

Establishing Provenance Before Coding: Traditional and Next-Generation Software Signing

Taylor R. Schorlemmer¹ | U.S. Army

Ethan H. Burmane², Kelechi G. Kalu³, Santiago Torres-Arias⁴, and James C. Davis⁵ | Purdue University

The integration of third-party components leaves the software supply chain vulnerable. Cryptographic signatures enable verifying the origin of components (provenance) before they are added. This article describes traditional software signing, its challenges, and the changes introduced by next-generation software signing platforms.

Nearly all modern software relies on other software components.¹ As illustrated in Figure 1, these components include libraries, operating systems, build tools, and deployment tools. As part of the planning process, software engineers decide which components they will use to build their software. These dependencies (and the components on which they depend in turn) create a software supply chain with implied trust relationships between software components and their users. When software engineers decide which components to use in their software, they are also deciding which components to trust.

Many recent cybersecurity attacks have exploited these trust relationships by targeting software components and supply chains.¹ In response, many researchers have proposed enhancements to software supply chain

security. Okafor et al.'s literature review summarized those proposals as addressing three distinct properties: separation to ensure that failures in one component are isolated, transparency to see the full supply chain, and validity to show that components are not changed unexpectedly (integrity).² Taken together, these latter two properties can describe the provenance of both an individual software component and of the resulting supply chain.

This article focuses on one specific technique for software component provenance: software signing. Software signing using public-key cryptography is the de facto method for assuring the origin of an artifact. Although signing is an old concept, there have been many works showing that using traditional approaches to software signing is easier said than done. However, next-generation signing platforms have emerged in recent years that improve usability.

In this article, we summarize software supply chains, traditional methods for software signing, and

Digital Object Identifier 10.1109/MSEC.2025.3537616

Date of publication 14 February 2025; date of current version 17 March 2025.

the transformative aspects of next-generation software signing platforms. We discuss the current state of software signing and describe how improvements to signing infrastructure are leading to better security practices. Our goal is for the reader to learn that, due to the changing software signing landscape, their component selection process can, and should, integrate provenance information.

A few years from now, everybody expects provenance. ... They don't install anything if they can't prove where it came from. ... Signing for open source is probably the single most important thing for software supply chain security.—Technical leader^a

Trust Before Coding

Components Influence Product Code

Before writing code, software engineers consider the components they will use to build their software. One of the primary decisions is this: Which external components should we build on? Figure 1 depicts the kinds of components on which software depends, aligned with the software factory model.⁴ The first-party code in a software product may interact with external libraries—which ones? It may rely on a framework or an operating system for interaction with the outside world—what is the resulting control flow, and which parts will be handled by the substrate? When being built, should it make use of Ant or Gradle or Bazel? When being deployed, should it rely on Docker, Podman, or Singularity as its virtualization scheme?

Each of these decisions affects the resulting first-party code. The selected libraries and substrates have an obvious effect: the code will call one application programming interface or another. More subtle influences are exercised downstream. Build tools may influence which language features are used, e.g., whether the product can make use of GNU Compiler Collection extensions to C or must rely on the C90 standard. Deployment tools may influence the parameterization of the system, e.g., the size of buffers and caches (and which of these knobs to expose).

Since dependency selection is a weighty matter, engineers evaluate candidate components along many dimensions. Functional considerations, such as correctness and performance, have long dominated this selection process. However, the rise of cyberattacks has made cybersecurity, and particularly the security risks of depending on external components, a top-of-mind concern.

Provenance in Software Supply Chains

Provenance is a security property that combines validity (data and actor integrity) and transparency. Provenance simultaneously demonstrates that the code has not been modified and gives evidence of the entity in control of the code at the time of release.^{2,10}

In the past, assessing the provenance of a component was left to the judgment of the engineering team. Now, the urgency of cyberattacks has driven greater top-down control to improve software supply chain security. Security requirements related to component provenance are being introduced through industry standards and government regulations. Relevant industry standards include the Linux Foundation's Supply Chain Levels for Software Artifacts,⁴ CNCF's Software Supply Chain Best Practices, and Microsoft's Secure Supply Chain Consumption Framework. The U.S. government has also published standards, such as the National Institute of Standards and Technology's (NIST's) Secure Software Development Framework and NIST 800-204D, along with regulations for contractors handling government contracts (U.S. executive order 14028⁵).

