 3 runs, we take the median value as the bar height in the graph, and the minimum and maximum values as the error bars. This experiment is conducted on a PC with an i3-7100T CPU and 32GB of RAM running Ubuntu 22.04. The results of this evaluation are shown in Figure 6.7.

Benchmarks outside of enclave. For around half of the benchmarks, the Wasm bytecode running outside of the enclave has a performance nearly identical to the native executable. For around 25% of the benchmarks, the overhead of running the Wasm bytecode outside of the enclave is 2x to 3x slower than the native executable. And the remaining benchmarks have an overhead of 3x to 7x slower than the native executable. The performance of the instrumented Wasm bytecode is generally slower than the uninstrumented Wasm bytecode, but the overhead is mostly not significant. In 4 of the benchmarks, the Wasm bytecode is even faster than the native executable, which is likely due to the optimizations that the JIT compiler performs on the Wasm bytecode; such a phenomenon is also observed in other works [81].

Benchmarks inside of enclave. 9 out of the 30 benchmarks have a performance that is close to the ones running outside of the enclave. Meanwhile, around 50% of the benchmarks have an overhead of 2x to 3x slower than the non-enclave benchmarks. The remaining 5 benchmarks

Num Requests	Gas Used	Amortized Cost per Request
10	132108	13210
100	219170	2191
200	314416	1572
500	607160	1214
1000	1086477	1086
5000	4943971	988
10000	9791820	979

Table 6.1: Gas cost for sending a checkpoint report for different requests.

have an overhead of 3x to 4x slower than the non-enclave benchmarks. And similar to the benchmarks running outside of the enclave, the instrumented Wasm bytecode has a performance that is generally slower than the uninstrumented, but the overhead is not significant.

6.6.2 Checkpoint Report Gas Cost

Our next evaluation looks into the on-chain cost for a provider submitting checkpoint reports. As mentioned in Section 6.5.6, AlacriTEE enclave runtime will periodically generate these reports and send them to the `SLA` contract to claim payments. Blocks in a blockchain network have a size limit, which limits the amount of requests that can be included in a single checkpoint report due to the space that each request takes up. We want to determine the optimal number of requests that a provider should include in a checkpoint report to minimize the cost of sending the reports. Table 6.1 shows the transaction gas cost for sending the report and the

corresponding cost per request for various request batch size. For checkpoint report with just ten requests, the cost per request is relatively high at 13210 gas. The reason for this high cost is that in addition to processing each request in the report, the smart contract will also perform a cross-contract call to notify the `Reputation` contract for it to update the reputation scores of the two parties and it will also emit an event to notify the client of the report submission, so there is an overhead associated with each report regardless of the number of requests in them. As the number of requests in the report increases, the average cost per request significantly decreases, with the cost per request for a report containing 5k requests being just 988 gas, which is an 93% reduction. However, this reduction of 93% would seem to be the limit since the cost per request remains the same for a report with 15k requests. Therefore, for each checkpoint report, a provider with a light load should aim for a batch size of at least a few hundred requests, and a provider with a heavy load should aim for a batch size of at least 1k requests to minimize the cost of sending the report.

Next, we will use the average cost per request to determine the maximum number of requests that a provider can include in a checkpoint report given the block size limit. Our evaluation is done on the Ethereum blockchain, which currently has a block size limit of 30 million gas.[3] Since our results show that the cost per request does not significantly decreases after 10k requests, a limit of 30 million gas would allow a provider to include almost 48k requests in a single checkpoint report, which would be the throughput limit for a single provider in our system. However, this throughput is dependent on the checkpoint interval, so a shorter interval will allow a contract to process more requests on-chain. For example, a checkpoint

[3]Though this limit can be adjusted by miners, we do not expect it to be significantly higher or lower than the current limit.

interval of 10 blocks, and a block time of 12 seconds per block means that the contract can process around 400 requests per second. For blockchain networks with faster block times, such as Polygon, which has a block time of 2 seconds, the throughput can be increased to 2.4k requests per second.

The throughput of our system relies on a purely on-chain solution for checkpoint reports. By including the individual requests, the system provides a higher degree of auditability for the client as well as transparency for reputation management. The cost of publishing these reports can be reduced by leveraging off-chain scaling solutions such as roll-ups or state channels [126], or by deploying the contracts on a layer-2 blockchain network such as Polygon and Optimism. Additionally, we have the flexibility to delegate more of the on-chain computation logic to the enclave since the messages it sends can be authenticated, and condense the checkpoint reports to include only a summary for the requests, which would also reduce the gas cost. Furthermore, the cost of sending the reports will not be a concern for providers, because they can set the rate for their resources to account for the costs associated with any on-chain communication.

6.7 Conclusion

In this chapter, we introduced the design, implementation, and evaluation of AlacriTEE, a fully-decentralized platform for serverless computing. AlacriTEE leverages the security guarantees of TEEs and WebAssembly to provide a two-way sandboxed execution environment for client workloads that preserves confidentiality and integrity and protects both clients and

providers from malicious behaviors. Our system provides fine-grained trusted resource accounting inside the enclave using an instrumenter that works directly on Wasm IR and supports dynamic weight mapping for instructions, allowing providers to adjust costs based on resource demands and application type. AlacriTEE's smart contracts enables clients and providers to establish service contracts in a completely decentralized manner, and contains mechanisms for on-chain resource reimbursement, service disputes, and reputation management.

Chapter 7

Conclusion

7.1 The Decent Framework

Building secure distributed applications with a decentralized trust model presents significant challenges. In this dissertation, we introduced the Decent Framework, which integrates secure enclaves and blockchain technology to provide confidentiality, integrity, and availability in a decentralized trust environment. The integration of secure enclaves with blockchain is a complex task, and this dissertation addresses several key challenges in achieving this integration.

We designed the Decent Framework infrastructure, where Decent Components utilize Self-Attestation to cache verifiable attestation reports. This approach improves throughput by 7.5x and reduces latency by 3.67x compared to the native SGX attestation. By embedding an AuthList in the self-attestation certificate, Decent Components support mutual attestation without requiring a universally trusted entity. These enhancements make the Decent Framework

more efficient and scalable, particularly for short-lived serverless functions. Using ProVerif, we formally verified that the protocol in the Decent Framework ensures the secrecy and authenticity of application data.

