Sigstore: Software Signing for Everybody

Zachary Newman
Chainguard
zjn@chainguard.dev

John Speed Meyers
Chainguard
jsmeyers@chainguard.dev

Santiago Torres-Arias
Purdue University
Department of Electrical and Computer Engineering
santiagotorres@purdue.edu

ABSTRACT

Software supply chain compromises are on the rise. From the effects of XCodeGhost to SolarWinds, hackers have identified that targeting weak points in the supply chain allows them to compromise high-value targets such as U.S. government agencies and corporate targets such as Google and Microsoft. Software signing, a promising mitigation for many of these attacks, has seen limited adoption in open-source and enterprise ecosystems.

In this paper, we propose Sigstore, a system to provide widespread software signing capabilities. To do so, we designed the system to provide baseline artifact signing capabilities that minimize the adoption barrier for developers. To this end, Sigstore leverages three distinct mechanisms: First, it uses a protocol similar to ACME to authenticate developers through OIDC, tying signatures to existing and widely-used identities. Second, it enables developers to use ephemeral keys to sign their artifacts, reducing the inconvenience and risk of key management. Finally, Sigstore enables user authentication by means of *artifact* and *identity* logs, bringing transparency to software signatures. Sigstore is quickly becoming a critical piece of Internet infrastructure with more than 2.2M signatures over critical software such as Kubernetes and Distroless.

CCS CONCEPTS

• **Security and privacy** → **Key management**; **Digital signatures**; **Security services**; *Usability in security and privacy*; • **Networks** → Security protocols.

KEYWORDS

security; code signing; distributed systems; software transparency

ACM Reference Format:
Zachary Newman, John Speed Meyers, and Santiago Torres-Arias. 2022. Sigstore: Software Signing for Everybody. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3548606.3560596

1 INTRODUCTION

Software supply chain compromises have garnered wide attention in recent years. Actors in the SolarWinds [65] and Codecov [62] compromises caused significant damage by attacking the chain of custody associated with developing, building, and distributing software. In other words, securing the supply chain is crucial to the overall security of a software product. An attacker able to control any step in this chain can introduce backdoors into source code or include vulnerable libraries in binaries or packages. Hence, attacks on the software supply chain are a high-impact mechanism for an attacker to affect many users at once. Moreover, attacks against the software supply chain are currently difficult to identify because they abuse processes that are normally trusted. Of particular importance is the last mile: the software delivery pipeline. Despite the existence of work documenting different mechanisms to protect the link between software users and software repositories, attacking the software delivery step still remains a common attack vector.

Perhaps surprisingly, software signing, one technique for mitigating the effect of software repository compromise, has not experienced widespread adoption. In fact, the vast majority of packages in popular software repositories that support signatures, such as the Python Package Index (PyPI), are unsigned [72]. Various efforts have been advanced to prepare signing infrastructure for these communities, yet there are often contentious issues related to ease of use, key management, and increased friction for maintainers.

In this paper we introduce Sigstore, a new open-source service that makes software signing part of an invisible and ubiquitous infrastructure. Drawing inspiration from Let's Encrypt and its effect on the adoption of HTTPS, Sigstore uses existing identity providers to issue short-lived certificates for individual package signing workflows. This way, users can sign using ephemeral keys ("keyless signing"), which allows developers to sign packages *without* managing their own cryptographic material; ephemeral keys, to our knowledge, have not been used for software signing. This not only allows for regular package upload processes, but also allows for automated workers (e.g., GitHub Actions) to sign and release a package on behalf of a developer.

By applying building blocks used by web public key infrastructure (PKI) and transparency log systems such as Golang's sumdb to the problem of artifact signatures, Sigstore enables linking software artifacts and identities. Like other transparency-log-backed package delivery security mechanisms, Sigstore uses a transparency log for signatures on packages, but it *also* publishes claims about identity in this log. The effect is that Sigstore not only prevents attackers without proper access from injecting new, malicious packages into a repository, but also enables auditors to detect repository tampering and account compromise.

Sigstore is gaining traction as a package signing solution and one approach to satisfying the software supply chain security controls associated with frameworks like Supply chain Levels for Software Artifacts (SLSA) [24]. Following adoption by GitHub Actions [39], popular Linux distributions such as Arch Linux [40] and Red Hat [37], and projects such as Google's Distroless [81], Sigstore

CCS '22, November 7–11, 2022, Los Angeles, CA, USA

ACM ISBN 978-1-4503-9450-5/22/11.
https://doi.org/10.1145/3548606.3560596

hosts more than two million (as of April 2022) different package signatures over more than 450 GitHub repositories.

This paper makes the following contributions:

(1) We identify limitations of prior approaches to artifact signing.
(2) We introduce a new signing mechanism using a protocol similar to Let's Encrypt and describe its threat model.
(3) We give applications of this mechanism and analyze package repository signing.
(4) We evaluate the performance and usage of Sigstore, and describe lessons learned in its deployment.
(5) We analyze the motivations and experiences of several projects and developers that have adopted Sigstore.

2 BACKGROUND

Sigstore relies on three technologies: artifact signing, transparency logging and identity providers.

2.1 Artifact and Repository Signing

Artifact signing. To sign an artifact, a signer generates a private/public keypair and uses the secret key to sign an arbitrary piece of data. In the context of package management, this is usually a signature over the package metadata, including the name of the package, the hash of the contents, and instructions for installation (such as pre-install hooks). Traditionally, the signer must keep the private signing key secure but accessible; this is the source of many usability issues [82].

Repository signing. Cappos et al. [12] show that attackers can tamper with repository metadata to conduct attacks even if packages are signed (for instance, delivering an out-of-date, vulnerable version of a legitimate package). In order to avoid these attacks, package repository administrators must also sign metadata about the repository's state. To resist compromise of the repository itself, a repository can delegate ownership of packages to individual package maintainers [43]. These maintainers would then need to manage keys; consequently, adoption of such systems has been low.

2.2 Transparency Logs and Web PKI

Web public key infrastructure (PKI) is a mature and widely deployed trust ecosystem. At its core, web PKI uses X.509 certificates [17] to pin the trust of web servers to a "certificate authority" (or CA). These CAs verify that an operator of a web server owns a particular domain, then issues them a certificate. Commonly, server operators use automated DNS challenge-response protocols to prove that they control a domain of choice.

One limitation of web PKI is the lack of top-level namespacing: multiple CAs can issue certificates to the same domain. For this reason, major CAs proposed and implemented certificate transparency (CT). Certificate Transparency keeps a public *transparency log* of issued certificates so that a third-party could notice if two CAs were to issue a certificate to the same domain (typically, a CA monitors the domains for which it issues certificates). Similarly, CT provides auditability and detection: domain owners can monitor the log, so that even the correct CA could be detected if it was compromised and issued a malicious certificate.

A transparency log ecosystem contains four main parties. First, *clients* submit entries. Second, the *log server* uses a historical Merkle tree [57] (which keeps records of past states) to record entries in the log, after a preliminary validation step. Third, a series of *auditors* monitor the log to avoid forking attacks and ensure that log server never removes any entries. Finally, verifiers can check that given entries are in the log (which forces other participants to publish entries) As a general construction, transparency logs provide an append only, global, non-forkable view (through the use of auditors [54]) of the state of a trust ecosystem. Various uses for these logs have been proposed by academics [41] and industry (e.g., Go's sumdb [30] or Firefox's binary transparency [6]). These logs are similar to a blockchain, in which entries added cannot be removed and, as further entries are performed, it is harder for an attacker to replace them. However, unlike a blockchain, a transparency log does not use an expensive consensus mechanism to support decentralization.

2.3 Identity Providers

Lastly, Sigstore uses identity providers to authenticate signers. We use OpenID Connect (OIDC) as the current construction used in production (the integration is modular, and other, similar systems could be used instead).

OIDC [61] is a widely-supported protocol allowing *relying parties* (applications) to verify the identity of *resource owners* (end users) as confirmed by *identity providers* (such as Google, Facebook, GitHub, or Okta). For example, an IniTech employee (resource owner) may use their Google (identity provider) account to log in to internal IniTech applications (relying party) by presenting an *identity token* issued by Google. Built from OAuth 2.0 [34], OIDC provides interoperability and wide language support, security against large classes of attack [49, 51], a well-defined identity specification, support for multi-factor authentication, and "workload identities" for non-interactive authentication (for example, by a build service).

3 SYSTEM GOALS AND THREAT MODEL

We base our threat model on historical attacks on the software delivery pipeline, especially account takeovers not linked to a compromised OIDC provider account [28, 85, 85] and registry compromises [31, 48, 79]. See data sets of supply chain compromises for additional examples [12, 15, 27, 43].

3.1 Parties and Roles

The Sigstore ecosystem consists of the following roles:

Signers Individuals vouching for the authenticity of content.
Verifiers Individuals checking that content is authentic.
Artifact Log Record of artifact metadata created by signers.
Identity Log Record of mappings from identities to signing keys.
OIDC Provider Mechanism vouching that an entity (individual) controls an identity (e.g., email account)
Certificate Authority Entity verifying OIDC identity tokens and issues cryptographic certificates to signers.
Monitors *Independent* service ensuring neither log lies about the content held (to avoid fork attacks [53]).
Package Repository *Independent service* hosting the artifacts referenced in the Artifact Log (e.g., PyPI).

Package Repositories are not essential to Sigstore, but are a core part of the specific use case we describe in Section 5.

3.2 System Goals

Borrowing from prior literature [4, 43, 67, 75], we envision the following security goals:

(1) Bind signatures over artifacts to OIDC identities.
(2) Provide a global, consistent view of the signing ecosystem.
(3) Provide an audit trail for compromise detection and post-incident analysis.

For the package signing case study (Section 5), we have the following additional goals:

(4) Bind software packages to the OIDC identities that can sign for them (maintainers).
(5) Provide a mechanism to ensure data is "fresh" (i.e., so as to avoid replay and freeze attacks).

3.3 Attacker Capabilities and Motivations

Attackers targeting the software supply chain are able to both compromise a server (e.g., a software package repository like DockerHub) and perform man-in-the-middle attacks on Internet communications. However, we assume that an attacker is unable to control cryptographic material or passwords from developers and packagers. We also assume that an attacker is unable to control a server for a prolonged period of time.