While security techniques typically ensure distinct properties (e.g., software attestations ensure transparency, and software signing ensures validity), combining security techniques is often preferred in practice to reinforce one or more of these properties, with provenance being one such combination. An example of a combination used to establish provenance involves signed attestations, which integrate validity (ensuring data and actor integrity) and transparency (identifying the entity in control at signing). The software bills of materials (SBOMs) is another method for establishing transparency that can be used in place of attestations. Next, we briefly summarize these methods to establish provenance for software components.

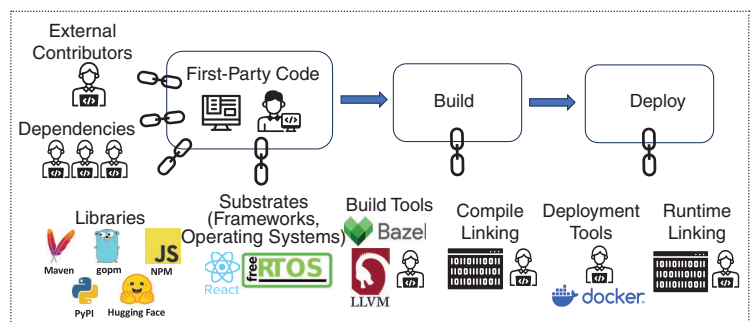


Figure 1. Depiction of software factory model to highlight external dependencies. Each chain symbol represents part of the software supply chain involved in producing software. Each component comes from some individual or organization. Provenance techniques enable downstream users to verify the origins of the components on which they depend.

^a This and other quotes are taken from interviews of industry cybersecurity experts conducted by Kalu et al. in 2025.⁶

Software signing. Software signing is a method of verifying the origin of software artifacts. Provenance techniques primarily rely on signing (e.g., digital signatures on source code, binaries, or even git commits) to ensure the integrity of both the actor and the data within components. Downstream users can verify the signature to ensure that the software artifact came from an expected source. Like other provenance techniques, code signing does not make any quality guarantees about the software's correctness (it might still have bugs!), but it does provide a way to verify the origin of a piece of software.

SBOMs. SBOMs contain a list of all the components that make up a piece of software. This list often includes the version of each component and the origin of the component. SBOMs can be used by downstream users to understand the risks associated with a piece of software. Furthermore, SBOMs can be used to ensure that a piece of software is built with the correct components. SBOMs can be signed to prove that the manifest has not changed.

Attestations. Attestations are pieces of metadata that describe some aspect of a software artifact. For example, build attestations describe how a piece of software was built, and review attestations indicate that a particular person has reviewed a piece of code, a component, a process, and so on. Typically, these attestations are signed to ensure their integrity. This enables users to verify that the metadata have not been tampered with.

In this article, we focus on the first of these techniques that underpin provenance: software signing. Like all security practices, signing is not a silver bullet. Although signing does not guarantee the correctness of a piece of software, it does provide a way to verify the origin of a piece of software and that it has not changed. Most industry standards highlighted earlier emphasize the importance of signing, which is prominently featured in their recommendations. For example, NIST 800-204D requires that attestations about software products (e.g., their bills of materials/SBOMs, build processes, and so on) be cryptographically signed with a secure key. NIST standards further require that consumers of software components verify these attestations before using them. We review some of the shortcomings of traditional signing methods and how newer techniques are attempting to resolve these issues. We first describe the foundations of software signing: public-key cryptography.

