SOFTWARE SUPPLY CHAIN SECURITY: ATTACKS, DEFENSES, AND THE ADOPTION OF SIGNATURES

by

Taylor R. Schorlemmer

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Master of Science in Electrical and Computer Engineering

School of Electrical and Computer Engineering

West Lafayette, Indiana

May 2024

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. James C. Davis, Chair
School of Electrical and Computer Engineering

Dr. Santiago Torres-Arias
School of Electrical and Computer Engineering

Dr. Saurabh Bagchi
School of Electrical and Computer Engineering

Approved by:
Dr. Milind Kulkarni

For His glory alone.

ACKNOWLEDGMENTS

Thank you to the many hands that have helped me along the way. I could not have done this without you.

To my friends: Thank you for the support and the reprieve you have provided. Getting my mind off of research was as important as the research itself.

To my family: This work is a product of the love and lessons you have given me.

To the Purdue Military Research Institute: Thank you for the organizational support you provided along the way. I always felt like I was a part of a larger team of like-minded military professionals.

To the Duality Lab (and friends thereof): Thank you for the camaraderie and the intellectual stimulation. You have motivated me to work harder and to think more deeply about the problems we face in research and beyond.

To my committee: Thank you for your guidance, feedback, and the knowledge you have shared with me.

To my advisor, James C. Davis: Thank you for inviting me to your lab and for your mentorship. You have taught me how to think more critically and how to communicate more effectively. I am better for having worked with you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Modern software relies heavily on third-party dependencies (often distributed via public package registries), making software supply chain attacks a growing threat. Prior work investigated attacks and defenses, but only taxonomized attacks or proposed defensive techniques, did not consistently define software supply chain attacks, and did not provide properties to assess the security of software supply chains. We do not have a unified definition of software supply chain attacks nor a set of properties that a secure software supply chain should follow.

Guaranteeing authorship in a software supply chain is also a challenge. Package maintainers can guarantee package authorship through *software signing*. However, it is unclear how common this practice is or if existing signatures are created properly. Prior work provided raw data on registry signing practices, but only measured single platforms, did not consider quality, did not consider time, and did not assess factors that may influence signing. We do not have up-to-date measurements of signing practices nor do we know the quality of existing signatures. Furthermore, we lack a comprehensive understanding of factors that influence signing adoption.

This thesis addresses these gaps. First, we systematize existing knowledge into: (1) a four-stage supply chain attack pattern; and (2) a set of properties for secure supply chains (transparency, validity, and separation). Next, we measure current signing quantity and quality across three kinds of package registries: traditional software (Maven Central, PyPI), container images (Docker Hub), and machine learning models (Hugging Face). Then, we examine longitudinal trends in signing practices. Finally, we use a quasi-experiment to estimate the effect that various factors had on software signing practices.

To summarize the findings of our quasi-experiment: (1) mandating signature adoption improves the *quantity* of signatures; (2) providing dedicated tooling improves the *quality* of signing; (3) getting started is the hard part — once a maintainer begins to sign, they tend to continue doing so; and (4) although many supply chain attacks are mitigable via signing, signing adoption is primarily affected by registry policy rather than by public knowledge of attacks, new engineering standards, etc. These findings highlight the importance of software package registry managers and signing infrastructure.

1. INTRODUCTION

1.1 Context and Problem Statement

Commercial and government software products incorporate open-source software packages [1], [2]. In a 2023 study of 1,703 commercial codebases across 17 sectors of industry, Synopsys found that 96% used open-source code, and 76% of the total application code was open-source [3]. Open-source software packages depend on other packages, creating *software supply chains* [4]. Malicious actors have begun to attack software supply chains, injecting malicious code into packages to gain access to downstream systems [4]. These attacks have affected critical infrastructure and national security [5]–[8].

The software engineering community has investigated how software supply chains can be attacked. Attack taxonomies help us understand how attackers compromise supply chains [9]; ecosystem analyses help us understand how the structure of software dependencies can make us more or less vulnerable when selecting external dependencies [10]–[12]; and data-science based efforts in the industry and academia have attempted to identify signals or indicators of compromise [13]. However, we still lack a unified definition of software supply chain attacks. Furthermore, we lack a comprehensive set of security properties that a secure software supply chain should have. This knowledge would guide future efforts to secure software supply chains.

The software engineering community has also proposed techniques to secure software supply chains. Some approaches seek to increase confidence in a package's *behavior*, *e.g.,* measuring use of best practices [14], [15], independent validation [16], and formal guarantees [17]; Other approaches target the package's *provenance*, *e.g.,* Software Bill of Materials (SBOMs) [18], [19] and "vendoring" trusted copies of dependencies [20]. The strongest guarantee of a package's provenance is a cryptographic signature by its maintainer. Prior work has noted that many packages are unsigned [21], [22]. However, we lack up-to-date measurements of software signing practices, and we do not know the general quality of existing signatures. Furthermore, we lack a deeper understanding of factors that affect adoption rates. This knowledge would guide future efforts to incentivize software signing, so that the provenance of software supply chains can be improved.

In light of these gaps, this thesis investigates two main topics:

- Chapter 3: What are the characteristics of software supply chain attacks and what security properties should a secure software supply chain have?
- Chapter 4: What are the current practices of software signing in public package registries and what factors influence software signing?

1.2 Thesis Statement

Since modern software relies increasingly on third party dependencies, software supply chain attacks are a growing, and challenging, threat. As a result, techniques like software signing must be effectively implemented to provide defensive benefits.

1.3 Contributions

This thesis makes two classes of contributions: (1) a systematization of existing knowledge into an attack pattern and a set of properties for secure software supply chains, and (2) measurements of software signing adoption in public package registries and an estimation of the effect of factors on signing adoption. We summarize these contributions below.

1.3.1 Supply Chain Attacks and Security Properties

In chapter 3, we provide a systematization of existing knowledge into an attack pattern and a set of properties for secure software supply chains. First, we propose a four-stage attack pattern for software supply chain attacks by extracting commonalities from existing taxonomies and case studies. This pattern consists of four stages: (1) *compromise*, (2) *alteration*, (3) *propagation*, and (4) *exploitation*. Next, we propose three properties for secure software supply chains: (1) *transparency*, (2) *validity*, and (3) *separation*.

1.3.2 Signature Adoption in Four Public Package Registries

In chapter 4, we measure software signing practices in four public software package registries, and we use that data to infer factors that influence software signing. We selected four registries for a quasi-experiment [23]: two with signing policies (Maven-positive, PyPI-negative), one with dedicated tooling (Docker Hub), and one with no stance on signing (Hugging Face). Under the assumption that maintainers behave similarly across registries, comparing signing practices in these registries will shed light on the factors that influence software signing. In addition to registry-dependent variables, we consider three registry-independent factors: organizational policy, dedicated signing tools, signing-related events such as high-profile cyberattacks, and the startup effort of signing.

Here are the highlights of our results. Registry-specific signing policies have a large effect on signing frequency: requiring signing yields near-perfect signing rates (Maven), while decreasing its emphasis reduces signing (PyPI). Signing remains difficult — only the registry with dedicated signing tools had perfect signature quality (Docker Hub), while the other three had signature quality rates of 68.5% (Maven), 50.2% (PyPI), and 20.2% (Hugging Face). We observed no effects from signing-related news, such as high-profile cyberattacks and new engineering standards that recommend software signing.

1.3.3 Summary

In summary, this thesis makes the following contributions:

1. An attack pattern for supply chain attacks (§3.1).
2. Properties for a secure software supply chain (§3.2).
3. Up-to-date measurements of software signing practices in four major package registries (§4.4).
4. A quasi-experiment to estimate the effect of factors on signing practices (§4.4).

1.4 Statement of authorship, attribution, and copyright

This section clarifies ownership of material presented in this thesis.

Several of the chapters of this thesis are derived from published works I co-authored with collaborators. Each chapter begins with a statement of attribution with a reference to the original publication. Chapter 2 uses material from [4], [24], which were published in the proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses and at the 2024 IEEE Symposium on Security and Privacy, respectively. Chapter 3 uses material from [4], which was published in the proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses. Chapter 4 uses material from [24], which was published at the 2024 IEEE Symposium on Security and Privacy. I have modified content to fit the narrative of this thesis, and I have added new material to provide context and continuity. I do not include intellectual contributions that are not my own. All figures and tables are my own work, unless otherwise noted.

2. BACKGROUND

Statement of Attribution: This chapter is drived from the work I published in the proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses [4] and at the 2024 IEEE Symposium on Security and Privacy [24].

§2.1 discusses software supply chains. §2.2 describes software signing, generally and in our target registries. §2.3 reviews related work and discusses the gap we address.

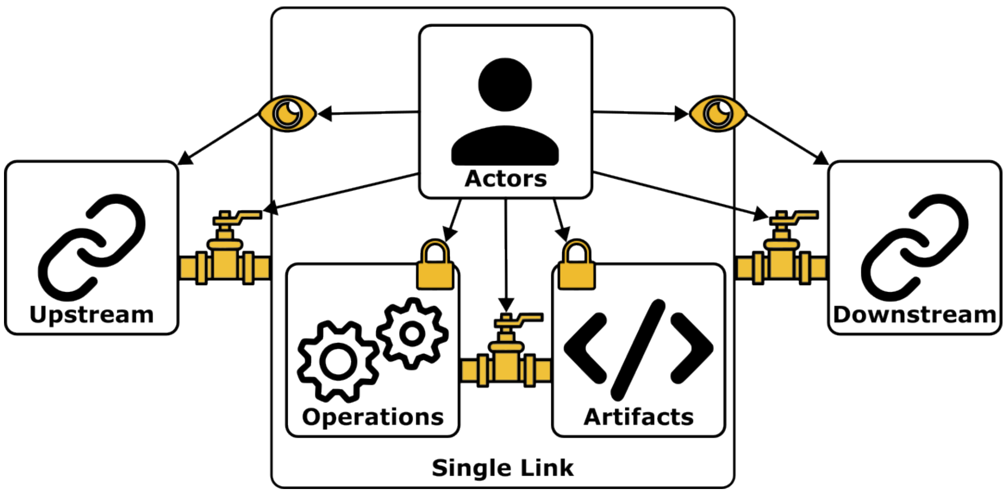

Figure 2.1. A software supply chain with focus on a single link. Actors manage components and connections within and between links. Therefore, actors manage security. Security depends on upstream and downstream transparency, link validity via component integrity and actor authentication, and logical separation between components and links.

2.1 Software Supply Chains

In general, a supply chain is a set of entities which interact to produce some product for an end consumer [25]. Each link in a supply chain contributes to the final product by providing a sub-product to reliant links. As a result, a network of dependencies forms between the links in a supply chain - terminating with the link representing the end consumer. Therefore, a supply chain is characterized by the connections and attributes of the entities used to create and ultimately consume a final product.

In computing, **software supply chains are a collection of systems, devices, and people which produce a final software product** [26]. Figure 2.1 depicts a typical software supply chain with a focus on an individual link. Each *link* in the software supply chain comprises the artifacts, operations, and actors needed to develop and deliver software products [27], [28]. *Actors* manipulate artifacts and operations within the supply chain to produce an output. *Artifacts* include the product team's code, development infrastructure, and software dependencies. *Operations* include productive steps such as fetching dependencies or compiling software, protective steps such as linting or security scans, and publishing steps such as deployment or distribution.

The structure of supply chains necessitates an interdependence between artifacts and operations within and between links. Actors manage the connections which form between components and between links in the chain. Responsibility for operations and artifacts is distributed among actors across different geographies, teams, companies, and legal jurisdictions. Modern software engineering is an international collaborative effort [29], [30]. A single link in the supply chain does not necessarily correspond to one group or organization. A single organization may be responsible for several links within a supply chain.

In modern software development, engineers commonly integrate and compose existing units of functionality to create novel applications [3]. Each unit of functionality is commonly distributed in the form of a *software package* [31]: software in source code or binary representation, accompanied by documentation, shared under a license, and distinguished by a version number. These units of functionality may be available directly from version control platforms (*e.g.,* source code [32], [33] or lightweight GitHub Packages [34])), but are more commonly distributed through separate *software package registries* [10], [35]. These registries serve both package maintainers (*e.g.,* providing storage and advertising) and package users (*e.g.,* indexing packages for search, and facilitating dependency management). These facilities for software reuse result in webs of dependencies comprising the *software supply chain* [4], [36].

Many empirical studies report the widespread use of software packages and the complexity of the resulting supply chains. Synopsys's 2023 Open Source Security and Risk Analysis (OSSRA) Report examined 1,703 commercial codebases across 17 industries [3], revealing that 96% of these codebases incorporate third-party open-source software components, averaging 595 distinct open-source dependencies per project. Similarly, Kumar *et al.* reported that over 90% of the top one million Alexa-ranked websites rely on external dependencies [37], and Wang *et al.* found that 90% of highly popular Java projects on GitHub use third-party packages [38].

Selecting and managing software dependencies is thus an important software engineering practice [39], [40]. Software engineers must decide which packages to use in their projects, *i.e.,* what to include in their application's software supply chain [41], [42]. Engineers consider many aspects, encompassing functionality, robustness, maintainability, compatibility, popularity, and security [31], [43]–[45]. Specific to security, various tools and methodologies have been proposed. These include in-toto [46], reproducible builds [47], testing [48], [49], LastPyMile [45], SBOMs [50], and BuildWatch [51]. Okafor *et al.* summarized these approaches in terms of three security properties for a project's software supply chain: validity (packages are what they claim to be), transparency (seeing the full chain), and separation of concerns [4]. Validity is a prerequisite property — if a individual package is invalid, transparency and separation will be of limited use.

2.2 Promoting Validity via Software Signing

Software signing is the standard method for establishing the validity of packages. Signing uses public key cryptography to bind an identity (*e.g.,* a package maintainer's private key) to an artifact (*e.g.,* a version of a package) [52]. With an artifact, a signature, and a public key, one can verify whether the artifact was indeed produced by the maintainer. Software signing is a development practice recommended by industry [14], [53], [54] and government [55], [56] leaders.

2.2.1 Signing Process and Failure Modes

Most software package registries require similar signing processes. Figure 2.2 illustrates this process, beginning with a maintainer and a (possibly separate) signer. They publish

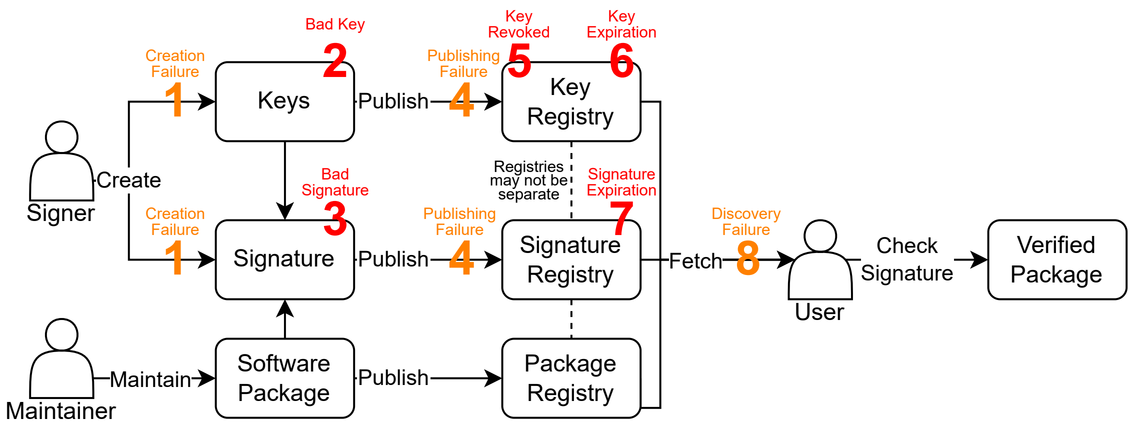

Figure 2.2. Maintainers create software packages and signers create keys which are used to create a signature. Each of these artifacts are published to a registry. Depending on ecosystem, the registries and the actors may or may not be separate. Users fetch these artifacts and can check signatures using infrastructure-specific tooling. This creates a verified package. Red and orange numbers indicate the failure modes described in §2.2.1. Red numbers indicate discernible failures. The orange numbers (modes 1, 4, and 8) are not distinguishable from one another by an external audit — when keys are missing, we cannot determine whether they were never created (mode 1), were not published (mode 4), or were undiscoverable by us (mode 8).

a signed package and separately the associated cryptographic material, so that a user can assess the validity of the result.

In package registries, there are two typical identities of the signer. In the *maintainer-signer* approach, the maintainer is also the signer (*e.g.,* Maven). In the *registry-signer* approach, the maintainer publishes a package and the registry signs it (*e.g.,* NPM). These approaches trade usability against security. Managing signatures is harder for maintainers, but the maintainer-signer approach gives the user a stronger guarantee: the user can verify they have the same package signed by the maintainer. The registry-signer approach is easier for maintainers, but users cannot detect malicious changes made during the package's handling by the package registry.

Figure 2.2 also depicts failure modes of the signing process. These modes stem from several factors, including the complexity of the signing process, the (non-)user-friendliness of the signing infrastructure, and the need for long-term management. We based these modes on the error cases of GPG [57] (*i.e.,* the possible output errors of GPG signature verification), but they are common to any software signing process based on public key cryptography. They are:

1. **Creation Failure:** The Signer does not create keys or signature files.