Attackers can attempt to impersonate a packager or developer to introduce malicious code. As such, they often target overlooked package dependencies. In addition, an attacker may perform targeted attacks by compromising packages used by a specific target.

In regards to Sigstore, we envision several types of compromise:

(1) An attacker is able to compromise a single identity provider used to perform OIDC authentication.
(2) An attacker is able to perform targeted man-in-the-middle attacks between a package repository and the client, a packager and a package repository, and identity provider. However, we assume that the attacker cannot intercept *all* traffic.
(3) Likewise, an attacker can compromise a package repository.

We assume secure distribution of the verification client itself, including public keys for an ecosystem *trust root*. This is analogous to secure distribution of a web PKI certificate bundle.

Key Compromise. We assume that verifiers know the public keys of project owners and that the attacker is not able to compromise the corresponding secret key. In addition, private keys of developers, continuous integration (CI) systems and other infrastructure public keys are known to a project owner and their corresponding secret keys are not known to the attacker. In Section 6 we explore additional threat models that result from different degrees of attacker access to the supply chain, including access to infrastructure and keys (both online and offline).

3.3.1 Implementation and Security Goals. To build a secure software supply chain that can combat the aforementioned threats, we design a system with the following properties:

Implementation transparency Sigstore should not require existing supply chains to change their practices in order to secure them; rather, it augments existing signature systems.

Graceful degradation of security properties Sigstore should not lose all security properties in the event of key compromise. That is, even if certain supply chain steps are compromised, the security of the system is not completely undermined.

In addition to these security goals, Sigstore is also geared towards practicality and, as such, it should maintain minimal operational, storage and network overheads and use widely-available cryptographic primitives.

4 SYSTEM DESIGN

In order to provide signatures over arbitrary artifacts, Sigstore performs three major operations (see Fig. 1 for an overview). The first, OIDC Issuance, vouches that a client is in control of an identity (e.g., an email account). The second associates short-lived public key certificates with these identities (from a Certificate Authority), and publishes these certificates to an Identity Log. The third publishes a long-lived signature over an artifact (or artifact meta-data) to an Artifact Log, allowing verifiers to check its validity.

The first and second operations are performed by a system called *Fulcio*, which functions as a Certificate Authority and transparency log for a namespace of OIDC identities (Identity Log). The third operation is performed by a system called *Rekor*, a transparency log for artifact signatures (Artifact Log). *Cosign* is the reference Sigstore client implementation, but Fulcio and Rekor have public API specifications and can be used by any client. With these components, Sigstore has three major phases: Trust Setup, Signing Flow and Verification Flow.

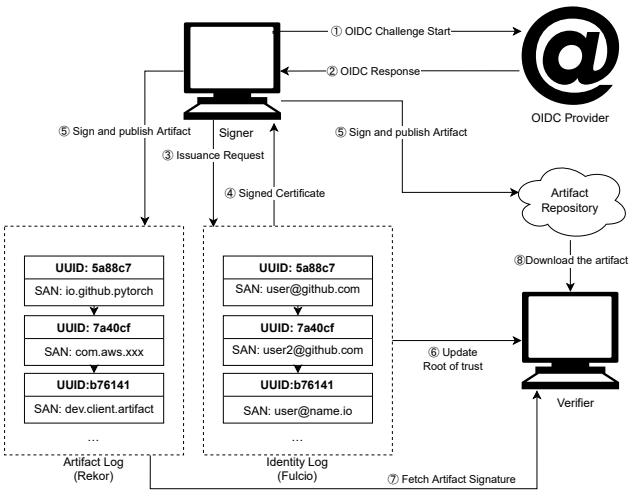

Figure 1: The Sigstore key issuance flow

4.1 Trust Setup: Root-of-trust via The Update Framework

Sigstore relies on The Update Framework (TUF) [4, 67] as a root-of-trust. TUF allows for efficient key rotation, key freshness and

delegation mechanisms. Web PKI does provide most of these benefits, but TUF also supports thresholds of signatures, and provides protection against subtle "freeze" or "replay" attacks in which an adversary prevents a client from updating its bundle to the latest set of certificates. It also handles revocations natively without any requirement for revocation lists. This is possible because all delegations within this framework fit in a file that clients can download, which is not the case at the scale of web PKI.

Sigstore uses TUF with 5 offline root keys, held by stakeholders in open-source, academia and industry. Once this root of trust is issued, it is published through various means. The trust root was created in a live-streamed key ceremony, with the root data published in real-time to public Git repositories and other services controlled by different entities. Once issued, this root of trust is included on client libraries for verification (described below).

In the Sigstore ecosystem, TUF manages *key material*, rather than artifacts themselves (the typical use of TUF). Sigstore uses TUF to distribute the root certificate for Fulcio and Rekor. This allows clients to ship with TUF root metadata containing the root keys, which can then be used to verify all future CA certificates.

4.2 Signing Flow

Signing in Sigstore requires interacting with three different online systems: an OIDC Provider, the Identity Log/Certificate Authority, and the Artifact Log. This process minimizes the amount of key management performed by signers while maintaining reasonable security properties on the signatures generated. To do so, we rely on *short-lived* certificates (with a validity period of 10 minutes) whose corresponding private keys are used to produce *a single signature*. This way, signers can generate signatures over arbitrary artifacts. Afterwards, they discard any private key material, which minimizes the likelihood of key compromise.

4.2.1 Public Key Infrastructure without Key Management. While web PKI has seen near-universal adoption over the last decade [13, 35] due in large part to services such as Let's Encrypt [1], most open-source developers, who may not control a domain or have spare money for an X.509 certificate from a commercial CA, have not adopted secure code-signing identities. Further, even if developers obtain such certificates, they will be faced with the well-known usability and security issues surrounding key management [26, 66].

Instead, Sigstore relies on existing digital identity providers that manage far more identities than PGP, comparable to DNS (1B [14]) or certificate transparency (1.5B certs [71] for 100M domains [33]). For instance, as of 2019, Google had 1.5B users for Gmail, their email service. OIDC [61], which Sigstore uses for authentication, is a standard used throughout the Internet to provide identity claims for people enrolled in popular services such as Gmail, GitHub, Facebook and more.

One fundamental difference exists between Let's Encrypt and Sigstore: while TLS connections are momentary, binaries are long-lasting. That is, in TLS, a certificate is valid if the current time is within its validity period. Traditional approaches to code signing tie the lifetime of the artifact to the lifetime of the certificate. This typically requires periodic re-signing, which is difficult in a community setting with actors that may come and go. This means that signing keys must periodically come online, which risks compromise.

Based on this insight, Sigstore introduces *ephemeral keys*: one-time-use key pairs and corresponding short-lived certificates. To sever the link between certificate lifetime and artifact lifetime, Sigstore introduces Rekor, an artifact log. In combination with Fulcio, this allows clients to validate that the signatures were made while a certificate was valid (even after it expires). Together, these services allow signers to remove reliance on key management (they get certificates by authenticating via OIDC, and destroy keys after a single use) and minimize the risk of key takeover (even if a single-use key is compromised, it cannot be reused after the short validity period). As we will explore in the security analysis, a compromise of an account does not fully subvert the trust on already-signed packages. Further, the logs make account compromise and maliciously issued certificates public, and they aid in investigation after an incident. Monitors can verify correct behavior in these logs to ensure the honesty of Rekor and Fulcio.

Finally, Sigstore proposes new mechanisms for signature and certificate distribution. While these are not technically complex, they enable new use cases by simplifying workflows for maintainers and end-users (see Section 5.2).

4.2.2 Signing Flow Operation. Algorithm 1 describes the client flow performed to generate a signature. This includes obtaining an OIDC identity token (Line 2) and generating a key pair and corresponding certificate signing request (Lines 3 and 4). With this information, the client will submit a signing request to Fulcio (the Identity Log and Certificate Authority) which will return a certificate signed by its root key (Line 5). This certificate will have Subject and Issuer fields associating it with the OIDC identity from the token. If successful, the client may now sign the data, and submit the signature to Rekor (the Artifact Log; Lines 6 to 8). Lastly, the same client tools can submit the artifact signed to a discoverable location, such as a package registry (Line 9).

Algorithm 1 Client flow to generate a signature

```
Require: provider, an OIDC provider
1: procedure Sigstore.Sign(artifact)
2:     tok ← OIDC.GetToken(provider)
3:     sk, pk ← GenerateKeyPair()
4:     chal ← Sign_sk(tok.sub)          ▷ certificate self-signature
5:     cert ← Fulcio.RequestCertificate(pk, tok, chal)
6:     sig ← Sign_sk(artifact)
7:     Discard sk
8:     Rekor.SubmitSignature(sig, pk)
9:     SubmitArtifact(artifact)                    ▷ to repository
```

We outline the operations performed by the server in Algorithm 2. These primarily include checks for correctness on the challenge-response. These include verifying (1) that the OIDC token was generated by a trusted provider and has a valid signature (Lines 2 to 3) and (2) that the subject field matches the claimed identity in the certificate (Lines 4 to 5). If these checks are all correct, Fulcio signs the certificate request, and the signed certificate returned to the client, who can now sign and submit a signature to Rekor.
As described above, once a short-lived certificate is created, a signer can submit a signature to the Rekor log. In order to submit a trusted

Algorithm 2 Server flow to accept a signature

Require: `providers`, a list of trusted OIDC providers
1: **procedure** Fulcio.RequestCertificate(pk, tok, chal)
2: **if** ¬OIDC.Verify(tok) or tok.iss ∉ providers **then**
3: **return** Err("OIDC signature invalid")
4: **if** ¬$\text{Verify}_{\text{pk}}$(tok.sub, chal) **then**
5: **return** Err("invalid self-signature: chal")
6: cert ← X509.NewCertificate(tok.iss, tok.sub, pk)
7: Sign_{sk}(cert)
8: Fulcio.PublishCertificate(cert)
9: **return** cert

signature into Rekor, a signer must provide both a public key and the signature itself. While this is not used by clients to correctly verify a signature, it is used to check for the entry's correctness. Note that while the signing verification procedure involves many steps and interactions with external systems, the interface for *end users* is simple: to sign using the `cosign` tool (see Section 7), a user simply invokes `cosign sign-blob` on the artifact and uses their browser to "log in to Sigstore" via OIDC.