Primer on Public-Key Cryptography

Public-key cryptography, also known as *asymmetric cryptography*, refers to the subset of cryptographic methods

that utilize a pair of public and private keys. The advantages of these methods, such as scalability for a large number of users and the elimination of the need for a secure channel, enable various applications (e.g., signing) where symmetric cryptography falls short. This section gives a brief primer; see the “References” section for a starting point on this topic.

Key Terms and Concepts

In a public-key cryptographic system, an application or user can generate a pair of public and private keys. The private key is kept secret, and the public key is shared. Data encrypted using one key can be decrypted using the other. This leads to two basic use cases: 1) holders of the public key encrypt messages that can be read only by holders of the private key, or 2) holders of the private key sign messages (typically by appending an encrypted hash of the message) that can be verified by holders of the public key. In either case, it is important to establish trust in the public key (i.e., that the public key is from the correct source and has not been modified). Methods such as public-key infrastructure and the web of trust have been used to strengthen this trust.

With respect to provenance, the second use case is especially important. After establishing trust in the public key, signatures provide strong guarantees about the origin of a message. This scheme is used by several computing applications, detailed next.

Applications in Computing

Public-key cryptography is widely used in computing. For example, Secure/Multipurpose Internet Mail Extensions (e-mail communication), Secure Sockets Layer/Transport Layer Security (network communication), Secure Shell (SSH) (remote network access), and even smart cards use forms of public-key cryptography for secure communication and/or authentication. Given the topic of this article, our primary interest is on the application of public-key cryptography to ensuring provenance in the software supply chain. For this reason, we focus on software-related signing tools (which also use public-key cryptography), such as Pretty Good Privacy (PGP) and Sigstore. Let's take a look!

Traditional Software Signing

Many platforms, such as software registries, support “traditional” signing as a way for engineers to verify the origin of the software components they rely on. Traditional signing is relatively simple, but it has drawbacks. When used correctly, traditional signing methods can cryptographically ensure the origin of an artifact. Unfortunately, these methods suffer from poor adoption and key management issues that can limit practical use.

Identity verification. Establishing trust in a public key is also particularly challenging. Once a user creates a key pair, they must find a way to share their public key. This is typically done through infrastructure like public key servers or through other file sharing means. Unfortunately, many of these methods do not provide strong identity guarantees for the creator of the public key. The web of trust was introduced as a way to introduce a more confident binding between an identity and a public key. There are, however, issues with this technique (i.e., relying on social interactions like signing parties).

Transparency. Additionally, there are transparency issues with traditional signing schemes. Users typically do not have knowledge about which artifacts were signed or what information should be present. Without knowledge about what modifications have been made to a project's metadata, users cannot verify that supply chain actors are behaving properly. For example, a downstream user may not be able to detect that the signature identity has changed in a project.

Current State

The current state of signing varies depending on the type and environment of the software being developed. In general, signing is more prevalent in commercial software than in open source software. This indicates that provenance is lacking in some open source environments.

Open source. In open source, the state of signing is generally poor. This is examined in Figure 3. The number of user-created signatures on high-profile platforms like Docker Hub, PyPI, and Hugging Face has historically been low. There are many potential reasons for this, but a simple explanation is that platforms do not require signatures.³ This is particularly evident when comparing Maven Central (a platform that requires signing) to the rest in Figure 3.

Regardless of the reason, the small amount of signing on these platforms decreases the effectiveness of signing. Since so few people sign their projects, verifying signatures on a project that relies on several dependencies is very hard to do. Furthermore, this lack of verification dissuades maintainers from going through the hassle of creating and managing signatures. PyPI is a prime example of this: after years of poor adoption, the registry maintainers decided to stop supporting PGP. In these circumstances, establishing provenance is hard.

The case is a bit different in ecosystems that require signing. In instances like the Debian project or on platforms like Maven Central, contributors must sign and maintain keys. This means that other users of those ecosystems can verify provenance. Unfortunately, it also means that users of these platforms have to deal with the drawbacks of traditional signing schemes.

