# Transparency for the Web using Confidential Computing

Shabsi Walfish

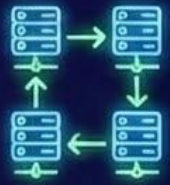
Google

# Overview

Motivation: Why Transparency for the Web?

Special Case: A Standalone Transparent Web Server

Background: Software and Certificate Transparency



Achieving the Goal: Transparent Web Services at Scale


Challenges: Web CAs, TLS, and Passports

Takeaways

# Why Transparent Web Services?

## Motivating Example: A Web Service for Age Verification

Recent Online Safety legislation increasingly mandates these

- Scan of your government issued photo ID
- Live video scan of your face
- Payment information

Users are rightfully concerned about abuse of their sensitive information:

Could be used for behavior tracking

and/or

Sold

and/or

Used for identity theft

# Trustworthy Privacy through Transparency

## THE CLAIM:

- Age Verification service claims no data is retained
- Perhaps they even publish their code or binaries for audit
- Seems reassuring, but...

## THE PROBLEM:

- How do you know you are really connected to the audited server?
- Maybe the server has been secretly and silently changed post-audit

## THE SOLUTION:

We need an assurance of continuous transparency

# The Foundation of Transparency: Transparency Logs

**Tamper Evident, Append-Only Log**

**Publicly Auditable & Verifiable Records**

- Anyone Can Monitor

**Builds Trust through Shared History (Witnessing)**

- Strong Guarantee of Consistency

# Software Transparency


**Software Manifest (e.g., SLSA)**
Create manifest with hash, publish to Log for inclusion evidence.

**Transparency Log**

**Verifiable Evidence**
Hash must match actual code measurements, binding software to manifest.

***Conceptually maps to "Workload Identity"***

**Progressive Transparency Layers**

**Source Code**
Gold Standard: Verifiable build from Source -> Binary -> Hash


**Binaries**
Allow behavioral analysis (reverse engineering)

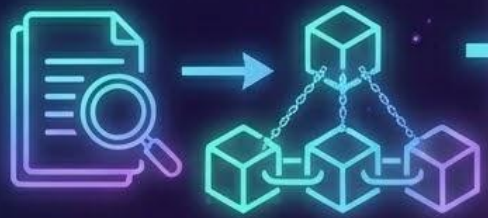
**Hashes**
Verify consistency (closed-source audits)

# Implementing Software Transparency with SigStore

**Cosign:** Tool for transparent software release signing and verification

Authenticates publisher's identity via Open ID Connect (OIDC)

Automatically logs the software manifest and authenticated publisher identity to a public, tamper-proof transparency log.



Transparency Log

Enables continuous, verifiable tracking of a publisher's release history.

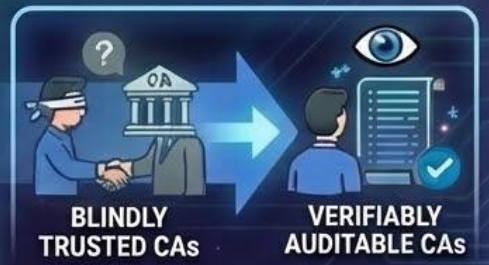
# Certificate Transparency

## PKI Problem: Compromised CA

A compromised Certificate Authority (CA) can secretly issue rogue certificates.


## Transparent Solution: Public Logging

All certificates must be verifiably published on a transparency log at issuance, not just signed by the CA.




## Paradigm Shift: Trust, but Audit

Blindly trusted CAs ⇒ Verifiably auditable CAs.

# Certificate Transparency (CT) on the Web


## Universal Logging

TLS certificates issued by Web CAs are now transparency logged.

## Strict Enforcement

Major web browsers strictly enforce CT for all HTTPS sites.

## Domain Auditing

Web domain owners can monitor and audit CT logs. Can revoke rogue certs and/or tarnish issuing CA's reputation.

# The Certificate Transparency Ecosystem

## Major Log Operators

Maintain highly available append-only ledgers on trusted infrastructure

Google Cloudflare Let's Encrypt

## Public Monitoring Services

Allow domain owners to search global logs and configure alerts.

Sectigo Cloudflare SSLMate

## Strict Browser Enforcement



Major browsers reject any certificate that lacks evidence of proper logging.

Chrome Safari Edge

# A Passport to the Web for Confidential Computing

# TLS Certificates are the “Passports” of the Web

The IETF RATS “Passport” Topology in Web PKI:

**The Attester**
(Web Server)

Generates a key pair and submits a CSR alongside Evidence of domain control (e.g., an ACME challenge).

**The Verifier**
(Web CA)

Appraises the domain control evidence and issues the Attestation Result (TLS Certificate).

**The Relying Party**
(Web Browser)

Accepts the “Passport” during the handshake. The browser trusts the connection without needing to contact the CA directly.

# Sure, but Web CAs don't EAT!

**The EAT** (Entity Attestation Token)

Confidential Computing has its own Evidence: Entity Attestation Tokens (EATs).

