Secure Distributed Applications the Decent Way

Haofan Zheng
hzheng6@ucsc.edu
UC Santa Cruz
Santa Cruz, CA

Owen Arden
owen@soe.ucsc.edu
UC Santa Cruz
Santa Cruz, CA

ABSTRACT

Remote attestation (RA) authenticates code running in trusted execution environments (TEEs), allowing trusted code to be deployed even on untrusted hosts. However, trust relationships established by one component in a distributed application may impact the security of other components, making it difficult to reason about the security of the application as a whole. Furthermore, traditional RA approaches interact badly with modern web service design, which tends to employ small interacting microservices, short session lifetimes, and little or no state.

This paper presents the Decent Application Platform, a framework for building secure decentralized applications. Decent applications authenticate and authorize distributed enclave components using a protocol based on *self-attestation certificates*, a reusable credential based on RA and verifiable by a third party. Components mutually authenticate each other not only based on their code, but also based on the other components they trust, ensuring that no transitively-connected components receive unauthorized information. While some other TEE frameworks support mutual authentication in some form, Decent is the only system that supports mutual authentication without requiring an additional trusted third party besides the trusted hardware's manufacturer. We have verified the secrecy and authenticity of Decent application data in ProVerif, and implemented two applications to evaluate Decent's expressiveness and performance: DecentRide, a ride-sharing service, and DecentHT, a distributed hash table. On the YCSB benchmark, we show that DecentHT achieves 7.5x higher throughput and 3.67x lower latency compared to a non-Decent implementation.

CCS CONCEPTS

• **Security and privacy** → **Trusted computing**; **Distributed systems security**; *Authentication*; *Authorization*; *Security protocols*; • **Networks** → *Security protocols*; Cloud computing; Peer-to-peer networks; • **Computer systems organization** → Cloud computing; Peer-to-peer architectures.

KEYWORDS

enclave; trusted execution environment; distributed enclave application; remote attestation; mutual attestation; mutual authentication

ASSS '21, June 7, 2021, Virtual Event, Hong Kong.

ACM ISBN 978-1-4503-8403-2/21/06.
https://doi.org/10.1145/3457340.3458304

ACM Reference Format:
Haofan Zheng and Owen Arden. 2021. Secure Distributed Applications the Decent Way. In *Proceedings of the 2021 International Symposium on Advanced Security on Software and Systems (ASSS '21), June 7, 2021, Virtual Event, Hong Kong*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3457340.3458304

1 INTRODUCTION

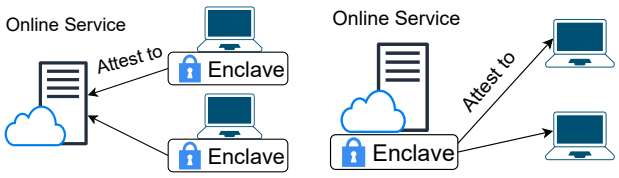
A fundamental challenge in building secure decentralized applications is that untrustworthy nodes may arbitrarily deviate from their expected behavior. A remote service may appear to execute a well-known system component when in fact it is executing a maliciously modified version. Trusted execution environments (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.

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 1a, or a client wishes to authenticate server code, as illustrated in Figure 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.

[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 paper, although that is demonstrably untrue for Intel SGX (e.g., [9]).



(a) Enclaves attest to a server. **(b) Enclave attests to clients.**

Figure 1: Traditional RA authentication

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 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 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., [18, 26, 31, 35]) 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. [6] 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, 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.

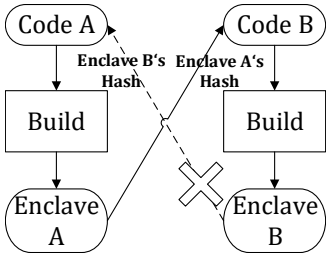


Figure 2: Mutual RA authentication. Enclave A and B cannot hardcode each other's identity directly in their code since it creates a circular dependency.

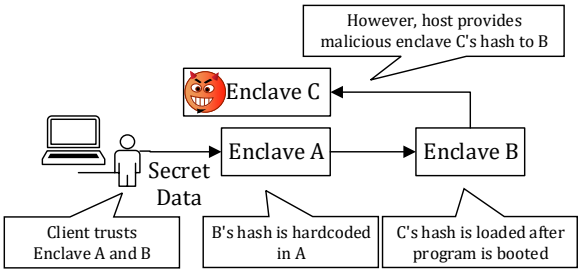


Figure 3: Allowing hosts to authorize enclaves independently could permit flows to malicious enclaves.