Commercial software. In commercial software, the implementation and utilization of signing are significantly more pronounced. Recent surveys^{8,9} indicate that organizations value signing (and, therefore, provenance). However, the cost of setting up a signing infrastructure is a general concern. Although there is diversity in the choice of tools, the perceived importance, and the challenges encountered, the adoption of signing as a practice remains widely recognized and actively employed.

Next-Generation Software Signing

Sigstore comes in ... keyless signing ... you don't have to worry about long-lasting static SSH keys, or keys being compromised, or rotating keys. ... You don't have big servers to store all these signatures—Manager [Sigstore's] other strength ... I can associate my identity with an OIDC identity as opposed to necessarily needing to generate a key and keep track of that key and yada, yada. So that's superuseful because I could say, "Oh, this is signed with [a] GitHub identity." So, unless my GitHub identity has been compromised, that's much better.—Member of senior management

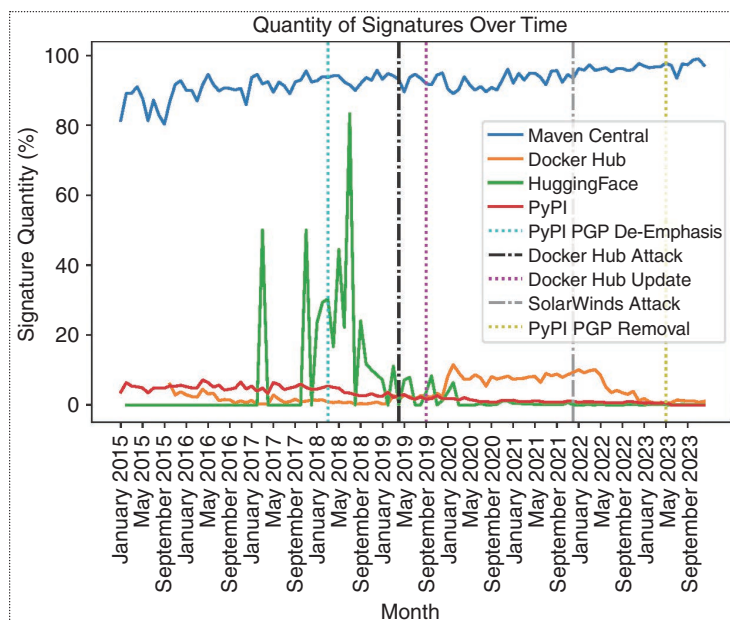


Figure 3. Software signing trends in four public software package registries. Figure from Schorlemmer et al.³

In the past several years, many structural improvements have been made to signing schemes. We call the result *next-generation signing tools*.

Changes to Signing

Next-generation signing tools are motivated by the challenges associated with traditional signing methods, such as key management, identity verification, and transparency. These tools still use some of the primitive constructs (e.g., cryptographic algorithms) of traditional signing. However, they change the signing workflow depicted in Figure 2 to address shortcomings in traditional signing tools.

Key management. One way to address key management issues is to use ephemeral keys. Ephemeral keys are temporary keys generated for a single signing session and discarded afterward. They are created using secure key generation algorithms, with the private key securely generated, used for signing, and discarded after a single use within a validity window. The corresponding public key is then bound to the signer's verified identity, ensuring authorship, and made available for signature verification. These keys do not need to be stored and managed by a user. This reduces the burden on users to ensure that they follow best practices for keys over their lifetime.

Identity verification. Linking a public key to an identity is a difficult problem. Unlike traditional signing methods that rely on public key infrastructure, the web of trust model, or even offhand verification (e.g., by meeting with the keyholder in person during a conference), next-generation signing tools leverage single sign-on protocols like OpenID Connect (OIDC), Security Assertion Markup Language, and Open Authorization (OAuth). Next-generation signatures bind ephemeral public keys to an account managed by an identity provider. For example, using the OAuth 2.0 protocol, a signing system might verify that a user has access to an account through an identity provider (e.g., Google). After verifying that the user has access to an account, the signing system binds an ephemeral public key (i.e., issues a certificate) to that account. This process links each signature to a verified identity, ensuring that it can be confidently traced back to the signer, thereby strengthening both security and trust.

