Signing in Four Public Software Package Registries: Quantity, Quality, and Influencing Factors

Taylor R. Schorlemmer
Purdue University
tschorle@purdue.edu

Kelechi G. Kalu
Purdue University
kalu@purdue.edu

Luke Chigges
Purdue University
lchigges@purdue.edu

Kyung Myung Ko
Purdue University
ko112@purdue.edu

Eman Abdul-Muhd Abu Ishgair
Purdue University
eabuishg@purdue.edu

Saurabh Bagchi
Purdue University
sbagchi@purdue.edu

Santiago Torres-Arias
Purdue University
santiagotorres@purdue.edu

James C. Davis
Purdue University
davisjam@purdue.edu

***Abstract*—Many software applications incorporate open-source third-party packages distributed by third-party package registries. Guaranteeing authorship along this supply chain is a challenge. Package maintainers can guarantee package authorship through *software signing*. However, it is unclear how common this practice is, and whether the resulting signatures are created properly. Prior work has provided raw data on signing practices, but measured single platforms, did not consider time, and did not provide insight on factors that may influence signing. We lack a comprehensive, multi-platform understanding of signing adoption and relevant factors.**

This study addresses this gap. We provide measurements across three kinds of package registries: traditional software (Maven, PyPi), container images (DockerHub), and machine learning models (HuggingFace). For each registry, we describe the nature of the signed artifacts as well as the current quantity and quality of signatures. Then, we examine longitudinal trends in signing practices. Finally, we use a quasi-experiment to estimate the effect that various events had on software signing practices. To summarize our findings: (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

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].

Many mitigations have been proposed for software supply chain attacks. Some approaches seek to increase confidence in a package's *behavior*, *e.g.,* measuring use of best practices [9], [10], independent validation [11], and formal guarantees [12]; Other approaches target the package's *provenance*, *e.g.,* Software Bill of Materials (SBOMs) [13], [14] and "vendoring" trusted copies of dependencies [15]. The strongest guarantee of a package's provenance is a cryptographic signature by its maintainer. Prior work has noted that many packages are unsigned [16], [17]. However, we lack up-to-date measurements of software signing practices, and we do not know the general quality of existing signatures nor what factors affect adoption rates. This knowledge would guide future efforts to incentivize software signing, so that the provenance of software supply chains can be improved.

Our work provides this knowledge: we measure software signing practices in four public software package registries to infer factors that influence software signing. We selected four registries for a quasi-experiment [18]:[1] two with signing policies (Maven-positive, PyPi-negative), one with dedicated tooling (DockerHub), and one with no stance on signing (HuggingFace). 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: the startup effort of signing, organizational policy, and signing-related events such as high-profile cyberattacks.

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). Sign-

1. A quasi-experiment seeks cause-and-effect relationships between independent and dependent variables without subject randomization.

ing remains difficult — only the registry with dedicated signing tools had perfect signature quality (DockerHub), while the other three had signature quality rates of approximately 80% (Maven), 50% (PyPi), and $< 2\%$ (HuggingFace). The first signature is the hardest: after a maintainer first signs a package, they are likely to continue signing that package. Finally, we observed no effects from signing-related news, such as high-profile cyberattacks and new engineering standards that recommend software signing.

To summarize our contributions:

1) We present up-to-date measurements of software signing practices — quantity and quality — in four major software package registries.
2) We use a quasi-experiment to estimate the effect of several factors on software signing practices. Registry policies have a notable effect on signing quantity. Registry tooling has a notable effect on signing quality.

2. Background

§2.1 discusses software supply chains. §2.2 describes software signing, generally and in our target registries.

2.1. Software Supply Chains

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* [19]: 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 [20], [21] or lightweight GitHub Packages [22])), but are more commonly distributed through separate *software package registries* [23], [24]. 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* [25], [26].

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 [27], and Wang *et al.* found that 90% of highly popular Java projects on GitHub use third-party packages [28].

Selecting and managing software dependencies is thus an important software engineering practice [29], [30]. Software engineers must decide which packages to use in their projects, *i.e.,* what to include in their application's software supply chain [31], [32]. Engineers consider many aspects, encompassing functionality, robustness, maintainability, compatibility, popularity, and security [33]–[36]. Specific to security, various tools and methodologies have been proposed. These include in-toto [37], reproducible builds [38], testing [39], [40], LastPyMile [41], SBOMs [42], and BuildWatch [43]. 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 [26]. 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) [44]. 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 [9], [45], [46] and government [47], [48] leaders.