These logs are architected for eventual consistency, and it may be hours before a submitted certificate appears in the log. To handle this case, since many artifacts are used shortly after publication, Rekor returns to users a *signed certificate timestamp* (SCT), which represents a timestamped promise (signed with the key in Rekor's root certificate) to incorporate the certificate into the log within a period known as the *maximum merge delay* (MMD). A typical MMD in certificate transparency is 24 hours. Sigstore has no published MMD, but currently only completes a request after it finishes merging a new log entry. These SCTs can act as signed timestamps to verify the signature generation time, instead of making an online query to the log. As in web PKI, clients accept SCTs indefinitely [16]).

4.2.3 Monitoring both logs. By moving certificates into the public view, any party can verify that Fulcio and Rekor act correctly. To do this, different actors within the Sigstore community run *monitors* that verify the log for consistency (that neither log creates a fork). The monitors can *also* look at semantic information within entries that may raise alarms. For example, monitors can take note of cases in which a signer changes OIDC providers, as this could be a case of somebody trying to impersonate them. Another example of such a check can be identifying typo-squatted identities (`user1234@gmail.com` and `user1234@outlook.com`) signing for different versions of the same package.

4.3 Verification Flow

In order to verify an artifact, a client runs Algorithm 3. First, a verifier ensures that the trusted keys are fresh by performing a TUF key update. With an updated root-of-trust, the first check ensures that the identity claimed was correctly verified by the identity log, and that the signature provided matches the content of the signed artifact. A final check verifies that the signature was created during the time window in which the certificate was valid.

To verify, a user runs `cosign verify-blob` and inspects the output to see the identity of the verifier (`cosign` also supports user-provided verification policies).

Algorithm 3 Verifier Flow

Require: `root`, a TUF root
1: **procedure** Sigstore.Verify(sig, artifact, cert)
2: ca ← TUF.UpdateTrust(root)
3: **if** ¬$\text{X509.Verify}_{\text{ca}}$(cert) **then**
4: **return** Err("invalid certificate")
5: **if** ¬Fulcio.VerifyInclusion(cert) **then**
6: **return** Err("certificate missing from Identity Log")
7: inclusion ← Rekor.VerifyInclusion(sig, artifact, cert)
8: **if** inclusion is Err **then**
9: **return** Err("signature missing from Artifact Log")
10: **if** inclusion.time ∉ cert.validityWindow **then**
11: **return** Err("invalid time for signature")
12: **if** ¬$\text{Verify}_{\text{cert.pk}}$(artifact, sig) **then**
13: **return** Err("invalid signature")

5 CASE STUDY: SECURING PACKAGE MANAGERS WITH SIGSTORE

Sigstore is a generic system for signing artifacts and linking those signatures to OIDC identities in an auditable way. Using the construction above, it is possible to build a system to provide package and repository signing with minor changes to existing infrastructure, and reducing the need for users to manage cryptographic material. In order to do so, some elements in Sigstore that were described in the abstract are replaced with concrete details; we also introduce another party, the Package Repository.

Though there is plenty of interest in providing signing capabilities for package managers, the uptake of these mechanisms in practice has been limited. For example, a 2020 analysis of RubyGems, which allows optional signing with SSL certificates, found that only 1.6% of the latest versions of packages were signed [84]. This may be caused by the difficulty of managing cryptographic material effectively, as well as the burden of operating certain cryptographic toolkits. This relatively small-scale adoption of PGP-based package signing mirrors the broader experience of PGP, which has struggled to achieve a wide user base [52, 78]. A 2016 analysis of PyPI found that only 4% of projects even list a signature and only 0.07% of users downloaded these signatures for verification [43]. Consequently, a system that allows for signatures without key management, like Sigstore, can popularize signing packages in this setting.

5.1 TUF Integration and Repository Federation

Sigstore acts as an *signing overlay* over existing package repositories so that packagers can sign artifacts without having to change existing infrastructure in popular package managers. However, client side tooling package managers (e.g., `pip` [63]) need to be Sigstore-aware so that can verify the signatures. When doing so, these tools need to check whether the subject name in the certificate generated is trusted to sign a particular package. This seems simple, as the package repository itself can provide the mapping. This, however, opens an avenue of attack in which a compromised package server lies about associations between package names and identities.

The traditional solution to this problem for software repositories is The Update Framework (TUF) [43, 67]. This provides protection

Table 1: Taxonomy of compromises and associated attacks when using Sigstore to publish to a package repository which implements The Update Framework (TUF).

Attacker controls:	Attacker can:			
	Forge signatures	Distribute packages	Reuse key[a]	Denial-of-service
Package repository	No	Yes	No	Yes
Rekor	No	No	Yes	Yes
Network (MITM)	Targeted user[b,c]	Targeted user	N/A	Targeted user
Signer	Signer only[b]	Signer only	N/A	Signer only
OIDC provider	*Some* identities[b,d]	No	N/A	Yes
Fulcio	*Any* identity[b]	No	N/A	Yes

[a] Given a previously-used signing key, can forge signatures.
[b] Signer/OIDC issuer can *detect* this activity in the log.
[c] With proposed OAuth 2.0 proof-of-possession enhancement [22], becomes "No."
[d] Any identity associated with this OIDC provider.

from compromised package repositories by requiring that packages be signed by their maintainers and securing the mapping from package name to maintainer. However, TUF requires that users (including package managers) manage their own keys. Instead, using Sigstore, TUF can be modified to delegate to *identities*, combining in the tamper-resistance of TUF and the convenience of Sigstore.

Rather than run their own instance of TUF, repository operators who already rely on Sigstore can use *repository federation* to root trust for their packages in the Sigstore ecosystem. A package repository is able to participate in an interactive protocol with the Sigstore infrastructure to claim a namespace within the Sigstore Artifact Log (e.g. `pypi.org/*`). Then, after a package repository claims a namespace, it can further delegate to particular packages names within the repository.[1]

Beyond being able to commit these username mappings to clients in the Sigstore ecosystem, this also allows the package repositories to *configure* the trust relationships between identity providers (e.g., by blocklisting or adding new identity providers), as well as signing repository metadata (so as to avoid freeze and mix-and-match attacks [12]). As a consequence, package repositories can use Sigstore to provide a signing overlay that provides TUF-like guarantees, while at the same time minimizing the need for packagers to maintain their own cryptographic material.

5.2 Sigstore user interviews

To complement the technical evaluations presented above, we also conducted interviews with seven individual users of Sigstore. Two of the individuals integrated Sigstore into the open source Kubernetes project, a popular container management platform. Another two individuals, both staff at Shopify, have been involved in a proposal to integrate Sigstore into RubyGems, a registry for Ruby programming language packages. The fifth individual has conducted design work and prototyping on a Java client for Sigstore. The sixth and seventh individuals are both open source software developers, creators of popular projects, and were only solely end-users of Sigstore (the other interviewees participated to varying extents as contributors to the project). All interviews were semi-structured, conducted via an online interview format and lasted between thirty minutes to an hour. We focused on what problems the user sought to solve with Sigstore, alternatives considered, and the interviewee's perceptions about the security benefits of Sigstore.

The individuals involved with Kubernetes initially sought to verify the base container images consumed by Kubernetes. These individuals were attracted to Sigstore over alternative signing solutions such as Notary [60] or PGP because of their view that Sigstore made signing relatively easy and the simplicity of key management for developers. To these individuals, many of the security benefits described elsewhere in this paper were secondary to immediate concerns about usability and speed of implementation.

The Shopify staff became interested in Sigstore as a means of protecting the open source software commons upon which their software development teams and business rely. These individuals identified the integrity of open source software packages and registries as a central concern, which eventually led to their proposal to integrate Sigstore into RubyGems, the Ruby package repository. Skeptical of PGP and notions such as web-of-trust, these individuals preferred to "trade a key management problem for an identity problem." In other words, Sigstore alleviated the traditional difficulty of binding an entity and a public key, leaving only the problem of deciding which identities to trust for a given software package. The interviewees also expressed strong support for Sigstore's use of a public ledger that "forces attackers to move in the open."

The software developer working on a Java client for Sigstore aims to provide tooling that makes integration of Sigstore into language ecosystems easier. While the developer noted that "getting rid of key management is nice," the interviewee was quick to emphasize that the security guarantees of Sigstore depend upon the security policies of downstream verifiers. The developer opined that Maven Central's current use of PGP "seems like it has reached the limit of what PGP was intended to be used for." The interviewee then expressed hope that Sigstore would enable a richer set of security policies related to artifact verification.

[1] Currently, federation in Sigstore uses the Secure Production Identity Framework for Everyone (SPIFFE) [73], which operates on similar principles to the construction described here, rather than using TUF directly.

The interview sample also included two open source software developers who participate in the Sigstore project only as end-users of the tools. One of these developers hoped to sign a container image he published and found the GitHub integration and keyless signing features of Sigstore attractive. Interestingly, this Sigstore user was relatively uncertain about the security benefits of digital signatures for software artifacts and yet still believed he "ought" to sign the image he publishes. The other open source software developer diligently explored alternative code signing approaches, concluding that acquiring a digital code signing certificate from a traditional certificate authority is onerous and expensive. For this user, Sigstore was much easier and had no direct costs. This user was also motivated by a need to help the Windows users of his open source project; these users sometimes faced issues from Windows Defender antivirus software when using the project he created and he hoped that Sigstore could one day help verify the integrity of his project and prevent these false positives (currently, the Windows antivirus software does not include Sigstore in its trust root).

In sum, factors such as ease of integration, reduced difficulties with key management, and the tight connection between identity and cryptographic material attract developers to use Sigstore over traditional package-signing solutions.

6 SECURITY ANALYSIS

In Section 3, we described a powerful adversary who can control Internet traffic between entities, as well as compromise actors in the ecosystem. However, we assume that attackers will not control servers for a long period of time, or control a majority of the servers on the Sigstore ecosystem at once.

With this in mind, we set forth to study the security properties of the system under normal operating conditions. After, we explore the impact of different types of compromises that are possible were an attacker to control the various elements of the system. Table 1 summarizes the following taxonomy of attacks.