Shift toward key and software transparency. Transparency allows verifiers to ensure that key issuers do not misbehave (e.g., by handing out a key to another party). Beyond allowing for easy key management for users, next-generation signing systems integrate transparency for keys issued and signatures created. This transparency most often takes the form of a public append-only logs for signatures and identity key bindings. This allows users to actively monitor signatures and ensure that they are not being withheld or reused.

How Next-Generation Signing Works

Compiling these changes to signing, we can see the general form of next-generation signing tools. Figure 2(b) shows an example protocol for next-generation software signing.

Components. There are several components in a next-generation signing system. Generally, the following services are used:

- An identity provider verifies that users have access to an account.
- A certificate authority (CA) acts as a trust root and binds public keys to accounts verified by identity providers. It issues a certificate to verify the binding.
- A certificate log keeps a public append-only copy of certificates issued by the CA to different accounts. This allows an account holder to verify that only authorized certificates have been issued.
- A signature log keeps a public append-only copy of signatures and their associated certificates. This allows verifiers to ensure that signatures were created during the lifespan of ephemeral keys and that the public key is valid.

Each of these services runs independently. As a result, users can check for the compromise of one component by auditing the others. For example, if an attacker attempted to modify an entry in the signature log, users could detect that either the certificate had changed (i.e., by referencing the certificate log) or that the signature had changed (i.e., by checking the signature with the public key embedded in the certificate).

Workflow. First, an author creates a package (A) and chooses one of the next-generation signing tools. The tool then follows a supported authentication protocol to redirect the signer to an identity provider of their choice (B). After successful authentication, the identity provider issues an identity token to the signing tool (C). This token serves as nonrepudiable proof that the user authenticated successfully. The tool then generates an ephemeral key pair (D).

The signer's identity, along with the public key, is bound through a short validity certificate issued by a CA, or alternatively, it may be bound directly within the issued token (E). While the specific method of binding may vary, the main goal is to link the public key to an identity. If a certificate is used, it is published to a public log (F). Once the identity binding is complete, the private key is used to sign the package during the short lifetime of the certificate (G), creating a signature whose authorship is verifiable.

To verify the software, a user retrieves the signed software, along with the certificate and the expected signer's identity. First, the certificate is verified against the root CA and key chain (H). Next, the certificate is checked against the certificate transparency log (I). The bound identity in the certificate is then compared with the expected identity. If all checks pass, the public key is retrieved from the certificate and used to verify the signature on the package. If all validations are successful, the user has confidence in the origin of the package.

Examples of Next-Generation Signing Platforms

Various systems have been engineered to address the challenges of traditional signing in the last couple of years. Perhaps one of the most visible ones is the Sigstore project, which simplifies signature creation and verification. Sigstore is an open source and free-to-use project that is supported by the Linux Foundation and several other organizations. To achieve this, Sigstore leverages existing standards, such as OIDC, for identity verification as well as transparency logs for software and key transparency.

Similarly to Sigstore, the Open PubKey (OPK) project provides a similar construction (identity providers to issue single-use ephemeral keys). However, it tries to minimize infrastructure requirements. Even though OPK is designed to be applicable to other use cases (e.g., SSH authentication using identity providers and end-to-end encrypted chat), it is also applicable for next-generation software signing. Currently, OPK is used by Docker-Hub to sign its official images.¹⁴ A generalization of this

construction is also being developed by standards organizations. Of particular interest, the Internet Engineering Task Force's Proof of Issuer Key Authority can be applied to the same use cases as Sigstore and OPK.¹³

Challenges

Although next-generation signing tools address many shortcomings of traditional signing methods, some challenges remain. Even when using next-generation signing tools, users must still decide whom to trust, watch for anomalous behaviors, and decide what to do when signature checks fail.

