Attestation Transparency[*]

Building secure Internet services for legacy clients

Jethro G. Beekman
University of California, Berkeley
jbeekman@eecs.berkeley.edu

John L. Manferdelli
Google
manferdelli@google.com

David Wagner
University of California, Berkeley
daw@cs.berkeley.edu

ABSTRACT

Internet services can provide a wealth of functionality, yet their usage raises privacy, security and integrity concerns for users. This is caused by a lack of guarantees about what is happening on the server side. As a worst case scenario, the service might be subjected to an insider attack. We use remote attestation of the server to obtain guarantees about the programming of the service. On top of that, we augment Certificate Transparency to distribute information about which services exist and what they do. Combined, this creates a platform that allows legacy clients to obtain security guarantees about Internet services.

CCS Concepts

•Security and privacy → Trusted computing; Software security engineering; •Computer systems organization → *Client-server architectures;* **•Networks →** *Cloud computing;*

Keywords

Remote attestation; Secure enclaves; Certificate Transparency; Cloud computing

1. INTRODUCTION

End users increasingly perform important computing activities online in the cloud. This is convenient for them but the guarantees they get about those activities are significantly reduced from an ideal desktop-computing model where applications are run on trusted machines, inaccessible to adversaries, using software installed and maintained by knowledgeable trusted personnel known to the end user. On well-managed desktop machines, users can be confident that the software they use is the version they expect, with known behavior and mechanisms to prevent unauthenticated access to their data and unauthorized modification to the software itself.

Cloud services typically do not provide similar guarantees, which raises privacy, security and integrity concerns. Who will have access to my data, intentionally or unintentionally? Will the service continue to work properly tomorrow? Can the service read my data and use it for purposes I didn't have in mind? Will my data still exist in the same form tomorrow? Could a malicious system administrator at the service read or modify my data? If a system administrator's credentials are breached, could an attacker gain access to my data? For current Internet services, the answers to these questions are often unsatisfying.

On the other hand, cloud services provide many benefits that desktop users don't get. Cloud services provide availability through redundancy and replication and they remove the burden of maintenance from the user. In addition, the sheer number of available services seems to be surpassing the number of applications available on the desktop.

The ideas presented in this paper aim to combine benefits from the cloud-based service model with some of the guarantees with which desktop computer users are familiar. We focus especially on defending against insider attacks. For instance, the insider might be a malicious system administrator, a system administrator whose credentials have been compromised, or even a government order that compels the service to provide access to user data [9]. In addition, we define a policy-based mechanism that allows users to choose which security properties they expect in a secure service.

Our basic approach is to use transparency to deter unauthorized access. We use hardware support for sealed storage to ensure that user data is stored in encrypted form, and not directly accessible by insiders. Then, we build mechanisms to ensure that insiders cannot modify or introduce backdoors into the software without detection. Our design makes use of recent advances in security technology such as hardware-supported remote attestation and Certificate Transparency. Services store a hash of the software that they are running in a public audit log, and (conceptually) clients verify that the server is running code that matches a hash in the public audit log. This means that insiders cannot make undetectable changes to the server-side code; at least in principle, any such changes can be detected through examination of the public audit log. One challenge is how to ensure that these servers can be used from legacy clients, such as existing web browsers. We show how to achieve this goal by building on Certificate Transparency.

[*]The extended version of this paper, which includes the related work section, is available at http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-12.html

ASIA CCS'16, May 30-June 03, 2016, Xi'an, China
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4233-9/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2897845.2897895

We claim the following contributions:

Insider attack protection We show how to build Internet services that are *unalterable secure services.* Such services provide integrity and confidentiality of computation and user data, are secure against insider attacks, and their functionality can be remotely verified.

Policy and update mechanism Although we describe services as unalterable, in fact, updates will often be desirable to fix bugs, improve usability or security or add functionality. Also, instances of a secure service should not be locked to a single platform. We demonstrate how these goals can be achieved, without user-visible downtime for updates, and without compromising security or privacy. Finally, secure services may—under policy control specified by a user—be authorized to collaborate with other user secure services while preserving the security promises; we show how this can be done as well.

Legacy client support In a manner similar to the way Certificate Transparency protects against CAs secretly issuing bad certificates, Attestation Transparency protects against service providers secretly changing the services they provide. This transparency provides a public record linking domain names to service implementations. Today's TLS clients immediately reap the benefits of the transparency framework, except in some cases when they are the victim of a targeted attack involving misissued certificates.

In the following section we review existing building blocks we use to build our framework. We provide a high-level overview of the paper in §3. In §4 we discuss the design of unalterable secure services while §5 describes how we augment Certificate Transparency to allow verification of these services. We evaluate our design in §6 and elaborate on potential use cases in §7. We then wrap up with the conclusion (§8).

2. BACKGROUND

2.1 Secure Enclaves

We define a *secure enclave* as an isolated process, executed on a platform that provides confidentiality and integrity of code and data as well as sealing and attestation. In general, these technologies allow initializing an isolated and perhaps encrypted block of memory with a known program. Access to application memory is restricted by hardware and external access to the software is similarly restricted to identified entry points into the code. The software loaded in an enclave is also measured[1], allowing the hardware to attest to another party that the expected software was properly loaded and initialized and that the enclave software is isolated from other software running on the computer. The platform also provides a way to encrypt data so that the encrypted data can only be decrypted by this particular instance of the code running on this particular hardware. Different technologies provide such secure enclaves, including Intel SGX [3, 18], IBM SecureBlue++ [5], TPM-based Flicker [17], and perhaps ARM Trustzone [26].

2.1.1 Attestation primitive

Attestation is a mechanism that allows software to make statements that can be verified remotely by communicating parties, using *attested statements.* When talking about secure enclaves, some hardware-based root of trust, H, will attest that it is running a program with identity (measurement) I. In order for such a program to communicate with the outside world securely, it will need an encryption key K, and a way to securely announce to the outside world that it controls that key. Attestation provides such a mechanism: the hardware makes a statement of the form[2]

$$A(I,K) = \ll H \text{ says ``}H \text{ runs } I \text{ which says } [K \text{ speaks for } I]\text{''} \gg .$$

Platforms providing secure enclaves often provide ways for an entity I_1 to endorse a particular program with identity I_2. For example, I_1 might cryptographically sign I_2, and this signature can be verified as part of loading I_2. Such an attestation is of the form

$$A(I_1 : I_2, K) = \ll H \text{ says ``}H \text{ runs } I_2 \text{ which says } [K \text{ speaks for } I_2]\text{'' and ``}I_1 \text{ endorses } I_2\text{''} \gg .$$

If the platform can not verify the endorsement itself, a similar statement can still be formed by including the endorsement directly, as in

$$A(I_1 : I_2, K) = \ll H \text{ says ``}H \text{ runs } I_2 \text{ which says } [K \text{ speaks for } I_2]\text{''} \gg \text{ and } \ll I_1 \text{ says ``}I_1 \text{ endorses } I_2\text{''} \gg .$$

2.1.2 Sealing and encryption primitives

The general secure enclave concept does not include secure persistent storage. This is generally solved by using an untrusted persistent store and storing data only in encrypted form. This provides a form of secure persistent storage. We discuss some limitations of this scheme (such as rollback attacks) in §7.2.