2. **Bad Key:** The Signer uses an invalid key, *e.g.,* the wrong key is used or it has become corrupted. Sometimes the key id associated with the signature refers to the wrong key. This leads to a mismatch between the key and the signature.

3. **Bad Signature:** The resulting signature is incorrect or unverifiable for non-malicious reasons, such as signing the wrong artifact or using an unsupported algorithm (*e.g.,* use of an unknown algorithm). The simplest example of this is a signature that does not match the artifact (*i.e.,* the artifact was modified or the signature was created on the wrong artifact).

4. **Publishing Failure:** The Signer does not publish the cryptographic material — signature and public keys — to locations accessible to the end user.

5. **Key Revoked:** The Signer revokes the key used to sign the artifact, *e.g.,* due to theft or a key rotation policy.

6. **Key Expired:** Some kinds of keys expire after a fixed lifespan. Associated signatures are no longer valid.

7. **Signature Expired:** Some signatures also expire.

8. **Discovery Failure:** The user may fail to retrieve signatures or keys. This case is distinct from Publishing Failure: the material may be available, but the user does not know where to look.

We omit from this list any failure modes associated with cryptographic strength (*e.g.,* short keys or broken ciphers), since these concerns vary by context [58]. We do, however, measure cryptographic algorithms and key lengths in §4.4.

2.2.2 Signing Targets

We detail the three kinds of signing used across the four registries considered in this work. These registries use maintainer-signing, which requires the maintainer to sign their package before publishing it. The registries support the signing of different artifact types.

Maven, PyPI—Packages: In Maven (Java) and PyPI (Python), the signing target is the software package.

Hugging Face—Commits: In Hugging Face (machine learning models), the signing target is the git commits that underlie the package. Signed commits may be interleaved with unsigned ones, reducing the security guarantee of a package that combines both kinds of commits. Hugging Face's commit-based approach means that signatures only ensure that the *changes* to package artifacts are authentic.

Docker Hub—Packages (container images): In Docker Hub (Docker container images), the signing target is the package, *i.e.,* the container image. Maintainers sign the packages, but unlike in Maven, PyPI, and Hugging Face, the cryptographic materials are stored and managed by a registry service called Notary that is run in conjunction with Docker Hub. This system provides a compromise between maintainer-signer and registry-signer: the maintainer attests to publication of the image, but the user must trust that the Notary service is not compromised (loss of cryptographic materials).

2.3 Related Works

We discuss work on software signing challenges (§2.3.1) and prior measurements of signing practices (§2.3.2).

2.3.1 Challenges of Software Signing

Signing by Novices

Like other cryptographic activities [59], [60], signing artifacts is difficult for people without cryptographic expertise. The ongoing line of "Why Johnny Can't Encrypt" works, begun in 1999 [61], enumerates confusion in the user interface [62], [63] and the user's understanding of the underlying public key model [64], [65], and other usability issues [66]. Automation is not a silver bullet — works by Fahl *et al.* [67] and Ruoti *et al.* [68] both found no significant difference in usability when comparing manual and automated encryption tools, though Atwater *et al.* [69] did observe a user preference for automated solutions.

Signing by Experts

Even when experts adopt signing, they have reported many challenges. Some concerns are specific to particular signing tools, *e.g.,* Pretty Good Privacy (PGP) has been criticized for issues such as over-emphasis on backward compatibility and metadata leaks [70], [71]. Other concerns relate to the broader problems of signing over time, *e.g.,* key management [55], [72], [73], key discovery [74], [75], cipher agility [76], and signature distribution [77]. Software signing processes and supporting automation remain an active topic of research [73], [78], [79]

Our Contribution

Our work measures the adoption of signing (quantity and quality) across multiple package registries. Our data indicates the common failure modes of software signing for the processes employed by the four studied registries. Our data somewhat rebukes ease-of-use literature, showing the importance of factors beyond perceived usability and individual preference.

2.3.2 Empirical Data on Software Signing Practices

Large-scale empirical measurements of software signing practices in software package registries are rare. In 2016, Kuppusamy *et al.* [21] reported that only ∼4% of PyPI projects listed a signature. In 2023, Zahan *et al.* examined signature propagation [14], reporting that only 0.1% (NPM) and 0.5% (PyPI) of packages publish the signed releases of their packages to the associated GitHub repositories. In 2023, Jiang *et al.* found a comparably low signing rate in Hugging Face [31]. In 2023, Woodruff reported that signing rates in PyPI were low, and that many signatures were of low quality (*e.g.,* unverifiable due to missing public keys) [22].

Our study complements existing research by aggregating and comparing the prevalence of signing across various software package registries, in contrast to previous studies that primarily focus on single registries. We publish the first measurements of signing in Maven and Docker Hub, and the first longitudinal measurements in Maven, Docker Hub, and Hugging Face. Our multi-registry approach allows us to both observe *and infer the causes of* variation in signature quantity and quality.

3. SOFTWARE SUPPLY CHAIN ATTACKS AND SECURITY PROPERTIES

Statement of Attribution: This chapter is derived from the work I published in the proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses [4].

3.1 Supply Chain Attacks

The software supply chain is an increasingly popular attack vector [80]. It is comprised of several connected links which share artifacts and conduct operations. Actors manage links and components. The difference between software supply chain attacks and other software attacks, however, is not clearly defined in literature. Ladisa *et al.* [80] and Ohm *et al.* [9] characterize supply chain attacks as the injection of malicious code into the supply chain to target downstream links. ENISA [26] defines a supply chain attack as a combination of at least two attacks — one attack on a supplier and a subsequent attack on intended targets. Other works such as Zimmermann *et al.* [10] and Zahan *et al.* [13] identify methods other than strict code injection for supply chain attacks.

Distilling the concept of software supply chain attacks from multiple sources, we arrive at a characteristic four stage attack pattern shown in Figure 3.1:

1. **Compromise:** First, an attacker finds and compromises an existing weakness within a supply chain.

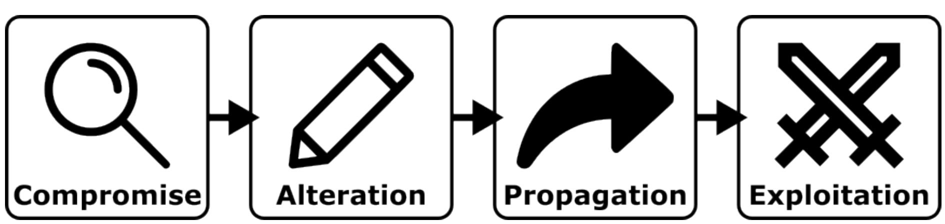

Figure 3.1. Four stage software supply chain attack pattern. First, attackers begin with an initial *compromise*. Second, they make some malicious *alteration* to the supply chain. Third, this change *propagates* down the supply chain. Finally, attackers *exploit* the introduced weakness(es).

2. **Alteration:** Second, an attacker leverages the initial compromise to alter the software supply chain.

3. **Propagation:** Third, the change introduced by the attacker propagates to downstream components and links.

4. **Exploitation:** Finally, the attacker exploits the alterations in a downstream link.

To illustrate this definition, consider the SOLARBURST compromise [6], [7]. In this supply chain attack, an attacker altered existing software from SolarWinds by injecting malicious code during the build process. This attack can be mapped to the four-stage attack from Figure 3.1 as follows: (1) The *existing weakness* compromised was the build infrastructure. (2) The *alteration* was malicious code injected by the compiler, permitting a user to bypass authentication in a SolarWinds product component. (3) The *propagation* was via SolarWinds's compromised product — its users include many companies and US government agencies like the IRS and NASA. (4) The *exploitation* was to leverage broken authentication mechanism to take control of affected machines. Such incidents are becoming common; software supply chain compromises have increased by a cumulative 650% in the last three years [81]–[83].

In contrast to this pattern, traditional attacks, such as those described by Lockheed Martin's Cyber Kill Chain [84], simply exploit an existing vulnerability (step 4). Attacks on software are not necessarily supply chain attacks just because the software exists within the context of a supply chain. For this reason, *an attack on software via the weakness of a dependency is not a supply chain attack unless it follows the attack pattern*; the attacker must both introduce the upstream change *and* subsequently exploit it downstream.

The European Union Agency for Cybersecurity (ENISA) [26] and Ohm *et al.* [9] highlight this difference. They distinguish between vulnerable and malicious dependencies. Vulnerable dependencies in the supply chain contain unintended weaknesses that may be exploited further downstream. These exploits are not supply chain attacks. On the other hand, malicious dependencies in the supply chain were intentionally designed to weaken the rest of the chain. Introducing and subsequently exploiting such weaknesses constitutes a supply chain attack.

Existing literature categorizes and documents known supply chain attacks [9], [10], [26], [80]. It is outside of the scope of this paper to enumerate individual attack types. Typically, this line of work differentiates between how attackers compromise and alter the supply chain.

3.2 Security Properties for Software Supply Chains

Components of a supply chain must be secured to mitigate the presence of vulnerabilities and the risk of attack. Supply chains become secure when attackers are unable to compromise components, alter the supply chain, or propagate malicious changes. In the literature on software supply chain security, we have identified three orthogonal and recurring security properties:

1. **Transparency:** Although actors only control portions of a supply chain, increased knowledge of the entire chain allows all parties to mitigate risk or employ specific countermeasures against an attack [27], [80], [85]–[87]. Transparency represents the availability of that knowledge to actors in the supply chain. Transparency applies to the entities connecting and comprising links in the chain.

2. **Validity:** Software supply chains should remain correct. Changes to actors, operations, or artifacts in a single link can compromise downstream entities. Validity comprises integrity of operations, integrity of artifacts, and authentication of actors. Each link in the supply chain contains a series of operations and artifacts which interact with other links in the chain. Secure supply chains require that these components remain unchanged by malicious parties [85]–[87]. Therefore, only authorized actors should make changes to link connections and components. Such changes must also receive permission to occur [13], [46], [86].

3. **Separation:** Secure supply chains embody a compartmentalized nature. Connections are an integral part of supply chains, but should only exist when necessary. These connections should be minimized to reduce attack surface area. Additionally, logically separate operations, artifacts, and actors should remain separate in practice to minimize unintended connections. By implementing measures such as mirroring, version

locking, containers etc. individual components can decrease reliance on security of others [13], [80], [85].

3.3 Analysis of Security Properties

The security properties discussed in §3.2 are only meaningful if, when applied ideally, they eliminate the risk of attack. For this to be the case, properties must be comprehensive.

To analyze these properties, we consider a hypothetical attack following the pattern discussed in §3.1 and apply transparency, validity, and separation. Since defenders can typically only address the first there stages of attack, we show how applying these properties prevents an attack from reaching the final stage: exploitation. Lastly, we note the difference between ideal conceptualizations and real-world embodiments of these security properties.

First, transparency primarily protects against the first stage of attack. Transparency, in its ideal state, enables perfect vision of all actors, operations, and artifacts across the supply chain. Such transparency would allow managers of a supply chain to identify link weaknesses before they are compromised. By securing weaknesses through patches, fixes, or other methods, managers block attempts at *compromise*. By identifying weaknesses first, managers prevent attackers from completing the first stage.

Second, validity primarily protects against the next stage of attack: *alteration*. By maintaining perfect integrity of operations, integrity of artifacts, and authentication of actors, no unauthorized changes can be made to the supply chain. As a result, attackers have no ability to maliciously alter the supply chain.

Finally, separation primarily protects against the third stage of attack: *propagation*. If a supply chain system can perfectly compartmentalize and moderate interactions between entities, then malicious changes cannot propagate downstream. In this case, connections between artifacts, operations, and actors are managed in such a way that malicious changes cannot affect other supply chain components. With ideal separation, only valid changes (*e.g.,* system updates and patches) can traverse the supply chain.

By preventing at least one of the three stages leading to exploitation, a supply chain attack cannot occur. While this hypothetical considers an ideal case, practical application is not so easy. Real techniques typically do not fully realize security properties, but they provide partial coverage (*e.g.,* attestations do not necessarily provide complete transparency). Real techniques also do not always implement security properties independently. In practice, techniques might require cohesion between multiple security properties. For example, a technique might require transparency to identify threats propagating through the supply chain before applying separation methods. Conversely, achieving close-to-ideal transparency may only be possible if the supply chain is sufficiently separated from other entities.

Since techniques do not perfectly embody security properties, defending in depth is critical [88]. Theoretically, a single technique could prevent all attacks if it provided a perfect implementation of transparency, validity, or separation. In practice, techniques have flaws. For this reason, using multiple techniques mapped to each of the security properties provides a more effective defense against attack.

4. SIGNATURE ADOPTION IN FOUR PUBLIC PACKAGE REGISTRIES

Statement of Attribution: This chapter is derived from the work I published at the 2024 IEEE Symposium on Security and Privacy [24].

4.1 Signing Adoption Theory

Although software signing is recommended by engineering leaders (§2.2), prior work shows that signing remains difficult (§2.3.1 and successful adoption is rare (§2.3.2). To promote the successful adoption of signing, we must understand what factors influence maintainers in their signing decisions. Even though using security techniques like signing is generally considered good practice [53], [54], maintainers do not always follow best practices [89]. Prior work has focused on the *usability* of signing techniques (§2.3), but we posit that a maintainer's *incentives* to sign are also important. This is because incentives may motivate maintainers to sign, even if the process is difficult.

Behavioral economics examines how incentives influence human behavior [90]. Economists typically distinguish between incentives that are intrinsic (internal) and extrinsic (external) [91]. Incentives can change how individuals make decisions, although the relationships are not always obvious [92], [93]. For example, Titmuss [94] found that paying blood donors could reduce the number of donations due to a perceived loss of altruism. In a similar way, we theorize that incentives influence how maintainers adopt software signing.

In Figure 4.1, we illustrate how incentives might apply to signature adoption. We define a *signing incentive* as a factor that influences signature adoption. Although intrinsic incentives might contribute to signature adoption (*e.g.,* altruism), we focus on extrinsic incentives because they are more easily observed. As summarized in Table 4.1, we examined four kinds of external incentives that might influence a maintainer's signing practices. We formulate hypotheses corresponding to each incentive in §4.2. Some of our hypotheses may seem obvious, but behavioral economics suggests that even these may not hold true (*i.e.,* some relationships may be counterintuitive).

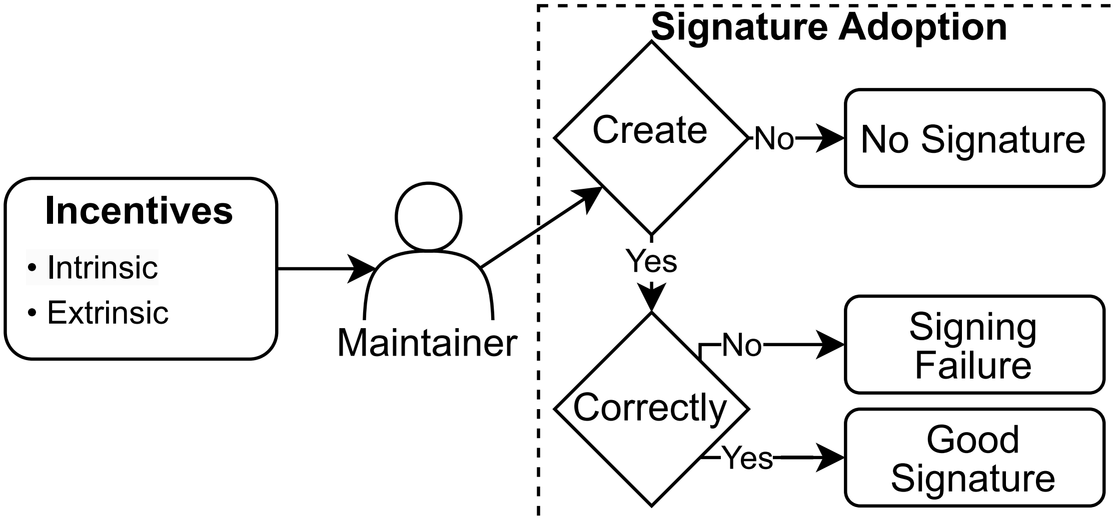

Figure 4.1. Incentives influence how maintainers adopt software signing. The maintainer decides weather or not to create a signature. This corresponds to the *quantity* of signatures. If a maintainer decides to create a signature, they can either follow the signing process (*i.e.,* Figure 2.2) correctly or not. Correctly following the signing process results in a good signature but incorrectly following the process results in a signing failure. This corresponds to the *quality* of signatures.

Table 4.1. Kinds of incentives considered. The second column is our hypotheses: whether the incentive was predicted to increase (↑) or decrease (↓) signature adoption (cf. §4.2). The third column is the observed effect (cf. §4.4).

Factor	Expectation	Observed Effect
Registry policies	↕	↕ quantity
Dedicated tooling	↑	↑ quality
Signing events	↑	None
High startup cost	↓	↓ quantity

To operationalize the concept of signing practices, we define *signature adoption* as a maintainer's decision to (1) create a signature (quantity), and (2) to follow the signing process correctly in doing so (quality). To secure software supply chains, we want to identify incentives that sway the behavior of the maintainer community, not just individuals. Going forward, we thus call *Signing Quantity* the number or proportion of signed artifacts present within a given registry, and *Signing Quality* the condition of the signatures which are present (*i.e.,* how many of them are sound, and how many display which of the failure modes indicated in Figure 2.2). For example, suppose that 90% of a registry's artifacts are signed, but only 10% of these signatures have available public keys. We would consider this registry as experiencing high quantity, but low quality, signing adoption.