To ensure that enclaves can verify the authenticity of blockchain blocks, we developed a novel algorithm that detects eclipse attacks from within enclaves by continuously monitoring blockchain difficulty levels. Statistical analysis shows that the algorithm's false negative rate is equivalent to the probability of brute-forcing a 128-bit key, and tests on Ethereum's historical data demonstrate that the false positive rate is negligible in practice.

Decentagram, our highly available publish/subscribe system, is built on secure enclaves and blockchain. By employing both on-chain and off-chain brokers, Decentagram achieves instant on-chain event delivery and timely off-chain event delivery. To enable smart contracts to authenticate messages from secure enclaves, we implemented the RA protocol within smart contracts, verifying attestation reports and signatures from enclaves and hardware manufacturers via the self-attestation certificate. Decentagram supports three revocation mechanisms: revocation by votes, by conflicting messages, and by leaked keys. Our evaluation shows that the gas costs for blockchain transactions are reasonable, and the off-chain broker processes new blocks faster than they arrive.

For service availability within the Decent Framework, we introduced AlacriTEE, a fully decentralized Function-as-a-Service (FaaS) platform that leverages secure enclaves and blockchain. AlacriTEE provides a two-way sandboxed execution environment, ensuring the integrity of code execution and the confidentiality of user data while protecting the host system from compromise. The service provider and host enter a Service Level Agreement (SLA)

via a smart contract, where AlacriTEE's WebAssembly (WASM) code instrumentation adds in resource accounting, ensuring the host is compensated. If the host fails to meet the SLA, it is penalized via the smart contract. Our evaluation demonstrates that the overhead of running instrumented WASM code within the enclave is minimal, and the associated gas costs for blockchain transactions are reasonable.

These contributions enable the Decent Framework to deliver all three essential guarantees of information security — confidentiality, integrity, and availability — for distributed applications operating within a decentralized trust model.

7.2 Code Availability

The implementation of the Decent Framework is open source and publicly accessible at: `https://github.com/lsd-ucsc/decent-framework`.

7.3 Future Work

7.3.1 Expanding Implementation Support to More Enclave Platforms

In this dissertation, our prototype implementations primarily use Intel SGX with EPID Remote Attestation (RA). However, the Decent Framework is designed to be platform-agnostic, and we aim to encourage the use of various secure enclave platforms to support our decentralized trust model. As of this writing, several enclave platforms are readily available, including Intel SGX, Intel TDX, AMD SEV, Apple Enclave, and ARM TrustZone, each with different underlying implementations. For example, Intel SGX is application-based, where the

code within the enclave is built as a single executable program. In contrast, AMD SEV and Intel TDX are Virtual Machine (VM) based, meaning that the entire VM runs inside the enclave, with applications protected within the VM.

Each implementation has its own advantages and drawbacks. Application-based enclaves, such as Intel SGX, offer a smaller Trusted Computing Base (TCB), which reduces the attack surface but requires greater effort to port existing applications into the enclave. VM-based enclaves, such as AMD SEV and Intel TDX, simplify the process of porting existing applications but have a much larger TCB, from the BIOS layer to the application layer. Despite these differences, all these platforms share the fundamental concept depicted in Figure 1.1: code and data are cryptographically protected in memory, and the CPU performs on-the-fly encryption and decryption. The Decent Framework is built on this same concept, making it feasible to support multiple platforms with the necessary implementations.

Additionally, there are multiple RA protocol implementations. Early Intel SGX versions relied on the Enhanced Privacy ID (EPID) protocol, which offered privacy and anonymity for end-users, but incurred high overhead. More recent versions use the Data Center Attestation Primitives (DCAP) protocol, which sacrifices privacy for reduced overhead, making it more suitable for data centers. AMD SEV uses a simple RA protocol that involves verifying a certificate chain and signature. While our current work is based on EPID, Intel has announced End-of-Life (EOL) support for EPID in April 2025. Regardless, all these protocols follow the fundamental model shown in Figure 2.3: the enclave hardware generates an attestation report, signed by a private key certified by the hardware manufacturer. Since the Decent Framework is built on this core concept, it can potentially support a wide range of RA protocols with appro-

priate implementations.

By adding support for more secure enclave platforms and their respective RA protocols, we can further advance the Decent Framework's goal of providing a decentralized trust model for distributed applications.

7.3.2 Dynamic Enclave Program Authorization

In Chapter 3, we introduced the AuthList, a list of authorized Decent Components. Only components listed in the AuthList can join the system with the specified responsibilities. In Chapter 5, we presented 3 automated mechanisms to revoke vulnerable components, enabling dynamic removal of components when they become compromised. However, the AuthList itself remains static, which prevents the addition of new Decent Components without causing system interruptions.

By porting a compiler and a formal Information Flow Control (IFC) verifier into the enclave, we could dynamically verify the code and data flow of new Decent Components and authorize them to join the system without interruption. Existing works [80, 127, 116, 9] have proposed static IFC verification for high-level languages. An enclave-based IFC verifier could load the source code of a new component, verify its IFC properties, and compile it into WASM bytecode for execution in the AlacriTEE runtime. The verifier enclave would also issue certificates for the new component, enabling it to authenticate via the RA protocol.

Several challenges arise in this approach. First, porting a compiler into the enclave is non-trivial. Many compilers have large codebases and rely on system calls, which are restricted within enclaves. This limits the choice of compilers and, consequently, the programming lan-

guages that can be supported by the verifier enclave. Ideally, the verifier should generate WASM bytecode to be compatible with the AlacriTEE runtime. The limited selection of compilers with WASM backends further constrains the available IFC verifiers. These limitations make the task of porting a compiler into the enclave particularly challenging.

Bibliography

[1] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance, December 2017.

[2] Abu Shohel Ahmed and Tuomas Aura. Turning trust around: Smart contract-assisted public key infrastructure. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 104–111, New York, NY, USA, August 2018. Institute of Electrical and Electronics Engineers.