Encrypting data in such a way that only a particular instance of a secure enclave can access it is called *sealing.* Generally, this is achieved by using a symmetric encryption key derived from both the program and hardware identity. The sealing operation turns the message m into the sealed text $s = E_{\text{seal}}(m)$, while unsealing is $m = D_{\text{seal}}(s)$.

In this paper we also use *authenticated encryption*, written $c = E(K, m)$ and $m = D(K, c)$.

2.2 Intel Software Guard Extensions

Intel Software Guard Extensions (SGX) [3, 10, 12, 18] are a recently announced hardware technology and instruction set extension providing secure enclaves. A special set of instructions can measure and encrypt a memory region before transferring execution control to it. The trusted computing base of SGX-based secure enclaves encompasses only the processor hardware, its microcode and the enclave image itself. In particular, the operating system is *not* part of the TCB. Data stored in memory regions belonging to the enclave is encrypted before it leaves the processor, so that the memory bus is also not part of the TCB. The security of this system is predicated on the correct functioning of the processor hardware and the SGX instruction set.

2.2.1 Attestation

SGX-enabled hardware can generate *reports*: integrity-protected statements about the enclave generated by the

[1]A *measurement* is typically a cryptographic hash of the software as loaded together with any configuration information which may affect the software behavior.

[2]Following the Taos language [27].

hardware:

$$\text{Report}_{\text{local}} = \text{MAC}\left(I_{\text{enclave}} \| I_{\text{signer}} \| D_{\text{user}}\right).$$

The MAC key is different for each processor and private to the enclave that requested the report—only that enclave on the same processor can verify the report. I_{enclave} is the measurement of the code of the enclave the report is generated of and I_{signer} is the public key that was used to sign that enclave before loading it. D_{user} is an arbitrary value that can be specified by the enclave when requesting the attestation report. This can be used to bind data to the attestation.

A special secure enclave provided by Intel, called the *quoting enclave*, can replace the MAC with a signature:

$$\text{Report}_{\text{remote}} = \text{Sign}\left(I_{\text{enclave}} \| I_{\text{signer}} \| D_{\text{user}}\right).$$

The signature private key is private to the processor and cannot be used improperly or for any purpose. The corresponding public key is published by the vendor, and a third party can use it to verify that the report was created by actual Intel hardware.

2.2.2 Sealed storage

A special instruction can generate an enclave-specific *sealing key*. The key is derived as

$$K_{\text{derived}} = H\left(I_{\text{enclave}} \| K_{\text{device}} \| \ldots\right)$$

where K_{device} is a hardware-embedded secret unique to this device. The enclave can use this key to encrypt data which can only be decrypted by the same enclave running the same code on the same hardware.

A different key can also be derived as $K_{\text{derived}} = H\left(I_{\text{signer}} \| K_{\text{device}} \| \ldots\right)$. This key can be used to transfer data between enclaves running on the same hardware that were signed by the same public key. In this work, we don't use this key since it gives too much control to the signer.

2.3 Certificate Transparency

The Certificate Transparency (CT) framework [14], as the name implies, aims to provide transparency to the issuance of TLS certificates. CT makes all legitimate TLS certificates a matter of public record, making it trivial to identify misissued certificates. The framework consists of several parts.

Public append-only logs A CT log server maintains a log of all certificates submitted to it. The log is structured as a Merkle tree which allows efficient verification of additions to the log. When submitting a certificate to the log server, the server will return a Signed Certificate Timestamp (SCT). The SCT is a promise that the server will include the certificate in the log within a certain time limit, the *maximum merge delay*. The SCT can be used as proof to other parties that a certificate is part of the public record.

Monitors A monitor watches one or more CT log servers for suspicious changes. For example, a domain owner might know that all its certificates are issued by a particular CA. If a certificate for their domain issued by a different CA appears in a log, the monitor raises an alarm. The administrator can then act upon that alarm, e.g., by demanding the revocation of the phony certificate.

Auditors An auditor watches one or more CT log servers for consistency. It checks that the Merkle tree is updated consistently and that certificates are included as promised by SCTs. If it detects any inconsistency, it raises an alarm. The CT log owner will then need to explain the discrepancy or risk being shut down.

Browsers Once the CT framework is fully operational, TLS clients such as browsers can demand proof from TLS servers that the server's certificate appears in a log. TLS servers can provide this proof in the form of SCTs. If a certificate does not appear in the logs, that is suspicious, and the client can choose to abort the connection attempt.

3. OVERVIEW

We want application service providers on the Internet to be able to host *secure services*. These services must be able to store and handle user data securely. By secure, we mean that the data's confidentiality and integrity is preserved, in the face of attacks within the scope of the threat model set forth below, which includes insider attacks. While we cannot absolutely prevent insiders from violating security, the transparency mechanism guarantees that such violations will be *publicly* detectable.

Users of a secure service must be able to verify that their client is connected to a specific service that is known to provide those security properties. Legacy clients must be supported: users must be able to obtain most of the security benefits without installing special software. Beyond legacy clients there must be an incremental deployment path. Performance loss compared to insecure services should be minimal. Services must also be updateable, and the security properties must be maintained in the update process.

3.1 Threat model

We assume the server hosting the service uses some secure enclave technology that prevents the adversary from accessing the code and data running in the enclave. We allow adversaries all the capabilities of an active network attacker as well as full control over non-enclave software running on the computer hosting the service (e.g., for SGX enclaves, this includes control over the operating system). For instance, an insider might add malicious software, or service provider personnel might accidentally misconfigure the service; these are included in the threat model. The adversary also has the ability to run its own servers mimicking real services. We assume the user's client is secure and cannot be tampered with. The threat model is depicted in Figure 1.

Availability is out of scope for this paper. If a malicious insider wishes to destroy all user data or deny access to the service, they can do so.

3.2 Design overview

The basic idea is to run all of the service code—including TLS session establishment, request handling, and storage—in an enclave on the server. This provides isolation and ensures that even insiders on the server cannot tamper with

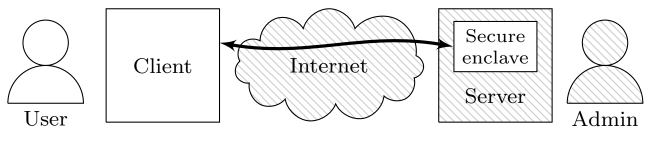

Figure 1: Secure service threat model. The shaded areas may be controlled by the adversary.

the running code or memory of the service. Also, we use sealed storage to prevent malicious insiders from reading or modifying data stored by the service on persistent storage: effectively, all data is encrypted before it leaves the enclave.

The user connects to the server, using TLS to establish a secure channel between the client and server. We use remote attestation to allow the user to verify what code is running in the enclave: secure hardware on the server provides a signed statement indicating what code has been loaded into the enclave. A fully attestation-aware client would then use this attestation to verify that the server is running the expected code.