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, 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.

Another challenge for decentralized TEE applications arises when components are replicated for scalability. For example, serverless applications such as those built on AWS Lambda [30], Google Cloud Functions [12], or Azure Functions [5], 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

[3]Strictly speaking, all entities in a system secured by TEEs must trust the TEE manufacturer. In this paper, 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 paper.

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 [1, 25].[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 authenticating 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 paper 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 [7] 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 [13] 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 [28], and others [17, 20, 36] 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 [2].

The rest of this paper is organized as follows: §2 motivates the design of Decent using our decentralized ride-sharing example. §3 gives a high-level overview and the design of the Decent Application Platform. §4 and §5 discuss the details of the Decent Authentication and Authorization. §6 outlines our implementation of Decent SDK, and §7 presents the composition and result of the formal verification for Decent. Moreover, §8 evaluates the expressiveness and performance of our example Decent applications. Furthermore, §9 provides discussions on existing works that are related to Decent. Finally, §10 concludes.

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 4 illustrates the interactions between components of DecentRide, a ridesharing service loosely based on Uber's microservice-based design [19].

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

[4]Intel also supports DCAP [28], 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 §4.

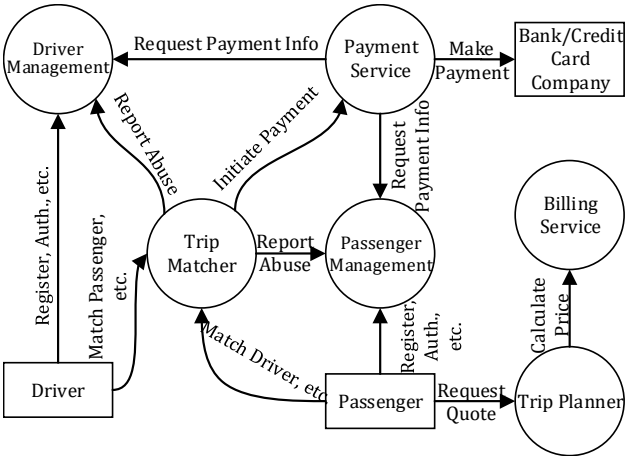


Figure 4: DecentRide, a decentralized ridesharing service.

circular dependencies described in §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 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) [1] to determine whether an attestation is authentic.

3 DECENT SYSTEM OVERVIEW

Figure 5 provides an overview of Decent framework.

Building Blocks. The Decent Framework includes Decent Components and Decent Servers. Trusted code in each Decent Component

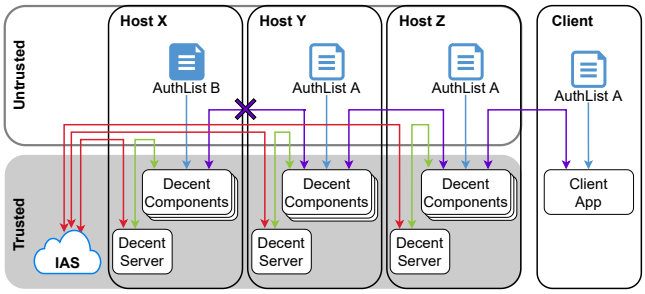


Figure 5: Overview of the Decent Framework; trust model shown is from the perspective of a client

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 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 §5.2 and §5.3, 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 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 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.

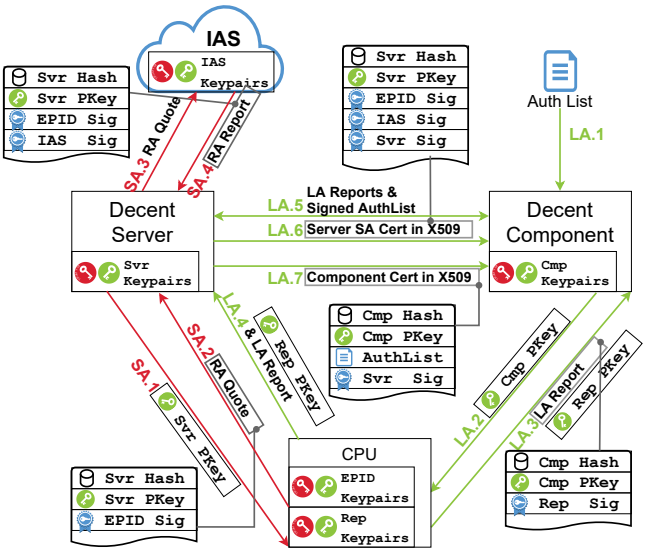


Figure 6: Process of creating certificates for the Decent Component authentication