[3] Mustafa Al-Bassam. SCPKI: A smart contract-based PKI and identity system. In *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*, BCC '17, pages 35–40, New York, NY, USA, April 2017. Association for Computing Machinery.

[4] Fritz Alder, N. Asokan, Arseny Kurnikov, Andrew Paverd, and Michael Steiner. S-FaaS: Trustworthy and accountable function-as-a-service using intel SGX. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, CCSW '19, pages 185–199, New York, NY, USA, November 2019. Association for Computing Machinery.

[5] Maher Alharby and Aad van Moorsel. Blockchain based smart contracts: A systematic mapping study. In *Fourth International Conference on Computer Science and Information Technology*, volume 7 of *CSIT '17*. Academy & Industry Research Collaboration Center (AIRCC), August 2017.

[6] Amazon. Amazon lambda service level agreement. `https://aws.amazon.com/lambda/sla/`.

[7] Amazon Web Services. Amazon SNS. `https://aws.amazon.com/sns/`, 2023.

[8] Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent R Scarlata. Innovative technology for CPU based attestation and sealing. Technical report, Intel Corporation, August 2013.

[9] Owen Arden, Michael D. George, Jed Liu, K. Vikram, Aslan Askarov, and Andrew C. Myers. Sharing mobile code securely with information flow control. In *2012 IEEE*

Symposium on Security and Privacy, IEEE S&P '12, pages 191–205, New York, NY, USA, May 2012. Institute of Electrical and Electronics Engineers.

[10] Frederik Armknecht, Colin Boyd, Christopher Carr, Kristian Gjøsteen, Angela Jäschke, Christian A. Reuter, and Martin Strand. A guide to fully homomorphic encryption. Cryptology ePrint Archive, Paper 2015/1192, December 2015.

[11] Sergei Arnautov, Andrey Brito, Pascal Felber, Christof Fetzer, Franz Gregor, Robert Krahn, Wojciech Ozga, André Martin, Valerio Schiavoni, Fábio Silva, Marcus Tenorio, and Nikolaus Thümmel. PubSub-SGX: Exploiting trusted execution environments for privacy-preserving publish/subscribe systems. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, SRDS '18, pages 123–132, New York, NY, USA, October 2018. Institute of Electrical and Electronics Engineers.

[12] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, pages 689–703, Berkeley, CA, USA, November 2016. USENIX Association.

[13] Maksym Arutyunyan, Andriy Berestovskyy, Adam Bratschi-Kaye, Ulan Degenbaev, Manu Drijvers, Islam El-Ashi, Stefan Kaestle, Roman Kashitsyn, Maciej Kot, Yvonne-Anne Pignolet, Rostislav Rumenov, Dimitris Sarlis, Alin Sinpalean, Alexandru Uta, Bogdan Warinschi, and Alexandra Zapuc. Decentralized and stateful serverless computing on the internet computer blockchain. In *2023 USENIX Annual Technical Conference*, USENIX ATC '23, pages 329–343, Berkeley, CA, USA, July 2023. USENIX Association.

[14] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 297–307, New York, NY, USA, October 2010. Association for Computing Machinery.

[15] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (SoK). In Matteo Maffei and Mark Ryan, editors, *Principles of Security and Trust*, POST '17, pages 164–186, Berlin, Heidelberg, April 2017. Springer Berlin Heidelberg.

[16] Abdullah Ayub Khan, Asif Ali Wagan, Asif Ali Laghari, Abdul Rehman Gilal, Izzatdin Abdul Aziz, and Bandeh Ali Talpur. BIoMT: A state-of-the-art consortium serverless network architecture for healthcare system using blockchain smart contracts. *IEEE Access*, 10:78887–78898, July 2022.

[17] Asaph Azaria, Ariel Ekblaw, Thiago Vieira, and Andrew Lippman. MedRec: Using blockchain for medical data access and permission management. In *2016 2nd Interna-*

tional Conference on Open and Big Data, OBD '16, pages 25–30, New York, NY, USA, August 2016. Institute of Electrical and Electronics Engineers.

[18] Microsoft Azure. Azure functions. `https://azure.microsoft.com/en-us/services/functions/`, 2019.

[19] Ataollah Fatahi Baarzi, George Kesidis, Carlee Joe-Wong, and Mohammad Shahrad. On merits and viability of multi-cloud serverless. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, pages 600–608, New York, NY, USA, 2021. Association for Computing Machinery.

[20] Jean Bacon, David M. Eyers, Jatinder Singh, and Peter R. Pietzuch. Access control in publish/subscribe systems. In *Proceedings of the Second International Conference on Distributed Event-Based Systems*, DEBS '08, pages 23–34, New York, NY, USA, July 2008. Association for Computing Machinery.

[21] Beaconscan. Statistics & charts, mainnet beacon chain (phase 0) ethereum 2.0 explorer. `https://beaconscan.com/statistics`, 2023.

[22] Jethro G. Beekman, John L. Manferdelli, and David Wagner. Attestation transparency: Building secure internet services for legacy clients. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 687–698, New York, NY, USA, May 2016. ACM.

[23] Iddo Bentov, Yan Ji, Fan Zhang, Lorenz Breidenbach, Philip Daian, and Ari Juels. Tesseract: Real-time cryptocurrency exchange using trusted hardware. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 1521–1538, New York, NY, USA, November 2019. Association for Computing Machinery.

[24] Christian Berger, Sadok Ben Toumia, and Hans P. Reiser. Does my BFT protocol implementation scale? In *Proceedings of the 3rd International Workshop on Distributed Infrastructure for the Common Good*, DICG '22, pages 19–24, New York, NY, USA, November 2022. Association for Computing Machinery.

[25] Alysson Bessani, João Sousa, and Eduardo E.P. Alchieri. State machine replication for the masses with BFT-SMART. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 355–362, New York, NY, USA, June 2014. Institute of Electrical and Electronics Engineers.

[26] Kenneth P. Birman. Replication and fault-tolerance in the ISIS system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, SOSP '85, pages 79–86, New York, NY, USA, December 1985. Association for Computing Machinery.