We also show how legacy clients can access the server and gain a subset of the security guarantees. Legacy clients also connect to the server via TLS. Conveniently, the TLS protocol is widely supported and provides a secure channel to the server, while verifying the authenticity of that server. As is usual for TLS, the client checks that the server's TLS certificate is valid and authenticates the server using the public key found in this certificate. Our system extends the guarantees provided by this authentication step by further constraining the use of the private key.

In particular, a secure service runs inside a secure enclave and it will generate its TLS private key there. The TLS private key will never leave the enclave in unencrypted form; it is stored using sealed storage, so that only the enclave can retrieve it. Thus, even insiders cannot learn the service's TLS private key.

When the service is first created, it publishes its TLS public key in an attested statement proving that the enclave was launched with a certain code and that that code generated the key. Legacy clients not built with Attestation Transparency in mind won't be able to verify these attestations, but the key idea of our system is that another party can do so on their behalf. Because the attestations are public, anyone can check what code the service will run, that the code is secure, that it will never reveal its TLS private key, and that it protects itself adequately from malicious insiders. This allows word to spread through out-of-band channels that the service is trustworthy. For instance, an expert might inspect the code published by `good.com` and determine that it is trustworthy and will never leak its TLS private key; inspect the attestation and TLS certificate and determine that the TLS keypair was generated by this enclave, and the public part is in the TLS certificate; and then spread the word that `good.com` can be trusted.

Of course, a malicious insider at `good.com` could always take down the secure service and replace it with malicious code, running outside an enclave. An attestation-aware client could detect this (because the attestation will change), but a legacy client could not. However, this attack is detectable. To mount such an attack, the insider would need to generate a TLS keypair and get a new certificate issued for it (because legacy clients expect to connect over TLS), and hand the new private key to the malicious code. This is detectable because it triggers issuance of a new certificate for `good.com`. In particular, we use Certificate Transparency to detect issuance of new certificates. In our design, secure services publicly commit to always publish a new attestation any time they update the service or obtain a new certificate. Thus, issuance of a new certificate without a corresponding published attestation indicates an attack or error. Crucially, because all of this information is public, anyone can monitor the published information and detect these situations, providing transparency.

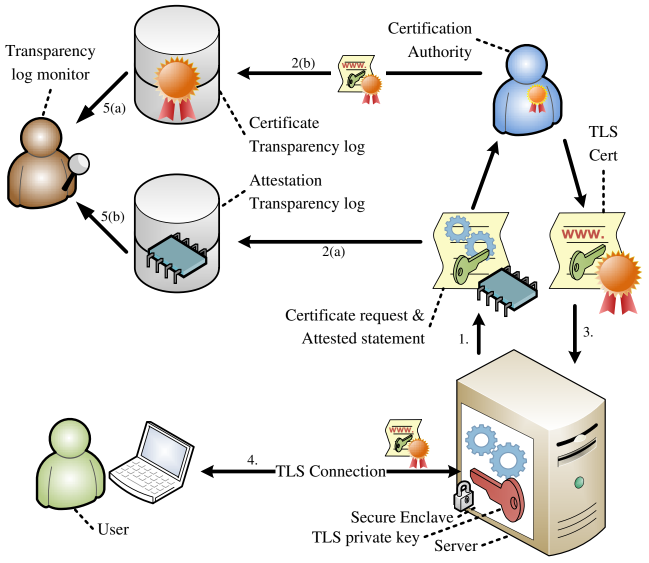

Figure 2: Overview of Attestation Transparency. (1) The *secure service* emits the certificate request and attested statement. (2) The attested statement and certificate are submitted to the *Attestation and Certificate Transparency logs*. (3) The secure service receives the certificate produced by the *CA*. (4) The *user* can now establish a regular TLS connection with the secure service. (5) The *transparency log monitor* independently monitors the transparency logs for possible violations.

Because our design focuses on transparency about what code the service will run, we call it *Attestation Transparency*. It extends Certificate Transparency to allow publishing these independently-auditable attestations. Legacy clients can rely on Certificate Transparency to ensure that attacks will be publicly detectable, while future attestation-aware clients can verify attestations themselves.

A diagram of the entire system is shown in Figure 2.

3.3 Policy model

To verify that a secure service will act "as promised", the user must verify that (a) the service code correctly implements the intended behavior and (b) no other program on the same computer will be able to interfere with the operation of the service code or observe its memory. The mechanisms described below allow a service to prove to a user what service code will execute on the server enclave. Thus, in principle, a user could examine all of the code and convince themselves that it will act "as expected" and will provide all the desired security guarantees. However, in practice few users will be able to do this: code analysis is expensive and beyond the capabilities of most end users.

To address this challenge, we provide several flexible mechanisms to enable users to verify that the service code will meet their needs (§5.3). One option is that the user may rely on a cryptographically signed statement from the secure service developer naming both the service identity and the (user comprehensible) promised behavior. Since the developer can produce the implementation corresponding to the identity, it can be verified by third parties or used as a basis for legal recourse in the case the service does not, in fact, conform to the promised behavior. Alternatively, a user can rely on either an authority (for example, in the case of an enterprise service, the enterprise "IT" department) or an auditor to decide which services are trustworthy. The authority cryptographically signs a statement representing that the service code conforms with expected behavior in all respects. These can be securely and automatically checked if so designated. Alternatively, a user may rely on a set of reviewers of the secure service code who cryptographically sign such statements. Reviewers might have different motivations, some altruistic and some self-serving. Further, a user may employ policy rules to automatically determine if the specified behaviors are adequate. For example, the user may insist that the EFF examine the service and certify that it meets a designated privacy policy, *and* that a consumer agency or a product reviewer also sign a conformance statement, *and* that the developer be one of a named set of developers that the user feels comfortable with. There are other alternatives that ensure compliance with the user's needs and relieve the user of the need to conduct extensive reviews themselves.

Usually, a user will use the same policy model to decide whether an update meets those same specified needs and, if it does, whether user data accessible to previous versions can be made available to subsequent versions.

4. SECURE SERVICE DESIGN

This section presents the architecture we use to implement *unalterable secure services.* Unalterable here means that the functionality of the service cannot be changed. This allows a client of the service to view the service as an extension of the client itself and not just a third-party program subject to the whims of another entity.

Our architecture runs services inside a secure enclave as well as to encourage secure software development. To reduce the attack surface, the architecture presents a limited interface to the programmer that should be sufficient for Internet services, and the interface is implemented in a *memory-safe and type-safe language,* Rust.[3]

4.1 Secure service interface

In our architecture (Figure 3), only the CPU and the code inside the secure enclave are trusted. The secure enclave has no input/output capabilities and relies on an *untrusted driver* for (insecure) access to the outside world. to communicate with the outside world. The untrusted driver is part of the host operating system and provides persistent storage (e.g., via C Standard I/O Streams), networking (e.g., via BSD sockets), and inter-process communication (e.g., via stdin/stdout/stderr). On top of this, the *secure enclave library* implements encrypted networking using TLS (e.g., using a standard TLS library), encrypted and sealed storage, attestation, and IPC. Software developers use these secure primitives to write their secure service. The secure enclave library and application code together form the secure enclave.