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 [24, 32]), timing channels (e.g., predictive mitigation [4]), and access pattern analysis (e.g., oblivious computing [23, 37]).

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. We plan to extend the Decent platform with additional mechanisms for ensuring an application's services and data remain available in future work.

4 AUTHENTICATING WITH SELF-ATTESTATIONS

Figure 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 EPID [8] 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 Appendix B.

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 [28] 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.

5 AUTHORIZATION

5.1 Authorization List (AuthList)

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 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

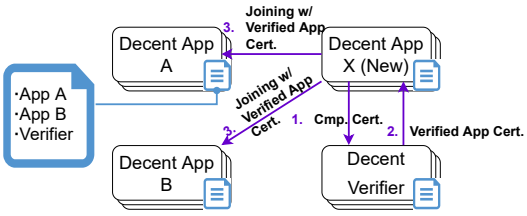


Figure 7: Procedures for the verifier

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.

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 7 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 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 [24] 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. If the source is successfully verified, the verifier compiles the source to machine code, calculates the hash of the new component and issues a signed approval. Any component presenting a valid SA certificate for the generated component will be signed by the verifier without the need for additional approvals.

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 (Byzentine Fault Tolerance) protocol could help application to tolerate compromised nodes, but it is beyond the scope of this paper. 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

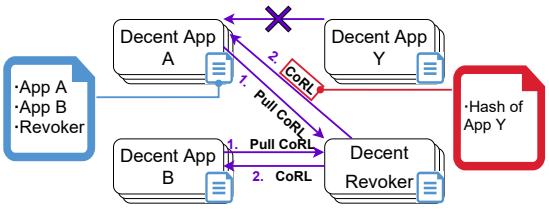


Figure 8: Procedures for the revoker

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]

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 reasonably 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 [14], 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 8 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 damage 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.

[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.

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.

6 USING THE DECENT SDK

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 mbedTLS 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.