[27] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, January 1987.

[28] Bitcoin.org. Block height and forking, 2020.

[29] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and proverif. *Foundations and Trends® in Privacy and Security*, 1(1-2):1–135, 2016.

[30] Mic Bowman, Andrea Miele, Michael Steiner, and Bruno Vavala. Private data objects: an overview. Technical report, Intel Labs, November 2018.

[31] Lorenz Breidenbach, Christian Cachin, Alex Coventry, Ari Juels, and Andrew Miller. Chainlink off-chain reporting protocol. Technical report, Chainlink Labs, February 2021.

[32] Stefan Brenner and Rüdiger Kapitza. Trust more, serverless. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, pages 33–43, New York, NY, USA, May 2019. Association for Computing Machinery.

[33] Ernie Brickell and Jiangtao Li. Enhanced privacy id from bilinear pairing. *IACR Cryptology ePrint Archive*, 2009:95, March 2009.

[34] Jo Van Bulck, Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium*, USENIX Security 18, page 991–1008, Berkeley, CA, USA, August 2018. USENIX Association.

[35] Vitalik Buterin. A next generation smart contract & decentralized application platform. Technical report, Ethereum Foundation, 2013. Ethereum White Paper.

[36] Vitalik Buterin. Critical update re: Dao vulnerability. `https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability/`, 2016.

[37] Vitalik Buterin. Hard fork completed. `https://blog.ethereum.org/2016/07/20/hard-fork-completed/`, 2016.

[38] Bytecode Alliance. Webassembly micro runtime (WAMR). `https://github.com/bytecodealliance/wasm-micro-runtime`, 2024.

[39] Jan Camenisch, Manu Drijvers, Timo Hanke, Yvonne-Anne Pignolet, Victor Shoup, and Dominic Williams. Internet computer consensus. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, PODC'22, pages 81–91, New York, NY, USA, July 2022. Association for Computing Machinery.

[40] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[41] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

[42] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. Secure routing for structured peer-to-peer overlay networks. *ACM SIGOPS Operating Systems Review*, 36(SI):299–314, December 2003.

[43] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, February 1999. USENIX Association.

[44] Chainlink. What is layer 2? `https://chain.link/education-hub/what-is-layer-2`, May 2023.

[45] Badrish Chandramouli and Jun Yang. End-to-end support for joins in large-scale publish/subscribe systems. *Proceedings of the VLDB Endowment*, 1(1):434–450, August 2008.

[46] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-sgx: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference*, USENIX ATC 17, pages 645–658, Berkeley, CA, USA, July 2017. USENIX Association.

[47] Guoxing Chen and Yinqian Zhang. MAGE: Mutual attestation for a group of enclaves without trusted third parties. In *31st USENIX Security Symposium*, USENIX Security '22, pages 4095–4110, Berkeley, CA, USA, August 2022. USENIX Association.

[48] Guoxing Chen, Yinqian Zhang, and Ten-Hwang Lai. Opera: Open remote attestation for intel's secure enclaves. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 2317–2331, New York, NY, USA, November 2019. Association for Computing Machinery.

[49] Wei Chen, Sam Toueg, and M.K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(5):561–580, May 2002.

[50] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 185–200. IEEE, 2019.

[51] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy*, EuroS&P '19, pages 185–200, New York, NY, USA, June 2019. Institute of Electrical and Electronics Engineers.

[52] Google Cloud. Cloud functions. `https://cloud.google.com/functions/`, 2019.

[53] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, June 2010. ACM.

[54] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile. RFC 5280 (Proposed Standard), May 2008. Updated by RFC 6818.

[55] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium*, USENIX Security 16, pages 857–874, Berkeley, CA, USA, August 2016. USENIX Association.

[56] CVEdetails. V8 security vulnerabilities. `https://www.cvedetails.com/vulnerability-list/vendor_id-1224/product_id-17734/Google-V8.html`.

[57] Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, pages 445–465, Berlin, Heidelberg, May 2010. Springer Berlin Heidelberg.

[58] Poulami Das, Lisa Eckey, Tommaso Frassetto, David Gens, Kristina Hostáková, Patrick Jauernig, Sebastian Faust, and Ahmad-Reza Sadeghi. FastKitten: Practical smart contracts on bitcoin. In *28th USENIX Security Symposium*, USENIX Security '19, pages 801–818, Berkeley, CA, USA, August 2019. USENIX Association.

[59] Chuntao Dong, Qingni Shen, Xuhua Ding, Daoqing Yu, Wu Luo, Pengfei Wu, and Zhonghai Wu. T-Counter: Trustworthy and efficient CPU resource measurement using SGX in the cloud. *IEEE Transactions on Dependable and Secure Computing*, 20(1):867–885, January 2022.

[60] John R Douceur. The sybil attack. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *International Workshop on Peer-to-Peer Systems*, pages 251–260, Berlin, Heidelberg, October 2002. Springer Berlin Heidelberg.

[61] John R Douceur. The sybil attack. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *International workshop on peer-to-peer systems*, pages 251–260, Berlin, Heidelberg, October 2002. Springer, Berlin, Heidelberg.

[62] Sisi Duan, Chao Liu, Xin Wang, Yusen Wu, Shuai Xu, Yelena Yesha, and Haibin Zhang. Intrusion-tolerant and confidentiality-preserving publish/subscribe messaging. In *2020 International Symposium on Reliable Distributed Systems (SRDS)*, pages 319–328, New York, NY, USA, September 2020. Institute of Electrical and Electronics Engineers.

[63] Open Enclave. Openenclave sdk. `https://openenclave.io`, 2019.

[64] Ayman Esmat, Martijn de Vos, Yashar Ghiassi-Farrokhfal, Peter Palensky, and Dick Epema. A novel decentralized platform for peer-to-peer energy trading market with blockchain technology. *Applied Energy*, 282:116123, January 2021.

[65] Ethereum Foundation. Go ethereum - official go implementation of the ethereum protocol. `https://github.com/ethereum/go-ethereum`, 2023.