Secure networking Secure networking is provided using TLS. The secure client interface allows connecting to an Internet address using TLS and verifying the connection using default methods (per RFC 5280 [7] and RFC 6125 [22]). There is also an option to connect to another secure service running in an enclave and have it attest to the secure channel parameters. The client will then verify that the attestation is valid and matches the expected server enclave identity. The secure server interface will listen on a specified port and accept TLS connections from clients. This is the main communication mechanism for a service using our architecture. There is also an option to accept connections from clients that are themselves running in an enclave and have that client identify itself and attest to the secure channel parameters.

[3]Available at https://github.com/jethrogb/secserv/.

Table 1: Secure storage types

Benefit	Sealed	Keyed	Both
Protect against blanket access after breach		✓	✓
Protect against offline attack versus weak user key	✓		✓
Recover data after hardware failure		✓	

Secure storage The secure storage interface allows the application to store persistent data safely. The interface provides access to different data objects identified by their name or path, while using an encrypted storage backend. There are three possible keying schemes for encrypting the data before storage: using the sealing key, a user key, or both (encrypted with a user key, then sealed). The different schemes have different benefits as shown in Table 1. In case of a breach—e.g., due to a faulty update (see §5.3), or a code bug—using sealing only, in its simplest form, is inadequate. Further, sealing—in its simplest form—is hardware-dependent and any sealed data is lost after a hardware failure. Using a per-user key based on the user's password enables password-guessing attacks if the password is weak. This is of particular concern since in addition to online attacks via the normal service authentication mechanism that all Internet services have to deal with, in our model an adversary can perform offline attacks on the stored data.

Attestation The attestation interface allows a secure enclave to have the hardware attest to a key. It can also verify that attested statements match a certain identity and extract the key that was attested to.

IPC Services may use inter-process communication, e.g., for inputting configuration data and logging. Since this channel is not secure, no sensitive information should be logged through this channel, and it must not be used for configuration that changes the security properties of the

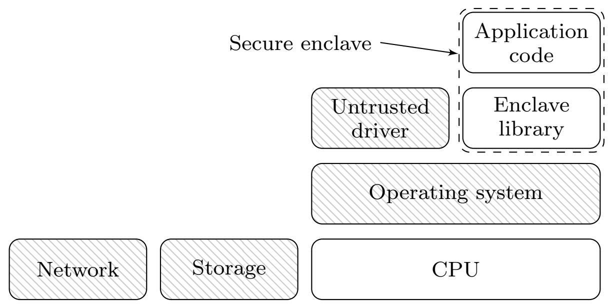

Figure 3: Secure service architecture. The shaded blocks are not trusted.

service. Instead, such configuration needs to be part of the enclave measurement.

4.2 Secure server

Using the primitives defined in the previous section, we can build an *unalterable secure service.* Keeping the private key K^{-1}_{server} of a TLS server in sealed storage and never exporting the key outside the enclave ensures only a particular secure enclave instance can have access to it. This means that when one establishes a TLS connection with a server that uses that K_{server} to authenticate its key exchange, the server endpoint is guaranteed to terminate inside the enclave.

Since the private key should never exist outside the enclave, it must be generated inside the enclave. The key setup procedure for the secure service enclave is shown in Figure 4. Input and output happens through the IPC channel.

All the service's static content (e.g., for a website, images and layout elements) must be included in the server binary that will be measured upon enclave startup. All dynamic/user content must be stored in secure storage.

4.2.1 Horizontal scaling

Once one instance of a service is running, another instance can connect to it and they can both verify that they're instances of the same program. After the verification, sensitive information can be shared (over a secure channel established using a secure key-exchange protocol). Any kind of distributed service can be supported this way.

4.2.2 Multiple enclaves

A service might consist of multiple parts, for example a database server and an application server. The enclaves should validate each other's identity and establish a secure channel between the two. There are at least two secure ways to implement this.

Consider an enclave A that accepts connections from clients and provides controlled access to information based on the client's identity. A second enclave B wants to use enclave A's service and has fixed A's identity in its loaded and measured code. Enclaves A and B establish a secure channel and both attest to their parameters. Enclave B can verify A's attestation and see that the identity matches what is expected. Enclave A can verify B's attestation and provide B access to the information B is authorized to access.

If both enclaves must verify each other's identity using embedded identities, there is a chicken-and-egg problem. Since the identity of a program changes when including a different identity, it's not possible for both programs to have the other's identity fixed in its code. Also, it's not secure to rely upon a system administrator to sign the identities of the two enclaves, since an insider could falsely sign the identity of a malicious enclave. One solution is to combine multiple programs into a single one with multiple operating modes. Now the same solution used for horizontal scaling can be applied.

4.2.3 Updates

When a service is updated, its persistent data will need to be updated too. Data encrypted with a user-dependent key can be used directly by the newer version. However, since the new service identity will be different from the previous version, all data stored in sealed storage is lost. Sealed data will need to be moved to the new version before the old

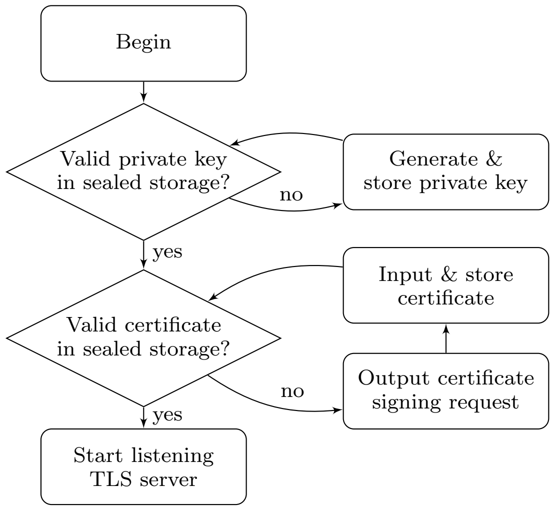
Begin
Valid private key in sealed storage?
Generate & store private key
no
yes
Input & store certificate
Valid certificate in sealed storage?
Output certificate signing request
no
yes
Start listening TLS server

Figure 4: Key management.

version can be retired.

A secure channel will need to be established between the old and the new version, see §5.3.1 for details on the authentication of this channel. Once the channel is established, the old version can unseal the data in question and send it across. The new version receives the data and immediately puts it in sealed storage. If there is too much data to be transferred over the secure channel, instead that data should be encrypted with an enclave-generated secret key. The key itself can then be stored in sealed storage and transferred for updates.

5. CLIENT VERIFICATION OF SECURE SERVICES

The previous section described how to construct a secure service. This section will explain how a client connecting to such a service can verify that service.

Consider the simple scenario in which the server sends an attestation $A(I_{\text{enclave}})$ to the client as part of establishing a secure channel. The client will need to verify both the attestation and the identity. As a straw-man proposal, envision a service provider distributing a client program that includes a fixed identity and can verify attestations expected for a particular service. This would require users to install a different client per service they want to use. Additionally, since the identity is fixed in the client, service updates would also require a client update.

A more general client could contain the logic to be able to verify all possible attestation mechanisms, as well as maintain a list of all acceptable identities. Done naively, this would be worse logistically, since now a new client needs to be distributed for every service update.