2.2.1. Signing Process and Failure Modes. Most software package registries require similar signing processes. Figure 1 illustrates this process, beginning with a maintainer and a (possibly separate) signer. They publish 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 1 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 [49], 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.
3) **Bad Signature:** The resulting signature is incorrect or unverifiable for non-malicious reasons, such as signing

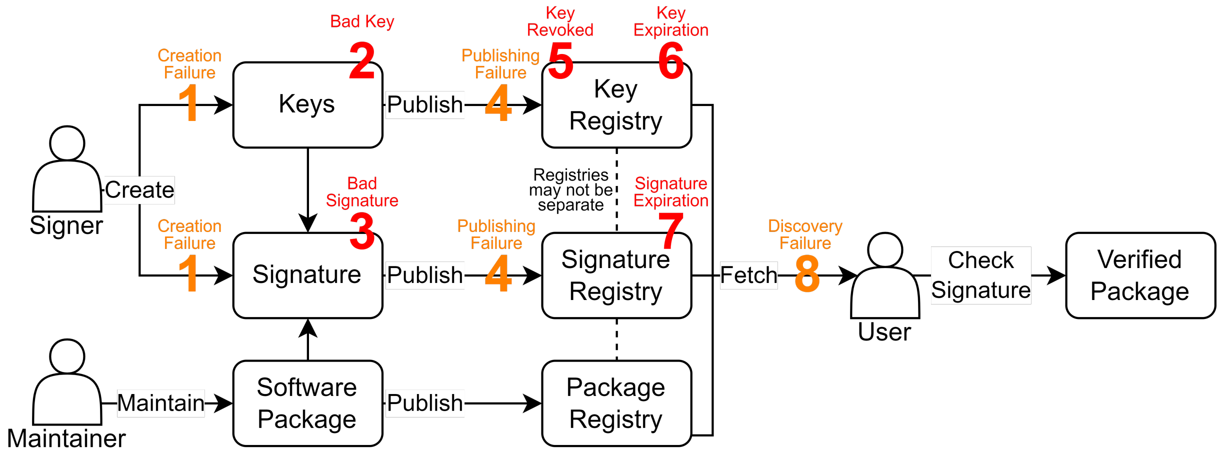

Figure 1. Maintainers create software packages and signers create keys which are used to create a signature. Each of these artifacts are published to a registry. Depending on ecosystem, the registries and the actors may or may not be separate. Users fetch these artifacts and can check signatures using 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).

the wrong artifact or using an unsupported algorithm (*e.g.,* use of an unknown algorithm).

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 [50].

2.2.2. Signing Targets.

We detail the three kinds of signing used across the four registries considered in this work. These registries largely follow the maintainer-signer style. The registries support the signing of different artifacts.

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

HuggingFace—Commits: In HuggingFace (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. HuggingFace's commit-based approach means that signatures only ensure that the *changes* to package artifacts are authentic.

DockerHub—Packages (container images): In DockerHub (Docker container images), the signing target is the package, *i.e.,* the container image. Maintainers sign the packages, but unlike in Maven, PyPi, and HuggingFace, the cryptographic materials are stored and managed by a registry service called Notary that is run in conjunction with DockerHub. 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).

3. Related Works

We discuss work on software signing challenges (§3.1) and prior measurements of signing practices (§3.2).

3.1. Challenges of Software Signing

3.1.1. Signing by Novices.

Like other cryptographic activities [51], [52], signing artifacts is difficult for people without cryptographic expertise. The ongoing line of "Why Johnny Can't Encrypt" works, begun in 1999 [53], enumerates confusion in the user interface [54], [55] and the user's understanding of the underlying public key model [56], [57], and other usability issues [58]. Automation is not a silver bullet — works by Fahl *et al.* [59] and Ruoti *et al.* [60] both found no significant difference in usability when comparing manual and automated encryption tools, though Atwater *et al.* [61] did observe a user preference for automated solutions.

3.1.2. 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 [62], [63]. Other concerns relate to the broader problems of signing over time, *e.g.,* key management [64]–[66], key discovery [67], [68], cipher agility [69], and signature distribution [70]. Software signing processes and supporting automation remain an active topic of research [66], [71], [72]

3.1.3. 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 rebuke this literature, showing the importance of factors beyond perceived usability and individual preference.

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.* [16] reported that only ∼4% of PyPI projects listed a signature. In 2023, Zahan *et al.* examined signature propagation [9], 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 HuggingFace [35]. 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) [17].