[66] Ethereum Team. Go ethereum: Official go implementation of the ethereum protocol. https://geth.ethereum.org/.

[67] Ethereum Team. Release let there be light (v1.5.0) ethereum/go-ethereum. https://github.com/ethereum/go-ethereum/releases/tag/v1.5.0, 2016.

[68] Ethereum Team. Byzantium hf announcement. https://blog.ethereum.org/2017/10/12/byzantium-hf-announcement/, 2017.

[69] Ethereum Team. Release version 0.4.14 ethereum/solidity. https://github.com/ethereum/solidity/releases/tag/v0.4.14, 2017.

[70] Etherscan. Forked blocks. https://etherscan.io/blocks_forked/, 2023.

[71] eth.wiki. Rlp, 2020.

[72] Benjamin Eze, Craig Kuziemsky, Liam Peyton, Grant Middleton, and Alain Mouttham. Policy-based data integration for e-health monitoring processes in a B2B environment: Experiences from canada. *Journal of Theoretical and Applied Electronic Commerce Research*, 5(1):56–70, April 2010.

[73] Flexera. 2024 state of the cloud report, 2024. Accessed: 2024-10-04.

[74] Sivanarayana Gaddam, Ranjit Kumaresan, Srinivasan Raghuraman, and Rohit Sinha. LucidiTEE: Scalable policy-based multiparty computation with fairness. In Jing Deng, Vladimir Kolesnikov, and Alexander A. Schwarzmann, editors, *Cryptology and Network Security*, volume 14342, pages 343–367, Singapore, October 2023. Springer Nature Singapore.

[75] Sara Ghaemi, Hamzeh Khazaei, and Petr Musilek. ChainFaaS: An open blockchain-based serverless platform. *IEEE Access*, 8:131760–131778, July 2020.

[76] go-ethereum. consensus.go. https://github.com/ethereum/go-ethereum/blob/4b2ff1457ac28fb2894485194e0e344e84c2bcd7/consensus/ethash/consensus.go, 2020.

[77] Muhammed Golec, Deepraj Chowdhury, Shivam Jaglan, Sukhpal Singh Gill, and Steve Uhlig. AIBLOCK: Blockchain based lightweight framework for serverless computing using AI. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing*, CCGrid '22, pages 886–892, New York, NY, USA, May 2022. Institute of Electrical and Electronics Engineers.

[78] Muhammed Golec, Sukhpal Singh Gill, Mustafa Golec, Minxian Xu, Soumya K Ghosh, Salil S Kanhere, Omer Rana, and Steve Uhlig. Blockfaas: Blockchain-enabled serverless computing framework for ai-driven iot healthcare applications. *Journal of Grid Computing*, 21(4):63, November 2023.

[79] Golem. Golem network. https://www.golem.network/.

[80] Anitha Gollamudi, Stephen Chong, and Owen Arden. Information flow control for distributed trusted execution environments. In *2019 IEEE 32nd Computer Security Foundations Symposium*, CSF '19, pages 304–30414, New York, NY, USA, June 2019. Institute of Electrical and Electronics Engineers.

[81] David Goltzsche, Manuel Nieke, Thomas Knauth, and Rüdiger Kapitza. AccTEE: A WebAssembly-based two-way sandbox for trusted resource accounting. In *Proceedings of the 20th International Middleware Conference*, Middleware '19, pages 123–135, New York, NY, USA, December 2019. Association for Computing Machinery.

[82] Yanwei Gong, Xiaolin Chang, Jelena Mišić, Vojislav B. Mišić, Jianhua Wang, and Haoran Zhu. Practical solutions in fully homomorphic encryption: a survey analyzing existing acceleration methods. *Cybersecurity*, 7(1):5, March 2024.

[83] Google. Goodbye PNaCl, hello WebAssembly! `https://blog.chromium.org/2017/05/goodbye-pnacl-hello-webassembly.html`, May 2017.

[84] Google Cloud. What is Pub/Sub? `https://cloud.google.com/pubsub/docs/overview`, May 2023.

[85] Trusted Computing Group. `https://trustedcomputinggroup.org/`, 2023.

[86] Andreas Gruhler, Bruno Rodrigues, and Burkhard Stiller. A reputation scheme for a blockchain-based network cooperative defense. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management*, IM '19, pages 71–79, New York, NY, USA, April 2019. Institute of Electrical and Electronics Engineers.

[87] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '17, pages 185–200, New York, NY, USA, June 2017. Association for Computing Machinery.

[88] Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, and Kate Stout. Java message service. Technical report, Sun Microsystems, Santa Clara, CA, December 2002.

[89] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 129–144, Berkeley, CA, USA, August 2015. USENIX Association.

[90] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. *ACM Transactions on Computer Systems*, 35(4):13:1–13:32, December 2018.

[91] iExec. iexec. `https://iex.ec/`.

[92] Infura, A Consensys Formation. Infura: Secure and scalable access to ethereum apis and ipfs gateways.

[93] Hudson Jameson. Faq: Upcoming ethereum hard fork. https://blog.ethereum.org/2016/10/18/faq-upcoming-ethereum-hard-fork/, 2016.

[94] Hudson Jameson. Hard fork no. 4: Spurious dragon. https://blog.ethereum.org/2016/11/18/hard-fork-no-4-spurious-dragon/, 2016.

[95] Hudson Jameson. Ethereum constantinople/st. petersburg upgrade announcement. https://blog.ethereum.org/2019/02/22/ethereum-constantinople-st-petersburg-upgrade-announcement/, 2019.

[96] Hudson Jameson. Ethereum istanbul upgrade announcement. https://blog.ethereum.org/2019/11/20/ethereum-istanbul-upgrade-announcement/, 2019.

[97] Hudson Jameson. Ethereum muir glacier upgrade announcement. https://blog.ethereum.org/2019/12/23/ethereum-muir-glacier-upgrade-announcement/, 2019.

[98] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel software guard extensions: EPID provisioning and attestation services. Technical report, Intel Corporation, March 2016.