Identity verification. Next-generation signing techniques rely on identity providers (e.g., Google and GitHub) to bind accounts with ephemeral public keys, but users must still decide which identities to trust. Signatures indicate only where the software came from, not whether the producer is trustworthy. If users decide to trust a bad actor, they might receive signed, but malicious, software artifacts.

Transparency. Next-generation signing still requires active analysis on the behalf of its users. Tools like transparency logs make this job easier, but users still need to watch for malicious behavior. For example, if identity providers are compromised (e.g., leaked/stolen credentials), then malicious signatures can be created. This requires the signer to actively monitor certificate and signature logs for unauthorized behavior and continuously check for account breaches.

Signature checks. Some software artifacts may still not have signatures even if next-generation signing is available. This is already a problem in current package registries where signing is not required.³ Unfortunately, this leaves the question of what to do with unsigned dependencies and bad signatures. Without widespread adoption in an ecosystem, the value of signing is greatly depreciated, and without a security policy for signatures, any benefit they provide may go unnoticed.

Current State

Next-generation software signing solutions have experienced a significant rise in popularity since their introduction. Industry (e.g., companies like Shopify, Autodesk, and Verizon) and open source (e.g., the python interpreter and Kubernetes) organizations alike have integrated these schemes to establish provenance.^{11,12} These successes have led to public recognition from government strategies, such as the U.S. Cybersecurity and Infrastructure Security Agency's "Improving Security of Open Source Software in Operational Technology and Industrial Control Systems," as well as industry organizations,

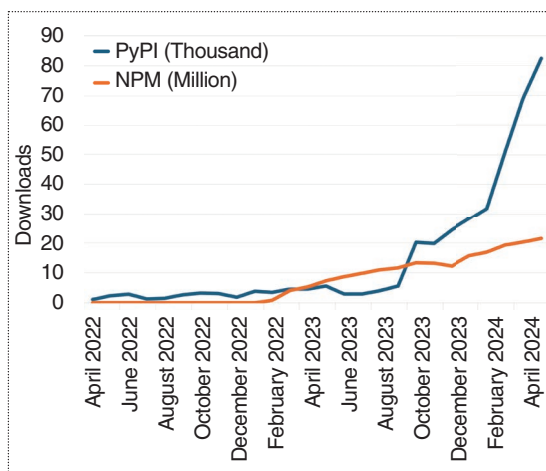


Figure 4. The Sigstore project has tooling available in several programming languages. This chart shows the downloads over time of the associated package for the Python PyPI package (blue, in thousands) and the JavaScript npm package (orange, in millions).

such as JPMorgan’s chief information security officer. However, concrete measurements of adoption of these technologies are lacking, and thus, it is difficult to fully assess whether this publicity is affecting adoption.

As a step toward quantifying the adoption status of next-generation signing tools, we looked at trends related to one of these tools, Sigstore. Our goal was to estimate its usage. We examined two aspects of usage: download rates of the Sigstore tool kit and analyzing the use and discussion of the tools and associated techniques, such as npm’s provenance feature,^b which use Sigstore under the hood.

To estimate the adoption rate of next-generation signing tools, we obtained the download statistics for various Sigstore-related tooling over the past 24 months. Two Sigstore tools were checked: npm and PyPI. The PyPI download data were collected using the PyPI database, as queried through Google’s BigQuery. The npm download data were collected using npm-stat. Figure 4 summarizes our results. Both PyPI and npm versions of the tooling are seeing increasing use, measured in the tens of thousands to millions of downloads over time.

To observe the usage and discussion of next-generation signing tools, we used Sourcegraph’s Code Search to query GitHub repositories for keywords relating to the tools. The results of these keyword searches are reported in Table 1. These keywords were determined based on package names (the top half of the table) and indicators of provenance workflows within repositories (the bottom half of the table).

Both measures (Figure 4 and Table 1) indicate increasing interest in Sigstore, the next-generation signing tool that we examined.