**Public Web CAs & Domain Control**

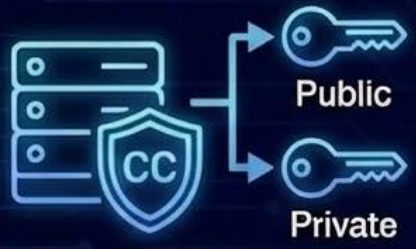

Public Web CAs check for Domain Control. They don't consume EATs.
We can't (and shouldn't) change global Web PKI.

**Private CAs *Should* EAT!**

If Internal CAs verify EATs before issuing certs, TLS works with CC attestation as-is.

# Living with Web CAs as they are

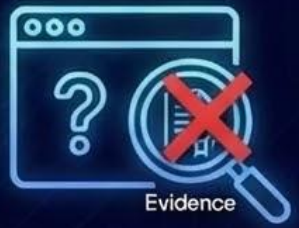
A Thought Experiment: Running a self-contained web Server on CC




1. Generates Key Pair


2. Contacts Web CA & Receives Certificate

3. Installs Certificate & Serves Traffic




But no one is checking Evidence.
If only we could solve that problem…

# What if we just EAT a PoP?

TLS
Public Key

EAT
Proof of Possession (PoP)

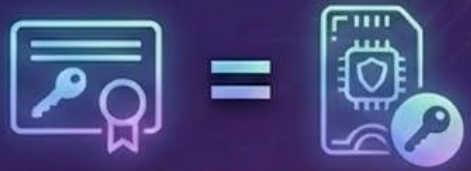
## Bind the TLS Public Key with an EAT Proof of Possession (PoP)

- Server generates EAT explicitly including its Public Key (RFC 9711 PoP claim)

## Replayable Evidence

Static, cacheable EAT asserts both TEE claims and ownership of the TLS key.

Store this EAT alongside the TLS Certificate.



Anyone can verify that TLS Cert Public Key == EAT PoP Public Key

**We have Evidence of the Web Server's claims, but who checks it?**
**Browsers don't EAT either!**

# Trust, but verify... Later

**The Foundation:**
Certificate Transparency (CT)

- Every TLS certificate is issued in public view on a CT log.

**The Big Idea:**
Asynchronous Appraisal

- Monitors watch the CT logs for your domain. Any new certs trigger...
- **Automatic Evidence Appraisal:** Request the matching EAT PoP!

**The Audit (Anyone can do this!):**

1. **Bind:** Verify that claimed EAT PoP Key == TLS Public Key.
2. **Measure:** Confirm the TEE is valid and extract software measurement.
3. **Verify Supply Chain:** Check that measurements match a transparent software release.
4. **Enforce:** Raise a public alarm if the evidence is missing or invalid.

# A Reality Check

Web servers require replication, software updates, and decomposition.

TEEs lose keys on reboot. New boot means new key and new TLS cert.

Server replicas generate new keys and TLS certs.

Software update requires reboot for transparency. More new keys and TLS certs.

**This would spam CT logs. Forget decomposition...**
**This all seems impractical...**

# Web Services at Scale, Part 1: Replication

## The Scaling Challenge

- **Problem**: Need multiple replicas to handle scale.
- **Requirement**: Replicas must all share the *same* TLS Certificate (for many reasons).

## The Key Insight: Measurement-Bound Key Sharing (MBKS)

Use a system that binds key access to software measurements.

## Implementation #1: Gossip Protocol

- Servers securely share the TLS key *only* to peers EATing identical measurements.
- (More MBKS implementations coming soon...)


## Current Limitation

Solved replication for stable certs, but assumes monolithic servers.

# Web Services at Scale, Part 2: Decomposition

## The Decomposition Challenge

**Reality:** Modern services are decomposed into heterogeneous components.

**Transitive Transparency:** Must prevent secret swapping of backend components.

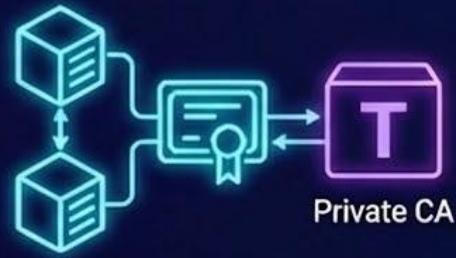
## The Measurement Problem


Components must use EAT authenticated channels.

Components have unique, frequently updating measurements.

**Uh-oh:** Rollouts can create combinatorial explosion of measurements!

## The Goal & New Tool




**Goal:** Transparent components should EAT each other via stable mTLS identities.

**New Tool:** Private CA that can consume EATs... and is itself transparent!

# Transparent CA (TCA ... with a capital TEE)


## TEE-based Private CA Service

### Runs Transparent Software
The CA code is published as a verifiable, transparent release.


### Exchanges EATs for Identity
Maps "good" EAT measurements to stable workload identifiers in standard Certificates. (Solves the combinatorial explosion!)