[99] Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. The eigentrust algorithm for reputation management in P2P networks. In *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, pages 640–651, New York, NY, USA, May 2003. Association for Computing Machinery.

[100] Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. Incentives for combatting freeriding on P2P networks. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Euro-Par 2003 Parallel Processing*, pages 1273–1279, Berlin, Heidelberg, August 2003. Springer Berlin Heidelberg.

[101] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. Cheapbft: Resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 295–308, New York, NY, USA, April 2012. Association for Computing Machinery.

[102] Sahiti Kappagantula. Microservice architecture — learn, build, and deploy applications. https://dzone.com/articles/microservice-architecture-learn-build-and-deploy-a, 07 2018.

[103] Reza Sherafat Kazemzadeh and Hans-Arno Jacobsen. Reliable and highly available distributed publish/subscribe service. In *2009 28th IEEE International Symposium on Reliable Distributed Systems*, pages 41–50, New York, NY, USA, September 2009. Institute of Electrical and Electronics Engineers.

[104] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. Integrating remote attestation with transport layer security. Technical report, Intel Corporation, January 2018.

[105] H. Krawczyk and P. Eronen. HMAC-based extract-and-expand key derivation function (HKDF). RFC 5869 (Informational), May 2010.

[106] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11 of *NetDB'11*, pages 1–7, New York, NY, USA, June 2011. Association for Computing Machinery.

[107] Seda Kul, Süleyman Eken, and Ahmet Sayar. Distributed and collaborative real-time vehicle detection and classification over the video streams. *International Journal of Advanced Robotic Systems*, 14(4), July 2017.

[108] Seda Kul, Isabek Tashiev, Ali Şentaş, and Ahmet Sayar. Event-based microservices with apache kafka streams: A real-time vehicle detection system based on type, color, and speed attributes. *IEEE Access*, 9:83137–83148, June 2021.

[109] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[110] Andy Lawrence and Lenny Simon. Annual outage analysis 2023. `https://datacenter.uptimeinstitute.com/rs/711-RIA-145/images/AnnualOutageAnalysis2023.03092023.pdf`, March 2023.

[111] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Dawn Song, and Krste Asanović. Keystone: An open framework for architecting tees. Technical report, UC Berkeley, 2019.

[112] Daniel Lehmann and Michael Pradel. Wasabi: A framework for dynamically analyzing webassembly. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 1045–1058, New York, NY, USA, April 2019. Association for Computing Machinery.

[113] Bo Li, Qiang He, Liang Yuan, Feifei Chen, Lingjuan Lyu, and Yun Yang. EdgeWatch: Collaborative investigation of data integrity at the edge based on blockchain. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '22, pages 3208–3218, New York, NY, USA, August 2022. Association for Computing Machinery.

[114] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*, pages 359–376, New York, NY, USA, May 2015. Institute of Electrical and Electronics Engineers.

[115] Jed Liu, Owen Arden, Michael D. George, and Andrew C. Myers. Fabric: Building open distributed systems securely by construction. *Journal of Computer Security*, 25(4–5):319–321, May 2017.

[116] Jed Liu, Owen Arden, Michael D George, and Andrew C Myers. Fabric: Building open distributed systems securely by construction. *Journal of Computer Security*, 25(4-5):367–426, July 2017.

[117] Ashwin Machanavajjhala, Erik Vee, Minos Garofalakis, and Jayavel Shanmugasundaram. Scalable ranked publish/subscribe. *Proceedings of the VLDB Endowment*, 1(1):451–462, August 2008.

[118] Stephanos Matsumoto and Raphael M. Reischuk. Ikp: Turning a pki around with decentralized automated incentives. In *2017 IEEE Symposium on Security and Privacy*, SP '17, pages 410–426, New York, NY, USA, May 2017. Institute of Electrical and Electronics Engineers.

[119] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, New York, NY, USA, June 2013. Association for Computing Machinery.

[120] John Mechalas. Code sample: Intel software guard extensions remote attestation end-to-end example. `https://software.intel.com/en-us/articles/code-sample-intel-software-guard-extensions-remote-attestation-end-to-end-example`, 2018.

[121] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. Twine: An embedded trusted runtime for WebAssembly. In *2021 IEEE 37th International Conference on Data Engineering*, ICDE '21, pages 205–216, New York, NY, USA, April 2021. Institute of Electrical and Electronics Engineers.

[122] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. WebAssembly as a common layer for the cloud-edge continuum. In *Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge*, FRAME '22, pages 3–8, New York, NY, USA, June 2022. Association for Computing Machinery.

[123] MesaTEE. MesaTEE, 2019. https://github.com/apache/incubator-mesatee/blob/master/README.md.

[124] MesaTEE. Mutual Attestation: Why and How. `https://github.com/apache/incubator-mesatee/blob/master/docs/mutual-attestation.md`, July 2019.

[125] Mariusz Michalowski. Cloud computing statistics 2023: Adoption & growth trends, March 2024.

[126] Andrew Miller, Iddo Bentov, Surya Bakshi, Ranjit Kumaresan, and Patrick McCorry. Sprites and state channels: Payment networks that go faster than lightning. In Ian Goldberg and Tyler Moore, editors, *Financial Cryptography and Data Security*, FC '19, pages 508–526, Cham, February 2019. Springer International Publishing.

[127] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241, New York, NY, USA, January 1999. Association for Computing Machinery.

[128] Michael Neuder, Daniel J Moroz, Rithvik Rao, and David C Parkes. Low-cost attacks on ethereum 2.0 by sub-1/3 stakeholders, February 2021.

[129] Joongun Park, Seunghyo Kang, Sanghyeon Lee, Taehoon Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Hardware-hardened sandbox enclaves for trusted serverless computing. *ACM Transactions on Architecture and Code Optimization*, 21(1), January 2024.

[130] Lauri I. W. Pesonen and Jean Bacon. Secure event types in content-based, multi-domain publish/subscribe systems. In *Proceedings of the 5th International Workshop on Software Engineering and Middleware*, SEM '05, pages 98–105, New York, NY, USA, September 2005. Association for Computing Machinery.

[131] PoolWatch.io. Best ethereum mining pools for 2020, 2020.