Given evidence to support the theorized relationship between these factors and signature adoption (Figure 4.1), one could predict the quantity and the quality of signatures in a given environment, as well as the effect of an intervention affecting maintainers' incentives.

4.2 Research Questions

We ask three questions across two themes:

Theme 1: Measuring Signing in Four Package Registries

Theme 1 will update the cybersecurity community's understanding on the adoption of software signing.

- **RQ1:** What is the current quantity and quality of signing in public registries?
- **RQ2:** How do signing practices change over time?

Theme 2: Signing Incentives

Theme 2 evaluates hypotheses about the effects that several types of external incentives have on software signing adoption.

- **RQ3:** How do signing incentives influence signature adoption?

To evaluate RQ3, we test four hypotheses.

H_1: *Registry policies that explicitly encourage or discourage software signing will have the corresponding direct effect on signing quantity.* This hypothesis is based on our experience as software engineers, "reading the manual" for the ecosystems in which we operate.

H_2: *Dedicated tooling to simplify signing will increase signing adoption (quantity and quality).* The basis for this hypothesis is the prior work showing that signing is difficult and that automation can be helpful §2.3.

H_3: *Cybersecurity events such as cyberattacks or the publication of relevant government or industry standards will increase signing adoption (quantity and quality).* The basis for this hypothesis is the hope that software engineers learn from failures and uphold best practices, per the ACM/IEEE code of ethics [95].

H_4: *The first signature is the hardest, i.e., after a package is configured for signing adoption, it will continue to be signed.* Like H_2, this hypothesis is based on prior work showing that signing is difficult — but knowing that the cost of learning to sign need be paid only once.

4.3 Methodology

This section describes our methods. In §4.3.1 we give our experimental design. Following that design, our methodology has five stages. As illustrated in Figure 4.2:

1. **Select Registries**(§4.3.2): Picking appropriate registries for our study.
2. **Collect Packages**(§4.3.3): Methods used to collect a list of packages from each registry.
3. **Filter Packages**(§4.3.4): Filter list of packages so that remaining packages are adequately versioned.
4. **Measure Signature Adoption**(§4.3.5): Measuring the quantity and quality of signatures for each registry.

5. **Evaluate Adoption Factors**(§4.3.6): Comparing signature adoption among registries and across time.

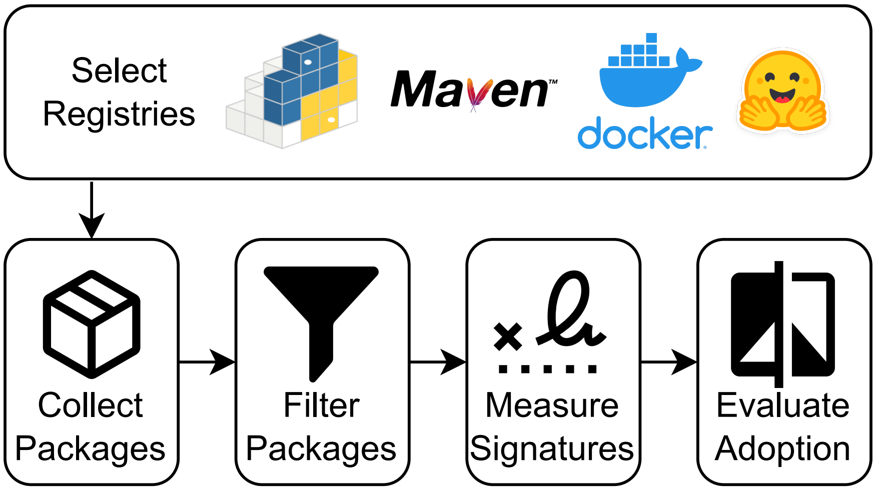

Figure 4.2. First, we select package registries that represent a range of software types and signing policies. Our selected registries include PyPI, Maven Central, Docker Hub, and Hugging Face. Next, we collect a list of packages for each platform. Then, we filter the list of packages to a sample of packages for each platform. On the remaining packages, we measure the quality and quantity of signatures. Finally, we use these measurements to evaluate factors influencing adoption.

4.3.1 Experimental Design

First, we provide an overview of the types of experiments needed to answer our RQs (§4.3.1). Next, we summarize quasi experiments and why they work well for our study (§4.3.1). Then, we discuss the assumptions that underlie our quasi experiment (**??** 1). Finally, we explain how we apply a quasi experiment to our study (**??** 1).

Overview

To answer RQ1, we measure the quantity and quality of signatures in the registries of interest. To answer RQ2, we examine trends in signature adoption and failure modes over time. The measurements themselves are fairly straightforward, following prior work on software signing quantity [14], [21], [31] and quality [22]. In §4.3.5 we define quality systematically, but our approach resembled prior work.

RQ3 requires us to test the correlation between incentives and signature adoption (quantity and quality). Our goal is similar in spirit to the usability studies performed by Whitten & Tygar [61], Sheng *et al.* [62], and Routi *et al.* [65] — like them, we are interested in factors influencing the adoption of signing. However, we are not interested in the usability of individual signing tools, in which context controlled experiments are feasible. Instead, we want to understand the factors affecting signing practices across package registries. A controlled experiment would randomly assign maintainers to platforms with different registry factors, hold other variables constant, and measure the effect for hypothesis tests. As this level of control is impractical, we instead use a *quasi experiment* to answer RQ3.

Quasi Experiments

Quasi experiments[1] are a form of experiment whereby (1) *treatments* (instances of independent variables) are applied to *subjects*; and (2) *outcomes* (dependent variables) are measured; but (3) the assignment of treatments to subjects is not random [23], [96]. Instead, the application of treatments is based on characteristics of the subjects themselves [97]. This method is often used where controlled experiment is infeasible, *e.g.,* measuring impacts of government policies on populations [96], [98], [99]. These studies produce the strongest results when treatments occur independently of the subjects, or are *exogenous* [96], [99].

A quasi experiment would allow us to test our hypotheses by leveraging naturally occurring differences between registries. If we can identify registries that vary along the dimensions of interest, then comparing signing adoption in these registries would allow us to infer cause-effect relationships from the incentives of interest. Furthermore, the existence of multiple time periods and comparison groups also strengthens the outcomes of this approach [99].

[1]↑ The terms *quasi experiment* and *natural experiment* are often used interchangeably, but some researchers distinguish between the two. Treatments applied in a natural experiment are not intended to influence the outcome, whereas treatments in a quasi experiment are planned [96]. Since some registry factors are intended to cause changes in signature adoption, our study is a quasi experiment, not a natural experiment.

Assumptions

Our quasi experiment relies on a major assumption:

Assumption of Structural Similarity: Our quasi-experiment assumes that there are no uncontrolled confounding factors. Such factors would differentially affect software signing adoption between registries. In other words, we assume that the registries are used by software engineers in similar ways, without registry-specific signing influences other than those considered.

We base this assumption in the cybersecurity and software engineering research literature. For example, Zahan *et al.* [19] compared NPM and PyPI and found comparable behaviors across a range of security measures and practices, including security policies, vulnerabilities, dependency update tools, maintenance procedures, signed-releases, and potentially risky workflows. Bogar *et al.* [100] offered further support for this assumption in a study of several software ecosystems, writing that "*Ecosystems tend to share many values but differentiate themselves based on a few distinctive values strongly related to their purpose and audience.*" While there is limited research on PTM registries, Jiang *et al.*'s [31] reported that the PTM re-use workflow for PTMs on Hugging Face to that of traditional software. This suggests comparability between the PTM space and the traditional software ecosystem space. These and other works suggest a level of uniformity across registries with respect to engineering practices.

Applying the Quasi Experiment

With this assumption in mind, we conduct the quasi experiment. In our study, registry factors are *treatments*, maintainers are *subjects*, and signature adoption is the *outcome*. As noted, applying random registry factors to maintainers is impractical — this is effectively the same as randomly assigning maintainers to different platforms. Instead, characteristics of the maintainers (the registries they naturally use) determine which registry factors they experience. Since maintainers' registry selections are not known to be influenced by signing infrastructure or policy, we can also consider registry factors as exogenous treatments.

We illustrate our experimental design with two evaluations, which we perform later in detail. Our four hypotheses for RQ3 include incentives that are registry-specific (H_1, H_2) and that are registry-independent (H_3, H_4). To assess the effect of registry-specific incentives, we examine whether the date of an associated event is correlated with a significant change in signing adoption within only the pertinent registry. For example, if a change in PyPI's signing policy changes the signing rates in PyPI, while rates in other registries are undisturbed, then we would view this as support for H_1. Conversely, to assess the effect of registry-independent incentives, we examine whether the date of an associated effect is correlated with a significant change in signing adoption in multiple registries. For example, if after a major software supply chain attack we see changes in signing adoption in PyPI and Maven, then we would view this as support for H_3.

Based on this design, we now proceed through the five stages indicated in Figure 4.2.

4.3.2 Stage 1: Select Registries

In this section, we explain what makes a registry a good candidate for this study (§4.3.2) and justify our use of PyPI, Maven Central, Docker Hub, and Hugging Face (§4.3.2).

Selection Requirements

We searched for software package registries which have natural variations in signing incentives. We selected some registries that have experienced changes to their signing infrastructure and policies over time. We focused on package registries with maintainer-signed signatures, as defined in §2.2, for two reasons: 1) they place a greater burden on the maintainer and thus the effect of incentives would be more observable; and 2) they provide a better guarantee of provenance than server-signed signatures (*i.e.,* the artifact hasn't been modified between the maintainer signing it and uploading it to a registry). Finally, the selected registries should be popular so that they are representative of publicly available software.

Selected Registries

Following these requirements, we identified 4 registries for study: PyPI, Maven Central, Docker Hub, and Hugging Face. Table 4.2 summarizes these registries. They represent some of the most popular programming languages [101] (Java, Python), the most popular container technology [102] (Docker), and the most popular ML model hub [35] (Hugging Face). Next we elaborate on each selected registry. All data is as of April 2024.

Table 4.2. The selected package registries, their associated software type, and signature type.

Registry Name	Software Type	Signature Type
PyPI	Python	PGP (now deprecated)
Maven Central	Java	PGP
Docker Hub	Containers	DCT
Hugging Face	ML Models	Git Commit Signing

(1) PyPI: PyPI is the primary registry for the exchange of software packages written in the Python programming language. PyPI hosts more than 520,000 packages [103]. PyPI allows maintainers to sign packages with PGP signatures. The PyPI registry owners deemphasized the use of PGP signatures on 22 Mar 2018 [104] and later deprecated them on 23 May 2023 [105]. Pre-existing signatures remain, but users cannot (easily) add new signatures.

(2) Maven Central: Maven Central (Maven) is the primary registry for the exchange of software packages written in the Java programming language. Maven hosts more than 499,000 packages [106]. Like PyPI, Maven allows maintainers to sign packages with PGP signatures. Unlike PyPI, on Maven, signatures have been mandatory since 2005 [107].

(3) Docker Hub: Docker Hub is the primary registry for the exchange of virtualized container images in the Docker format. Docker Hub hosts more than 1,000,000 container images [106]. Docker Hub uses Docker Content Trust (DCT) [108] to sign container images. As noted in §2.2, Docker Hub has dedicated tooling for signing, integrated in the Docker CLI. Sigstore's Cosign also supports signing docker images [109], but is not yet well integrated in Docker Hub. In 2019, Docker Hub began including more information about image provenance (*e.g.,*

author, OS, digest, architecture) on its web UI, but does not mandate signing like Maven does.

(4) Hugging Face: Hugging Face is the primary registry for the exchange of neural network models [35]. Hugging Face hosts more than 590,000 models [110]. Hugging Face supports the signing of git commits [111]. Hugging Face has no stated policy towards signing, and we are aware of no signing events specific to Hugging Face.

4.3.3 Stage 2: Collect Packages

Next, our goal was to collect a list of the packages from each of the selected registries, so that we could sample from it and measure signing practices. For each registry, we attempted to enumerate all packages available on the platform. We used ecosyste.ms [106] as an index for the packages available from Maven Central and Docker Hub. For Hugging Face, we used the Hugging Face API to collect packages. For PyPI, we used the Google BigQuery dataset [112] to collect packages. In the remainder of this subsection, we describe the package structure and collection techniques used for each registry.

The Ecosyste.ms Cross-registry Package Index

Ecosyste.ms provides a comprehensive cross-registry package index, aggregating package data from multiple registries into a database. Periodically, ecosyste.ms releases datasets that can be downloaded and subjected to detailed queries. In this work, we used the most recent version of the dataset — 01 Mar 2024. This dataset indexed PyPI, Docker Hub, and Maven Central, but not Hugging Face. We did not use this dataset for PyPI because the BigQuery dataset contains signing information.

We sought to obtain data up to 31 Dec 2023 from each registry. We arbitrarily set the start date to 01 Jan 2015 imitating [22], so that the reported data would not be too different from current practices.

Details Per-Registry

PyPI: In PyPI, packages are distributed by version as *wheels* or *source distributions* (*i.e.,* each package may have multiple versions each with their own distributions). For example, the latest version of the *requests* package is *2.31.0* (as of this writing) and has two distribution files: (1) a wheel file named *requests-2.31.0-py3-none-any.whl* and (2) a source distribution file named *requests-2.31.0.tar.gz*. Both of these files can be signed, so we collect each of these files during our assessment of PyPI.

To collect all packages from PyPI between 01 Jan 2015 and 31 Dec 2023, we used Google's BigQuery PyPI dataset. This dataset contains a list of package distributions and associated metadata for each package hosted on PyPI.

Since this study is only concerned with the quality and quantity of signatures, we did not need to download any packages without signatures (we only need to count how many packages have no signature). We downloaded signed packages to assess the quality of their signatures.

Maven Central: Maven Central packages are stored in a directory structure organized by namespace, package name, and version number. Each version of a package contains several files for use by the downstream user. These typically include *.jar*, *.pom*, *.xml*, and *.json* files which include the package, source distributions, tests, documentation, or manifest information. Each of the files included in a package version typically has a corresponding PGP signature file with a *.asc* extension.

We used ecosyste.ms' data dump from 01 Mar 2024 to collect packages from Maven Central. This data dump contains a list of package distributions and associated metadata for each package hosted on Maven Central.

Docker Hub: Docker Hub packages are organized into *repositories* which are collections of *images*. Each repository contains *tags* which are versions of the image. These tags can be signed using Docker Content Trust (DCT). DCT is a Docker-specific signing tool built on Notary [113].

Table 4.3. Packages available after each stage of the pipeline. Collect packages refers to the total number of packages available in the registry. Filter packages refers to the number of packages with ≥ 5 versions between 01 Jan 2015 and 31 Dec 2023 and non-gated Hugging Face models. Measure signatures refers to the number of packages we attempted to measure. Due to download rates, we only measure a random sample of the filtered Maven Central packages (10% of the total population).

Stage	**PyPI**	**Maven**	**Docker**	**HF**
Collect Packages	623,346	499,588	1,001,771	559,517
Filter Packages	205,513	243,191	91,719	128,338
Measure Signatures	205,513	49,959	91,719	128,338

To collect all packages from Docker Hub between 01 Jan 2015 and 31 Dec 2023, we use ecosyste.ms' data dump from 01 Mar 2024. This data dump contains a list of package distributions and associated metadata for each package hosted on Docker Hub.

Hugging Face: Hugging Face hosts models, datasets, and spaces for machine learning. For the purpose of this study, we only focus on the models, which are stored as git repositories. Hugging Face uses git commit signing, *i.e.,* signatures occur on a per-commit basis for each repository.

To collect all of the model packages on Hugging Face between 01 Jan 2015 and 31 Dec 2023, we use the Hugging Face Hub Python interface to generate a list of packages (and metadata) in our date range. We then iteratively clone all repositories from Hugging Face.

4.3.4 Stage 3: Filter Packages

After obtaining the lists of packages, we filtered them for packages of interest to our study. Since our study was interested in effects over time, our primary filter was for packages with multiple versions. In all of the registries, we filtered for all packages with ≥ 5 versions between 01 Jan 2015 and 31 Dec 2023.[2]

[2]↑ Note that for Docker Hub we filtered for packages with ≥ 5 tags and for Hugging Face we filtered for packages with ≥ 5 commits. The notion of a version for ML models is less clear than for software packages, so we used commits as a proxy. Hugging Face has some mechanisms (*e.g.,* tags) for versioning, but we found that these are not consistently used.

On Hugging Face, we also filter models that are gated (*i.e.,* they have some sort of access control). Examples include *pyannote/segmentation* which requires users to agree to terms of use. These models account for 0.9% (5,138) of the total models. We are unable to tell how many of these models have ≥ 5 commits in the time period, without gaining access. In many cases, this requires agreeing to terms of use, which we are unable to do at scale.

See Table 4.3 for the number of packages available after each stage of the pipeline.

4.3.5 Stage 4: Measure Signatures

After filtering, we measured the quantity and quality of the signatures associated with the surviving packages.