6.1 Normal Operating Conditions

Under normal operating conditions, a verifier knows that a software package signed by a Sigstore identity is legitimate; the following is an argument that this guarantee holds:

First, an identity is trusted to sign a package if it controls that package's namespace within a repository. To verify this, a client performs a lookup for that package as in The Update Framework (TUF) [67]. Then, the client is convinced that the given identity is trusted to sign the package; otherwise, this violates the security of TUF delegations as presented by Kuppusamy et al. [43].

Then, the package is legitimate as long as the signature comes from a party who holds that identity. Suppose that some party who does not hold the identity can convince the client to accept their signature. Then, they must have produced a signature that validates with the public key in the certificate issued by Fulcio, so they must hold the corresponding private key (otherwise, this violates the security of the digital signature scheme). Similarly, the timestamp on the signature must be in the validity period of the certificate (as attested by a signed certificate timestamp from Rekor); this is possible only if the signature was submitted to Rekor during that window. Additionally, the certificate must have that identity as the subject, must be valid, and must be signed by Fulcio. This means that Fulcio must have issued them such a certificate (otherwise, this violates the security of the signature scheme for Fulcio's signatures). It does this only if presented with a valid OIDC identity token corresponding to the subject in the certificate. To be valid, this token must bear the signature of the OIDC provider. However, the OIDC provider only signs identity tokens for users that *do* authenticate with the corresponding identity; this is a contradiction.

Additionally, because the client checks both that the signature is present in the Artifact Log and that the certificate is present in the Identity log, any valid signature must have both corresponding entries. Even if a signer *does* control the corresponding identity, they cannot issue signatures for artifacts without detection.

Having settled the regular operation conditions, we move on to explore the cases in which different parts of the system are compromised.

6.2 Man-in-the-Middle Attacks

An attacker able to man-in-the-middle conversations is able to carry out attacks on different conversations that make up the protocol. If the attacker can do *both* of the following, they can publish packages as the user (this is analogous to the case of compromising the user themselves).

Signature submission flow. When a packager submits a package to the repository, an attacker could replace the package with a tampered one. However, the client and log will reject the package if the signature is invalid.

OIDC flow. An attacker able to intercept an OIDC token may be able to forge a single-short package signature (by requesting a certificate from Fulcio). A proposed enhancement to OAuth 2.0 [22] would eliminate this risk, by binding an OAuth token to the private key corresponding to the public key in the certificate.

6.3 Package Server Compromise

If the package repository is compromised, an attacker will be able to submit packages to the repository. But without the ability to generate signatures through the OIDC flow the packages will be rejected by the client. Instead, an attacker can leverage compromises to remove existing versions from the repository (e.g., to force users to use downgraded/vulnerable versions of a package). This, however, can be detected by verifying signatures generated in the log (i.e., by confirming that new signatures were generated that do not have a corresponding package in the repository). Further, by using The Update Framework (TUF) [67] in combination with Sigstore, clients can detect such attacks.

There is a risk if the identity provider and package server are controlled by the same party. For instance, GitHub is an identity provider in Sigstore. If GitHub is used to distribute releases as well, using GitHub as an identity provider does not provide much additional security against a compromise of GitHub (however, monitoring the transparency log can detect misbehavior).

6.4 Sigstore Compromise

Similarly, an attacker can compromise Fulcio or Rekor. By compromising the Artifact or Identity Log, an attacker can remove

Table 2: Source code statistics for tools in the Sigstore ecosystem.

Name	Description	Version	Languages	Lines of Code
Cosign	Client Libraries and Tools	1.8.0	Go	28,236
Fulcio	Identity Log and Certificate Authority	0.4.0	Go, Python	5,562
Rekor	Artifact Log	0.6.0	Go	16,675
maven-plugin	Maven Integration	1.0	Java	819

signatures for packages (e.g., to cause a denial of service). Both of these attacks are detectable, and can only cause denial of service attacks when cross-referencing information between the logs and the package repository.

A more concerning attack is such in which an attacker attempts to issue certificates for arbitrary OIDC providers in the Identity Log. However, this attack is detectable by the OIDC providers, who can cross-reference their logs and notice unaccounted-for identities in the log. Because OIDC provider public keys are widely available, future work could extend the protocol to allow for *any* monitor to detect unaccounted-for certificates.

If an attacker controls Rekor *and* a compromised one-time-use key from a legitimate user, it can reuse this key by injecting a signature in the transparency log at the time the certificate holding that key was valid. However, this attack is detectable by *any* monitor, as it requires modifying earlier entries in the log.

6.5 Package Maintainer/Publisher Compromise

It is possible that automated signing pipelines (e.g., through the GitHub action provider) may be compromised by an attacker. In such a case, an attacker should be able to carry out an OIDC flow, generate an arbitrary trusted key and submit a maliciously tampered version of the package. However, the compromise is limited to an individual run of the automated signer. This is because keys are one-shot and correspond to an individual OIDC flow. As such, future and previous package signatures (i.e., for other package versions) are not affected by this compromise.

An attacker could attempt to keep cryptographic material to sign future versions (as the certificates are trusted by the identity log). However, the signatures will be rejected after a short time window (e.g., 30 minutes). This is because the certificates are short-lived, and the signatures are only trusted if they were generated and logged within the appropriate time window.

Requiring thresholds of signatures for a package mitigates this attack, as the attacker would need to control more than one account.

6.6 OIDC Issuer Compromise

An attacker in control of an OIDC provider (e.g., Google or Microsoft) can impersonate any of their users. In this case, an attacker can not only generate arbitrary certificates for a user, but may also be able to authenticate on behalf of the user with the package repository. As such, an attacker may be able to submit tampered packages to the repository and generate signatures accordingly. In this case, Sigstore will not prevent or detect an attack. However, such attacks can be detected by the impersonated individuals who did not request certificates for their identity.

Multiple-vantage-point OIDC Verification. It is possible to mitigate this attack, however, by using a similar approach used by technologies like Let's Encrypt [10]. In this case, Sigstore can request not one OIDC flow, but *multiple* flows to authenticate a user. This is trivially performed at the protocol level (i.e., before admitting a certificate in the identity/packaging log), by requesting proof that the user can control multiple identities related to the same package (e.g., a GitHub login and a Google email). In this case, an attacker needs to compromise multiple OIDC providers at the same time, which is much more difficult.

7 IMPLEMENTATION AND EVALUATION

We evaluated Sigstore on three dimensions: (1) usage frequency of the public service, and how the service is used, (2) latency for signing and verification with Sigstore, and (3) scalability. To perform these evaluations, we gathered data from the public transparency log and undertook micro-benchmarking of Sigstore-associated tools.

We find that: (1) Sigstore adoption is increasing, especially for automated releases via GitHub actions, and (2) Sigstore scales to real-world loads and requires minimal client-side work.

7.1 Implementation

We implemented the Sigstore infrastructure (Fulcio and Rekor) and associated client tooling to allow for package signing on popular package repositories (see Table 2). In order to ease adoption, we built our primary client tool, `cosign`, to allow for signing and verifying container images without relying on repository federation. This tool is widely used within cloud ecosystems. This tool also includes the `policy-controller` webhook to verify the admission of container images to cloud deployments.

In order to build both transparency log implementations, we built upon the popular Trillian [29] framework for transparency logs. Trillian supports logs of arbitrary data; to add additional structure and validation, developers can implement *Trillian personalities*. We developed two *personalities* for Trillian, one for the Identity Log (Fulcio), and one for the Artifact Log (Rekor). We deployed public instances of both logs to allow users of `cosign` to sign arbitrary artifacts. We implemented OIDC integration in Fulcio with support for three providers: Google, GitHub, and Microsoft. We use 10 minutes as the default certificate lifetime.

Notice that while the Fulcio implementation is only ≈5,500 lines of code, Rekor is almost three times as large. Most of this code is in support of pluggable entry types for Rekor. This includes X.509 signing of packages, TUF metadata (as mentioned above), and in-toto metadata (for supply chain provenance), as well as more traditional Linux signing formats for Alpine Package Keeper (APK)

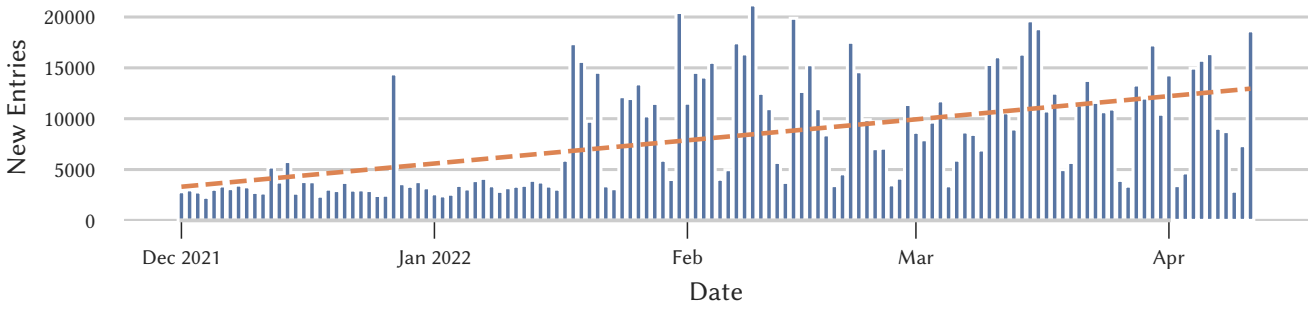

Figure 2: Daily count of new entries in the public Rekor log. December 1, 2021–April 11, 2022.

or RedHat Package Manager (RPM) packages. This code describes entry formats and sanitization (to ensure entry correctness), and should require minimal maintenance effort.

GitHub Actions Runner And Automated Signing Pipelines. In December 2021, GitHub published a mechanism to integrate `cosign` into release workflows using GitHub Actions. This interoperates with GitHub as an OIDC provider, and it allows for automated pipelines to sign and publish on behalf of users registered in their platform [39]. This approach links built artifacts to source code (by its commit hash), preventing attacks in which a malicious user uploads artifacts containing code that is not present in the public source repository. As we will see below, this is the most widespread mechanism used within the Sigstore ecosystem to sign packages.