### EATs a PoP
Generates a self-signed Root Certificate alongside an EAT PoP as evidence of key provenance.

### But how should we store the root key?

## MBKS Implementation #2: Cloud KMS with Public Audit Log

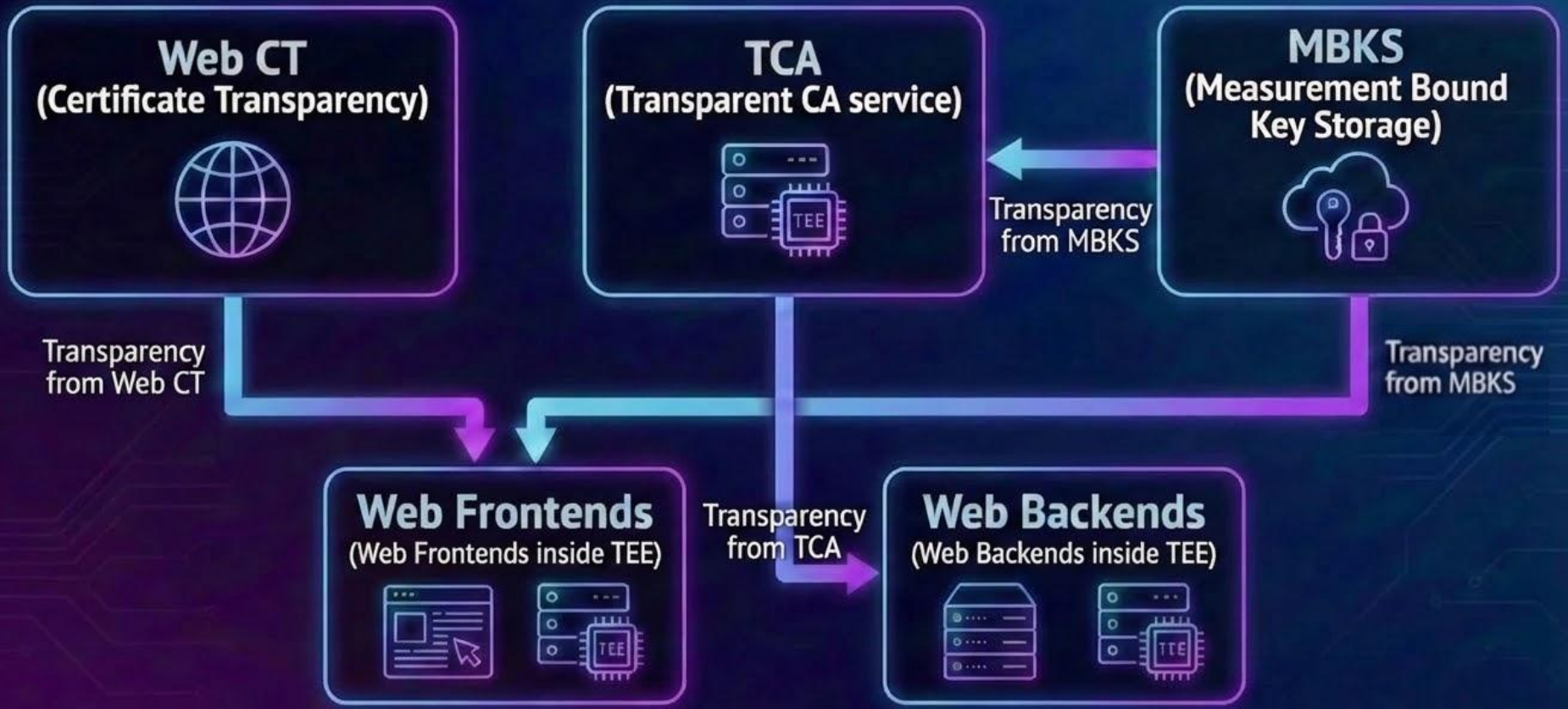
### Durability
Stores the Root Key Pair transparently via Cloud KMS to keep the Root Cert stable across reboots.

### Measurement-Bound Access
KMS access policy restricts key release strictly to the CA's authorized measurement.

### Provable Transparency
Publish the tamper-proof Cloud KMS audit trails showing access policy changes.

### The Result
Anyone can verify cloud provider only releases the root key to the published transparent CA software.

# Bringing It All Together



# Takeaways: The Transparent Confidential Web

- **Transparency is Deployable Today:** No need to change Web CAs or Web Browsers!
- **Decouple Identity & Evidence:** Use standard [m]TLS even for CC attested services.
  - "EAT a PoP" to cryptographically bind the TLS key to attestation evidence.
- **Trust, but Verify... Later:** Conduct audits on public Certificate Transparency (CT) logs.
  - Log monitors verify all issued certs are backed by valid EATs and transparent software.
- **Scale with TCA & MBKS:** Deploy a Transparent CA (in a TEE) to exchange EATs for TLS Identity.
  - Use Measurement-Bound Key Storage (in auditable Cloud KMS) to store and replicate keys.