5.1 Attestation Transparency

Instead of creating this new verification mechanism that clients would need to implement, we build a verification mechanism on top of an existing mechanism that clients already know how to use: Public-Key Infrastructure. Under our proposed scheme, all the client needs to do to trust the secure service is verify the TLS server certificate using standard existing methods.

Our scheme, called *Attestation Transparency*, is an extension of the Certificate Transparency framework [14]. Remember that in our unalterable secure service model, demonstrating posession of K_{server}^{-1} by an entity implies that it is a secure service instance. The core idea of *Attestation Transparency* is that a secure service provider publishes (once) in the Attestation Transparency log an attested statement $A(I_{\text{enclave}}, K_{\text{server}})$. With this, they announce to the world that an entity possessing K_{server}^{-1} is an instance of I_{enclave}. The secure service provider also obtains a valid TLS server certificate for K_{server} through normal means and inputs it into the enclave. The certificate binds a Common Name (CN) to the key K_{server}, and the published attested statement binds that to the identity I_{enclave}. When a client establishes a TLS connection with the enclave, it verifies the certificate and the enclave uses its K_{server}^{-1} to sign the key exchange, after which the client has established a secure channel with the enclave. The whole process is depicted in Figure 2 on page .

An *Attestation Transparency monitor* performs a similar function to a Certificate Transparency monitor. The AT monitor continuously watches the CT logs for certificates issued to the CN identifying the secure service it is interested in. Once a new certificate with public key K is logged, the monitor checks the AT logs to see if any attested statements $A(I, K)$ with that same key K exist. If such an attested statement does exist, the monitor checks whether the identity I is valid for that service. If the identity is invalid, or no attested statement was found in the log, the monitor raises the alarm.

To prevent spamming, an AT log might require proof of existence of a valid certificate in the CT logs before accepting statements for submission. As such, there can be a short period of time where a certificate will exist in the CT logs without a corresponding statement in the AT logs. Monitors will need to take this into account and choose an appropriate wait time (e.g. two maximum merge delays) before raising the alarm. This wait time is the maximum time during which clients could be vulnerable to attack, before it could be noticed.

5.2 Incremental deployment — logs

While from the previous description it sounds like the AT log is separate from the CT log, this is not necessarily the case. Instead, attested statements can be included in a certificate as a X.509 Certificate extension.[4] The secure service can output the attested statement in the requested extensions section of its certificate signing request. As Certificate Transparency is already being deployed, this means Attestation Transparency does not require any new infrastructure. We propose minor changes to CT to support AT, along with an incremental deployment path towards a smoother process in the future.

5.2.1 Fake attestation certificates

Currently the only data that can be included in the CT logs are certificates and precertificates.[5] To prevent spam, the only certificates accepted in the logs are those signed by a known CA.

In order to publish attested statements in the CT logs, we propose that CT logs also accept (pre)certificates from an 'Attestation Authority' (AA). This is a fake Certification Authority that only issues pre-certificates and is not trusted by regular TLS clients. The AA follows a simple procedure: it takes as input a certificate, a Signed Certificate Timestamp and a certificate signing request that includes a statement as an extension. The AA verifies the certificate and SCT and it verifies the CSR includes the same public key. It will then issue a precertificate with the same Subject Name and public key, including the statement extension and a pointer to the real certificate. The AA will only issue one precertificate per real certificate.

5.2.2 Attested statement log entries

An alternate first step in the deployment process is to move the Attestation Authority's responsibilities into the CT log server. This requires a change in the CT specification to add a new entry type for attested statements. The inputs for the submission procedure will be the same, the verification and spam protection measures will be the same, only the output will be an attested statement-type entry in the CT log as well as an SCT for this entry.

While this setup increases the functionality and complexity of the CT log, it reduces the logistical complexity compared to using an Attestation Authority.

5.2.3 Certificate extensions

It would be much more convenient to just include the attested statement as an extension in the actual valid end-entity TLS certificate. This would eliminate the need for any changes to the current CT system. It would also solve the issue of a potential delay between the appearance of the certificate and the attested statement in the logs.

It is currently practically infeasible to obtain certificates with such an extension. We contacted a total of 9 subsidiaries of the largest 6 Certification Authorities (Comodo, Symantec Group, Go Daddy Group, GlobalSign, DigiCert, StartCom) to see if they would issue certificates with this extension. Of the CAs we contacted, 5 did not respond to our inquiry or did not understand the request, 3 were unable to provide such certificates, and 1 was unsure whether it would be possible, but if it was, it would cost an additional US$5,000. We considered (ab)using an existing extension, but were unable to find a suitable one for the type of data we'd want to include.

We encourage CAs to support Attestation Transparency extensions in the future.

5.3 Validating enclave identities

The previous discussion depends on being able to determine what is a valid enclave identity. This is mostly a matter of policy, and as such we present a mechanism that supports different policies. For each service, some entity or a group of entities—known as the *policy administrator*—is in charge of verifying the policy for an enclave identity. The policy administrator maintains a private key for each service policy. After verification, the policy administrator signs the enclave code indicating that the policy was met. For ex-

[4]We have allocated OID arc 1.3.6.1.4.1.4995.1000.4.1 for this purpose.

[5]Precertificates are similar to regular certificates, conveying the same information. However, they are constructed in such a way that they can't be used in place of a regular certificate.

ample, the EFF could establish a service that audits code for privacy violations and certify complying code by signing it. These compliance certificates can be used as an automated or semi-automated mechanism by client software to determine whether it trusts the code.

When a system runs such a signed enclave, it will issue attestation statements of the form $A(I_{\text{signer}} : I_{\text{enclave}}, K_{\text{server}})$. An AT monitor will maintain a list of policy administrators it trusts for a specific CN. Now, the monitor need not itself verify the enclave identity in an AT log entry, it can instead rely on a valid signature from the policy administrator.

5.3.1 Handling updates

This mechanism also enables code updates to services by having an old version of the service check the policy for the new version. This check is embedded in the code for the old service, and has undergone the same vetting process as the rest of the code. The whole process for updating from an old service S_1 to a new service S_2 is as follows.

$$D \xrightarrow{S_2} P \tag{1}$$

When a developer D is ready to update their service, they will send their binary S_2 and optionally documenting materials to the policy administrator P.

$$P \xrightarrow{\text{Sig}_P(I_2)} D \tag{2}$$

The administrator will verify that the new code meets the policy and sign it.

$$SP : S_2 \xrightarrow{\text{CSR}(K), A(P:I_2,K)} CA, AT \tag{3}$$

The service provider SP will launch the signed enclave S_2, which will output a certificate signing request including the attested statement. The service provider submits the CSR to a CA and the attested statement will be published to the AT logs.

$$CA \xrightarrow{\text{Cert}_{CA}(K)} SP : S_2, CT \tag{4}$$

CA will sign a certificate. The certificate will be submitted to the CT logs. With this publication, the policy administrator has announced to the world that I_{enclave} conforms to the policy established by I_{signer}. The service provider inputs the certificate into the signed enclave and launches the service. AT monitors will see the new service with the new certificate and can use the CT/AT log to verify that everything is in order.

$$S_2 \xleftrightarrow{\text{secure channel}} S_1 \tag{5}$$

S_2 establishes a mutually authenticated secure channel with S_1. Both sides must verify the code identity of the other side through attestation, as well as check the Certificate Transparency proofs for their keys. Doing both validates the service code *and* ensures that there is a public record for this particular service instance.

$$S_2 \xrightarrow{\text{Proof}(A(P:I_2,K)\in AT)} S_1 \tag{6}$$

S_2 provides S_1—which is configured to accept policy statements from policy administrator P—with proof that an attestation $A(P : I_2, K)$ appears in the AT logs.

$$S_2 \xleftarrow{\text{sealed data}} S_1 \tag{7}$$

S_1 will subsequently transfer its sealed data to S_2.