7.2 Usage Frequency and Details

To understand the usage patterns of the public instance of the Rekor transparency log, we gathered data on the number of entries in Rekor, the daily usage pattern, and the relative percentage of different entry types. As of April 11, 2022, there were 1,969,836 entries in the public Rekor log. Figure 2 below provides a daily count and moving average of new entries in the Rekor log from December 1, 2021 to April 11, 2022.

Table 3 displays a breakdown of the type of entries found in Rekor as of April 11, 2022. Different entry types represent different types of claims about the associated artifact. For instance, the `in-toto` entry represents an attestation about an artifact using the in-toto framework, an approach to software integrity that emphasizes transparency. The `rfc3161` entry type represents an attestation that data existed before a particular time. For descriptions of the other types, see Table 3. Over 99% of entries are `rekord`, `hashedrekord`, and `intoto` types.

We also calculated the number of Fulcio signatures by OIDC identity provider. Identity providers are entities that authenticate a user. Each Fulcio entry represents a pairing of an identity (e.g. an email address) with a public key. The results are in the following table:

Issuer	**Usage (%)**
GitHub Actions	80.2
Google	17.5
Kubernetes	1.5
GitHub OAuth	0.7
Other	0.1

Rekor users can encode the public keys in their entries in a number of different *signature formats*, allowing interoperability with existing signing systems like OpenPGP or SSH. We calculated the percentage of `rekord` entries using different signature format types (as of April 11, 2022). Each format requires a specific set of data fields. The results (as of April 11, 2022) can be found in the following table:

Signature Format	X.509	minisign	PGP	SSH
Usage (%)	82.8	11.2	6.0	<0.0001

7.3 Performance

Signing and verification latency. The following table presents a breakdown of the time `cosign` spends on various phases of signing and verification. We measured these by modifying the `cosign` source to record timings at each stage; the times presented are an average over 10 trials on a developer workstation (32 GiB RAM, AMD Ryzen 5 5600G CPU) with about 55 ms round-trip ping times to the Sigstore infrastructure. The bulk of the time is spent updating TUF metadata, which (as in the specification) requires several round-trips to a remote repository. Remote calls to Fulcio and Rekor take well under a second. Local computation is negligible (about 1 ms total). We omit time performing OIDC sign-in (relevant only to signing), as this phase requires user interaction.

	Time (ms)			
Phase	Local	Fulcio	TUF	Rekor
Sign	0.5	397.2	1,154.0	509.7
Verify (online)	1.0	-	1,162.2	361.7
Verify (offline)	1.2	-	1,183.9	-

Table 3: Breakdown of Entry Types in the Public Rekor Log, as of April 11, 2022.

Type	Description	Count
`rekord`	Signature on arbitrary blob (includes full blob)	1,037,961
`hashedrekord`	Signature on arbitrary blob (only includes hash)	564,138
`intoto`	Attestation on artifact in the in-toto framework [74]	367,229
`rfc3161`	Response from RFC3161 [2] timestamping authority	187
`helm`	Signature on a package for Helm (Kubernetes package manager)	129
`tuf`	Signed TUF metadata and associated root-of-trust	99
`jar`	Signature on a Java Archive (JAR) package	78
`rpm`	Signature on a RedHat Package Manger (RPM) package	14
`alpine`	Signature on an Alpine Package Keeper (APK) package	1

Scalability. The Sigstore project performed a load test in August 2021 to test its ability to withstand high usage. The Rekor log received over 347,000 entries on August 18th. Per-hour usage peaked at 17,755 requests and per minute usage peaked at 437 requests. The log service remained healthy throughout the test.

7.4 Lessons Learned

The real-world deployment of Sigstore has surfaced a number of practical concerns.

First, making key management easier does not solve all identity problems. Users still need to figure out *which identities* are trusted to sign a particular artifact. It's important to discourage users from assuming an artifact is secure because it is signed at all, rather than signed by a specific, trusted party. Identities in Sigstore are more legible to humans than opaque public keys, but this is a weakness as well as a strength. Any workflow which requires a human to eyeball an identity in order to decide whether to trust it is vulnerable to typosquatting or type confusion attacks. Further, users must look beyond the "email" string and consider the "issuer" and other fields as well: a certificate associated with an `@gmail.com` address based on an OIDC login with Google as the provider is generally stronger than a similar certificate based on a login from another provider. We recommend that systems integrating with Sigstore also implement a system like The Update Framework (TUF) [67] to help manage the association between identities and artifacts (for instance, to specify which maintainers can sign a particular software package).

Like similar systems, Sigstore faces a tradeoff between transparency and privacy. Many code-signing certificates use emails or other personally identifiable information as the primary identifier. In an open-source setting, many users value anonymity, and would prefer *not* to reveal their email address, even if it is pseudonymous. This may be for fear of additional requests from the consumers of their software; enterprises have been known to demand that open-source maintainers perform security work for free [80]. Sigstore has implemented support for identities that *are not* associated with individuals, and instead refer to public information like the name of a GitHub repository; it also supports using traditional public keys in concert with its artifact log. Even if users consent to putting their information on a log now, they may later want to remove that information. Any system that hosts user-submitted data is also vulnerable to abusive content. Planned work will allow the removal of log entries, but this may impact the availability of the artifacts from those entries and may be a vector for denial-of-service attacks and censorship. Future work should explore ways to mitigate privacy concerns while providing the usability benefits of ephemeral keys.

Finally, there are practical deployment considerations. Requiring an OIDC login flow per artifact may not be appropriate in all settings. For instance, Java packages can have thousands of artifacts in a single release of a package: a developer would balk at logging in thousands of times in order to release a package, and even if credentials are cached, this imposes a large burst of load on the identity and transparency logs which may lead to performance issues. Fortunately, Rekor supports pluggable types, and new types have been proposed which consolidate signatures over many artifacts. Similarly, Sigstore has encountered many of the same scalability and availability concerns as certificate transparency in web PKI. An indefinitely-growing log leads to slower requests over time, since operations are super-linear in the number of entries; to deal with this, we implement temporal sharding, where logs are retired and replaced periodically (approximately annually). In most applications, the additional latency incurred by a roundtrip to the log server is prohibitive, so many clients choose to verify using Signed Certificate Timestamps (SCTs; see Section 4.2) instead.

8 RELATED WORK

Sigstore overlaps with various efforts in the systems security space. Research areas such as Public-Key Infrastructure (PKI), software supply-chain security, and identity-proof systems overlap with different components of Sigstore. In this section, we describe academic and industry state-of-the-art in the aforementioned areas.

8.1 Artifact Signing

Signed artifacts enable confidence in the integrity of software dependencies and related code artifacts such as build outputs. Past efforts from academic and industry include package signing, repository signing and package transparency.

Package signing. Currently, most software distribution pipelines (e.g., package repositories) support or plan to support a package signing solution. Perhaps the most widely-known approach for package signing is PGP. The Pretty Good Privacy (PGP) standards

and tooling, along with the GNU Privacy Guard (GPG) implementation and the related "web-of-trust" model allow users to use digital signing to enforce content integrity. Some open source software registries, such as Maven Central, a Java registry, have actually mandated the use of PGP in order to upload artifacts [84]. However, the majority of registries that support PGP signatures have made their use optional. Unsurprisingly, key elements of PGP have therefore been central to the debate on the merits of code signing, especially the usability and security assumptions of PGP itself, including the use of long-lived secret keys and the difficulty of manually finding and verifying developer keys [43]. While Sigstore does not replace PGP as a package signing solution, it allows developers to simplify their signing workflow by eliminating cryptographic key management.

Repository signing. Cappos et al. [12] identified limitations with individual package signing. In particular, attackers able to compromise a package repository are able to replace non-signed metadata (usually, repository state information). This allows attackers to effectively bypass package signing and force developers to install outdated, vulnerable dependencies, as well as force the installation of attacker-controlled malicious dependencies. A similar attack, shown by Torres-Arias et al. [75], can be carried out on Git repositories. Samuel et al. [67] introduced The Update Framework (TUF), the most widely deployed repository signing model. TUF has inspired open-source and industry designs, including Docker's Notary [60] as well as F-Droid [21], among others. Further designs for specific use cases, like automotive (Uptane [42, 76]) as well as IoT devices (SUIT [36]) have been proposed to manage TUF's limitations within individual ecosystems. While Sigstore does not present a repository signing mechanism, it uses TUF metadata to handle repository and package signing use-cases.

Several deployed repositories do require code signing, including OS package repositories (like those run for the Debian operating system), and mobile app stores (for both Android and Apple devices). However, in all such cases, developers are expected to manage their own keys and request certificates from the platform. None of these platforms supports short-lived certificates, so the risk associated with a leaked key is greater, or timestamping, so end users cannot check whether a given artifact was signed while the certificate was active. Finally, the root-of-trust is the same party that hosts the applications, which means that an app store or repository, if compromised, could substitute their own binaries.

Supply chain provenance signing & compliance. Beyond individual artifact and repository signing, there is a widespread push by industry [23], academia [3], and government [9, 77] to provide more insightful information about the how a software artifact is developed. Of particular importance, Torres-Arias et al. [74] introduced in-toto, which allows developers to embed certificates of supply chain operations (e.g., build provenance). This design has been widely adopted in industry settings such as Solarwind's Trebuchet [65], as well as Tekton Chains [25]. Other initiatives have attempted to perform automated compliance over this information. Of note, the Open Source Security Foundation's standards for Supply-chain Levels for Software Artifacts (SLSA) [24] require granular provenance data, like in-toto attestations, and make trust over an artifact contingent on the properties therein.

Other efforts in the space attempt to ensure the software-to-binary mapping is deterministic, which allows third parties to assess the quality of the software based on the properties of the corresponding software. Of note, the reproducible-builds project [44, 64] attempt to minimize the effects of host build systems in the output of a build. Navarro Leija's DetTrace [58] uses tracing mechanisms to observe build processes, identify and remove sources of "divergence". This enables systems such as Trebuchet (mentioned above).

8.2 Public Key Infrastructure and Key Management

Web PKI. The most commonly used PKI deployment is the system used to authenticate and encrypt traffic during web browsing and related communication, sometimes known as web PKI. In this model, a trusted third party called a *certificate authority* (CA) issues certificates attesting to the identity of the holder of a public key. Typically, this identity is a domain name in the Domain Name System (DNS); this is the approach favored by Let's Encrypt [1], a CA that uses a protocol called ACME (see Section 8.4) to automatically verify control of a domain. Let's Encrypt is, in fact, one of the main inspirations for Sigstore's design as a solution to increase uptake in package, repository and supply-chain signing. However, existing ACME system supports software signatures with ephemeral keys, as signatures are not timestamped.