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 DockerHub, and the first longitudinal measurements in Maven, DockerHub, and HuggingFace. Our multi-registry approach allows us to both observe *and infer the causes of* variation in signature quantity and quality.

4. Signing Adoption Theory

Although software signing is recommended by engineering leaders (§2.2), prior work shows that signing remains difficult (§3.1 and successful adoption is rare (§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 [45], [46], maintainers do not always follow best practices [73]. Prior work has focused on the *usability* of signing techniques (§3), but we posit that a maintainer's *incentives* to sign are also important.

Behavioral economics examines how incentives influence human behavior [74]. Economists typically distinguish between incentives that are intrinsic (internal) and extrinsic (external) [75]. Incentives can change how individuals make decisions, although the relationships are not always obvious [76], [77]. For example, Titmuss [78] 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 2, 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 1, we examined four kinds of external incentives that might influence a maintainer's signing practices. We formulated hypotheses as to their effects, but behavioral economics suggests that even the most obvious hypothesis must be tested.

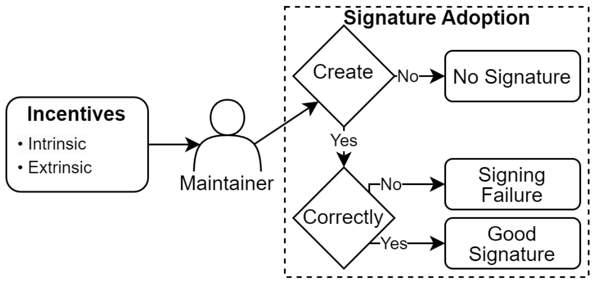

Figure 2. Incentives influence how maintainers adopt software signing. The maintainer decides weather or not to create a signature. If a maintainer decides to create a signature, they can either follow the signing process (*i.e.,* Figure 1) correctly or not. Correctly following the signing process results in a good signature but incorrectly following the process results in a signing failure.

TABLE 1. KINDS OF INCENTIVES CONSIDERED. THE SECOND COLUMN IS OUR HYPOTHESES: WHETHER THE INCENTIVE WAS PREDICTED TO INCREASE (↑) OR DECREASE (↓) SIGNATURE ADOPTION (CF. §5). THE THIRD COLUMN IS THE OBSERVED EFFECT (CF. §7).

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 1). 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 2), 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.

5. 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 §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 [79].

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.

6. Methodology

This section describes our methods. In §6.1 we give our experimental design. Following that design, our methodology has five stages (Figure 3):

1) **Select Registries**(§6.2): Picking appropriate registries for our study.
2) **Collect Packages**(§6.3): Methods used to collect a list of packages from each registry.
3) **Filter Packages**(§6.4): Filter list of packages so that remaining packages are adequately versioned.
4) **Measure Signature Adoption**(§6.5): Measuring the quantity and quality of signatures for each registry.
5) **Evaluate Adoption Factors**(§6.6): Comparing signature adoption among registries and across time.

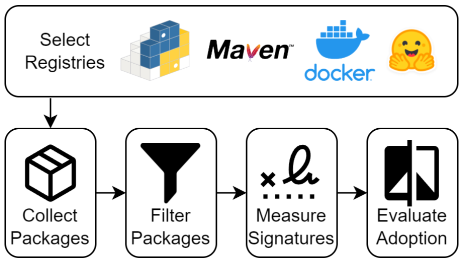

Figure 3. First, we select package registries that represent a range of software types and signing policies. Our selected registries include PyPI, Maven Central, Docker Hub, and Hugging Face. Next, we collect a list of packages for each platform. Then, we filter the list of packages to a sample of packages 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.

6.1. Experimental Design

We first describe the types of experiments needed to answer our RQs (§6.1.1) then briefly explain quasi experiments and why they work well for our study (§6.1.2).

6.1.1. Answering our RQs. 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 [9], [16], [80] and quality [17]. In §6.5 we define quality systematically, but our approach resembled prior work.

RQ3 requires us to test the relationship between incentives and signature adoption (quantity and quality). Our goal is similar in spirit to the usability studies performed by Whitten & Tygar [53], Sheng *et al.* [54], and Routi *et al.* [57] — 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.

6.1.2. Quasi Experiments. *Quasi experiments* [2] 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 [18], [81]. Instead, the application of treatments is based on the characteristics of the subjects themselves [82]. This method is often used

2. The terms *quasi* and *natural experiments* 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 [81]. Since registry factors are intended to cause changes in signature adoption, our study is a quasi experiment, not a natural experiment.

where controlled experiment is not possible, *e.g.,* to measure impacts of government policies on target populations [81], [83], [84]. These studies produce the strongest results when treatments occur independently of the subjects, or are *exogenous* [81], [84].