The update process can be performed without downtime for the users. Users can keep using the old version of the service as long as its certificate is still valid. Once the new certificate has been obtained and the required publications in the Transparency logs have been made for the updated service, it can start accepting connections. From then on, clients will see the new certificate and Transparency log proofs, indicating that they are now using the updated service.

5.3.2 Enclave policies

A policy shall at least require that the TLS private key will not be leaked and that updates shall be considered valid only when accompanied with a proof that they appear in the Attestation Transparency logs. While a policy administrator may issue a signed policy statement erroneously, the statement will be ineffective until published. Once published, others can hold the policy administrator accountable. It is also important that an entity controlling a Certificate Transparency log signing key must not also be an entity controlling a policy signing key. Such an entity would be able to issue signed policy statements and obtain a signed 'proof of inclusion' from the log without actually publishing the statement.

Policies can cover a variety of use cases from most transparent to not transparent. Care must be taken when a single party has a fair amount of control over what would be considered a secure service under their policy. Such constructions should always be paired with the ability for independent entities to verify their claims *post facto* using transparency. We propose the following policies:

Open-source reproducible builds The software developer publishes their source code publicly with a mechanism for reproducible builds. The same developer doubles as the policy administrator and builds the binary and signs it before handing it off to the service provider.

Independent audit The software developer hands their source code to an independent auditor. The auditor vets the secure service and describes the security properties the service has in a policy. It will sign the binary and publish the policy. When, later, the developer submits an updated version of the software, the auditor checks whether it meets the security requirements per the established policy.
As an extension to this scheme, an independent auditor could maintain several standard policies. For example they might have a 'secure IMAP service' policy. Anyone will be able to write software that adheres to the policy, and the auditor can verify and sign all such software. This effectively creates an interoperable system of secure IMAP services, where data can be transferred from one service to the other while maintaining the security properties.

Self-published with legal obligations The software developer hands their source code to a publisher. The publisher builds the binary and signs it before handing it off to the service provider. The publisher also promises (e.g. by incorporating a clause in their terms and conditions) that enclaves they sign exhibit certain properties.

Enterprise-local audit An enterprise might maintain a set of policies for secure services it runs internally. They can have an internal audit team that vets service updates. This way they can have the benefits of protection from insider attacks as well as local control.

5.4 Incremental deployment — clients

We present an incremental deployment path for Attestation Transparency that makes it immediately useful for today's clients while improving security guarantees for future clients.

Current clients Initially, clients without Certificate Transparency support will benefit from the existence of the CT/AT ecosystem, as independent entities can monitor the published certificates and statements. However, there are no guarantees for such clients and targeted attacks are possible. While the CT logs might include a valid certificate for some domain, a client without CT support can be presented with a valid certificate that does not appear in the logs, and the client would be none the wiser.

Clients supporting Certificate Transparency Once clients support Certificate Transparency, a process which is already in motion, they will get most of the benefits of Attestation Transparency as well. Suppose a service has subscribed to our secure service paradigm, promising to publish in the AT log in conjunction with the CT log. Then, by checking the Signed Certificate Timestamps when setting up a connection, a client can be sure that the server published its attestation if it has one. A user still needs to rely on manual verification or word-of-mouth to know whether a particular service at a particular domain is in fact a *secure service*.

Clients supporting Attestation Transparency A client that can check the attested statements will be able to indicate to the user that it is in fact connected to a *secure service*.

Clients supporting remote attestation Clients supporting remote attestation get even stronger guarantees than those just supporting Certificate Transparency. With remote attestation, a client can verify that the server they are connected to is actually running inside a secure enclave. This is helpful in case a server's TLS private key got leaked inadvertently. A third party could run a modified service with the TLS private key thus impersonating the secure service under the previous three mechanisms. When using remote attestation directly, this third party could not produce a correct attestation if the service were modified.

6. EVALUATION

Since Intel SGX is not yet available at the time of writing,[6] we performed our evaluation by implementing secure services on top of CloudProxy [16]. CloudProxy provides an abstraction over security primitives similar to the ones Intel SGX provides. A current implementation is available using Linux with a hardware root-of-trust based on the Trusted Platform Module (TPM) interface. This implementation has a much larger TCB than pure SGX would have, including the bootloader, the CloudProxy hypervisor, and the entire OS kernel.[7]

[6]Hardware with SGX is currently available in both Core and Xeon E3 processors, but Intel had yet to release the required additional software support.

[7]The OS kernel could be significantly reduced or even replaced by a small run-time library but we have not done this.

We do two case studies on secure services to evaluate developer effort and performance loss. First, to see how difficult programming with our secure service paradigm would be, we implemented a secure web server "from scratch."Second, we molded an existing web application stack into our secure service model to test the performance loss incurred. The original CloudProxy technical report [16] includes further performance measurements. We do expect slightly better performance on CloudProxy than we would see on real SGX hardware, since there is no runtime memory encryption overhead. All our tests were performed on an Intel NUC 5i5MYHE with Intel Core i5-5300 processor, 8GB RAM and a Samsung 850 EVO SSD, running Ubuntu 15.04 with a Linux 4.0.7 kernel.

6.1 File hosting service

We implemented the secure service interface in a memory-safe and type-safe but fast language, Rust [21]. The interface library is designed in a way that makes it easy to swap out different parts (such as OpenSSL for another TLS library or CloudProxy bindings for an SGX runtime).

On top of that we implemented an HTTPS server that runs a simple file storage service. A user can login using a username and password. They can then upload files which will be encrypted with a key derived from their password. Users can later retrieve these files by logging in again. The webserver is entirely self-contained—the binary includes all HTML, JavaScript, CSS and images that are going to be served to the user.