The root of trust in web PKI is a set of certificates shipped with an OS or browser, typically referred to as a CA bundle. A consortium of CA and browser vendors, called the CA/Browser forum, establishes requirements and performs vetting for inclusion in standard bundles. Lowering barriers for domain owners to obtain certificates has enabled widespread adoption [1] including 94% of the Alexa top 1 million domains [38]. However, web PKI is a poor fit for artifact signing. First, identities in web PKI correspond to domains, and users without domains are unlikely to require web PKI certificates; for artifact signing, many individuals *do not* own relevant domains, and obtaining web PKI certificates represents a cost of both money and effort. Further, encrypting and signing web traffic with these certificates provides a straightforward guarantee: clients can trust that they are sending traffic to the domain operator. There is no such straightforward mapping for artifacts: a mirror might host artifacts from many different maintainers, each with their own identity, and domain identities do not help clients make a decision about whether to trust the artifact. Finally, web PKI certificates are time-bound: they expire after a fixed period of time (for instance, Google's certificates are valid for only three months). However, software artifacts may go indefinitely without a release. Expiration requires maintainers to periodically resign, with consequences for security and availability in the case of lapsed signatures, lost keys, or transfer of maintainership. Further, domain names are permitted to transfer between users, and a new owner could sign malicious artifacts associated with a domain (a *resurrection attack*).

Certificate transparency (CT). Trust in web PKI hinges on the certificate authorities. In several high-profile instances, CAs have issued improper certificates due to malice or failures in procedure [16]. To manage the risk, Laurie et al. [46] propose *certificate transparency*: publishing all issued certificates to an append-only *transparency log* that can be publicly verified. Then, if clients reject

certificates that are not in the log, bad actors can only use certificates available publicly, enabling detection of misbehaving CAs. In other words, CAs can look for certificates they did not issue, and domain owners can monitor for certificates they did not request.

Parties called *monitors* perform verification of these public logs. The log servers publish short digests of the log, constructed using Merkle trees [57], which allows end users to efficiently check that entries *are* in the log, and monitors to check that new digests contain *all* of the records from previous digests. Then, misbehavior by logs can be detected; for instance, monitors caught a cosmic-ray bit flip [5] in a log operated by DigiCert. Currently, monitors and log operators use mailing lists to coordinate, but ideally monitors would implement a full-fledged gossip protocol (see Meiklejohn et al. [54] for one proposal).

Since its proposal, many web PKI participants have adopted certificate transparency. A non-standards-track IETF document (RFC9162 [47]) describes protocols for CT. In aggregate, deployed CT logs contain 1.5B certificates for 100M domains [33]. Enough CAs are posting all certificates to CT logs that Chrome has started rejecting new certificates without corresponding entries and Safari also requires CT log submission in some cases.

Web of trust. In contrast to web PKI, PGP supports a relatively horizontal "web-of-trust" model. In such a model, users can endorse each others' keys as belonging to specific identities: for instance, after Alice has verified Bob's keys in person, if she sees Bob's attestation that a public key belongs to Charlie, Alice can trust this key. However, in the dependency management setting, Alice may want to use a package published by Dave even if Dave is a perfect stranger and not connected to her "web." This has shown some limitations in handling package signing in major Linux distributions [12, 83]: decentralized web-of-trust networks do not allow for canonical ownership of packages in a namespace. Package manager adoption of PGP tooling typically does not rely on web-of-trust and instead places the responsibility for key management on repositories. This, in turn, places complications in system integrators, as they are required to remove web-of-trust semantics from PGP-enabled signing solutions using custom tooling.

The limited uptake of package signing and verification has seen relatively low exploration. In the wild, the user experience difficulties of using PGP and the tenuous security assumptions of PGP are well documented [32, 45, 68, 82]. These PGP-specific critiques likely account for a significant portion of the difficulties that package signing has faced in gaining wide adoption. While a PGP package signing user study has, to our knowledge, not been undertaken, our anecdotal conversations with open source software developers suggest four particularly important issues. First, the usability of PGP tools is widely perceived as low. Second, package signing has, like PGP, generally required the creation and storage of long-lived private keys. Ensuring this key remains private places a demanding responsibility on users and creates security risks when a private key is lost. Third, determining the true public key associated with a developer requires significant manual work and, unfortunately, is far from fool-proof given that attackers can distribute fake keys [43]. Fourth, the security benefits of tying an identity to an artifact could be perceived as uninformative without additional attestations related to the provenance and properties of the artifact [74]. In other words, the exact meaning of a signature has been hard to ascertain in most formulations of artifact signing. This should not be a surprise, however, as the limitations of PGP for *encryption* have been widely explored [68, 82]. Of note, in the past decade practical tools for code signing (other than PGP) and conceptual alternatives to PGP for artifact signing have emerged [18, 60].

Automatic enrollment and delegation mechanisms. Similar attempts to provide automatic key-management and authentication have been proposed. For example, Kumar et al. proposed JEDI [41] to manage automatic issuance and validation of device identity on dense smart-device deployments (e.g., a smart building). It uses standard X.509 certificate issuance and transparency logs to enroll devices, issue certificates in a distributed setting and communicate key information efficiently. Similarly, CONIKS [55] is a robust key transparency system that attempts to provide a "best of both worlds" design between regular web-of-trust and web PKI approaches. Instead of using device-based identities, or self-generated identity claims, Sigstore leverages common proof-of-identity protocols to allow signers to show that they are who they claim to be.

8.3 Transparency for Software Artifacts

Several recent systems propose using transparency techniques for software artifacts. Mozilla's binary transparency [6], for Firefox binaries, is designed but not implemented. This design inspired `f-droid` [21] to make a third-party "transparency log" (a Git repository, which provides some of the same guarantees) for Google releases of the Android SDK. Linderud [50] proposes a system similar to Trebuchet that uses in-toto evidence in a transparency log to verify build reproducibility. Nikitin et al.'s CHAINIAC [59] uses a new structure called a *skipchain* and reproducible builds to ensure a source-to-binary mapping. `pacman-bintrans` [40], for Arch Linux packages, uses Rekor internally. All of the above are in a single-publisher setting, and don't support community contributions. Further, none address the problem of linking identities to keys or support ephemeral signing keys.

8.4 Identity Management

Like all computer security-related systems, Sigstore must also deal with the notion of identity, by defining a computer's representation of a unique entity and binding that representation to an identity internal to the computer [11, p. 211]. Identity has long been a staple of computer security research [8, 19]. To tie a user's identity to corresponding cryptographic material, Sigstore could use any of the building block technologies described below (though we limited the initial implementation to OIDC).

Automatic Certificate Management Environment (ACME). Automatic Certificate Management Environment (ACME) [7] is a suite of "challenges" for proving ownership of a domain. These challenges can be based on HTTP, DNS, or TLS, and are used by Let's Encrypt to automatically issue HTTPS certificates to hundreds of millions of domains without user interaction [1]. As such ACME is perfectly suited to prove the ownership of a particular *live* domain. However, for other types of identities (e.g., a user account), ACME may not be suitable. An extension to ACME [56] allows email-based challenges

for issuing end-user S/MIME certificates; however, they are not *ephemeral*, as there is no authority to timestamp signed artifacts.

OIDC. In contrast to ACME, OpenID Connect (OIDC) can perform identity mapping by an *identity provider*. That is, an individual server can vouch for the fact that an individual controls an account with the given "subject." For example, a Facebook server can *claim* to know who owns a particular email address (e.g., because they engaged in email registration). By engaging in an OIDC protocol, a third party server (e.g., Sigstore/Fulcio) can confirm such claims about a particular email address or GitHub username. To our knowledge, no system uses OIDC identities for signing certificates. There is precedent for very short-lived SSH certificates [69] based on OIDC, but these cannot be used for signing artifacts unless they are long-lived or they have been timestamped by additional service. OIDC is a rather restrictive protocol, and it does not allow for more generalizable identity claims. For example, web3 (i.e., blockchain-based) identity claims may not fully fit the model originally designed for major web platforms.

DIDs. In contrast with OIDC, Decentralized Identifiers (DIDs) [70] generalize the generation of identity claims by any provider. To avoid these restrictions, it allows a loosely-federated protocol in which identity providers can register their own semantics. As a consequence, the aforementioned and otherwise hard-to-manage providers may provide identity mappings by brokering information on a blockchain to an identity claim transparently. While Sigstore uses OIDC at this moment due to its widespread deployment (including support by Microsoft, GitHub, Meta, Google, and Apple), replacing OIDC with DIDs would be trivial.

9 DISCUSSION AND FUTURE WORK

Although Sigstore is a mature, production-ready system, there are plenty of directions in which the ecosystem may grow. In this section, we present alternative building blocks that Sigstore may leverage, as well as future extensions to the system.

9.1 Privacy Concerns

A limitation of Sigstore as presented in this paper is privacy: an identity needs to be made public in order to sign for an artifact. Although this expectation is standard in many popular package managers, there is a realistic concern that a signer's email will be published in a tamper-evident log. As this may affect privacy conscious users, it may affect adoption of Sigstore in privacy-sensitive communities. In prior work, Eskandarian et al. [20] use zero-knowledge proofs to protect the privacy of *users* submitting information to a transparency log, and allow for logs to contain private entries for private use. However, this solution does not support private entries for *public* use, as is the case in the private package-signing setting. A possible solution to tackle this challenge is the use of cryptographic techniques to hide the *identity* of the signer, but convince verifiers of the *continuity* of the identity. We note that simply using symmetric signing keys achieves this, but gives privacy-conscious users the burden of key management. Future work related to privacy might also use pairwise pseudonymous identifiers, an OIDC feature that gives users a long-lived but pseudonymous identity for Sigstore. Another construction, which we reserve for future work, uses zero-knowledge proofs and authenticated data structures to prove signature compliance with TUF-like delegations.