[132] Louis-Noel Pouchet and Tomofumi Yuki. Polybench/c benchmark suite. `https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1`, 2016.

[133] Thomas Prantl, Simon Engel, Lukas Horn, Dennis Kaiser, Lukas Iffländer, André Bauer, Christian Krupitzer, and Samuel Kounev. Performance impact analysis of homomorphic encryption: A case study using linear regression as an example. In Weizhi Meng, Zheng Yan, and Vincenzo Piuri, editors, *Information Security Practice and Experience*, pages 284–298, Singapore, November 2023. Springer Nature Singapore.

[134] Weizhong Qiang, Zezhao Dong, and Hai Jin. Se-Lambda: Securing privacy-sensitive serverless applications using SGX enclave. In Raheem Beyah, Bing Chang, Yingjiu Li, and Sencun Zhu, editors, *Security and Privacy in Communication Networks*, SecureComm '18, pages 451–470, Cham, August 2018. Springer International Publishing.

[135] RabbitMQ. RabbitMQ. `https://www.rabbitmq.com/`, 2023.

[136] Gowri Sankar Ramachandran, Kwame-Lante Wright, Licheng Zheng, Pavas Navaney, Muhammad Naveed, Bhaskar Krishnamachari, and Jagjit Dhaliwal. Trinity: A byzantine fault-tolerant distributed publish-subscribe system with immutable blockchain-based persistence. In *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, ICBC '19, pages 227–235, New York, NY, USA, May 2019. Institute of Electrical and Electronics Engineers.

[137] Mark Russinovich, Edward Ashton, Christine Avanessians, Miguel Castro, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Cédric Fournet, Matthew Kerner, Sid Krishna, Julien Maffre, Thomas Moscibroda, Kartik Nayak, Olga Ohrimenko, Felix Schuster, Roy Schuster, Alex Shamis, Olga Vrousgou, and Christoph M. Wintersteiger. Ccf: A framework for building confidential verifiable replicated services. Technical report, Technical Report MSR-TR-2019-16, Microsoft, Redmond, WA, USA, 2019.

[138] Ryan Schneider. Bloom filter false positive rate w/ ERC-20/721. `https://github.com/ethereum/go-ethereum/issues/17613`, 2018.

[139] Tania Saleem, Muhammad Umar Janjua, Muhammad Hassan, Talha Ahmad, Filza Tariq, Khadija Hafeez, Muhammad Ahsan Salal, and Muhammad Danish Bilal. Proofchain: An x.509-compatible blockchain-based pki framework with decentralized trust. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 213(C), August 2022.

[140] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig. Transport layer security (tls) session resumption without server-side state. RFC 5077 (Proposed Standard), January 2008.

[141] Vinnie Scarlata, Simon Johnson, James Beaney, and Piotr Zmijewski. Supporting third party attestation for intel sgx with intel data center attestation primitives. Technical report, Intel Corporation, 2018.

[142] Vinnie Scarlata, Simon Johnson, James Beaney, and Piotr Zmijewski. Supporting third party attestation for Intel SGX with Intel data center attestation primitives. Technical report, Intel Corporation, April 2019.

[143] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *2015 IEEE Symposium on Security and Privacy*, pages 38–54, New York, NY, USA, May 2015. IEEE.

[144] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary CPU architectures. In *19th USENIX Security Symposium*, USENIX Security '10, Berkeley, CA, USA, August 2010. USENIX Association.

[145] Amazon Web Services. Aws lambda. `https://aws.amazon.com/lambda/`, 2019.

[146] Haiying Shen. Content-based publish/subscribe systems. In Xuemin Shen, Heather Yu, John Buford, and Mursalin Akon, editors, *Handbook of Peer-to-Peer Networking*, pages 1333–1366. Springer US, Boston, MA, October 2009.

[147] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-tcb linux applications with sgx enclaves. In *Proceedings 2017 Network and Distributed System Security Symposium*, NDSS '17, San Diego, CA, 2017. Internet Society.

[148] Corwin Smith, Sebastian Supreme, Chirag Badhe, and Tyler Pfledderer. Ethereum proof-of-stake attack and defense. https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/attack-and-defense/, June 2023.

[149] SONM. SONM whitepaper. Technical report, SONM, May 2017.

[150] Deian Stefan, Amit Levy, Alejandro Russo, and David Maziéres. Building secure systems with LIO (demo). In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 93–94, New York, NY, USA, September 2014. Association for Computing Machinery.

[151] Jutta Steiner. Security alert [consensus issue]. https://blog.ethereum.org/2015/08/20/security-alert-consensus-issue/, 2015.

[152] Jutta Steiner. Security alert – [implementation bug in go clients causing increase in difficulty – fixed – miners check and update go clients]. https://blog.ethereum.org/2015/09/03/security-alert-implementation-bug-in-go-clients-causing-increase-in-difficulty-fixed-miners-check-and-update-go-clients-if-necessary/, 2015.

[153] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, August 2001. Association for Computing Machinery.

[154] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit A. Seshia. A formal foundation for secure remote execution of enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2435–2450, New York, NY, USA, October 2017. Association for Computing Machinery.

[155] The Investopedia Team. 51% attack definition: What it is, how it works, and example, May 2024. Reviewed by Erika Rasure, Fact-checked by Suzanne Kvilhaug.

[156] Shruti Tople, Soyeon Park, Min Suk Kang, and Prateek Saxena. VeriCount: Verifiable resource accounting using hardware and software isolation. In Bart Preneel and Frederik Vercauteren, editors, *Applied Cryptography and Network Security*, ACNS '18, pages 657–677, Cham, July 2018. Springer International Publishing.

[157] Tuan Tran, Haofan Zheng, and Owen Arden. AlacriTEE: A platform for decentralized serverless computing using trusted execution environments. July 2024.

[158] Truffle Suite. What is ganache? https://trufflesuite.com/docs/ganache/, 2023.

[159] Stephan Tual. Ethereum protocol update 1. https://blog.ethereum.org/2015/08/04/ethereum-protocol-update-1/, 2015.