The webserver itself is 425 SLOC of Rust and 168 SLOC of HTML/JavaScript/CSS plus the jQuery framework and the Bootstrap theme. The interface library is 983 SLOC of Rust, plus an additional 262 SLOC of Rust and 1154 SLOC of C/C++ for the CloudProxy bindings. In addition, the webserver and library pull in an additional 59000 SLOC of Rust for statically linked dependencies. These numbers represent an upper bound on the actual number of lines as they include inline unit tests and feature-gated code that is not compiled—such as 18000 lines in unused Windows API bindings, and 5000 lines in an unused HTTP/2 implementation. The CloudProxy bindings pull in another 17000 SLOC of C/C++ for statically linked dependencies in addition to libprotobuf. We link to OpenSSL and common system libraries as well.[8]

6.2 Web forum

While the goal of this paper is not to show that legacy applications can be ported to our architecture—others have already shown similar results [4]—we do want to show that the architecture is fit for running large and complex Internet services. We mold an existing web server stack consisting of Apache, PHP, SQLite and phpBB to fit in our secure service architecture using CloudProxy and Linux OS features. The insecure filesystem, networking and IPC are provided as normal by the OS. Secure storage and attestation is provided by a small set of binaries that provide access to the CloudProxy interface. Secure networking is provided by Apache's mod_ssl and the keying interface by OpenSSL. Database encryption is provided by SQLCipher, [28] an enhanced version of SQLite that encrypts the on-disk files using AES. The database encryption key itself is stored in sealed storage.

[8]To be precise: libz, libdl, libpthread, libm, libstdc++, libgcc_s, libc.

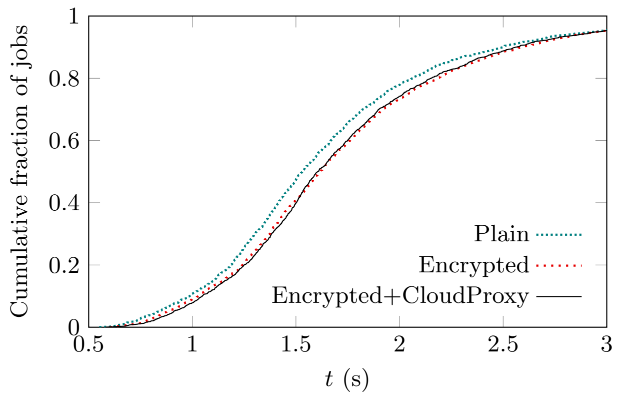

Figure 5: Cumulative Distribution Functions of the processing times for 3 different phpBB setups. For each line, $n = 1692$.

To make the web stack measurable, all files (binaries, system libraries, web content, etc.) are bundled into a single bootstrap binary. The bootstrap binary creates a new root filesystem namespace—isolating this process's view of the filesystem from the rest of the system—from an empty RAM-disk and fills it with the bundled files. It then starts an initialization script that sets up the keys according to Figure 4 as well as the database encryption key and then launches the Apache web server. The bundle is about 76MB.

We measure the performance of three related web forum setups to see the effect encryption and the CloudProxy platform have. The first setup, *plain* is a vanilla Apache, PHP, SQLite and phpBB installation. The second, *encrypted*, swaps out SQLite for SQLCipher. The final setup, *encrypted+CloudProxy*, runs the aforementioned bundled measured web stack.

For each setup, we load a small real forum dataset with 21 users, 94 topics and about 1300 initial posts. We then run a set of workers that access the forum over HTTPS simultaneously. Each user gets their own worker. Each worker selects a random topic they want to post to and performs the following procedure: login, navigate to the appropriate topic listing, visit the topic, go to the last page, go to the reply screen, and post a reply. Each topic is visited 6 times in total. We measure the aggregate time it takes to complete each topic-posting procedure from login to post. We perform the entire test procedure 3 times, for a total of $3 \cdot 6 \cdot 94 = 1692$ measurements per setup total.

The average processing time for the *plain* version is $\mu = 1.671$s ($\sigma = 0.693$). For the *encrypted* version $\mu = 1.731$s ($\sigma = 0.633$) and for the *encrypted+CloudProxy* version $\mu = 1.738$s ($\sigma = 0.646$). The *encrypted*/*encrypted+CloudProxy* setups are about 4% slower than the *plain* version. There is no significant difference between the *encrypted* and *encrypted+CloudProxy* setups (Two-sample Kolmogorov-Smirnov test, $p > 0.73$). The CDFs in Figure 5 show that all requests in the encrypted setups are just slightly slower than in the plain version, as opposed to the timing distribution being completely different. It also clearly shows the lack of difference between the *encrypted* and *encrypted+CloudProxy* setups, either setup being faster than the other at different percentiles.

7. DISCUSSION

7.1 Possible applications

7.1.1 Browser-based cryptographic systems

One of the arguments against doing in-browser cryptography using JavaScript is its poor auditability [15,20]. Even if a user assures themselves of the quality of the cryptographic system by carefully inspecting a page's DOM tree, there is no guarantee the server will send you the exact same page the next time you visit it. With an Attestation Transparency-supported secure service, a user does get that guarantee. Because the logic for sending HTTP responses is fixed within the unalterable secure service's identity, a client will receive the same script every time. This in combination with the Web Crypto API [23] brings us closer to being able to do browser-based crypto properly and securely.

7.1.2 Bootstrapping secure web applications

In the web application world, many production updates are pushed out every day. Having to go through the update process and requesting a new TLS certificate every time might not be practical. It is not necessary, however, to include an entire website within the secure enclave.

Instead, one can create a small *core web page* at a well-known URL (e.g. https://example.com) that will load further web content. Even untrusted content (e.g. from not-audited.example.com) can be included when using a technique such as HTML5 privilege separation [2]. The small core is secure and verified and provides only security functionality to the web application, which should require infrequent changes. The untrusted part of the website can be developed and updated frequently as normal, while not being able to cause harm because of the privilege separation.

Including static external content, e.g. from Content Delivery Networks, is supported securely through the recent Subresource Integrity draft [1]. Websites can include a hash with a URL on an external resource which will be checked by the browser.

Including dynamic external content is trickier. If an external site is known to be a secure service defined in this paper, verifying its known public key should be sufficient to ensure the safety of loading its contents. The Subresource Integrity mechanism could be extended to allow public key pinning on an external resource.

7.1.3 Encrypted e-mail storage server

An e-mail provider could run their SMTP/IMAP stack as two separate secure services. The IMAP server, storing the user's e-mails, will maintain an internal directory of users and corresponding encryption keys. Only the user will have access to their e-mails which are encrypted at rest. The SMTP server, when receiving mail for a local user, will obtain the local public key for that user from the IMAP server and encrypt the received message before handing it to the IMAP server for storage.

This setup provides secure encrypted e-mail storage for legacy IMAP clients including the inability of an insider to obtain the user's e-mails or credentials. Additionally, an SMTP client could verify the server's identity before submitting mail, making sure that the e-mail will get delivered to a secure mailbox.

7.2 Limitations and open research questions

The research in this paper does not address availability questions at all. Denial of service is a valid attack that an adversary might perform. Worse, destruction of user data is also possible. In order to get the cloud environment closer to the desktop model, these issues need to be resolved.

Secure enclaves don't keep any presistent state. As such enclaves can't know whether the untrusted party it's relying one for data storage is returning the most recent version of its encrypted state. This attack against the *freshness* of data is called a *rollback attack*. Generic solutions for this problem—such as Memoir [19]—exist and can be applied to the design proposed in this paper.