In §4.1 we defined signature quantity as the fraction of signed artifacts in a registry, and quality as the fraction of good signatures among those. Since signatures apply to different units of analysis in each registry, we defined quantity and quality somewhat differently for each registry. The nature of the signable artifacts was discussed earlier in this work. However, certain aspects of quality cannot be measured in all registries (*e.g.,* signature expiration is not applicable to git commits). We use the remainder of this subsection to clarify signature quality for each registry.

For quality, not all failure modes can be measured in each registry. Recall that Figure 2.2 indicated failure modes in a typical signing scheme. However, some registries use unique signing schemes, such as Docker Content Trust (DCT). These schemes do not allow us to measure quality in the same manner as other registries. Although they use similar cryptographic methods, we cannot measure points of failure in the same manner as other registries. Using the numbering system of Figure 2.2, in Table 4.4 we indicate which signature failures can be measured on each platform.

PGP

PyPI and Maven Central use PGP signatures to secure packages. The measurements for these registries are similar, so we describe commonalities here.

Table 4.4. Signature statuses and whether or not they are measurable on each platform. ✓: Measurable. ✗: Not measurable. –: Theoretically measurable. PK: Public key.

Status	**Failure #**	**PyPI**	**Maven**	**Docker**	**HF**
Good Signature	–	✓	✓	✓	✓
No Signature	1	✓	✓	✓	✓
Bad Signature	3	✓	✓	✗	✓
Expired Signature	7	✓	✓	✗	–
Expired PK	6	✓	✓	✗	–
Missing PK	4,8	✓	✓	✗	–
Revoked PK	5	✓	✓	✗	–
Bad PK	2	✓	✓	✗	–

Discovery: For PGP, we need to discover the public keys associated with each signature. The most common method for sharing public PGP keys is to use a public key server. Sonatype recommends the Ubuntu, OpenPGP, and MIT servers [114]. We found 5 more servers via Google searches.

On this set of 8 servers, we conducted a small-scale experiment to determine which servers to use in our study. Using a sample of ∼3,800 keys from PyPI and Maven Central, we found that only 4 servers worked reliably (*i.e.,* some servers are no longer functional or perform slowly). For example, the famous *pgp.mit.edu* server often times out.

We found that four servers responded consistently: *keyserver.ubuntu.com keys.openpgp.org keyring.debian.org*, and *pgp.surf.nl* For each key, we queried each of these servers in order to find the public key associated with the signature. If we were unable to find the key in any of these servers, we marked the signature as having a missing public key.

Among our selected servers, the Ubuntu server was able to discover the most keys, followed by the Surf, OpenPGP, and Debian servers. The Surf server is queried last because has the most overlap with the Ubuntu server (*i.e.,* we are more likely to find the key by looking at other servers first).

Verification: To verify a PGP signature, we use the *gpg* command line utility. This utility provides a *verify* command which can be used to verify the validity of a signature. This command returns the status of the signature verification. We parse this status to determine the quality of a signature.

Expiration: For our measurements, we consider a key to be expired if the key's expiration had passed at the time of our measurement. There is no definitive way to determine if a signature was created before or after the key expired.

Cryptographic Algorithm: The strength of a PGP key is determined by the cryptographic algorithm. The *gp* command line utility provides a *list-packets* command which can be used to extract metadata from a signature. This command returns the cryptographic algorithm and, particularly important for RSA keys, the key length. We compare this to NIST's recommendations [115].

Details Per-Registry

PyPI: PyPI uses PGP signatures to secure packages. As a result, we use the methods described in §4.3.5 to measure the quality and quantity of signatures on PyPI packages.

Maven Central: Maven Central requires PGP signatures on all artifacts. As a result, we use the methods described in §4.3.5 to measure the quality and quantity of signatures on PyPI packages.

For Maven Central, we were unable to sample the entire filter population. Due to the high adoption quantity and the number of files associated with each Maven Central package, verifying Maven Central packages requires a large amount of network traffic to download every file and its signature. For this reason, we choose to only check a random sample of the filter population (10% of the total population). See Table 4.3 for the number of packages we attempted to measure versus the total number of packages available.

Docker Hub: Docker Hub uses Docker Content Trust (DCT) to sign image tags. This tool has first-class support in the Docker CLI, so we use the *docker trust inspect* command to verify the validity of signatures. Since DCT automates much of the signature process, we can only measure the presence of a signature (*i.e.,* you cannot upload a bad signature with DCT). For this reason, all signatures are considered valid unless they are missing.

Hugging Face: Hugging Face uses git commit signing for its models. Typically, git commit signatures are verified using the *git verify-commit* command. This uses PGP under the hood, and still requires a public key to verify the signature. On platforms like GitHub, the public key is stored on the user's profile. However, Hugging Face does not currently expose the public keys for its users [116]. Since users upload keys to Hugging Face they do not necessarily upload them to a public key server as in §4.3.5 — we tried and had a low discovery rate. For this reason, verifying signatures with *git verify-commit* is not actually representative of the signature quality. Instead, we use Hugging Face's UI to verify the validity of signatures. Verified signatures are marked as good, and unverified signatures are marked as bad.

4.3.6 Stage 5: Evaluate Adoption

For RQ1 and RQ2, our method is to report and analyze the statistics for each registry.

To answer RQ3, we need to evaluate hypotheses H_1–H_4. For these hypotheses, we need distinct factors corresponding to each class of incentive. We identified a set of factors through web searches such as "software supply chain attack" or "government software signing standard". Four authors collaborated in this process to reduce individual bias.

The resulting set of factors are given in Table 4.5. We identified changes in registry policies and in three kinds of signing events: software supply chain attacks, government actions, and industry standards.

Using these factors, we perform exploratory data analysis (visual analysis) to identify trends in signing adoption. This analysis was performed on another sample of the data to avoid biasing our final statistical tests. On the final data, we use statistical tests to evaluate hypotheses H_1–H_3.

We conduct statistical tests as follows:

- We assume that if an event (incentive) impacts a given registry, then the effect will be measurable within 6 months. This time horizon was selected to permit some amount of lag, while not introducing too many possible confounding events within the time frame.

- The data used in the tests are the daily signing rates from each registry (calculation: Number of signed units / Total number of units).

- We select a 99% confidence level (p-values should be < 0.01 to be considered significant)

We note two caveats: *First*, there are many comparisons that could be performed (4 registries x 15 events in Table 4.5), but every additional test increases the risk of Type-1 errors (false positives). For this reason, we only test the hypotheses that are supported by exploratory data analysis and apply a Bonferroni correction to our p-values. *Second*, where present, statistically significant results indicate correlation, not causation. Our research is motivated by an underlying predictive theory. If the predicted results occur at a statistically significant level, they support the theory.

Table 4.5. Factors that could provide signing incentives. Factors were identified via web searches by four authors, and categorized by incentive type per Table 4.1.

Factor	Category	Date
PyPI De-emph	Registry policy	Mar 2018
Docker Hub Update	Registry policy	Sep 2019
PyPI Removed PGP	Registry policy	May 2023
NotPetya	Signing event (attack)	Jun 2017
CCleaner	Signing event (attack)	Sep 2017
Magecart	Signing event (attack)	Apr 2018
Docker Hub Hack	Signing event (attack)	Apr 2019
SolarWinds	Signing event (attack)	Dec 2020
Log4j	Signing event (attack)	Dec 2021
NIST code signing	Signing event (government)	Jan 2018
CISA Publication	Signing event (government)	Apr 2021
Exec. Ord. 14028	Signing event (government)	May 2021
CMMC	Signing event (standard)	Jan 2020
CNCF Best Practices	Signing event (standard)	May 2021
SLSA Framework	Signing event (standard)	Jun 2021

We conducted 3 kinds of statistical tests: (1) a one-way ANOVA test on overall differences between registry adoption quantity; (2) a subsequent Tukey tests; and (3) selected independent t-tests to compare signing before and after selected events. All tested distributions meet the assumptions inherent in the corresponding tests.

4.4 Results

For RQ1, we present the quantity and quality of signatures we measured in each registry in §4.4.1. For RQ2, we describe changes in signing practices over time in §4.4.2. Finally, for RQ3, we assess the influence of incentives on signing adoption in §4.4.3.

4.4.1 RQ1: Quantity and Quality of Signatures

In Table 4.6, we show the quantity and quality of signatures in each registry. We show the total amount of signable artifacts in each registry, how many of those are signed, and

Table 4.6. The number of assessed packages and artifacts from each registry, the percent with and without signatures, and the breakdown of signature status for signed artifacts. For each measurement, we show the most recent year and the entire measurement period. "—": Not measurable; Hugging Face hides keys, only disclosing whether validation succeeded.

Registry	**PyPI** 1 year (Total)	**Maven Central** 1 year (Total)	**Docker Hub** 1 year (Total)	**Hugging Face** 1 year (Total)
Total Packages	93.2K (206K)	24.1K (50K)	70K (91.7K)	100K (128K)
Total Versions	1.2M (4.83M)	378K (2.04M)	5M (9.52M)	2.02M (2.66M)
Total Artifacts	2.75M (9.68M)	1.55M (7.96M)	5M (9.52M)	2.02M (2.66M)
Unsigned Artifacts	99.8% (98.8%)	2.9% (5.9%)	99.0% (97.5%)	99.9% (99.9%)
Signed Artifacts	0.2% (1.2%)	97.1% (94.1%)	1.0% (2.5%)	0.1% (0.1%)
Good Signature	48.4% (50.2%)	72.9% (68.5%)	100% (100%)	23.9% (20.2%)
Bad Signature	0.0% (0.2%)	0.2% (0.7%)	0.0% (0.0%)	—
Expired Signature	0.0% (0.0%)	0.0% (0.0%)	0.0% (0.0%)	—
Expired Public Key	6.6% (16.0%)	15.0% (23.0%)	0.0% (0.0%)	—
Missing Public Key	23.4% (16.0%)	7.3% (3.8%)	0.0% (0.0%)	—
Public Key Revoked	18.5% (15.7%)	0.5% (2.3%)	0.0% (0.0%)	—
Bad Public Key	3.1% (1.9%)	4.2% (1.6%)	0.0% (0.0%)	—

the status of the subset of signed signatures. We show both the most recent year of data (Jan-Dec 2023) and the entire time period (01 Jan 2015 to 31 Dec 2023).

Quantity of Artifact Signatures

With respect to the quantity (proportion) of signed artifacts, the registries lie in three groups by order of magnitude. First, *Maven Central* experiences the highest signing rate with 97.1% of artifacts signed in 2023. We conjecture that this degree of signing occurs only when signing is mandatory.[3] Second, *Docker Hub* has a low adoption rate with 1.0% of tags signed in 2023. Third, *Hugging Face* and *PyPI* currently have a negligible amount of signatures with 0.1% and 0.2% of artifacts signed in 2023, respectively. Keep in mind that *PyPI's* low signing rate includes data since the feature was removed in 23 May 2023. Even considering this, the relative number of signed artifacts is still the lowest on *Hugging Face*.

[3]↑ Not all Maven Central packages are signed. Some are ingested from other Java package registries and the Maven signing requirement is waived. *Source: Personal communication with Maven team.*

Out of all 2.02M commits across the 100K packages on *Hugging Face* in 2023, only 1.24K are signed.

Finding 1: Between 01 Jan 2023 and 31 Dec 2023, all registries aside from Maven Central had less than 2% of artifacts signed. Maven Central, the only registry in our study that mandates signing, had 97.1% of artifacts signed in that same time period.

Quality of Artifact Signatures

Failure Modes: With respect to quality, each registry has a distinct flavor. On *Docker Hub* signatures either exist or not — we can only tell if maintainers correctly signed a tag. On *Hugging Face*, we can only tell if the signature was valid. On *Maven Central* and *PyPI*, we can measure the failure modes of signatures.

Of the registries with measurable quality, *Maven Central* has the best with 72.9% of signatures valid since 01 Jan 2023. *PyPI* has the next best quality with 48.4% of signatures valid since 01 Jan 2023. Not only does *Hugging Face* have the lowest quantity of signed artifacts, but it also has the lowest quality of signed artifacts. Of the 1.24K signed artifacts, only 296 (23.9%) were valid.

We observed differences between signing failure modes by registry. The three most common failure modes on *Maven Central* were expired public keys, missing public keys, and bad public keys. Failures related to public keys accounted for over 99% of all Maven signing failures in 2023. The three most common failure modes on *PyPI* were missing public keys, revoked public keys, and expired public keys. Expired signatures are very rare, the only instances we could find were from 2014 versions of the leekspin package on PyPI Similar to Maven Central, public key related failures accounted for over 99% of all PyPI signing failures in 2023. On Hugging Face, the registry records but does not publish the public keys disclosed by package maintainers.[4] Due to the lack of published public keys, we were unable to determine the cause of the invalid signatures. Finally, we observed no signing failures in Docker Hub.

[4]↑We asked the Hugging Face engineers for access. They declined.

Table 4.7. Cryptographic algorithms used in signatures.

Algorithm	Maven Central	PyPI
RSA	**96.01%**	**85.82%**
DSA	1.79%	**11.22%**
EdDSALegacy	2.15%	2.71%
RSA Sign Only	0.03%	0.00%
ECDSA	0.01%	0.26%

Table 4.8. RSA key lengths used in signatures.

Length	Maven Central	PyPI
8192	0.005%	0.142%
4608	0.006%	0.000%
4096	**38.162%**	**49.317%**
3072	**13.892%**	2.174%
2048	**47.662%**	**48.223%**
1536	0.000%	0.001%
1024	0.271%	0.143%

Finding 2: Signing failures were common in three of the four studied registries. On Maven Central and PyPI, 24.0% and 53.1% of signatures between 01 Jan 2023 and 31 Dec 2023 were invalid, respectively. On Hugging Face the situation is worse, with 76.1% invalid signatures. Lastly, on Docker Hub, we observed no signing failures.

Cryptographic Algorithms: For Maven Central and PyPI, we observed the use of several cryptographic algorithms. We show the distribution of algorithms in Table 4.7. RSA was the most common algorithm used in both Maven Central and PyPI. In Maven Central, RSA was used in 96.01% of signatures. In PyPI, RSA was used in 85.82% of signatures.

RSA signature security is dependent on the key length. In Table 4.8, we show the distribution of RSA key lengths used in Maven Central and PyPI. Of the RSA signatures in Maven Central, most of them used either 2048 or 4096 bit keys. The same is true for PyPI. These key sizes comply with the US NIST's SP-800-78-5 baseline [115] for keysizes for this decade. Only a small fraction of signatures used keys larger than 4096 bits or smaller than 2048 bits.

Finding 3: For both Maven Central and PyPI, RSA was the most common cryptographic algorithm used in signatures. Most RSA signatures used 2048 or 4096 bit keys. A small fraction used insecurely-small key sizes (<2048 bits) or very large key sizes (>4096 bits).

Expired Keys: We report expired keys as a failure mode regardless of the publishing time of the artifact. For old artifacts, this may be unfair, since the key was presumably valid at time of publication, and since a newer version of the package may have been available. We investigated both of these aspects for signatures whose keys had expired.

First, we examined the typical time of validity, *i.e.,* the time remaining in the public key's lifespan at time of signature creation. Surprisingly, a substantial proportion of signatures are created **after** the expiration of the associated public key, *i.e.,* it was never a valid signature. For Maven Central, 16.8% of the artifacts whose public key eventually expired had signatures created after the expiration. For PyPI, 11.4% of the artifacts whose public key eventually expired had signatures created after the expiration. For signatures created with a still-valid public key, signatures on Maven Central had a median of 1.37 years remaining in the public key's lifespan and those on PyPI had a median of 1.93 years remaining in the public key's lifespan.

Second, we examined the availability of an upgrade path from an expired to an unexpired version of a package. On Maven Central, in only 26.7% of cases was there a newer version of the package with an unexpired signature. On PyPI, the number was 8.0%. Thus, upgrade paths are usually unavailable, suggesting that signature expiration is not well managed in these ecosystems. This result may be confounded by abandoned packages.

4.4.2 RQ2: Change in Signing Practices Over Time

Quantity of Artifact Signatures

In Figure 4.3, we show the quantity of signed artifacts over time. We measured signing rates for the signable artifacts published in that month. This figure shows how many such artifacts were signed, grouped by registry.