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 [20] 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.

```
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 9: Creating a TLS channel with AuthList

Figure 9 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 configures 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.

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.

7 FORMAL VERIFICATION

We have formalized and verified the secrecy and authentication properties of the Decent protocol design using ProVerif [7]. 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 §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.

We defined six 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 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 13 in Appendix C illustrates the process diagram for our 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 [2].

8 PERFORMANCE EVALUATION

In order to evaluate Decent's overhead using a standard benchmark workload, we implemented a simple Distributed Hash Table (DHT) in Decent based on the Chord [33] 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 [16] 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.

8.1 Experiment setup

We evaluate the overhead of Decent authentication and authorization by comparing the performance of DecentHT on the YCSB benchmark [13] 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 channels 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.

For our experimental setup, 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. 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. 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 [27]. In SGX RA only group, we have also implemented an RA session ticket, a process similar to the TLS ticket, 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. The IAS simulator replays a single hardcoded response from the official IAS, and the nonce in RA report is ignored during the verification. 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.

More technical details about experiment setup is available in our tech report and GitHub repository [2?].

8.2 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 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.

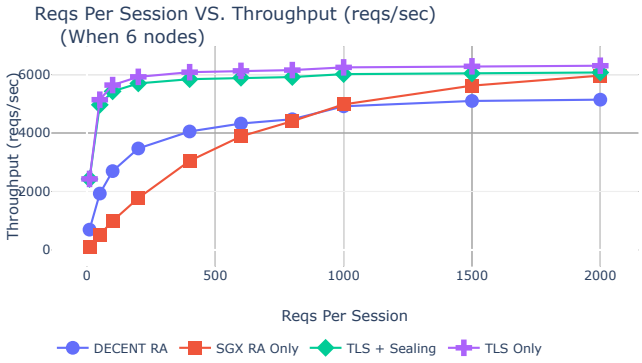


Figure 10: Requests per session versus throughput

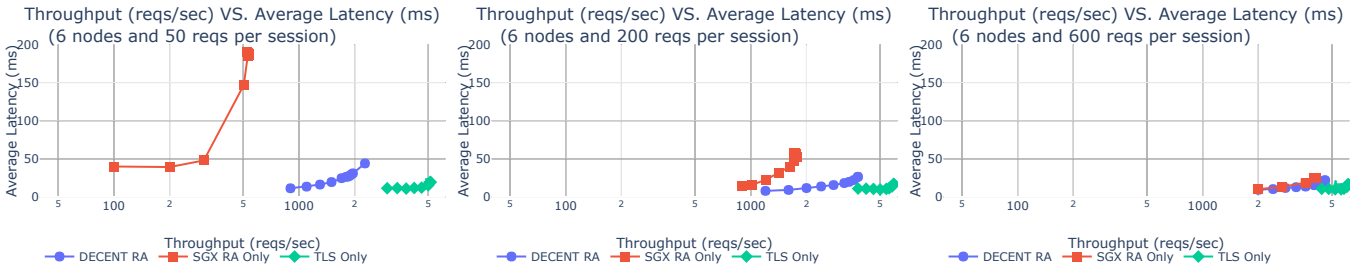
Figure 10 presents the results of these experiments. The non-enclave performance improvement plateaus at about 200 requests per session, with the TLS+Sealing implementation achieving a lower throughput due to the extra overhead of encrypting and decrypting the stored records. Between 10 and 400 requests 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.

Figure 11 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. As expected, latency in both implementations is largely unaffected by the small increase in nodes. However, the average latency for Decent RA (50ms) is almost four times lower than SGX RA (190ms).

9 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.[6] suggest a work-around for an application with two components by



(a) Each session includes 50 requests **(b) Each session includes 200 requests** **(c) Each session includes 600 requests**

Figure 11: Average latency versus throughput

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 [29] 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 [18] 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 [34] solves the problem of mutual authentication by relying on third-party *auditors* [35], 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 [31] 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 [26] 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 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.

Decent's SA process is similar to a now-common approach to authenticating TLS connections with enclaves. Knauth *et al.* [20] 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 [36] and Open Enclave [17] offer similar support. Similarly, OPERA [11] 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 §8 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 [3] and Graphene-SGX [10] 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 [15] and Keystone [22] 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 (Appendix A). 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

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

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.

10 CONCLUSION

In this paper, 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.

ACKNOWLEDGMENTS

We thank Tuan Tran for feedback on early drafts and Xiaowei Chu for assistance designing and building DecentHT. Partial funding for this research provided by NSF CAREER grant CNS-1750060.