9.2 Beyond Package Managers

We presented Sigstore as a general-purpose artifact signing solution not exclusive to package manager. Sigstore can be used as-is (without the ephemeral signing flow) in any setting that currently uses symmetric package signatures to provide *transparency* (e.g., Arch Linux [40] or signing container images on DockerHub). Similarly, any ecosystem that currently uses symmetric signatures can adopt Sigstore to base those signatures on *identities*, rather than keys which must be managed. Other possible target ecosystems can be automated delegation mechanisms, similar to those presented in JEDI [41], as well as logging automated process (e.g., APIs or serverless pipelines).

10 CONCLUSIONS

This paper introduced Sigstore, a mechanism to provide a PKI-like system that associates identity providers with short-lived cryptographic material. Through this system, it is possible to create a transparency log-backed signing repository with minimal friction for integration, while maintaining reasonable security guarantees. Much like Let's Encrypt, Sigstore allows increased adoption for package signing, using centralized-but-accountable (using transparency) infrastructure, as in web PKI, to reduce the risk and burden of key management for users. While Sigstore already boasts wide adoption among community repositories, further enhancements (e.g., integration with Distributed Identifiers) remain an open question for future work.

ACKNOWLEDGMENTS

Building a universally available, open source, and all-encompassing code signing platform is not the endeavor of a couple of people, but the tireless labor of hundreds of passionate community members. While it is infeasible to fit all these names within the page limit, we would like to highlight some of the community members that helped us drive this work to fruition. First, we would like to thank Luke Hinds, Dan Lorenc, and Bob Callaway, as well the whole Sigstore community for their hard work making Sigstore the exciting and successful software platform that it is. Likewise, we would like to thank Priya Wadhwa, Simon Kent and Hayden Blauzbern for their work on the Sigstore General Availability project, as well as Asra Ali for maintaining the root of trust infrastructure, and members of the end-user libraries such as `sigstore-python` (Willian Woodruff, Dustin Igram, and Jussi Kukkonen), `-rust` (Flavio Castelli, Lily Sturmann, and Victor Cuadrado), `-java` (Appu Goundan, Patrick Flynn, Vladimir Sitnikov), and Cosign (Carlos Panato, Matthew Moore, and Batuhan Apaydın). Other notable people that keep the community thriving are Tracy Miranda and Lisa Tagliaferri. We would also like to extend a special thank you to the end users that participated in the interviews needed for these paper. We're also grateful to Nathan Smith for his extremely helpful creation of a Rekor dashboard. Lastly, we thank the anonymous reviewers, as well as our shepherd for their insightful feedback.

REFERENCES

[1] Josh Aas, Richard Barnes, Benton Case, Zakir Durumeric, Peter Eckersley, Alan Flores-López, J Alex Halderman, Jacob Hoffman-Andrews, James Kasten, Eric Rescorla, et al. 2019. Let's Encrypt: an automated certificate authority to encrypt the entire web. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, London, UK, 2473–2487.

[2] C. Adams, P. Cain, D. Pinka, and R. Zuccherato. 2001. *Internet X.509 Public Key Infrastructure Time-Stamp Protocol.* Technical Report. Internet Engineering Task Force. http://tools.ietf.org/html/rfc3161

[3] Bilal Al Sabbagh and Stewart Kowalski. 2015. A socio-technical framework for threat modeling a software supply chain. *IEEE Security & Privacy* 13, 4 (2015), 30–39.

[4] The Update Framework authors. 2022. TUF. https://github.com/theupdateframework/tuf.

[5] Andrew Ayer. 2021. Yeti 2022 not furnishing entries for STH 65569149. https://groups.google.com/a/chromium.org/g/ct-policy/c/PCkKU357M2Q/. Accessed: 2022-04-30.

[6] Richard Barnes. 2017. [meta] Binary Transparency on Firefox. https://bugzilla.mozilla.org/show_bug.cgi?id=1341395.

[7] R. Barnes, J. Hoffman-Andrews, D. McCarney, and J. Kasten. 2019. *Automatic Certificate Management Environment (ACME).* Technical Report. Internet Engineering Task Force. http://tools.ietf.org/html/rfc8555

[8] Steven M. Bellovin. 2010. Identity and Security. *IEEE Security & Privacy* 8, 2 (March–April 2010), 88–88. https://www.cs.columbia.edu/~smb/papers/cleartext-2010-02.pdf

[9] Joseph R. Biden Jr. 2021. Executive Order on Improving the Nation's Cybersecurity. Accessed: 2022-04-30.

[10] Henry Birge-Lee, Liang Wang, Daniel McCarney, Roland Shoemaker, Jennifer Rexford, and Prateek Mittal. 2021. Experiences Deploying Multi-Vantage-Point Domain Validation at Let's Encrypt. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, Virtual Event, 4311–4327. https://www.usenix.org/conference/usenixsecurity21/presentation/birge-lee

[11] Matt Bishop. 2004. *Introduction to Computer Security.* Addison-Wesley Professional, Boston.

[12] Justin Cappos, Justin Samuel, Scott Baker, and John Hartman. 2008. A Look in the Mirror: Attacks on Package Managers. In *The 15th ACM Conference on Computer and Communications Security (CCS '08)* (Alexandria, Virginia). ACM, New York, NY, USA, 565–574.

[13] Chia-ling Chan, Romain Fontugne, Kenjiro Cho, and Shigeki Goto. 2018. Monitoring TLS adoption using backbone and edge traffic. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, IEEE, Honolulu, HI, 208–213.

[14] Timothy Chen. 2018. One Billion Domains. https://www.domaintools.com/resources/blog/one-billion-domains. Accessed: 2022-04-30.

[15] Cloud Native Computing Foundation. 2021. *Software Supply Chain Best Practices.* Technical Report. Cloud Native Computing Foundation.

[16] Mozilla Developer Network Contributors. 2022. Certificate Transparency. https://developer.mozilla.org/en-US/docs/Web/Security/Certificate_Transparency. Accessed: 2022-04-30.

[17] M. Cooper, Y. Dzambasow, P. Hesse, S. Joseph, and R. Nicholas. 2005. *Internet X.509 Public Key Infrastructure: Certification Path Building (RFC 5280).* Technical Report. Internet Engineering Task Force. http://tools.ietf.org/html/rfc5280

[18] Frank Dennis. 2015. Minisign. https://jedisct1.github.io/minisign/. Accessed: 2020-04-30.

[19] Carl M Ellison. 2000. Naming and certificates. In *Proceedings of the tenth conference on Computers, freedom and privacy: challenging the assumptions*. ACM, Toronto, Canada, 213–217.

[20] Saba Eskandarian, Eran Messeri, Joseph Bonneau, and Dan Boneh. 2017. Certificate Transparency with Privacy. *Proc. Priv. Enhancing Technol.* 2017, 4 (2017), 329–344. https://doi.org/10.1515/popets-2017-0052

[21] F-Droid. 2022. Security Model. https://f-droid.org/docs/Security_Model/. Accessed: 2022-04-30.

[22] D. Fett, B. Campbell, J. Bradley, T. Lodderstedt, M. Jones, and D. Waite. 2022. *Internet Draft: OAuth 2.0 Demonstrating Proof-of-Possession at the Application Layer (DPoP).* Technical Report. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-oauth-dpop

[23] The Linux Foundation. 2019. The Linux Foundation's Automated Compliance Work Garners New Funding, Advances Tools Development. https://www.linuxfoundation.org/press-release/the-linux-foundations-automated-compliance-work-garners-new-funding-advances-tools-development/. Accessed: 2022-04-30.

[24] The Linux Foundation. 2022. SLSA: Supply-chain levels for software artifacts. https://slsa.dev. Accessed: 2022-04-30.

[25] The Linux Foundation. 2022. Tekton. https://tekton.dev. Accessed: 2022-04-30.

[26] Simson L Garfinkel and Robert C Miller. 2005. Johnny 2: a user test of key continuity management with S/MIME and Outlook Express. In *Proceedings of the 2005 symposium on Usable privacy and security*. ACM, Pittsburgh, PA, 13–24.

[27] Dan Geer, Bentz Tozer, and John Speed Meyers. 2020. For good measure: Counting broken links: A quant's view of software supply chain security. In *USENIX; Login:, Vol. 45, no. 4.* USENIX Association, Berkeley, California.

[28] Dan Goodin. 2011. Fedora servers breached after external compromise. *The Register* (January 2011). https://www.theregister.com/2011/01/25/fedora_server_compromised/. Accessed: 2022-04-30.

[29] Inc. Google. 2016. Trillian. https://github.com/google/trillian. Accessed: 2022-04-30.

[30] Google, Inc. 2019. Go Module Mirror, Index, and Checksum Database. https://sum.golang.org/. Accessed: 2022-04-30.

[31] Patrick Gray. 2003. Gentoo Linux server compromised. https://www.zdnet.com/article/gentoo-linux-server-compromised/. *ZDNet* (2003). Accessed: 2022-04-30.

[32] Matthew Green. 2014. What's the Matter with PGP? https://blog.cryptographyengineering.com/2014/08/13/whats-matter-with-pgp/. Accessed: 2020-04-30.

[33] Kaspar Hageman, René Rydhof Hansen, and Jens Myrup Pedersen. 2021. Gollector: Measuring Domain Name Dark Matter from Different Vantage Points. In *Nordic Conference on Secure IT Systems*. Springer, Springer, Virtual Event, 133–152.

[34] D. Hardt. 2012. *The OAuth 2.0 Authorization Framework.* Technical Report. Internet Engineering Task Force. http://tools.ietf.org/html/rfc6749

[35] Scott Helme. 2021. Top 1 Million Analysis – November 2021. https://scotthelme.co.uk/top-1-million-analysis-november-2021/. Accessed: 2022-04-30.

[36] José L Hernández-Ramos, Gianmarco Baldini, Sara N Matheu, and Antonio Skarmeta. 2020. Updating IoT devices: challenges and potential approaches. In *2020 Global Internet of Things Summit (GIoTS)*. IEEE, IEEE, Dublin, Ireland, 1–5.

[37] Luke Hinds. 2021. Sigstore: An open answer to software supply chain trust and security. https://www.redhat.com/en/blog/sigstore-open-answer-software-supply-chain-trust-and-security. Accessed: 2022-04-30.