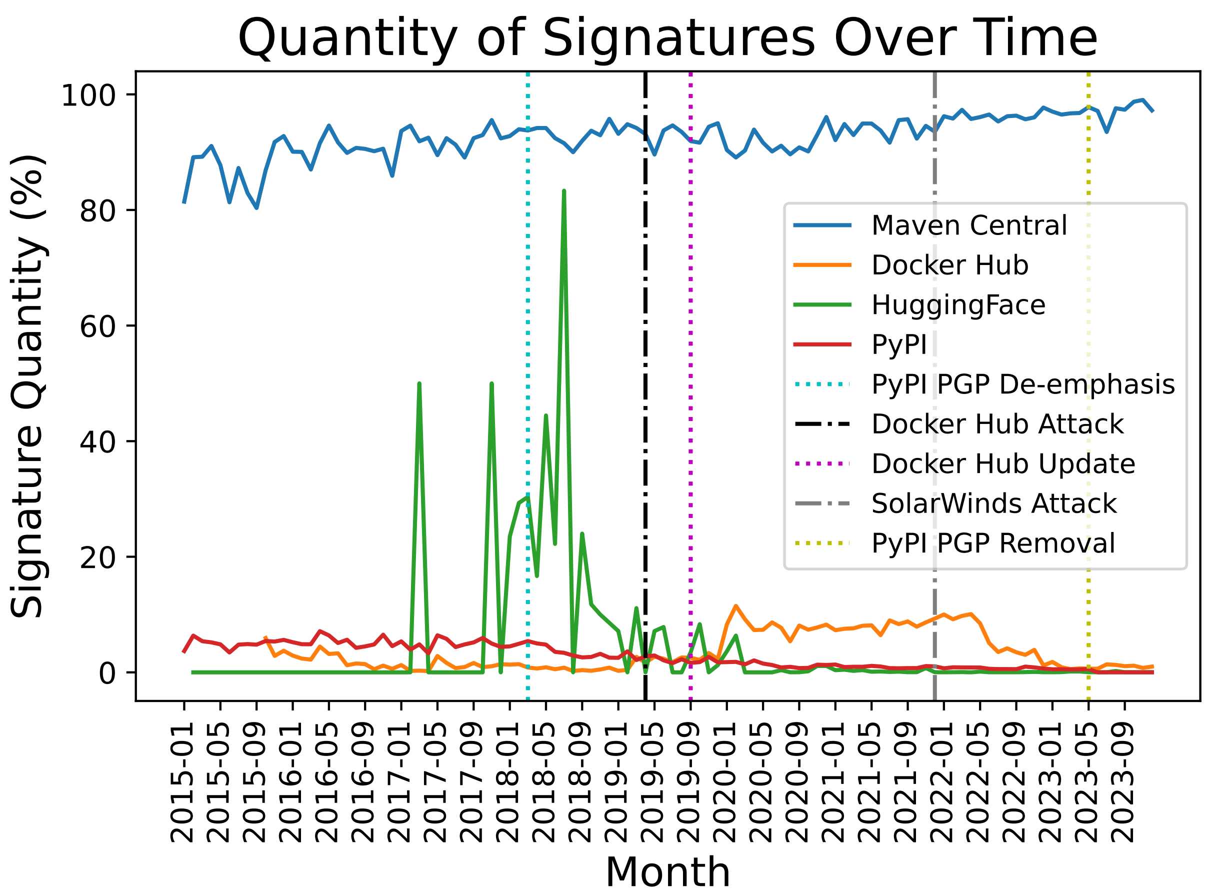

Figure 4.3. Quantity of signed artifacts over time. Axes are time (monthly increments) and the percentage of signed artifacts per registry.

We observe a stark contrast in signing quantity between Maven Central and the other registries. Because of its mandatory signing policy as of 2005, Maven Central had a high quantity of signed artifacts throughout the period we measured. In contrast, the other registries have had a low quantity of signed artifacts throughout. PyPI, for example, has had a historically decreasing quantity of signed artifacts until they were ultimately removed in 23 May 2023. Hugging Face has also experienced a low signing rate across its lifespan.[5] Before late 2019, Docker Hub had a worse signing rate than PyPI. From early 2020 through the middle of 2022, Docker experienced a notable increase in signing.

Finding 4: Maven Central is the only registry with a consistently high quantity of signed artifacts.

Quality of Artifact Signatures

Failure Modes: In Figure 4.4, we show the quality of signed artifacts over time. This figure shows how many of the signed artifacts in each registry were signed correctly in a given month. Within each registry, we observe no change in the quality over time. Between registries, we observe perfect quality in Docker Hub; high quality in Maven, lower and variable quality in PyPI, and spikes (due to the low number of signatures) of quality in Hugging Face.

Next we consider the failure modes of signatures by registry. For *PyPI*, see Figure 4.5. There are several common failure modes. They vary in relative frequency and none dominates. For signing failure modes in *Maven Central* see Figure 4.6. The primary failure mode in our study period is an expired public key. Revoked public keys have become less of a concern over time and missing public keys have become more of a concern since the end of 2019. Bad public keys are also on the rise. Public key creation and distribution remains challenging.

We omit figures for Docker Hub and Hugging Face. Since Docker Hub signatures are either valid or invalid (and all we measured were valid), we cannot distinguish the failure

[5]↑ The Hugging Face spikes in Figure 4.3 are due to the small number of commits in Jan 2017–Dec 2019 (only 1,266 commits in this period).

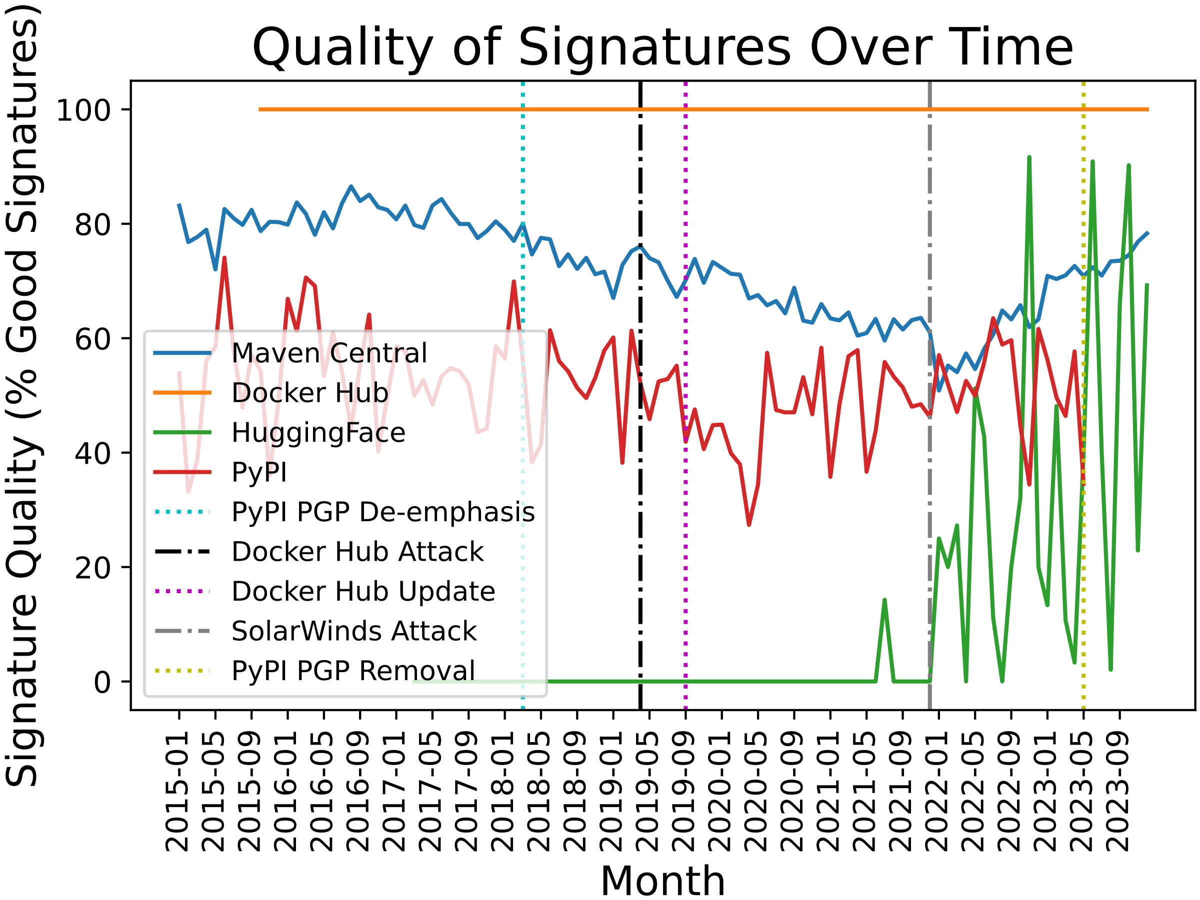

Figure 4.4. Quality of signed artifacts over time. X-axis shows time (monthly increments). Y-axis shows percentage of signatures with good status.

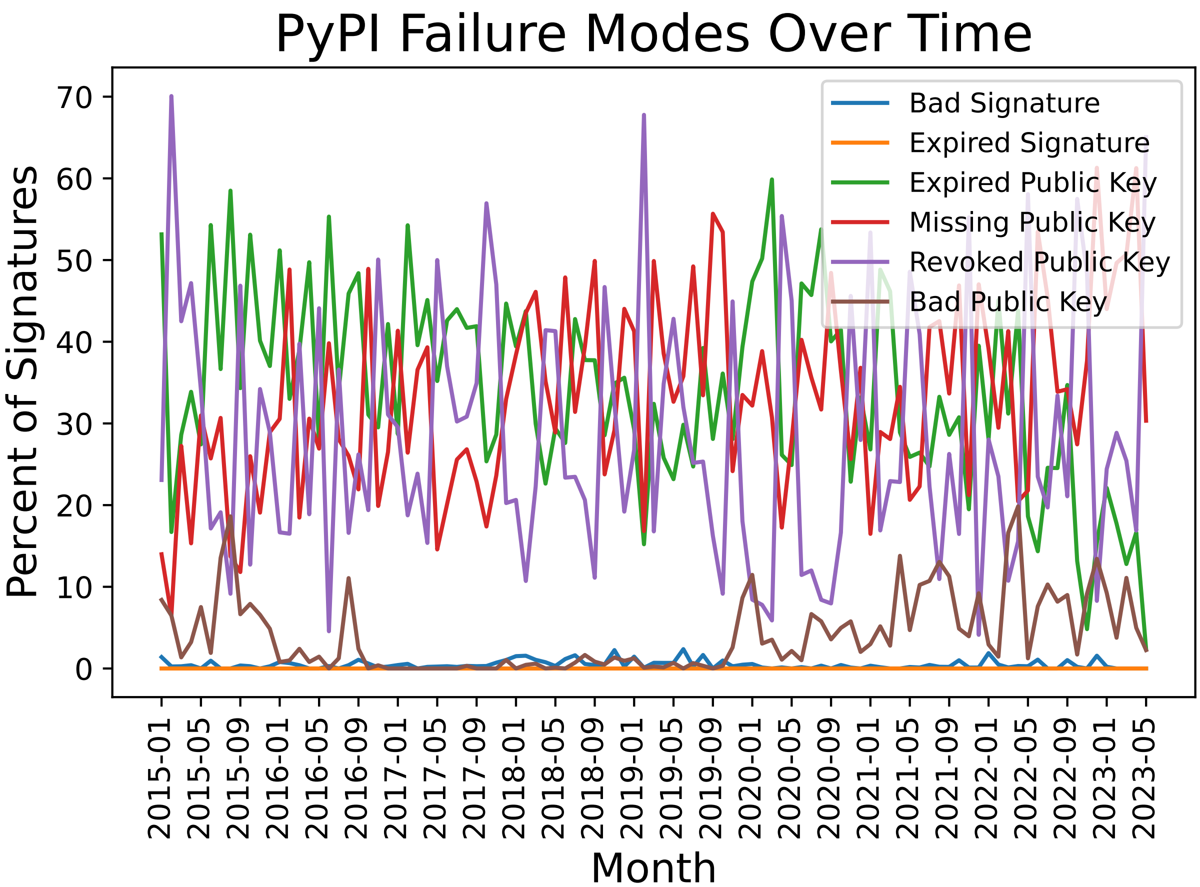

Figure 4.5. Failure modes of signatures on PyPI.

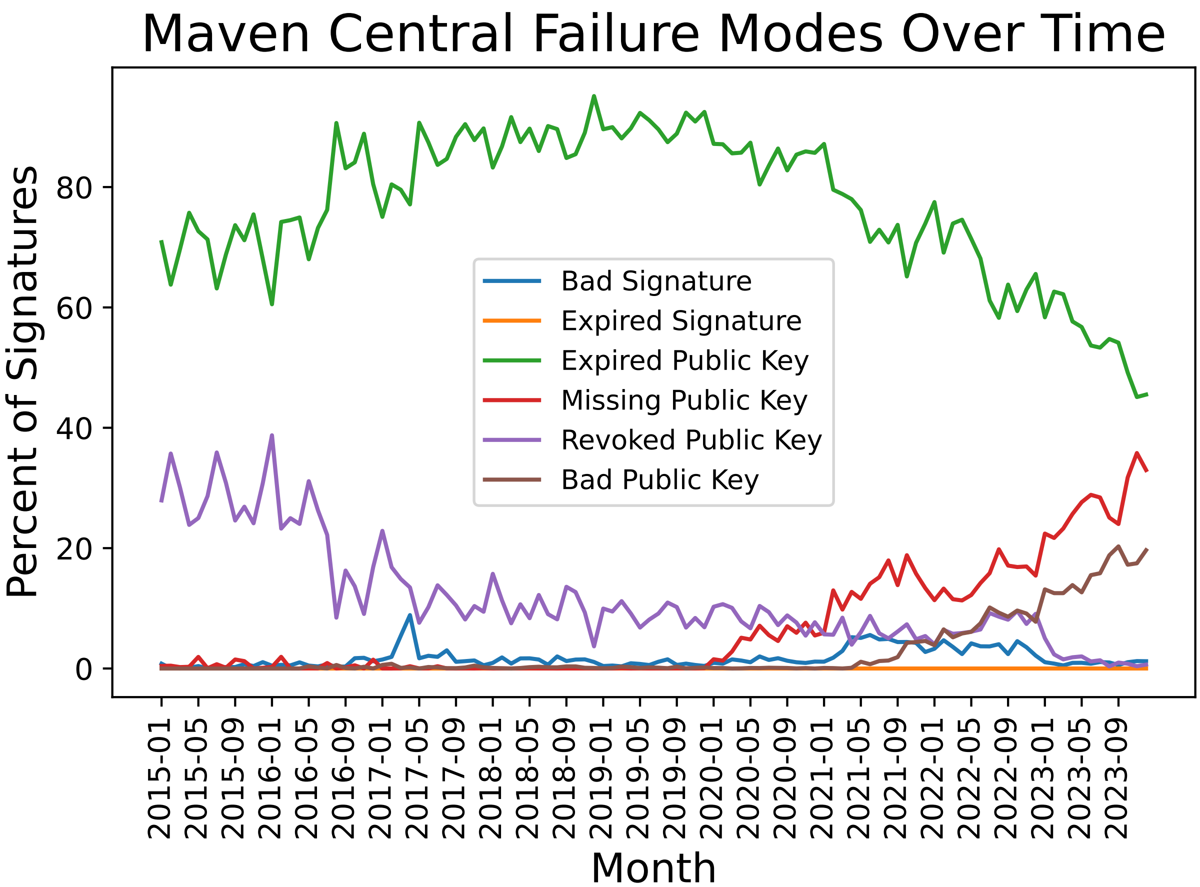

Figure 4.6. Failure modes of signatures on Maven Central.

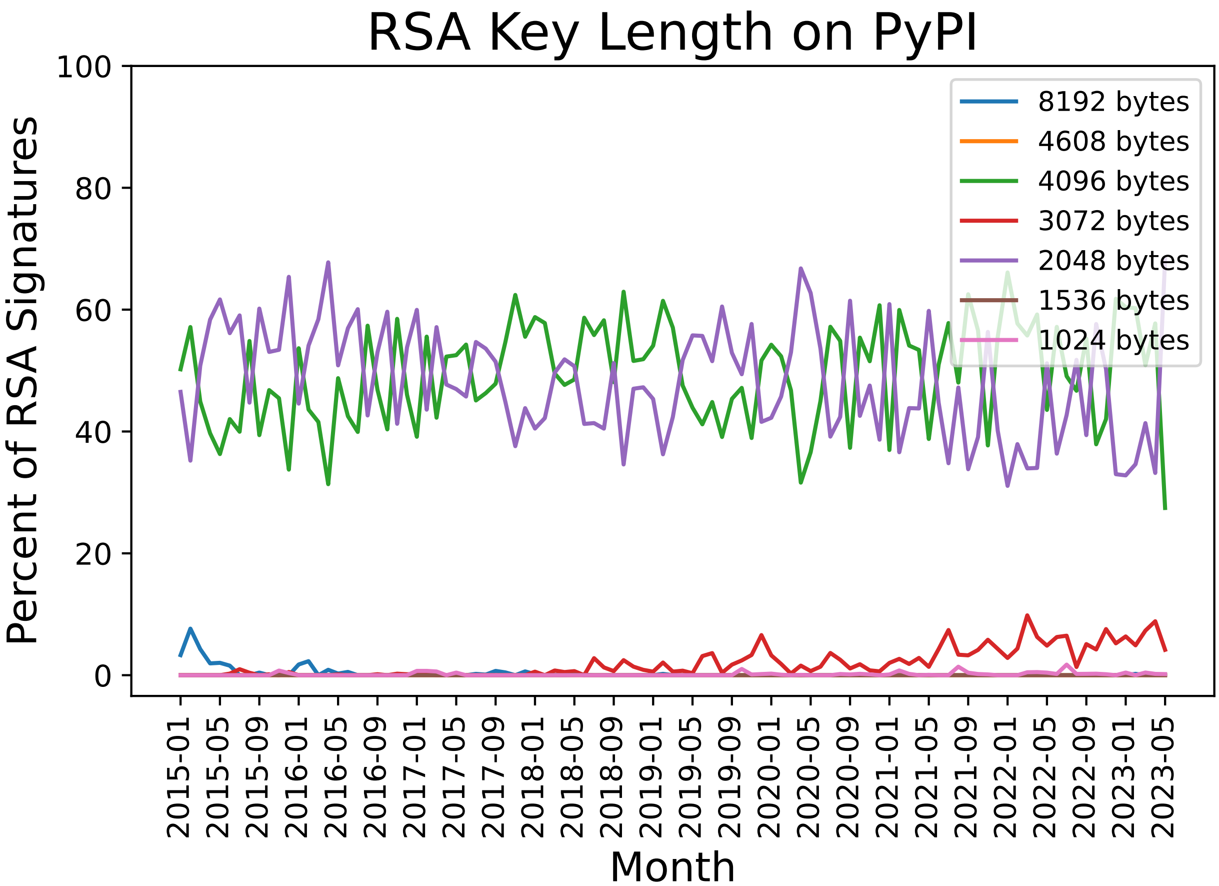

Figure 4.7. RSA key length over time in PyPI.

modes. On Hugging Face, we cannot access the maintainers' PGP keys, so we cannot analyze the failure modes of signatures there.