REFERENCES

[1] Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent R Scarlata. 2013. *Innovative Technology for CPU Based Attestation and Sealing.* Technical Report. Intel Corporation.

[2] Anonymous. 2020. Decent Code Repositories. https://github.com/decent-ra.

[3] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 689–703.

[4] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. 2010. Predictive Black-Box Mitigation of Timing Channels. In *17th ACM Conf. on Computer and Communications Security (CCS)*. 297–307.

[5] Microsoft Azure. 2019. Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[6] Jethro G. Beekman, John L. Manferdelli, and David Wagner. 2016. Attestation Transparency: Building Secure Internet Services for Legacy Clients. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* (Xi'an, China) *(ASIA CCS '16)*. ACM, New York, NY, USA, 687–698. https://doi.org/10.1145/2897845.2897895

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

[8] Ernie Brickell and Jiangtao Li. 2009. Enhanced Privacy ID from Bilinear Pairing. *IACR Cryptology ePrint Archive* 2009 (2009), 95.

[9] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 991–1008.

[10] Chia che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 645–658.

[11] Guoxing Chen, Yinqian Zhang, and Ten-Hwang Lai. 2019. OPERA: Open Remote Attestation for Intel's Secure Enclaves. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 2317–2331. https://doi.org/10.1145/3319535.3354220

[12] Google Cloud. 2019. Cloud Functions. https://cloud.google.com/functions/.

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[14] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. 2008. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard). Updated by RFC 6818.

[15] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 857–874.

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

[17] Open Enclave. 2019. OpenEnclave SDK. https://openenclave.io.

[18] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. 2018. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. *ACM Trans. Comput. Syst.* 35, 4, Article 13 (Dec. 2018), 32 pages. https://doi.org/10.1145/3231594

[19] Sahiti Kappagantula. 2018. Microservice Architecture — Learn, Build, and Deploy Applications. https://dzone.com/articles/microservice-architecture-learn-build-and-deploy-a.

[20] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. 2018. *Integrating Remote Attestation with Transport Layer Security.* Technical Report. Intel Corporation. arXiv:1801.05863 [cs.CR]

[21] H. Krawczyk and P. Eronen. 2010. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869 (Informational).

[22] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Dawn Song, and Krste Asanović. 2019. *Keystone: An Open Framework for Architecting TEEs.* Technical Report. UC Berkeley. arXiv:1907.10119 [cs.CR]

[23] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. 2015. ObliVM: A Programming Framework for Secure Computation. In *2015 IEEE Symposium on Security and Privacy*. IEEE, San Jose, CA, USA, 359–376. https://doi.org/10.1109/SP.2015.29

[24] Jed Liu, Owen Arden, Michael D. George, and Andrew C. Myers. 2017. Fabric: Building Open Distributed Systems Securely by Construction. *J. Computer Security* 25, 4–5 (May 2017), 319–321. https://doi.org/10.3233/JCS-0559

[25] John M. 2018. 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.

[26] Mark Russinovich, Edward Ashton, Christine Avanessians, Miguel Castro, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Cédric Fournet, Matthew Kerner, Sid Krishna, et al. 2019. *CCF: A framework for building confidential verifiable replicated services.* Technical Report. Technical Report MSR-TR-2019-16, Microsoft.

[27] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig. 2008. Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 5077 (Proposed Standard).

[28] Vinnie Scarlata, Simon Johnson, James Beaney, and Piotr Zmijewski. 2018. *Supporting Third Party Attestation for Intel SGX with Intel Data Center Attestation Primitives.* Technical Report. Intel Corporation. 1–8 pages.

[29] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *2015 IEEE Symposium on Security and Privacy*. IEEE, San Jose, CA, 38–54. https://doi.org/10.1109/SP.2015.10

[30] Amazon Web Services. 2019. AWS Lambda. https://aws.amazon.com/lambda/.

[31] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. Panoply: Low-TCB Linux Applications With SGX Enclaves.. In *Proceedings 2017 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA, 15. https://doi.org/10.14722/ndss.2017.23500