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 [84].

In our study, registry factors are *treatments*, maintainers are *subjects*, and signature adoption is the *outcome*. As already 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 themselves (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 now illustrate our experimental design with two examples. 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 trends in PyPi, while trends 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 following 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.

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.

Based on this design, we proceed through five stages (§6.1).

6.2. Stage 1: Select Registries

In this section, we explain what makes a registry a good candidate for this study (§6.2.1) and justify our use of PyPI, Maven Central, Docker Hub, and Hugging Face (§6.2.2).

6.2.1. 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, because these place a greater burden on the maintainer and thus the effect of incentives would be more observable. Finally, the selected registries should be popular so that they are representative of publicly available software.

6.2.2. Selected Registries. Following these requirements, we identified 4 registries for study: PyPI, Maven Central, Docker Hub, and Hugging Face. Table 2 summarizes these registries. They represent some of the most popular programming languages [85] (Java, Python), the most popular container technology [86] (Docker), and the most popular ML model hub [23] (Hugging Face). Next we elaborate on each selected registry.

TABLE 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
HuggingFace	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 525,000 packages [87]. PyPI allows maintainers to sign packages with PGP signatures. The PyPI registry owners deemphasized the use of PGP signatures on 22 Mar 2018 [88] and later deprecated them on 23 May 2023 [89]. 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. Like PyPi, Maven allows maintainers to sign packages with PGP signatures. Unlike PyPi, on Maven, signatures have been mandatory since 2005 [90].

(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. Docker Hub uses Docker Content Trust (DCT) [91] to sign container images. As noted in §2, Docker Hub has dedicated tooling for signing, integrated in the Docker CLI. Since 2019, Docker Hub has encouraged signing by allowing users to filter images by signature status, but does not mandate signing like Maven does. Another relevant event for Docker Hub was the addition of support from Cosign on 28 Jul 2021 [92].

(4) HuggingFace: Hugging Face is the primary registry for the exchange of neural network models. Hugging Face hosts more than 300,000 models. Hugging Face supports the signing of git commits [93]. Hugging Face has no stated policy towards signing, and we are aware of no signing events specific to Hugging Face.

6.3. 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 [87] 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 to collect packages. In the remainder of this subsection, we describe the package structure and collection techniques used for each registry.

6.3.1. The Ecosyste.ms Cross-registry Package Index. Ecosyste.ms serves as a comprehensive cross-registry package index, aggregating package data from multiple registries into a singular 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 — August 8, 2023. This dataset was missing some information from Maven, and did not index the HuggingFace registry.

We sought to obtain data up to August 2023 from each registry. We arbitrarily set the start date to January 2015, imitating [17], so that the reported data would not be too different from current practices.

6.3.2. 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 Jan 2015 and Sep 2023, we used ecosyste.ms' data dump from 22 Oct 2023. This data dump 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.

In a similar fashion to PyPI, we used ecosyste.ms' data dump from 22 Oct 2023 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.

TABLE 3. Packages available after each stage of the pipeline. Note that Maven and HF measurement populations are less than the filter populations. Because of download rates we chose to limit Maven to 10% of the original package population. Some HF models require users to agree to terms and conditions — we skip these models.

Stage	PyPI	Maven	Docker	HF
Collect Packages	588,965	499,245	950,155	387,566
Filter Packages	213,851	118,516	73,973	50,632
Measure Signatures	213,851	49,925	73,973	49,865

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) or Sigstore's Cosign. DCT is a Docker-specific signing tool which uses Notary [94] under the hood. Cosign is a general-purpose signing tool which can be used to sign any container image.

To collect all packages from Docker Hub between Jan 2015 and Sep 2023, we use ecosyste.ms' data dump [3] from 22 Oct 2023. This data dump contains a list of package distributions and associated metadata for each package hosted on Docker Hub.

HuggingFace: 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 Jan 2015 and Sep 2023, we use the HuggingFace Hub Python interface to generate a list of packages (and metadata) in our date range. Using this information, we are able to iteratively clone all repositories from Hugging Face.

6.4. 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 preferred filter was for packages with multiple versions. In Docker Hub, Maven Central, and PyPI we filtered for all packages with ≥ 5 versions. Hugging Face does not support versions in the same way, so we instead filtered for Hugging Face models with at least 5 downloads (a proxy for popularity).

See Table 3 for the size of the original lists and the effect of the filter.