Finding 5: Docker Hub is the only registry with perfect quality. For Maven Central and PyPI, the most common failure modes are related to public keys in our study period.

Cryptographic Algorithms: For RSA keys on PyPI, Figure 4.7 shows the key lengths used over time. Note that 2048 and 4096 bit keys trade off as the most common over time. 3072 bit keys are also used starting in mid-2018, but remain much less common than the other two key lengths.

For RSA keys on Maven Central, Figure 4.8 shows the key lengths used over time. 2048 bit keys are initially the most common, but then drop to a similar level as the 4096 bit keys.

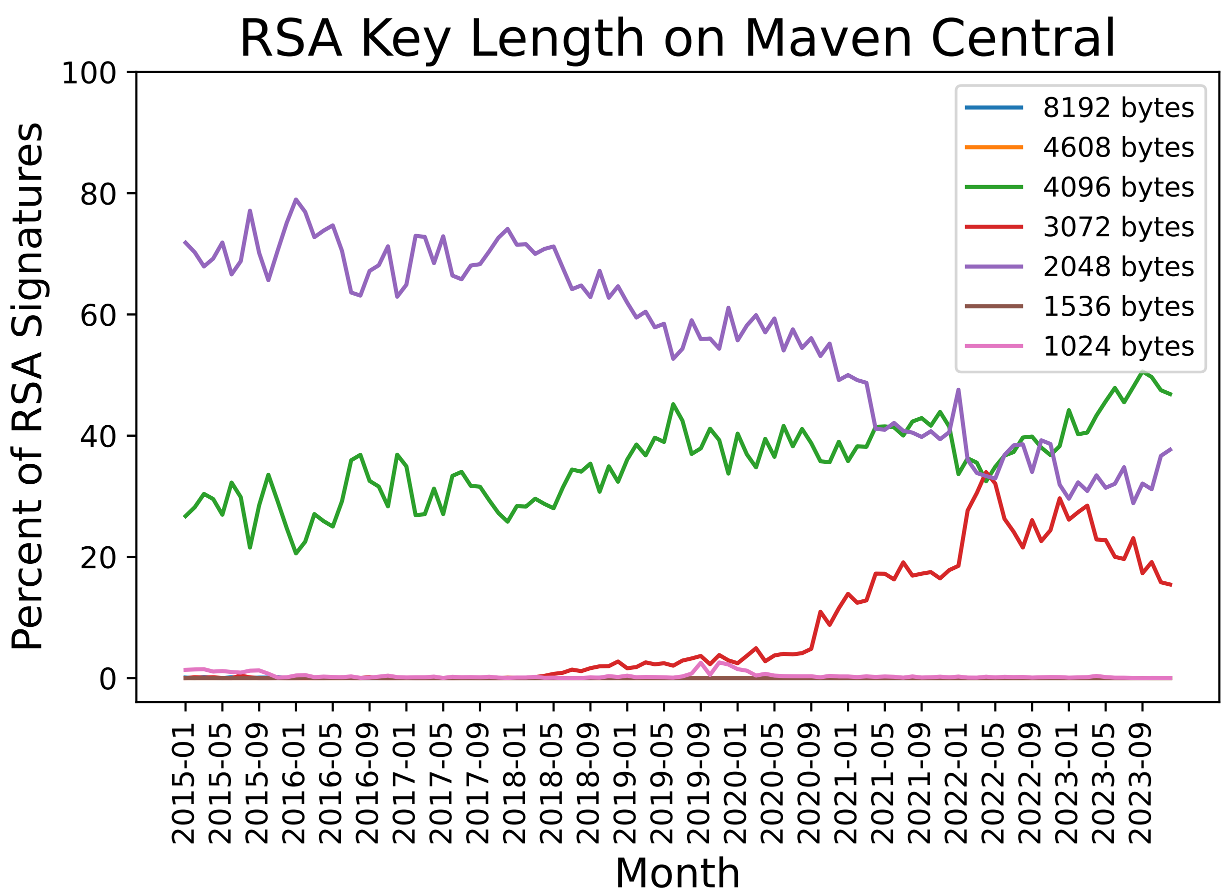

Figure 4.8. RSA key length over time in Maven Central.

As in PyPI, 3072 bit keys start to be used in mid-2018. 3072 bit keys start to replace 2048 bit keys in late 2020 but are still less common than 2048 and 4096 bit keys.

Finding 6: 2048 and 4096 bit RSA keys have remained the most common key lengths in both Maven Central and PyPI between 01 Jan 2015 and 31 Dec 2023. On Maven Central, 3072 bit keys started to replace 2048 bit keys in late 2020.

No bias from new packages

Software package registries grow over time [106]. One consideration about our longitudinal analysis is therefore whether the signing practices of new packages dominates our measure in each time window. To assess this possibility, Figure 4.9 shows the number of first-versions (*i.e.,* a new package) and subsequent-versions (*i.e.,* a new version of an existing package) of packages on PyPI, binned monthly. We note that most of the artifacts on PyPI are from subsequent-versions of packages. Maven Central, Docker Hub, and Hugging Face follow the same trend. Hence, our results reflect the ongoing practice of existing maintainers rather than the recurring mistakes of new maintainers.

4.4.3 RQ3: Influence of Incentive

To test our hypotheses, we performed a one-way ANOVA test, subsequent Tukey tests, and a selection of one-sided, two-way t-tests. The dependent variable is signature quantity, *i.e.,* daily signing rate (percent of signed artifacts).

For the one-way ANOVA test, we compared the signing rates over the entire sample period between each of our registries. We received a p-value of 0.000 and an F-statistic of 116135.82. This indicates a significant difference in signing rates between the registries.

To determine which registries are different, we performed a Tukey test. The Tukey test performs pairwise comparisons between the registries. In all cases except for the comparison between PyPI and Hugging Face (p-value of 0.000), we observed a significant difference in signing rates (all p-values of 0.000). This indicates that the signing rates of all registries are significantly different from each other except for PyPI and Hugging Face.

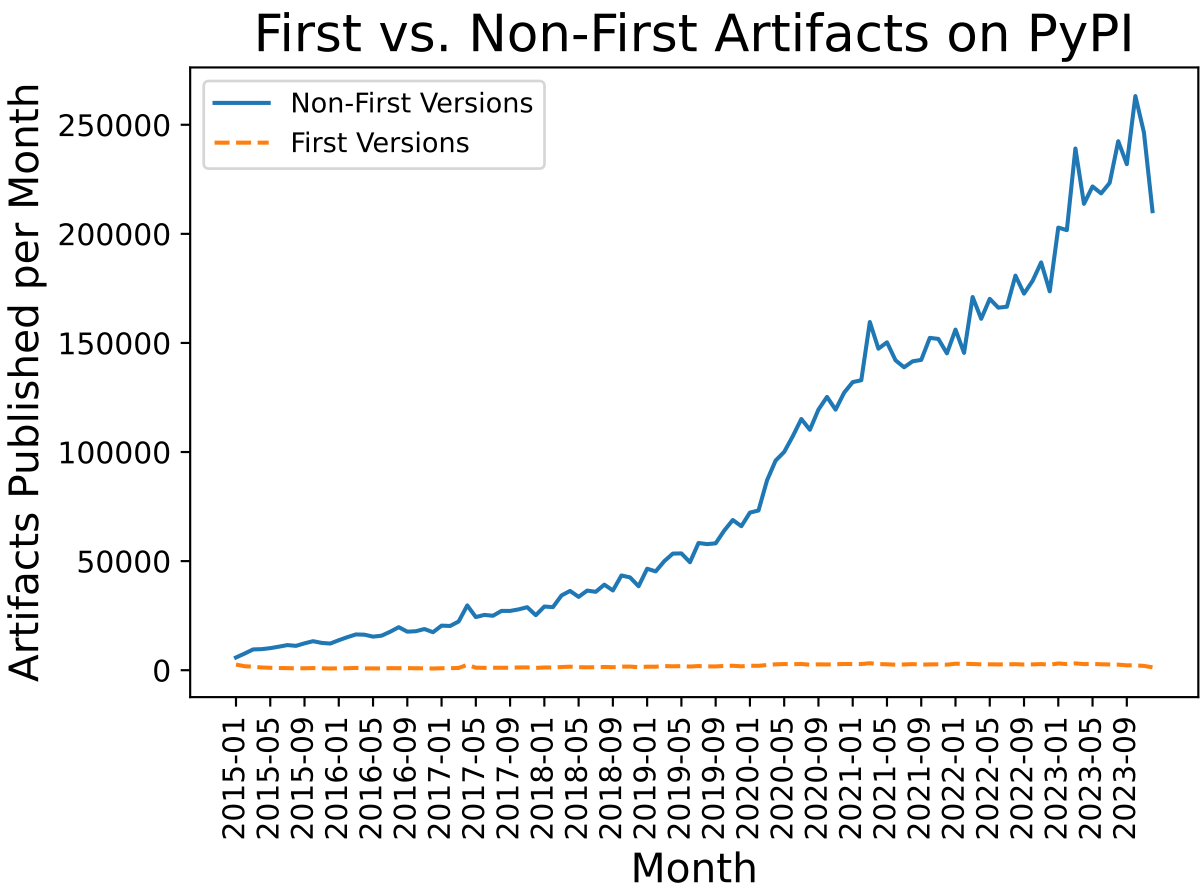

Figure 4.9. Number of first-artifacts and subsequent-artifacts of packages on PyPI.

Table 4.9. Results of t-tests for each hypothesis. Per event, we show the registry, adjusted p-value, and effect size.
*: Statistically significant value at $p < 0.01$.

Event	Registry	Adj. P-Val	Effect Size
PyPI De-emph	PyPI	**0.000***	0.228
PyPI De-emph	Maven Central	0.081	0.085
PyPI De-emph	Docker Hub	**0.005***	0.149
Docker Hub Update	PyPI	0.971	0.100
Docker Hub Update	Maven Central	0.971	0.066
Docker Hub Update	Docker Hub	**0.000***	0.392
Docker Hub Hack	PyPI	0.999	0.170
Docker Hub Hack	Maven Central	0.999	0.117
Docker Hub Hack	Docker Hub	0.021	0.129
SolarWinds	PyPI	0.922	0.015
SolarWinds	Maven Central	0.021	0.144
SolarWinds	Docker Hub	0.021	0.122

We use these results, and a series of one-sided, two-way t-tests to evaluate our hypotheses. The summary of those t-tests is shown in Table 4.9. Note that we do not include Hugging Face in our t-tests because it has a negligible quantity of signed artifacts during the time periods surrounding the selected events.

H_1: Registry Policies

H_1 predicts that registry policies will either encourage or discourage signing adoption quantity. If this is the case, we would expect to see corresponding changes in the measurements shown in Figure 4.3. For Maven Central and Hugging Face, we did not identify any changes in registry policies during the time period of this study. We did, however, identify two policy changes in PyPI and one change in Docker Hub. These are shown in Table 4.5 and appear as vertical lines in Figure 4.3.

Three observations from this figure support hypothesis H_1. On 22 Mar 2018, PyPI de-emphasized the use of PGP by removing UI elements that encouraged signing. *Our t-tests show a statistically significant change (decrease) in signing rates on PyPI after this event.*

We also measured an effect on Docker Hub, although the effect size is small and referring to Figure 4.3 there is no clear long-term trend.

Second, Docker Hub's quantity of signatures experienced an appreciable increase at the start of 2020. This increase follows a provenance visibility related update to Docker Hub, published on 05 Sep 2019.[6] In this update, Docker Hub increased tag visibility and updated security scan summaries. This update may have encouraged maintainers to sign their tags. *Our t-tests show that the only registry with a statistically significant change in signing rate after the Docker Hub update was Docker Hub.*

Third, the notable difference in signing quantity between Maven Central and the other registries suggests that mandatory signing policies encourage adoption. Since Maven Central has a mandatory signing policy, we expected, and observe, a high quantity of signatures. *Our ANOVA test, discussed earlier, found that the signing rate of Maven Central is significantly different (higher) than the others.*

H_2: Dedicated Tooling

H_2 predicts that dedicated tooling will affect both the quantity and quality of signatures. Since Docker Hub is the only registry in our study that has dedicated tooling, H_2 predicts that Docker Hub would have a higher quantity and quality of signatures than the other registries.

We do not observe support for the quantity aspect of H_2. In Figure 4.3, we observe that Docker Hub has a lower quantity of signatures than Maven Central and had lower quantity of signatures than PyPI before 2020. This suggests that the dedicated tooling on Docker Hub did not significantly impact the quantity of signatures. After all, between 01 Jan 2023 and 31 Dec 2023, Docker Hub's signing rate was only 1.0%.

However, we do find support for the quality aspect of H_2. In Figure 4.4, we observe that Docker Hub has perfect signature quality — something no other registry can achieve. This is because all signatures on Docker Hub must be created with through the DCT which, in itself, checks to make sure signatures are created correctly.

[6]↑See https://docs.docker.com/docker-hub/release-notes/#2019-09-05.

H_3: Cybersecurity Events

H_3 predicts that cybersecurity events will encourage signing adoption quantity and quality. Since these events are not specific to any registry, we would expect to see corresponding changes in the measurements shown in Figure 4.3 and Figure 4.4 from multiple registries. Table 4.5 lists several influential software supply chain attacks and cybersecurity events. However, neither preliminary visual analysis nor t-tests show any significant changes in signing rate or quality after these events.

We illustrate this with two cases. First, consider the registry-specific Docker Hub hack in April 2019. Although this attack was widely publicized and required over 100,000 users to take action to secure their accounts, this attack had little observable effect on the quantity of signatures on Docker Hub. The large increase in signing adoption on Docker Hub occurred at the start of 2020, about 9 months after the Docker Hub hack (and just after a registry-specific policy change, to the visibility of package signatures). This implies that the attack did not even have an impact on the quantity of signatures of its victim registry. Other registries were not affected at all.

Second, the SolarWinds attack in December 2020 had little effect on the quantity of signatures for any registry. This attack was one of the largest software supply chain attacks in history. It led to several government initiatives to improve software supply chain security. However, SolarWinds (and those government initiatives, *e.g.,* the subsequent executive order and NIST guidance) had no discernible effect on signing adoption in the studied registries.

H_4: Startup Cost

H_4 predicts that the first signature in a package will encourage subsequent signing. This is relatively simple to measure. First, we determine the probability of an artifact having a signature in each registry. We then determine the probability of an artifact having a signature if one of the previous artifacts from the same package has been signed. We then compare these two probabilities to determine if the first signature predicts subsequent signing.

In Table 4.10, we show both of these probabilities for each of our four registries. All registries experience an increase from the raw probability to the probability after the first signature. This suggests that overcoming the burden of signing for the first time encourages subsequent signing. The magnitude of this increase varies by registry. On Maven Central, we observe a small increase in signing probability, but this is expected since Maven Central has a mandatory signing policy. On the other platforms, the increase was ∼40x. These changes suggest that the initial burden of signing is a significant barrier to adoption.

Table 4.10. The probability of an artifact having a signature. Raw probability describes the likelihood of any artifact in the registry having a signature. After 1^{st} Signature describes the probability that an artifact will be signed if one of the previous artifacts from the same package has been signed.

Registry Name	**Raw Probability**	**After 1^{st} Signature**
PyPI	1.25%	42.60%
Maven Central	94.14%	96.00%
Docker Hub	2.47%	88.30%
Hugging Face	0.07%	15.10%

4.5 Discussion

We highlight three points for discussion.

First, our findings suggest that the long line of literature on the usability of signing tools (§2.3) may benefit from extending its perspective from an individual view to ecosystem-level considerations. Two registries, Maven and PyPI, use the same PGP-based signing method. We observe significant variations in signing adoption between these two registries, both in signature quantity and in signature quality/failure modes. Our answers to RQ3 suggest that the registry policies have a substantial effect on signing adoption, regardless of the available tooling. However, we acknowledge that our data do substantiate their concern about signature quality — our data expose major issues with signature quality in both the Maven and the PyPI registries, and that the dedicated tooling available in Docker Hub appears to eliminate the issues of signing quality.

Second, our findings suggest that registry operators control the largest incentives for software signing. Mandating signatures has not apparently decreased the popularity of Maven — we recommend that other registries do so. Registry operators can also learn from the success of Docker Hub, whose dedicated tooling results in perfect signing quality. No registry currently mandates signatures *and* provides dedicated tooling. Our results predict that the combination would result in high signature quantity and quality.