[32] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. 2011. Flexible Dynamic Information Flow Control in Haskell. In *Proceedings of the 4th ACM Symposium on Haskell* (Tokyo, Japan) *(Haskell '11)*. Association for Computing Machinery, New York, NY, USA, 95–106. https://doi.org/10.1145/2034675.2034688

[33] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 2001. 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* (San Diego, California, USA) *(SIGCOMM '01)*. Association for Computing Machinery, New York, NY, USA, 149–160. https://doi.org/10.1145/383059.383071

[34] Mingshen Sun et al. 2019. MesaTEE. https://github.com/apache/incubator-mesatee/blob/master/README.md.

[35] Mingshen Sun et al. 2019. Mutual Attestation: Why and How. , 1 pages. https://github.com/apache/incubator-mesatee/blob/master/docs/mutual_attestation.md.

[36] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. 2019. Towards Memory Safe Enclave Programming with Rust-SGX. In *Proceedings of the 2019 ACM SIGSAC Conference*

on Computer and Communications Security (London, United Kingdom) *(CCS '19)*. ACM, New York, NY, USA, 2333–2350. https://doi.org/10.1145/3319535.3354241

[37] Samee Zahur and David Evans. 2015. Obliv-C: A Language for Extensible Data-Oblivious Computation. *IACR Cryptology ePrint Archive* 2015 (2015), 1153.

A ACCESSING AND MIGRATING SEALED DATA

Persistent data stored by an enclave must be encrypted with keys that the enclave maintains access to across reboots of the enclave process. Embedding secret keys in enclave code is obviously insecure, and keys that are stored in the (encrypted) enclave memory will be lost if the process exits. Intel SGX provides two key-derivation schemes for creating *seal keys* that encrypt data for persistent storage outside of the enclave. One scheme, `MRSIGNER`, 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. The other scheme, `MRENCLAVE`, uses the enclave's code hash to derive keys, meaning that only that enclave may decrypt sealed data.

Neither of these key-derivation schemes alone are appropriate for Decent applications. The author of an enclave has no special authority in Decent. Entities do not place trust in the *authors* of components, only in the code itself. However, allowing a Decent Component to access sealed data could be insecure if the component belongs to a different application instance than the one that sealed the data. Therefore, the Decent SDK uses a HMAC-based key-derivation function [21] (HKDF) to derive a key from the `MRENCLAVE`-derived key that binds the key to a specific AuthList.

Decent's HKDF scheme prevents malicious hosts from using legitimate components to leak sealed data to a malicious components (since the AuthList would be different), and malicious components from directly deriving another component's seal key (since the enclave's hash would be different). However, it 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 authority is revoked.

Note that TEE mechanisms like Intel SGX cannot on their own be used to guarantee recovery of sealed data. A malicious host may deny access to sealed data at any time. We are currently investigating mechanisms to integrate into the Decent framework that would support availability guarantees in additional to the current confidentiality and integrity guarantees.

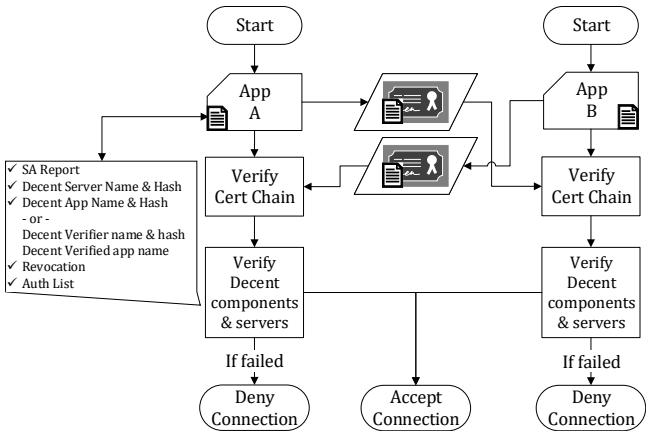


Figure 12: Decent handshake workflow

B DECENT HANDSHAKE PROTOCOL

The Decent handshake extends the usual TLS handshake where both sides authenticate with X.509 certificates. Figure 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 X.509 certificate chain by verifying all signatures on the certificate

(3) If signatures are successfully verified, a callback function is executed to verify the customized contents in the X.509 certificate

(4) The RA report contained in server's certificate is verified with Intel's public Report Key

(5) The local AuthList is consulted to ensure the Decent Server's hash appears under the "DecentServer" service name.

(6) Authenticating Decent App
   - Decent App's hash must either appear under the expected component service name in the AuthList
   - Or, Decent App's certificate must be signed by a verifier whose certificate is included (and verified) in the certificate chain, and whose hash appears under the expected verifier service name

(7) The local revocation list is consulted to ensure all hashes—apps, and verifiers—are still valid and have not been revoked [7]

(8) the AuthLists included in each component's certificate are compared to the local AuthList to ensure they match.

(9) If all checks are successful, the connection is accepted, otherwise it is rejected

[7] Our current prototype does not have revocation implemented yet.

C PROVERIF PROCESS GRAPH

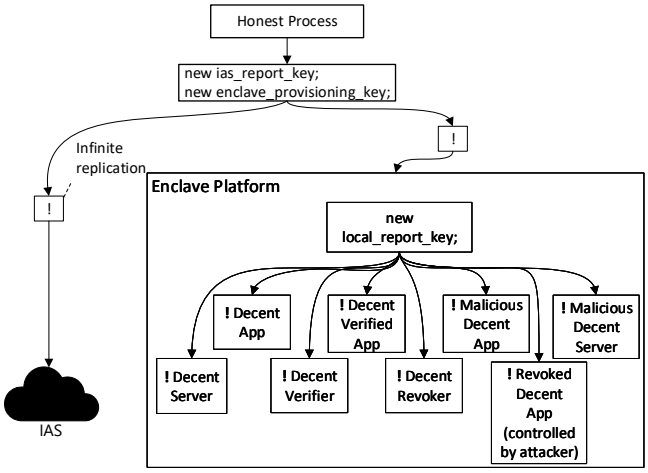


Figure 13: Verification Processes Overview