3. We use https://ecosyste.ms/ to get this list, however, after a review of their API source, we found that their list of packages is built by continually appending a "list" with the most recently uploaded packages from Docker Hub. This may lead to small discrepancies between what is on the "list" and what is on Docker Hub.

TABLE 4. SIGNATURE STATUSES AND WHETHER OR NOT THEY ARE MEASURABLE ON EACH PLATFORM. 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	✓	✓	✗	–

6.5. Stage 4: Measure Signatures

After filtering, we attempted to measure the quantity and quality of signatures over the entire filter population. For Maven Central and Hugging Face, 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 10% of the filter population. Some of the Hugging Face repositories require the user to sign additional agreements before use (*e.g.,* the *pyannote/segmentation* model requires users to agree to terms of use) — we do not include these repositories in this study.

In §4 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 1 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. Table 4 indicates which signature failures can be measured on each platform. We use this table to guide our evaluation of signature quality in each registry.

PyPI: PyPI uses PGP signatures to secure packages. These signatures are uploaded to the registry along side the package. Since PyPI does not provide a public key server, we use the Ubuntu public key servers to verify signatures. [4] We use the *gpg* command line utility to verify signatures. This utility provides a *verify* command which can be used to verify the validity of a signature. This command returns a status code which indicates whether or not the signature is valid. We use this status code to determine whether or not a signature is valid.

4. See https://keyserver.ubuntu.com/.

Maven Central: Maven Central requires PGP signing on all artifacts within the registry. As as a result, there is a PGP signature for each file present in the registry. For example, a *package.jar* or *package.pom* file will have a corresponding *package.jar.asc* or *package.pom.asc* file located in the same directory. In a similar manner to PyPI, we use the *gpg* command line utility to verify signatures. We use the *verify* command and Ubuntu public key servers to verify the validity of signatures.

Docker Hub: Docker Hub uses two different signing schemes, Docker Content Trust (DCT) and SigStore's Cosign. DCT is the default signing scheme for Docker Hub. However, SigStore has recently updated Cosign to support Docker Hub. Both of these signing tools have command line utilities which can be used to verify signatures. We use these utilities to verify the validity of signatures. For DCT, we use the *docker trust inspect* command to verify signatures. For Cosign, we use the *cosign verify* command to verify signatures.

HuggingFace: To measure the adoption of selected packages on Hugging Face, we adapted a script used by Jiang *et al.* to create PTMTorrent [95]. This script uses a Python wrapper for Git to download repositories locally. Since repositories are often large and we only care about commit signing, we download bare repositories to speed up the process. We then iterate through all commits in the downloaded repository and use the built-in *git verify-commit* command to check for signatures on each commit. We log output of this command for later analysis.

6.6. 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 events corresponding to each class of incentive. We identified a set of events 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 events are given in Table 5. We identified changes in registry policies and in dedicated tooling, as well as three kinds of signing events: software supply chain attacks, government actions, and industry standards.

7. Results

For RQ1, we present the quantity and quality of signatures we measured in each registry in §7.1. For RQ2, we describe changes in signing practices over time in §7.2. Finally, for RQ3, we assess the influence of incentives on signing adoption in §7.3.

TABLE 5. EVENTS THAT COULD PROVIDE SIGNING INCENTIVES. EVENTS WERE IDENTIFIED VIA WEB SEARCHES BY FOUR AUTHORS, AND WERE CATEGORIZED ACCORDING TO THE KINDS OF INCENTIVES CONJECTURED IN TABLE 1.

Event	Category	Date
PyPI De-emph	Registry policy	Mar 2018
Docker Hub Update	Registry policy	Sep 2019
PyPI Removed PGP	Registry policy	May 2023
Cosign Support	Dedicated tooling	Jul 2021
NotPetya	Signing event (attack)	Jun 2017
CCleaner	Signing event (attack)	Sep 2017
Magecart	Signing event (attack)	Apr 2018
DockerHub 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

7.1. RQ1: Current Quantity & Quality of Signatures

In Table 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 the status of the subset of signed signatures. We show both the most recent year of data (Oct 2022 to Sep 2023) and the entire time period of data (Jan 2015 to Sep 2023).

The rest of this section refers to Table 6.