Third, we were disturbed at the non-impact of signing events — software supply chain attacks, government orders, and industry standards. Good engineering practice (not to mention engineering codes of ethics) calls for engineers to recognize and respond to known failure modes. Our contrary results motivate continued research into engineering ethics and a failure-aware software development lifecycle [117].

4.6 Threats to Validity

We distinguish three kinds of threats: construct, internal, and external.

Construct: Our study operationalized several constructs. We defined signature adoption in terms of quantity (proportion) and quality (frequency of no failure). We believe our notion of signature quantity is unobjectionable. However, signature quality is somewhat subjective. We made four assumptions that may bias our results: (1) We defined quality based on failure modes derived from the error cases of GPG; (2) We considered expired and revoked keys as failures even if the keys were valid at the time of signing; (3) We reported the cryptographic algorithm and key size for signatures on PyPI and Maven Central but did not include this as a factor in quality; and (4) We relied on the correctness of specific (albeit widely used) tools to measure the signatures.

We acknowledge that the limitations of our data sources may potentially impact the robustness of our results. Our reliance on third party metadata (*i.e.,* ecosyste.ms and BigQuery) may introduce errors and our key discovery methodology may fail to find keys that exist on small websites or that have been shared through other means. In addition, our insights into failure modes were limited by the data made available by the signing infrastructure of our target registries.

Internal: We evaluated a theory of incentive-based software signing adoption based on several hypotheses. Due to the difficulty of conducting controlled experiments of this theory, we used a quasi-experiment, and assumed no uncontrolled confounding factors. Among

Maven Central, PyPI, and Docker Hub, we have no reason to believe there would be such factors. In Hugging Face, there may be confounding factors related to the nature of the platform itself. As noted by Jiang *et al.*, Hugging Face is characterized by a "research to practice pipeline" more than traditional software package registries are [35]. Researchers have little incentive to follow secure engineering practices. This difference could comprise an uncontrolled confounding factor. However, Hugging Face had little variation in signing quantity and none of our main results relied on Hugging Face phenomena.

External: All empirical studies are limited in generalizability by the subjects they study. Our work examines four of the most popular software package registries, across three kinds of packages (traditional software, Docker containers, and machine learning models). Our results thus have some generalizability. However, our results may not generalize to contexts with substantially different properties, *e.g.,* registries more influenced by government policy or more dominated by individual organizations.

4.7 Conclusion

In this study, we assessed signing in four public software package registries (PyPI, Maven Central, Docker Hub, and Hugging Face). We measured signature adoption (quantity and quality). We found that, aside from Maven Central, the quantity of signatures in package registries is low. Aside from Docker Hub, the quality of signatures in package registries is low. To explain these observations, we proposed and evaluated an incentive-basd theory explaining maintainer's decisions to adopt signatures. We used quasi-experiments to test four hypotheses. We found that incentives do influence signing adoption, and some incentives are more influential than others. Registry policies and startup costs seem to have the largest impact on signing adoption. Cybersecurity events do not appear to have a significant impact on signing adoption.

We hope that our results will encourage the software engineering community to improve their software signing efforts, enhancing the overall security of software systems. Our findings suggest specific incentives that could significantly improve software signing adoption rates.

4.8 Data Availability

The tools used to collect and analyze the data are available at https://github.com/PurdueDualityLab/signature-adoption. This repository also contains the reported data in a relational database snapshot.

5. CONCLUSIONS AND RECOMMENDATIONS

5.1 Summary

In §1.2, I stated my thesis:

> Since modern software relies increasingly on third party dependencies, software supply chain attacks are a growing, and challenging, threat. As a result, techniques like software signing must be effectively implemented to provide defensive benefits.

In chapters 3 and 4, I provided evidence supporting this thesis. I summarize this evidence in the remainder of this section.

5.1.1 SSC Attacks and Security Properties

In chapter 3, I systematized knowledge of software supply chain attacks and defenses. I compiled existing literature into a four stage attack pattern to describe software supply chain attacks. I also identified a set of three properties that characterize secure software supply chains. These contributions reveal the present state of software supply chain attacks and defenses.

Current academic, industry, and government literature shows that software supply chain attacks are increasing in frequency and severity. The previous lack of a unified attack pattern suggested that the software engineering community did not have a common understanding of software supply chain attacks. Furthermore, the absence of comprehensive security properties (and the current development of multiple security frameworks) indicates that the community is still developing a shared understanding of a secure software supply chain. Therefore, the threats to software supply chains, and our defenses against them, are still evolving.

5.1.2 Signature Adoption in Four Public Package Registries

In chapter 4, I assessed software signing, a foundational security technique, across four public software package registries. I measured signature adoption (quantity and quality) and also evaluated an incentive-based theory explaining maintainer's decisions to adopt signatures. I found that the state of signing in package registries is generally poor. Except for Maven Central, the quantity of signatures in package registries is low. Except for Docker Hub, the quality of signatures in package registries is low. These contributions reveal that software signing is not being used to its full potential.

Signing cannot prevent all software supply chain attacks by itself, but it is a common and mature technique to increase security. As supply chain attacks evolve, so too do our defenses against them. Even though these defenses take many forms, they must be correctly used to provide defensive benefits. Our results show that there is a significant gap between the potential and actual use of software signing in package registries. As a fundamental security technique, the current state of software signing in package registries is concerning. The software engineering community must work together to close the gap between the potential and actual use of techniques like signing.

5.2 Future Work

We suggest several directions for future research.

Further Diversification of Registries: Our findings are provocative, but we recommend diversifying the types of ecosystems under study. Further work could go beyond open-source registries to include commercial and proprietary registries (*e.g.,* app stores), and ecosystems implementing different forms of signing solutions. This approach will facilitate a comprehensive exploration of other factors, incentives and cost trade-offs that influence the adoption of software signing for these types of ecosystems.

Incorporating Human Factors: Our approach is grounded in a theory of software signing based in incentives. Qualitative data — *e.g.,* surveys and interviews of engineers in the registries of interest — would shed light on the relative weight of the factors we identified. Such studies could expose new factors for quantitative evaluation.

Identifying Machine Learning (ML) Software and Pre-Trained Model Signing requirements: Signing adoption rates in the Hugging Face registry are much lower than in all other studied registries. We recommend research on signing practices in this context. The issue might be an odd signing target — commits rather than packages. The challenge might be more fundamental, clarifying the nature of effective signatures for ML models and training regimes [118].

Apply the results: As noted in §4.5, registry operators appear to have a strong influence on software signing quantity and quality. Partnering with registry operators, researchers can apply our results to empirically validate them.

REFERENCES

[1] G. 18F, *18F: Digital service delivery | Open source policy*, en-US, https://18f.gsa.gov/open-source-policy/.

[2] D. of Defense, *Open Source Software (OSS) in the Department of Defense (DoD)*, May 2003.

[3] Synopsys, "2023 OSSRA Report," en, Synopsys, Tech. Rep., 2023, https://www.synopsys.com/software-integrity/engage/ossra/rep-ossra-2023-pdf.

[4] 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 *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, ser. SCORED'22, Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 15–24, ISBN: 9781450398855. DOI: 10.1145/3560835.3564556. [Online]. Available: https://doi.org/10.1145/3560835.3564556.

[5] FireEye, *Highly evasive attacker leverages solarwinds supply chain to compromise multiple global victims with sunburst backdoor*, https://www.fireeye.com/blog/threat-research/2020/12/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor.html.

[6] Microsoft Security Response Center, *Customer guidance on recent nation-state cyber attacks*, https://msrc-blog.microsoft.com/2020/12/13/customer-guidance-on-recent-nation-state-cyber-attacks/. [Online]. Available: https://msrc-blog.microsoft.com/2020/12/13/customer-guidance-on-recent-nation-state-cyber-attacks/.

[7] Catalin Cimpanu, *Microsoft, fireeye confirm solarwinds supply chain attack*, https://www.zdnet.com/article/microsoft-fireeye-confirm-solarwinds-supply-chain-attack/. [Online]. Available: https://www.zdnet.com/article/microsoft-fireeye-confirm-solarwinds-supply-chain-attack/.

[8] Sonatype, "State of the Software Supply Chain: 2022," Sonatype, Tech. Rep. 8th Annual, 2022, https://www.sonatype.com/state-of-the-software-supply-chain/open-source-supply-demand-security.

[9] M. Ohm, H. Plate, A. Sykosch, and M. Meier, “Backstabbers knife collection: A review of open source software supply chain attacks,” in *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 2426, 2020, Proceedings*, Lisbon, Portugal: Springer-Verlag, 2020, pp. 23–43, ISBN: 978-3-030-52682-5. DOI: 10.1007/978-3-030-52683-2_2. [Online]. Available: https://doi.org/10.1007/978-3-030-52683-2_2.

[10] M. Zimmermann, C.-A. Staicu, and M. Pradel, “Small World with High Risks: A Study of Security Threats in the npm Ecosystem,” in *USENIX Security Symposium*, 2019.

[11] A. Decan, T. Mens, and E. Constantinou, “On the impact of security vulnerabilities in the npm package dependency network,” in *International Conference on Mining Software Repositories (MSR)*, 2018.

[12] A. Zerouali, T. Mens, A. Decan, and C. De Roover, “On the impact of security vulnerabilities in the npm and rubygems dependency networks,” *Empirical Software Engineering*, vol. 27, no. 5, pp. 1–45, 2022.

[13] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, and L. Williams, “What are weak links in the npm supply chain?” In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP ’22, Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 331–340, ISBN: 9781450392266. DOI: 10.1145/3510457.3513044. [Online]. Available: https://doi.org/10.1145/3510457.3513044.

[14] N. Zahan, P. Kanakiya, B. Hambleton, S. Shohan, and L. Williams, *OpenSSF Scorecard: On the Path Toward Ecosystem-wide Automated Security Metrics*, http://arxiv.org/abs/2208.03412, Jan. 2023. DOI: 10.48550/arXiv.2208.03412.

[15] N. Zahan, S. Shohan, D. Harris, and L. Williams, “Do software security practices yield fewer vulnerabilities?” In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, IEEE, 2023, pp. 292–303.

[16] K. Serebryany, “{OSS-Fuzz} - Google’s continuous fuzzing service for open source software,” en, 2017. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany.

[17] N. Luo, T. Antonopoulos, W. R. Harris, R. Piskac, E. Tromer, and X. Wang, "Proving unsat in zero knowledge," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22, Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 2203–2217, ISBN: 9781450394505. DOI: 10.1145/3548606.3559373. [Online]. Available: https://doi.org/10.1145/3548606.3559373.

[18] M. Balliu, B. Baudry, S. Bobadilla, *et al.*, "Challenges of producing software bill of materials for java," *arXiv preprint arXiv:2303.11102*, 2023.

[19] N. Zahan, E. Lin, M. Tamanna, W. Enck, and L. Williams, "Software bills of materials are required. are we there yet?" *IEEE Security & Privacy*, vol. 21, no. 2, pp. 82–88, 2023.

[20] T. Winters, T. Manshreck, and H. Wright, *Software engineering at Google: lessons learned from programming over time*, First edition. Beijing [China]: O'Reilly Media, 2020, ISBN: 978-1-4920-8279-8.

[21] T. K. Kuppusamy, S. Torres-Arias, V. Diaz, and J. Cappos, "Diplomat: Using delegations to protect community repositories," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/kuppusamy, Santa Clara, CA: USENIX Association, Mar. 2016, pp. 567–581, ISBN: 978-1-931971-29-4.

[22] W. Woodruff, *PGP signatures on PyPI: Worse than useless*, en, https://blog.yossarian.net/2023/05/21/PGP-signatures-on-PyPI-worse-than-useless, May 2023.

[23] M. Felderer and G. H. Travassos, Eds., *Contemporary empirical methods in software engineering*, eng. Cham: Springer, 2020, ISBN: 978-3-030-32488-9.

[24] T. R. Schorlemmer, K. G. Kalu, L. Chigges, *et al.*, "Signing in four public software package registries: Quantity, quality, and influencing factors," in *2024 IEEE Symposium on Security and Privacy (SP)*, Los Alamitos, CA, USA: IEEE Computer Society, May 2024.

[25] H. Stadtler and C. Kilger, Eds., *Supply Chain Management and Advanced Planning*, en, 4th. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, ISBN: 978-3-540-74511-2. DOI: 10.1007/978-3-540-74512-9. [Online]. Available: http://link.springer.com/10.1007/978-3-540-74512-9.

[26] European Union Agency for Cybersecurity, “ENISA threat landscape for supply chain attacks.,” en, Publications Office, LU, Tech. Rep., Jul. 2021. [Online]. Available: https://data.europa.eu/doi/10.2824/168593.

[27] R. J. Ellison, J. B. Goodenough, C. B. Weinstock, and C. Woody, “Evaluating and Mitigating Software Supply Chain Security Risks,” en, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, Tech. Rep., May 2010, Section: Technical Reports. [Online]. Available: https://apps.dtic.mil/sti/citations/ADA522538.

[28] C. Nissen, J. E. Gronager, R. S. Metzger, and H. Rishikof, “Deliver uncompromised: A strategy for supply chain security and resilience in response to the changing character of war,” MITRE CORP MCLEAN VA, Tech. Rep., 2018.

[29] J. D. Herbsleb, “Global software engineering: The future of socio-technical coordination,” in *Future of Software Engineering (FOSE'07)*, IEEE, 2007, pp. 188–198.

[30] D. mite, C. Wohlin, T. Gorschek, and R. Feldt, “Empirical evidence in global software engineering: A systematic review,” *Empirical Softw. Engg.*, vol. 15, no. 1, pp. 91–118, Feb. 2010, ISSN: 1382-3256. DOI: 10.1007/s10664-009-9123-y. [Online]. Available: https://doi.org/10.1007/s10664-009-9123-y.

[31] W. Jiang, N. Synovic, M. Hyatt, *et al.*, “An empirical study of pre-trained model reuse in the hugging face deep learning model registry,” in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23, Melbourne, Victoria, Australia: IEEE Press, 2023, pp. 2463–2475, ISBN: 9781665457019. DOI: 10.1109/ICSE48619.2023.00206. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00206.

[32] SourceForge, *Compare, Download & Develop Open Source & Business Software - SourceForge*, https://sourceforge.net/.

[33] GitLab, *The DevSecOps Platform*, en-us, https://about.gitlab.com/.

[34] *GitHub Packages*, https://github.com/features/packages, Accessed: 2023-12-06.

[35] W. Jiang, N. Synovic, R. Sethi, *et al.*, "An empirical study of artifacts and security risks in the pre-trained model supply chain," in *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, ser. SCORED'22, Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 105–114, ISBN: 9781450398855. DOI: 10.1145/3560835.3564547. [Online]. Available: https://doi.org/10.1145/3560835.3564547.

[36] W. Enck and L. Williams, "Top five challenges in software supply chain security: Observations from 30 industry and government organizations," *IEEE Security & Privacy*, vol. 20, no. 2, pp. 96–100, 2022. DOI: 10.1109/MSEC.2022.3142338.

[37] D. Kumar, Z. Ma, Z. Durumeric, *et al.*, "Security challenges in an increasingly tangled web," in *Proceedings of the 26th International Conference on World Wide Web*, 2017, pp. 677–684.

[38] Y. Wang, B. Chen, K. Huang, *et al.*, "An empirical study of usages, updates and risks of third-party libraries in java projects," en, in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Adelaide, Australia: IEEE, Sep. 2020, pp. 35–45, ISBN: 978-1-72815-619-4. DOI: 10.1109/ICSME46990.2020.00014. [Online]. Available: https://ieeexplore.ieee.org/document/9240619/.

[39] C. R. de Souza and D. F. Redmiles, "An empirical study of software developers' management of dependencies and changes," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 241–250.

[40] I. Pashchenko, D.-L. Vu, and F. Massacci, "A qualitative study of dependency management and its security implications," in *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, 2020, pp. 1513–1531.

[41] A. S. Jadhav and R. M. Sonar, "Evaluating and selecting software packages: A review," *Information and software technology*, vol. 51, no. 3, pp. 555–563, 2009.

[42] A. S. Jadhav and R. M. Sonar, "Framework for evaluation and selection of the software packages: A hybrid knowledge based system approach," *Journal of Systems and Software*, vol. 84, no. 8, pp. 1394–1407, 2011.

[43] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, vol. 24, pp. 381–416, 2019.

[44] J. Ghofrani, P. Heravi, K. A. Babaei, and M. D. Soorati, “Trust challenges in reusing open source software: An interview-based initial study,” in *Proceedings of the 26th ACM International Systems and Software Product Line Conference-Volume B*, 2022, pp. 110–116.

[45] D.-L. Vu, F. Massacci, I. Pashchenko, H. Plate, and A. Sabetta, “LastPyMile: Identifying the discrepancy between sources and packages,” en, in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Athens Greece: ACM, Aug. 2021, pp. 780–792, ISBN: 978-1-4503-8562-6. DOI: 10.1145/3468264.3468592. [Online]. Available: https://dl.acm.org/doi/10.1145/3468264.3468592.

[46] S. Torres-Arias, H. Afzali, T. K. Kuppusamy, R. Curtmola, and J. Cappos, “In-toto: Providing farm-to-table guarantees for bits and bytes,” in *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1393–1410, ISBN: 978-1-939133-06-9. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/torres-arias.