In order for a user to be able to fully trust a 'secure web application' as defined in the previous section, they need to know that the data they see or the input they provide is handled securely. The current web hardly provides such mechanisms. JavaScript and CSS on a page can arbitrarily change page elements to re-order information, capture user input, or even read cross-origin data [25]. More research effort is needed to provide the user with a secure and trustworthy user interface on the web.

The security of our system relies on an adversary not being able to break in to the secure enclave. Even if the hardware is infallible—which it isn't—a simple software bug could leak sensitive information or provide an attacker with code execution capabilities. Bugs are exacerbated by being completely transparent about the code running on a machine. The transparency makes it much easier for an attacker to automate exploitation of known vulnerabilities. We propose using only safe languages such as Rust to write secure services, but even then there is no guarantee against compiler bugs or developer errors. Further guarantees could be obtained by using formal methods.

While SGX in theory provides good isolation, in practice it might have security flaws. In addition, SGX does not aim to protect against side-channel attacks [13]. The operating system is in an excellent position to mount side-channel attacks as well as Iago attacks [6]. SGX also has software components that are critical to its security, which might be more easily compromised than the hardware [8]. Compromise of SGX on the system running the secure service provides an attacker with access similar to that of directly compromising the secure service. But, even if SGX is broken, future systems might provide better secure enclave functionality that can be used for the secure service design in this paper.

7.3 Adoption

Previous Trusted Computing approaches have not seen much practical use. Our scheme differs from most approaches in that the required hardware and software support is only needed on the server side. A single entity can decide to adopt our approach and make it happen without being dependent on their customer's hardware or software choices.

Some service providers may be reluctant to adopt our scheme. However, we believe there is motivation to strongly consider it. The ITIF has predicted that the U.S. cloud computing industry could lose up to \$35 billion by 2016 due to loss of trust in the post-Snowden era [11]. Forrester Research has predicted that losses could eventually be up to \$180 billion [24]. Our scheme provides a mechanism that partially addresses these trust concerns.

8. CONCLUSION

In this paper we have shown how to build secure services and how to enable clients to verify those services. This brings to clients the benefits of the cloud—including scalability, availability, elasticity, maintainability—while guarding against principal attacks (e.g. from insiders) that make cloud usage worrisome.

We have presented a system enabling flexible policy options to let users meaningfully choose security properties. Policies can be established by anyone, including software developers, auditors, independent organizations and communities. The policies are enforced through a compulsory transparency mechanism that brings malicious intent to light. This deters bad actors since they can be easily identified for purposes of legal action or reputation damage.

We have extended the certificate transparency model to code, providing a technical mechanism for users to rely on the security principal which ultimately ensures security properties—code. In addition, we provide flexible trust models that allow any user to meaningfully adduce behavior guarantees from actual implementations. We demonstrate that resulting systems can be nearly as efficient and scalable as existing services and provide strong protection from mischievous providers, foreign governments and sloppy cloud data center operations.

All proposed mechanisms include incremental deployment paths which make our techniques usable now for present-day clients, whereas future deployment will increase security guarantees. In conclusion, we have presented a flexible, practical mechanism to build secure Internet services.

9. ACKNOWLEDGEMENTS

We thank Jon McCune and our anonymous reviewers for their feedback. This work was supported by Intel through the ISTC for Secure Computing, AFOSR under MURI award FA9550-12-1-0040, and NSF under CCF-0424422.

10. REFERENCES

[1] D. Akhawe, F. Marier, F. Braun, and J. Weinberger. Subresource Integrity. W3C working draft, W3C, July 2015. URL: http://www.w3.org/TR/SRI/.

[2] D. Akhawe, P. Saxena, and D. Song. Privilege Separation in HTML5 Applications. In *21st USENIX Security Symposium*, pages 429–444. USENIX, Aug. 2012.

[3] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[4] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *11th USENIX Symposium on Operating Systems Design and Implementation*, pages 267–283. USENIX Association, Oct. 2014.

[5] R. Boivie and P. Williams. SecureBlue++: CPU support for secure execution. Technical report, IBM Research Report, 2013.

[6] S. Checkoway and H. Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages*

and Operating Systems, ASPLOS '13, pages 253–264. ACM, 2013.

[7] 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. URL: http://www.ietf.org/rfc/rfc5280.txt.

[8] V. Costan and S. Devadas. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint.iacr.org/2016/086.

[9] E. Felten. A court order is an insider attack, 2013. URL: https://freedom-to-tinker.com/blog/felten/a-court-order-is-an-insider-attack/.

[10] M. Hoekstra, R. Lal, P. Pappachan, C. Rozas, V. Phegade, and J. del Cuvillo. Using Innovative Instructions to Create Trustworthy Software Solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[11] Information Technology and Innovation Foundation. How much will PRISM cost the U.S. cloud computing industry?, 2013. URL: http://www.itif.org/2013-cloud-computing-costs.pdf.

[12] Intel Corporation. Intel Software Guard Extensions Programming Reference, October 2014.

[13] Intel Corporation. Intel Software Guard Extensions Enclave Writer's Guide, 2015.

[14] B. Laurie, A. Langley, and E. Kasper. Certificate Transparency. RFC 6962 (Experimental), June 2013. URL: http://www.ietf.org/rfc/rfc6962.txt.

[15] N. Lawson. Final post on Javascript crypto, 2010. URL: http://rdist.root.org/2010/11/29/final-post-on-javascript-crypto/.

[16] J. L. Manferdelli, T. Roeder, and F. B. Schneider. The CloudProxy Tao for Trusted Computing. Technical Report UCB/EECS-2013-135, EECS Department, University of California, Berkeley, Jul 2013. URL: https://github.com/jlmucb/cloudproxy.

[17] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 315–328. ACM, 2008.

[18] F. McKeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. 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*, 2013.

[19] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. In *32nd IEEE Symposium on Security and Privacy*, pages 379–394, 2011.

[20] T. Ptacek. Javascript Cryptography Considered Harmful, 2011. URL: http://www.matasano.com/articles/javascript-cryptography/.

[21] Rust programming language. https://www.rust-lang.org/.

[22] P. Saint-Andre and J. Hodges. Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS). RFC 6125 (Proposed Standard), Mar. 2011. URL: http://www.ietf.org/rfc/rfc6125.txt.

[23] R. Sleevi and M. Watson. Web Cryptography API. W3C candidate recommendation, W3C, Dec. 2014. URL: http://www.w3.org/TR/WebCryptoAPI/.

[24] J. Staten. The cost of PRISM will be larger than ITIF projects. Forrester Research, 2013. URL: http://blogs.forrester.com/james_staten/13-08-14-the_cost_of_prism_will_be_larger_than_itif_projects.

[25] P. Stone. Pixel perfect timing attacks with HTML5. Presented at Black Hat USA 2013, 2013. URL: http://www.contextis.com/documents/2/Browser_Timing_Attacks.pdf.

[26] J. Winter. Trusted Computing Building Blocks for Embedded Linux-based ARM Trustzone Platforms. In *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing*, pages 21–30. ACM, 2008.

[27] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Trans. Computer Systems*, 12(1):3–32, 1994.

[28] Zetetic LLC. SQLCipher. https://www.zetetic.net/sqlcipher/.