[160] Rafael Brundo Uriarte and Rocco DeNicola. Blockchain-based decentralized cloud/fog solutions: Challenges, opportunities, and standards. *IEEE Communications Standards Magazine*, 2(3):22–28, October 2018.

[161] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave: secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, March 2019. Association for Computing Machinery.

[162] Chenyu Wang, Zhipeng Cai, and Yingshu Li. Sustainable blockchain-based digital twin management architecture for IoT devices. *IEEE Internet of Things Journal*, 10(8):6535–6548, February 2022.

[163] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. Towards memory safe enclave programming with rust-sgx. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 2333–2350, New York, NY, USA, November 2019. ACM.

[164] Wenhao Wang, Benjamin Ferrell, Xiaoyang Xu, Kevin W. Hamlen, and Shuang Hao. SEISMIC: Secure in-lined script monitors for interrupting cryptojacks. In Javier Lopez, Jianying Zhou, and Miguel Soriano, editors, *Computer Security*, ESORICS '18, pages 122–142, Cham, September 2018. Springer International Publishing.

[165] Wasabi. Wasabi issue 11. `https://github.com/danleh/wasabi/issues/11`, 2019.

[166] WebAssembly. The great renaming. `https://github.com/WebAssembly/wabt/commit/052d2864ec4cc45a3aca4bab1a833d1cc45e29d6`, 2018.

[167] WebAssembly. WABT renaming function references. `https://github.com/WebAssembly/wabt/commit/93640e2a4ef8559c412f9860aa8ed1497d756abc`, 2019.

[168] WebAssembly. WABT renaming memory operations. `https://github.com/WebAssembly/wabt/commit/ceef14583e7634230126c5b6974e697c5a58b2c2`, 2019.

[169] WebAssembly. WABT: The webassembly binary toolkit. `https://github.com/WebAssembly/wabt`, 2024.

[170] Jeffrey Wilcke. The ethereum network is currently undergoing a dos attack. `https://blog.ethereum.org/2016/09/22/ethereum-network-currently-undergoing-dos-attack/`, 2016.

[171] Jeffrey Wilcke. Homestead release. `https://blog.ethereum.org/2016/02/29/homestead-release/`, 2016.

[172] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. Technical report, Ethereum Foundation, 2022.

[173] Kwame-Lante Wright, Martin Martinez, Uday Chadha, and Bhaskar Krishnamachari. Smartedge: A smart contract for edge computing. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1685–1690, New York, NY, USA, July 2018. Institute of Electrical and Electronics Engineers.

[174] Qi Xia, Emmanuel Boateng Sifah, Kwame Omono Asamoah, Jianbin Gao, Xiaojiang Du, and Mohsen Guizani. MeDShare: Trust-less medical data sharing among cloud service providers via blockchain. *IEEE Access*, 5:14757–14767, July 2017.

[175] YCHARTS. Ethereum average gas limit. `https://ycharts.com/indicators/ethereum_average_gas_limit`, 2023.

[176] Liang Yuan, Qiang He, Siyu Tan, Bo Li, Jiangshan Yu, Feifei Chen, Hai Jin, and Yun Yang. CoopEdge: A decentralized blockchain-based platform for cooperative edge computing. In *Proceedings of the Web Conference 2021*, WWW '21, pages 2245–2257, New York, NY, USA, June 2021. Association for Computing Machinery.

[177] Liang Yuan, Qiang He, Siyu Tan, Bo Li, Jiangshan Yu, Feifei Chen, and Yun Yang. CoopEdge+: Enabling decentralized, secure and cooperative multi-access edge computing based on blockchain. *IEEE Transactions on Parallel and Distributed Systems*, 34(3):894–908, December 2022.

[178] Samee Zahur and David Evans. Obliv-c: A language for extensible data-oblivious computation. *IACR Cryptology ePrint Archive*, 2015:1153, November 2015.

[179] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 270–282, New York, NY, USA, October 2016. Association for Computing Machinery.

[180] Shijie Zhang and Jong-Hyouk Lee. Eclipse-based stake-bleeding attacks in PoS blockchain systems. In *Proceedings of the 2019 ACM International Symposium on Blockchain and Secure Critical Infrastructure*, BSCI '19, pages 67–72, New York, NY, USA, July 2019. Association for Computing Machinery.

[181] Yuanyuan Zhao and Daniel C. Sturman. Dynamic access control in a content-based publish/subscribe system with delivery guarantees. In *26th IEEE International Conference on Distributed Computing Systems*, ICDCS '06, pages 60–60, New York, NY, USA, July 2006. Institute of Electrical and Electronics Engineers.

[182] Haofan Zheng. Decent code repositories. `https://github.com/lsd-ucsc/decent-framework`, 2020.

[183] Haofan Zheng and Owen Arden. Secure distributed applications the decent way. In *Proceedings of the 2021 International Symposium on Advanced Security on Software and Systems*, ASSS '21, pages 29–42, New York, NY, USA, June 2021. Association for Computing Machinery.

[184] Haofan Zheng and Tuan Tran. Decentdiffmonitorexpr, 2020.

[185] Haofan Zheng and Tuan Tran. Gethdbreader, 2020.

[186] Haofan Zheng, Tuan Tran, and Owen Arden. Total eclipse of the enclave: Detecting eclipse attacks from inside TEEs. In *2021 IEEE International Conference on Blockchain and Cryptocurrency*, ICBC '21, pages 1–5, New York, NY, USA, May 2021. Institute of Electrical and Electronics Engineers.

[187] Haofan Zheng, Tuan Tran, Roy Shadmon, and Owen Arden. Decentagram: Highly-available decentralized publish/subscribe systems. In *The 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '24, pages 274–287, New York, NY, USA, June 2024. Institute of Electrical and Electronics Engineers.

[188] Nejc Zupan, Kaiwen Zhang, and Hans-Arno Jacobsen. Hyperpubsub: A decentralized, permissioned, publish/subscribe service using blockchains: Demo. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Posters and Demos*, Middleware '17, pages 15–16, New York, NY, USA, December 2017. Association for Computing Machinery.