[47] C. Lamb and S. Zacchiroli, “Reproducible Builds: Increasing the Integrity of Software Supply Chains,” *IEEE Software*, vol. 39, no. 2, pp. 62–70, Mar. 2022, Conference Name: IEEE Software, ISSN: 1937-4194. DOI: 10.1109/MS.2021.3073045.

[48] J. Hejderup, “On the Use of Tests for Software Supply Chain Threats,” en, in *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, Los Angeles CA USA: ACM, Nov. 2022, pp. 47–49, ISBN: 978-1-4503-9885-5. DOI: 10.1145/3560835.3564557. [Online]. Available: https://dl.acm.org/doi/10.1145/3560835.3564557.

[49] G. Benedetti, “Automatic Security Assessment of GitHub Actions Workflows,” en, *Los Angeles*, 2022, https://dl.acm.org/doi/abs/10.1145/3560835.3564554.

[50] *The Minimum Elements For a Software Bill of Materials (SBOM) | National Telecommunications and Information Administration*. [Online]. Available: https://www.ntia.gov/report/2021/minimum-elements-software-bill-materials-sbom.

[51] M. Ohm, A. Sykosch, and M. Meier, “Towards detection of software supply chain attacks by forensic artifacts,” in *Proceedings of the 15th International Conference on Availability, Reliability and Security*, ser. ARES ’20, New York, NY, USA: Association for Computing Machinery, Aug. 2020, pp. 1–6, ISBN: 978-1-4503-8833-7. DOI: 10.1145/3407023.3409183. [Online]. Available: https://doi.org/10.1145/3407023.3409183.

[52] R. Shirey, "Internet security glossary, version 2," Network Working Group, RFC 4949, Aug. 2007, FYI: 36. Obsoletes: 2828. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4949.html.

[53] Security Technical Advisory Group, "Software Supply Chain Best Practices," en, Cloud Native Computing Foundation, Tech. Rep., May 2021. [Online]. Available: https://project.linuxfoundation.org/hubfs/CNCF_SSCP_v1.pdf.

[54] *Supply-chain Levels for Software Artifacts*, en. [Online]. Available: https://slsa.dev/.

[55] David Cooper, Andrew Regenscheid, Murugiah Souppaya, *et al.*, "Security Considerations for Code Signing," en, *NIST Cybersecurity White Paper*, Jan. 2018, https://csrc.nist.rip/external/nvlpubs.nist.gov/nistpubs/CSWP/NIST.CSWP.01262018.pdf.

[56] Cybersecurity and Infrastructure Security Agency, "Defending against software supply chain attacks," Cybersecurity and Infrastructure Security Agency, Technical Report, Apr. 2021. [Online]. Available: https://www.cisa.gov/sites/default/files/publications/defending_against_software_supply_chain_attacks_508_1.pdf.

[57] Project, The People of the GnuPG, *The GNU Privacy Guard*, en, Publisher: The GnuPG Project, Apr. 2023. [Online]. Available: https://gnupg.org/.

[58] N. K. Gopalakrishna, D. Anandayuvaraj, A. Detti, F. L. Bland, S. Rahaman, and J. C. Davis, "" if security is required" engineering and security practices for machine learning-based iot devices," in *Proceedings of the 4th International Workshop on Software Engineering Research and Practice for the IoT*, 2022, pp. 1–8.

[59] N. Meng, S. Nagy, D. Yao, W. Zhuang, and G. A. Argoty, "Secure coding practices in java: Challenges and vulnerabilities," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 372–383.

[60] M. Chen, F. Fischer, N. Meng, X. Wang, and J. Grossklags, "How reliable is the crowdsourced knowledge of security implementation?" In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, 2019, pp. 536–547.

[61] A. Whitten and J. D. Tygar, "Why Johnny can't encrypt: A usability evaluation of PGP 5.0," in *Proceedings of the 8th conference on USENIX Security Symposium - Volume 8*, ser. SSYM'99, USA: USENIX Association, Aug. 1999, p. 14.

[62] S. Ruoti, J. Andersen, D. Zappala, and K. Seamons, "Why johnny still, still can't encrypt: Evaluating the usability of a modern pgp client," *arXiv*, 2016, https://arxiv.org/abs/1510.08555. arXiv: 1510.08555 [cs.CR].

[63] Gillian Andrews, *Usability Report: Proposed Mailpile Features.* OpenITP, https://openitp.org/field-notes/user-tests-mailpile-features.html Accessed on 27/06/2023, December 2014.

[64] C. Braz and J.-M. Robert, "Security and usability: The case of the user authentication methods," in *Proceedings of the 18th Conference on l'Interaction Homme -Machine*, ser. IHM '06, https://dl.acm.org/doi/10.1145/1132736.1132768, New York, NY, USA: Association for Computing Machinery, Apr. 2006, pp. 199–203, ISBN: 978-1-59593-350-8. DOI: 10.1145/1132736.1132768.

[65] S. Ruoti, N. Kim, B. Burgon, T. van der Horst, and K. Seamons, "Confused Johnny: When automatic encryption leads to confusion and mistakes," in *Proceedings of the Ninth Symposium on Usable Privacy and Security*, ser. SOUPS '13, New York, NY, USA: Association for Computing Machinery, Jul. 2013, pp. 1–12, ISBN: 978-1-4503-2319-2. DOI: 10.1145/2501604.2501609. [Online]. Available: https://dl.acm.org/doi/10.1145/2501604.2501609.

[66] A. Reuter, K. Boudaoud, M. Winckler, A. Abdelmaksoud, and W. Lemrazzeq, "Secure Email - A Usability Study," en, in *Financial Cryptography and Data Security*, M. Bernhard, A. Bracciali, L. J. Camp, *et al.*, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2020, pp. 36–46, ISBN: 978-3-030-54455-3. DOI: 10.1007/978-3-030-54455-3_3.

[67] S. Fahl, M. Harbach, T. Muders, M. Smith, and U. Sander, "Helping Johnny 2.0 to encrypt his Facebook conversations," in *Proceedings of the Eighth Symposium on Usable Privacy and Security*, ser. SOUPS '12, https://dl.acm.org/doi/10.1145/2335356.2335371, New York, NY, USA: Association for Computing Machinery, Jul. 2012, pp. 1–17, ISBN: 978-1-4503-1532-6. DOI: 10.1145/2335356.2335371.

[68] S. Ruoti, J. Andersen, T. Hendershot, D. Zappala, and K. Seamons, *Private Webmail 2.0: Simple and Easy-to-Use Secure Email*, http://arxiv.org/abs/1510.08435, Oct. 2015. DOI: 10.1145/2984511:2984580.

[69] E. Atwater, C. Bocovich, U. Hengartner, E. Lank, and I. Goldberg, "Leading Johnny to water: Designing for usability and trust," in *Proceedings of the Eleventh USENIX Conference on Usable Privacy and Security*, ser. SOUPS '15, https://www.usenix.org/system/files/conference/soups2015/soups15-paper-atwater.pdf, USA: USENIX Association, Jul. 2015, pp. 69–88, ISBN: 978-1-931971-24-9.

[70] Latacora, *The PGP Problem*, https://latacora.micro.blog/2019/07/16/the-pgp-problem.html, Jul. 2019.

[71] M. Green, *What's the matter with PGP?* en, https://blog.cryptographyengineering.com/2014/08/13/whats-matter-with-pgp/, Aug. 2014.

[72] R. Chandramouli, M. Iorga, and S. Chokhani, "Cryptographic key management issues and challenges in cloud services," *Secure Cloud Computing*, pp. 1–30, 2013.

[73] Z. Newman, J. S. Meyers, and S. Torres-Arias, "Sigstore: Software Signing for Everybody," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22, New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 2353–2367, ISBN: 978-1-4503-9450-5. DOI: 10.1145/3548606.3560596. [Online]. Available: https://dl.acm.org/doi/10.1145/3548606.3560596.

[74] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "{Coniks}: Bringing key transparency to end users," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 383–398.

[75] H. Malvai, L. Kokoris-Kogias, A. Sonnino, *et al.*, "Parakeet: Practical key transparency for end-to-end encrypted messaging," *Cryptology ePrint Archive*, 2023.

[76] E. Heftrig, H. Shulman, and M. Waidner, "Poster: The unintended consequences of algorithm agility in dnssec," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 3363–3365.

[77] A. Bellissimo, J. Burgess, and K. Fu, "Secure software updates: Disappointments and new challenges," in *First USENIX Workshop on Hot Topics in Security (HotSec 06)*, Vancouver, B.C. Canada: USENIX Association, Jul. 2006. [Online]. Available: https://www.usenix.org/conference/hotsec-06/secure-software-updates-disappointments-and-new-challenges.

[78] K. Merrill, Z. Newman, S. Torres-Arias, and K. Sollins, *Speranza: Usable, privacy-friendly software signing*, http://arxiv.org/abs/2305.06463, May 2023.

[79] OpenBSD, *Signify: Securing openbsd from us to you*, "https://www.openbsd.org/papers/bsdcan-signify.html".

[80] P. Ladisa, H. Plate, M. Martinez, and O. Barais, “Sok: Taxonomy of attacks on open-source software supply chains,” in *2023 IEEE Symposium on Security and Privacy (SP)*, Los Alamitos, CA, USA: IEEE Computer Society, May 2023, pp. 1509–1526. DOI: 10.1109/SP46215.2023.10179304. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.10179304.

[81] Sonatype, “State of the software supply chain: 2021,” Sonatype, Tech. Rep., 2021, https://www.sonatype.com/resources/state-of-the-software-supply-chain-2021.

[82] Symantec Corporation, *Internet threat security report: 2018*, https://www.symantec.com/content/dam/symantec/docs/reports/istr-23-2018-en.pdf, 2018.

[83] Symantec Corporation, *Internet threat security report: 2019*, https://www.symantec.com/content/dam/symantec/docs/reports/istr-24-2019-en.pdf, 2019.

[84] L. Martin, *Cyber Kill Chainő*, en, Jun. 2022. [Online]. Available: https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html.

[85] Microsoft, “3 ways to mitigate risk when using private package feeds,” en, Microsoft, Tech. Rep., Mar. 2021, p. 15. [Online]. Available: https://azure.microsoft.com/mediahandler/files/resourcefiles/3-ways-to-mitigate-risk-using-private-package-feeds/3%20Ways%20to%20Mitigate%20Risk%20When%20Using%20Private%20Package%20Feeds%20-%20v1.0.pdf.

[86] M. Ensor and D. Stevens, “Shifting left on security - Securing software supply chains,” en, Google Cloud, Tech. Rep., Feb. 2021, https://cloud.google.com/solutions/shifting-left-on-security, p. 36.

[87] J. Boyens, A. Smith, N. Bartol, K. Winkler, A. Holbrook, and M. Fallon, “Cybersecurity Supply Chain Risk Management Practices for Systems and Organizations,” en, National Institute of Standards and Technology, Tech. Rep. NIST Special Publication (SP) 800-161 Rev. 1, May 2022. DOI: 10.6028/NIST.SP.800-161r1. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-161/rev-1/final.

[88] Joint Task Force Interagency Working Group, “Security and Privacy Controls for Information Systems and Organizations,” en, National Institute of Standards and Technology, Tech. Rep., Sep. 2020, Edition: Revision 5. DOI: 10.6028/NIST.SP.800-53r5. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r5.pdf.

[89] D. C. Brown, "Digital nudges for encouraging developer behaviors," English, Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2023-07-19, Ph.D. dissertation, 2021, p. 160, ISBN: 9798522954932. [Online]. Available: https://www.proquest.com/dissertations-theses/digital-nudges-encouraging-developer-behaviors/docview/2566242524/se-2.

[90] E. L. Deci, "Effects of externally mediated rewards on intrinsic motivation," *Journal of Personality and Social Psychology*, vol. 18, no. 1, pp. 105–115, 1971, Place: US Publisher: American Psychological Association, ISSN: 1939-1315. DOI: 10.1037/h0030644.

[91] M. Baddeley, *Behavioural Economics: A Very Short Introduction*. Oxford University Press, Jan. 2017, ISBN: 9780198754992. DOI: 10.1093/actrade/9780198754992.001.0001. [Online]. Available: https://doi.org/10.1093/actrade/9780198754992.001.0001.

[92] E. Kamenica, "Behavioral economics and psychology of incentives," *Annual Review of Economics*, vol. 4, no. 1, pp. 427–452, 2012. DOI: 10.1146/annurev-economics-080511-110909. eprint: https://doi.org/10.1146/annurev-economics-080511-110909. [Online]. Available: https://doi.org/10.1146/annurev-economics-080511-110909.

[93] U. Gneezy and A. Rustichini, "Pay enough or don't pay at all," *The Quarterly Journal of Economics*, vol. 115, no. 3, pp. 791–810, 2000, ISSN: 00335533, 15314650. [Online]. Available: http://www.jstor.org/stable/2586896.

[94] R. M. Titmuss, *The gift relationship (reissue): From human blood to social policy*, 1st ed. Bristol University Press, 2018. [Online]. Available: http://www.jstor.org/stable/j.ctv6zdcmh.

[95] D. Gotterbarn, K. Miller, and S. Rogerson, "Software engineering code of ethics," *Communications of the ACM*, vol. 40, no. 11, pp. 110–118, 1997.

[96] D. K. Remler and G. G. Van Ryzin, *Research methods in practice: strategies for description and causation*, Second edition. Los Angeles: SAGE, 2015, ISBN: 978-1-4522-7640-3.

[97] C. Wohlin, *Experimentation in software engineering*. New York: Springer, 2012, ISBN: 978-3-642-29043-5.

[98] J. M. Wooldridge, *Introductory econometrics: a modern approach*, 5th ed. Mason, OH: South-Western Cengage Learning, 2013, OCLC: ocn827938223, ISBN: 978-1-111-53104-1.

[99] B. D. Meyer, "Natural and Quasi-Experiments in Economics," *Journal of Business & Economic Statistics*, vol. 13, no. 2, pp. 151–161, 1995, https://www.jstor.org/stable/1392369, ISSN: 0735-0015. DOI: 10.2307/1392369.

[100] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "When and how to make breaking changes: Policies and practices in 18 open source software ecosystems," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 4, Jul. 2021, ISSN: 1049-331X. DOI: 10.1145/3447245. [Online]. Available: https://doi.org/10.1145/3447245.

[101] S. Cass, *The top programming languages 2023*, https://spectrum.ieee.org/the-top-programming-languages-2023, Accessed: 2023-12-07, Aug. 2023. [Online]. Available: https://spectrum.ieee.org/the-top-programming-languages-2023.

[102] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: A state-of-the-art review," *IEEE Transactions on Cloud Computing*, vol. 7, no. 3, pp. 677–692, 2017.

[103] PyPI, *The python package index*, https://pypi.org/, 2024.

[104] D. Stufft, *PGP signatures are not displayed / Issue #3356*, en, Mar. 2018. [Online]. Available: https://github.com/pypi/warehouse/issues/3356.

[105] D. Stufft, *Removing PGP from PyPI - The Python Package Index*, en, May 2023. [Online]. Available: https://blog.pypi.org/posts/2023-05-23-removing-pgp/.

[106] Open Collective, *Ecosyste.ms*, https://ecosyste.ms/, 2023.

[107] A. Linskens, *PGP vs. sigstore: A Recap of the Match at Maven Central*, en-us, 2023. [Online]. Available: https://blog.sonatype.com/pgp-vs.-sigstore-a-recap-of-the-match-at-maven-central.

[108] Docker, *Content trust in Docker*, en, https://docs.docker.com/engine/security/trust/, 200. [Online]. Available: https://docs.docker.com/engine/security/trust/.

[109] D. Lorenc, *Cosign 1.0! - Sigstore Blog*, https://blog.sigstore.dev/cosign-1-0-e82f006f7bc4/, Jul. 2021. [Online]. Available: https://blog.sigstore.dev/cosign-1-0-e82f006f7bc4/.

[110] Hugging Face, *Hugging face*, https://huggingface.co/, 2024.

[111] Hugging Face, *Security*, 2023. [Online]. Available: https://huggingface.co/docs/hub/security.

[112] PyPA, *Analyzing pypi package downloads*, https://packaging.python.org/en/latest/guides/analyzing-pypi-package-downloads/#public-dataset, 2024.

[113] Notary Project, *Notice*, original-date: 2015-06-19T20:07:53Z, Oct. 2023. [Online]. Available: https://github.com/notaryproject/notary.

[114] Sonatype, *Working with pgp signatures*, https://central.sonatype.org/publish/requirements/gpg/#signing-a-file, 2024.

[115] H. Ferraiolo and A. Regenscheid, *Cryptographic Algorithms and Key Sizes for Personal Identity Verification*. Sep. 2023. DOI: 10.6028/nist.sp.800-78-5.ipd. [Online]. Available: http://dx.doi.org/10.6028/NIST.SP.800-78-5.ipd.

[116] Hugging Face, *Location of public gpg keys*, https://discuss.huggingface.co/t/location-of-public-gpg-keys/62915, 2024.

[117] D. Anandayuvaraj and J. C. Davis, "Reflecting on recurring failures in iot development," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–5.

[118] J. C. Davis, P. Jajal, W. Jiang, T. R. Schorlemmer, N. Synovic, and G. K. Thiruvathukal, "Reusing deep learning models: Challenges and directions in software engineering," in *Proceedings of the IEEE John Vincent Atanasoff Symposium on Modern Computing*, 2023.