[38] Ralph Holz, Johanna Amann, Abbas Razaghpanah, and Narseo Vallina-Rodriguez. 2019. The Era of TLS 1.3: Measuring Deployment and Use with Active and Passive Methods. *CoRR* abs/1907.12762 (2019). arXiv:1907.12762 http://arxiv.org/abs/1907.12762

[39] Justin Hutchings. 2021. Safeguard your containers with new container signing capability in GitHub Actions. https://github.blog/2021-12-06-safeguard-container-signing-capability-actions. Accessed: 2022-04-30.

[40] kpcyrd. 2021. pacman-bintrans. https://github.com/kpcyrd/pacman-bintrans. Accessed: 2022-04-30.

[41] Sam Kumar, Yuncong Hu, Michael P Andersen, Raluca Ada Popa, and David E. Culler. 2019. JEDI: Many-to-Many End-to-End Encryption and Key Delegation for IoT. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1519–1536. https://www.usenix.org/conference/usenixsecurity19/presentation/kumar-sam

[42] Trishank Karthik Kuppusamy, Akan Brown, Sebastien Awwad, Damon McCoy, Russ Bielawski, Cameron Mott, Sam Lauzon, André Weimerskirch, and Justin Cappos. 2016. Uptane: Securing Software Updates for Automobiles. *14th ESCAR Europe* (2016).

[43] Trishank Karthik Kuppusamy, Santiago Torres-Arias, Vladimir Diaz, and Justin Cappos. 2016. Diplomat: Using delegations to protect community repositories. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX, Santa Clara, CA, 567–581.

[44] Chris Lamb and Stefano Zacchiroli. 2021. Reproducible Builds: Increasing the Integrity of Software Supply Chains. *CoRR* abs/2104.06020 (2021). arXiv:2104.06020 https://arxiv.org/abs/2104.06020

[45] Latacora, LLC. 2019. The PGP Problem. https://latacora.micro.blog/2019/07/16/the-pgp-problem.html. Accessed: 2020-04-30.

[46] Ben Laurie, Adam Langley, and Emilia Kasper. 2013. *Certificate Transparency.* Technical Report. Internet Engineering Task Force. http://tools.ietf.org/html/rfc6962

[47] Ben Laurie, E. Messeri, and R. Stradling. 2021. *Certificate Transparency Version 2.0.* Technical Report. Internet Engineering Task Force. http://tools.ietf.org/html/rfc9162

[48] Renai LeMay. 2008. Fedora reboots updates after hack. https://www.zdnet.com/article/fedora-reboots-updates-after-hack/. *ZDNet* (2008). Accessed: 2022-04-30.

[49] Wanpeng Li and Chris J Mitchell. 2016. Analysing the Security of Google's implementation of OpenID Connect. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, Springer, San Sebastián, Spain, 357–376.

[50] Morten Linderud. 2019. *Reproducible Builds: Break a log, good things come in trees.* Master's thesis. The University of Bergen.

[51] Christian Mainka, Vladislav Mladenov, Jörg Schwenk, and Tobias Wich. 2017. SoK: single sign-on security—an evaluation of openID connect. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, IEEE, Paris, France, 251–266.

[52] Moxie Marlinspike. 2015. GPG and me. https://moxie.org/2015/02/24/gpg-and-me.html. Accessed: 2022-04-30.

[53] David Mazières and Dennis Shasha. 2002. Building secure file systems out of Byzantine storage. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing* (Monterey, California). ACM, New York,

NY, USA, 108–117. https://doi.org/10.1145/571825.571840
[54] Sarah Meiklejohn, Pavel Kalinnikov, Cindy S. Lin, Martin Hutchinson, Gary Belvin, Mariana Raykova, and Al Cutter. 2020. Think Global, Act Local: Gossip and Client Audits in Verifiable Data Structures. *CoRR* abs/2011.04551 (2020). arXiv:2011.04551 https://arxiv.org/abs/2011.04551
[55] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. 2015. {CONIKS}: Bringing Key Transparency to End Users. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, DC, 383–398.
[56] A. Melnikov. 2021. *Extensions to Automatic Certificate Management Environment for end-user S/MIME certificates*. Technical Report. Interenet Engineering Task Force. http://tool.ietf.org/html/rfc8823
[57] Ralph C Merkle. 1987. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*. Springer, Springer, Berlin, 369–378.
[58] Omar S. Navarro Leija, Kelly Shiptoski, Ryan G. Scott, Baojun Wang, Nicholas Renner, Ryan R. Newton, and Joseph Devietti. 2020. *Reproducible Containers*. Association for Computing Machinery, New York, NY, USA, 167–182. https://doi.org/10.1145/3373376.3378519
[59] K. Nikitin, L. Kokoris-Kogias, P. Jovanovic, N. Gailly, L. Gasser, I. Khoffi, J. Cappos, and Ford. B. 2017. CHAINIAC: Software-Update Transparency via Collectively Signed Skipchains and Verified Builds. In *26th USENIX Security Symposium (USENIX Security '17)*. USENIX Association, Vancouver, BC, Canada, 1271–1287.
[60] Notary Project. 2015. Notary. https://github.com/notaryproject/notary. Accessed: 2022-04-30.
[61] OpenID Foundation. 2014. OpenID Connect. https://openid.net/connect/. Accessed: 2022-04-30.
[62] Justin Pagano. 2021. Securing the Supply Chain: Lessons Learned from the Codecov Compromise. https://www.rapid7.com/blog/post/2021/07/09/securing-the-supply-chain-lessons-learned-from-the-codecov-compromise/. Accessed: 2022-04-30.
[63] Python Packaging Authority (PyPA). 2022. Project Summaries. https://packaging.python.org/en/latest/key_projects/. Accessed: 2022-04-30.
[64] Reproducible Builds. 2015. Reproducible Builds: A set of software development practices that create an independently-verifiable path from source to binary code. https://reproducible-builds.org/. Accessed: 2022-04-30.
[65] Reuters Staff. 2021. SolarWinds hack was 'largest and most sophisticated attack' ever: Microsoft president. https://www.reuters.com/article/us-cyber-solarwinds-microsoft/solarwinds-hack-was-largest-and-most-sophisticated-attack-ever-microsoft-president-idUSKBN2AF03R. Accessed: 2022-04-30.
[66] Scott Ruoti, Jeff Andersen, Tyler Monson, Daniel Zappala, and Kent Seamons. 2018. A comparative usability study of key management in secure email. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*. USENIX Association, Baltimore, MD, 375–394.
[67] Justin Samuel, Nick Mathewson, Justin Cappos, and Roger Dingledine. 2010. Survivable Key Compromise in Software Update Systems. In *The 17th ACM Conference on Computer and Communications Security (CCS '10)* (Chicago, IL). ACM, Chicago, IL.
[68] Steve Sheng, Levi Broderick, Colleen Alison Koranda, and Jeremy J Hyland. 2006. Why johnny still can't encrypt: evaluating the usability of email encryption software. In *Symposium On Usable Privacy and Security*. ACM, ACM, Pittsburgh, PA, 3–4.
[69] Smallstep Labs, Inc. 2020. smallstep/certificates: A private certificate authority. https://github.com/smallstep/certificates. Accessed: 2022-08-22.
[70] Manu Sporny, Dave Longley, Markus Sabadello, Drummond Reed, Orie Steele, and Christopher Allen. 2021. *Decentralized Identifiers (DIDs) v1.0*. Technical Report. World Wide Web Consortium (W3C).
[71] SSLMate. 2022. Certificate Transparency Log Growth. https://sslmate.com/labs/ct_growth/. Accessed: 2022-04-30.
[72] Donald Stufft. 2016. PyPI and GPG Signatures. https://mail.python.org/pipermail/distutils-sig/2016-May/028933.html Accessed: 2022-09-07.
[73] The SPIFFE authors. 2022. Secure Production Identity Framework for Everyone. https://spiffe.io/. Accessed: 2022-04-30.
[74] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. 2019. in-toto: Providing farm-to-table guarantees for bits and bytes. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1393–1410.
[75] Santiago Torres-Arias, Anil Kumar Ammula, Reza Curtmola, and J Cappos. 2016. On omitting commits and committing omissions: Preventing Git metadata tampering that (re) introduces software vulnerabilities. In *25th USENIX Security Symposium (USENIX Security '16)*. USENIX Association, Austin, TX, 379–395.
[76] Uptane Alliance. 2018. Uptane – IEEE-ISTO 6100.1.0.0 Uptane Standard for Design and Implementation. https://uptane.github.io/papers/ieee-isto-6100.1.0.0.uptane-standard.html.
[77] U.S. Department of Commerce and U.S. Department of Homeland Security. 2022. *Assessment of the critical supply chains supporting the U.S. information and communications technology industry*. Technical Report. U.S. Department of Commerce and U.S. Department of Homeland Security.
[78] Filippo Valsorda. 2016. I'm throwing in the towel on PGP, and I work in security. *Ars Technica* (2016). https://arstechnica.com/information-technology/2016/12/op-ed-im-giving-up-on-pgp/ Accessed: 2022-04-30.
[79] Steven Vaughan-Nichols. 2015. Red Hat's Ceph and Inktank code repositories were cracked. https://www.zdnet.com/article/red-hats-ceph-and-inktank-code-repositories-were-cracked/. *ZDNet* (2015). Accessed: 2022-04-30.
[80] Luis Villa. 2021. Setting new expectations for open source maintainers. https://opensource.com/article/21/8/open-source-maintainers. Accessed: 2022-08-12.
[81] Priya Wadhwa and Appu Goundan. 2021. Distroless Builds are now SLSA 2. https://security.googleblog.com/2021/09/distroless-builds-are-now-slsa-2.html. Accessed: 2022-04-30.
[82] Alma Whitten and J Doug Tygar. 1999. Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0.. In *USENIX security symposium*, Vol. 348. USENIX Association, Washington, DC, 169–184.
[83] Debian Wiki. 2005. SecureApt. http://wiki.debian.org/SecureApt. Accessed: 2022-04-30.
[84] Tieg Zaharia. 2020. The state of package signing across package managers. https://blog.tidelift.com/the-state-of-package-signing-across-package-managers. Accessed: 2022-04-30.
[85] Henry Zhu. 2018. Postmortem for Malicious Packages Published on July 12th, 2018. https://eslint.org/blog/2018/07/postmortem-for-malicious-package-publishes. Accessed: 2022-04-30.