Software supply chains are not new, but they are more visible than ever before. Software signing—of code, of attestations, of bills of materials, and of all other aspects of consuming a software component—is a key technique in ensuring that supply chain artifacts have not been modified since they were authored by trusted parties. Although the previous generation of signing tools left much to be desired, the next generation of tools has addressed many of those challenges and is seeing widespread adoption.

The technological capability is now present for any component provider to easily sign its products. Software signature creation and verification are already commonplace. While selecting components, software engineers can and should rely only on components bearing the signature of the author. Conversely, software producers should expect that their consumers will

Table 1. Estimates of the use of next-generation signing tools via keyword searches of open source repository files.

Approach	Measure	Mentions	Repository files
Sigstore	Name	143,830	5,229
Sigstore, Python	Name	1,041	396
Sigstore, Java	Name	194	21
Sigstore, Rust	Name	235	23
In-toto	Name	68,752	985
NPM provenance workflow	GitHub Workflow	37,363	26,255
NPM repository publishing with provenance	GitHub Workflow	1,427	896

The method was as follows. For the tools in the top half of the table, we searched for the tool name as a string, anywhere in the project. For the tools in the lower half of the table, we checked flags or fields in GitHub workflow files that indicate the use of Sigstore. In npm provenance workflows, we checked whether the “id-token” field was set to “write.” For npm publishing with provenance, we checked for the “-provenance” flag.

begin to demand these signatures, as a result of due diligence or the need to meet current or anticipated regulations regarding software provenance. Soon, checking signatures before coding will be a normal part of component selection.

Finally, we caution the reader that provenance—even cryptographically assured provenance using signatures—does not guarantee correctness or security. Provenance information reduces the risk of certain classes of attacks, such as man-in-the-middle substitutions. Provenance information also improves the engineer’s ability to estimate the quality of the components they rely on, e.g., based on the reputation of the supplier. But guaranteeing the correctness and security of a component (not to mention the resulting system) is a task for formal methods, which is another endeavor for computing. ■

Acknowledgment

We thank Zach Steindler, of GitHub, for his feedback on our estimates of npm provenance adoption, and Chinenye Okafor for her input on the design of next-generation signing systems. This work was supported by the U.S. National Science Foundation, under Award 2229703, and by funds from Google and Cisco.

References

1. “9th annual state of the software supply chain,” Sonatype, Fulton, MD, USA, 2023. [Online]. Available: <https://www.sonatype.com/hubfs/SSC/2023%20Sonatype-%209th%20Annual%20State%20of%20the%20Software%20Supply%20Chain-%20Update.pdf>

^b See <https://docs.npmjs.com/generating-provenance-statements>.

2. C. Okafor, T. R. Schorlemmer, S. Torres-Arias, and J. C. Davis, "SoK: Analysis of software supply chain security by establishing secure design properties," in *Proc. ACM Workshop Softw. Supply Chain Offensive Research Ecosystem Defenses (SCORED)*, 2022, pp. 15–24, doi: [10.1145/3560835.3564556](https://doi.org/10.1145/3560835.3564556).
 3. T. R. Schorlemmer et al., "Signing in four public software package registries: Quantity, quality, and influencing factors," in *Proc. IEEE Symp. Secur. Privacy (S&P)*, 2024, pp. 1160–1178, doi: [10.1109/SP54263.2024.00215](https://doi.org/10.1109/SP54263.2024.00215).
 4. The Linux Foundation. "What is SLSA? (Supply-chain Levels for Software Artifacts)." SLSA. Accessed: Jun. 17, 2024. [Online]. Available: <https://slsa.dev>
 5. J. R. Biden Jr., "Executive order 14028: Improving the nation's cybersecurity," *The White House*, May 12, 2021. [Online]. Available: <https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/>
 6. K. G. Kalu, T. Singla, C. Okafor, S. Torres-Arias, and J. C. Davis, "An industry interview study of software signing for supply chain security," in *Proc. USENIX Secur. Symp.*, 2025, pp. 1–20.
 7. A. Whitten and J. D. Tygar, "Why Johnny can't encrypt: A usability evaluation of PGP 5.0," in *Proc. USENIX Secur. Symp.*, 1999, pp. 169–183.
 8. P. Ladisa, H. Plate, M. Martinez, and O. Barais, "SoK: Taxonomy of attacks on open-source software supply chains," in *Proc. IEEE Symp. Secur. Privacy (S&P)*, 2023, pp. 1509–1526, doi: [10.1109/SP46215.2023.10179304](https://doi.org/10.1109/SP46215.2023.10179304).
 9. D. A. Wheeler, J. S. Meyers, M. Barbero, and R. Rumbul, "New SLSA++ survey reveals real-world developer approaches to software supply chain security," *OpenSSF*, Mar. 15, 2023. [Online]. Available: <https://openssf.org/blog/2023/03/15/new-slsa-survey-reveals-real-world-developer-approaches-to-software-supply-chain-security/>
 10. D. Cooper et al., "Security considerations for code signing," NIST Cybersecurity White Paper 5, 2018. [Online]. Available: <https://www.nist.gov/publications/security-considerations-code-signing>
 11. Z. Newman, J. S. Meyers, and S. Torres-Arias, "Sigstore: Software signing for everybody," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2022, pp. 2353–2367.
 12. K. Merrill, Z. Newman, S. Torres-Arias, and K. R. Sollins. "Speranza: Usable, privacy-friendly software signing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2023, pp. 3388–3402, doi: [10.1145/3576915.3623200](https://doi.org/10.1145/3576915.3623200).
 13. R. L. Barnes and S. Goldberg. "Proof of issuer key authority (PIKA)." IETF. Accessed: Jul. 1, 2024. [Online]. Available: <https://www.ietf.org/archive/id/draft-barnes-oauth-pika-00.html>
 14. J. Stoten. "Signing Docker official images using OpenPubkey." Docker. Accessed: Jun. 20, 2024. [Online]. Available: <https://www.docker.com/blog/signing-docker-official-images-using-openpubkey/>
 15. "OpenPubkey." BastionZero. Accessed: Jun. 18, 2024. [Online]. Available: <https://www.bastionzero.com/openpubkey>
-
- Taylor R. Schorlemmer** is a lieutenant in the U.S. Army, West Lafayette, IN 47907 USA. His research interests include software engineering and cybersecurity, with an emphasis on software signing and software supply chains. Schorlemmer received his M.Sc. in computer engineering from Purdue University. He is a Member of IEEE. Contact him at tschorle@purdue.edu.
-
- Ethan H. Burmane** received a B.Sc. in computer engineering from the Elmore Family School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907 USA. His research interests include software engineering and cybersecurity. He is a Member of IEEE. Contact him at eburmane@purdue.edu.
-
- Kelechi G. Kalu** is a Ph.D. student at the Elmore Family School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907 USA. His research interests include software engineering and cybersecurity, with a focus on software supply chains. Kalu received his B.Sc. in electronic and computer engineering from Nnamdi Azikiwe University, Nigeria. He is a Member of IEEE. Contact him at kalu@purdue.edu.
-
- Santiago Torres-Arias** is an assistant professor at the Elmore Family School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907 USA. His research interests include cybersecurity, cryptography, and computer systems, with an emphasis on software supply chain security. Torres-Arias received his Ph.D. in computer science from New York University, USA. He is a Member of IEEE and the Association for Computing Machinery. Contact him at santiagotorres@purdue.edu.
-
- James C. Davis** is an assistant professor at the Elmore Family School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907 USA. His research interests include human and technical aspects of software engineering and cybersecurity, with an emphasis on the analysis of failures. Davis received his Ph.D. in computer science from Virginia Tech, USA. He is a member of the Association for Computing Machinery and a Senior Member of IEEE. Contact him at davisjam@purdue.edu.