7.1.1. Quantity of Artifact Signatures. With respect to the quantity (proportion) of signed artifacts, we can group the registries into three groups by order of magnitude. First, *Maven Central* experiences the highest signing rate with 95.3% of file signed since Oct 2022. We conjecture that this degree of signing occurs only when signing is mandatory. Second, *Docker Hub* has a low adoption rate with 1.4% of tags signed since Oct 2022. Third, *Hugging Face* and *PyPI* currently have a negligible amount of signatures with 0.1% and 0.4% of artifacts signed since Oct 2022, 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*. Out of all 261K commits across 49.9K on *Hugging Face*, only 253 are signed.

Finding 1: Between Oct 2022 and Sep 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 95.3% of artifacts signed in that same time period.

7.1.2. Quality of Artifact Signatures. With respect to quality, each registry has a distinct flavor. On *Docker Hub* signatures either exist or do not — we can only tell if maintainers correctly signed a tag. Of the registries with measurable quality, *Maven Central* has the best with 76.0% of signatures valid since Oct 2022. *PyPI* has the next best quality with 46.9% of signatures valid since Oct 2022. Not only does *Hugging Face* have the lowest quantity of signed artifacts, but it also has the lowest quality of signed artifacts. Of the 253 signed artifacts, only 57 (22.5%) were valid.

We observed differences between signing failure modes by registry. On Maven, the three most common failure modes were expired public keys, missing public keys, and revoked public keys. Failures related to public keys accounted for over 99% of all Maven signing failures between Oct 2022 and Sep 2023. The three most common failure modes on *PyPI* were revoked public keys, bad public keys, and missing public keys. Similar to Maven, public key related failures accounted for over 99% of all PyPi signing failures between Oct 2022 and Sep 2023. On HuggingFace, the registry records but does not publish the public keys disclosed by package maintainers.[5] 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.

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 Oct 2022 and Sep 2023 were invalid, respectively. On Hugging Face the situation is worse, with 77.5% invalid signatures. Lastly, on Docker Hub, we observed no signing failures.

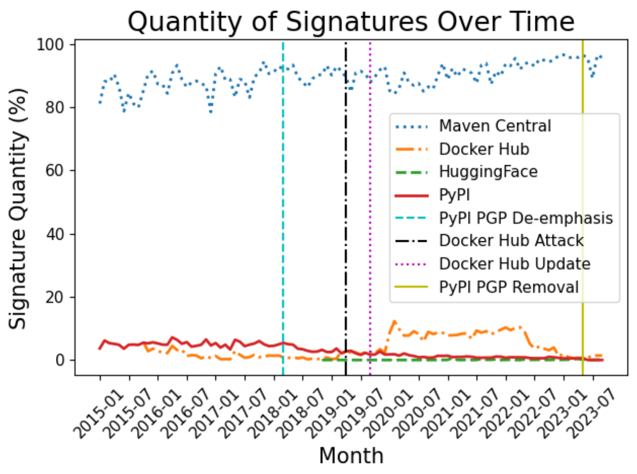

Figure 4. Quantity of signed artifacts over time. X-axis shows time in monthly increments. Y-axis shows the percentage of signed artifacts in each registry.

7.2. RQ2: Change in Signing Practices Over Time

7.2.1. Quantity of Artifact Signatures. In Figure 4, we show the quantity of signed artifacts over time. We measured signing rates for the signable artifacts published in that

5. We asked the HuggingFace engineers for access. They declined.

TABLE 6. The number of assessed artifacts from each registry, the percent with and without signatures, and the breakdown of signature status for signed artifacts. For each measurement, we show both the most recent year of data and the entire time period of data. "—": Not measurable; HuggingFace hides keys and only discloses whether their validation was successful.

Registry	**PyPI** 1 year (all time)	**Maven Central** 1 year (all time)	**Docker Hub** 1 year (all time)	**Hugging Face** 1 year (all time)
Total Artifacts	**2.56M (9.65M)**	**1.72M (9.8M)**	**4.42M (7.94M)**	**261K (562K)**
Unsigned Artifacts	99.6% (98.6%)	4.7% (9.6%)	98.6% (97.1%)	99.9% (99.9%)
Signed Artifacts	0.4% (1.4%)	95.3% (90.4%)	1.4% (2.9%)	0.1% (0.1%)
Good Signature	46.9% (49.8%)	76.0% (75.2%)	100% (100%)	22.5% (30.7%)
Bad Signature	0.1% (0.2%)	0.2% (0.3%)	0.0% (0.0%)	—
Expired Signature	0.0% (0.1%)	0.0% (0.0%)	0.0% (0.0%)	—
Expired Public Key	6.1% (16.4%)	9.9% (17.2%)	0.0% (0.0%)	—
Missing Public Key	12.9% (15.7%)	8.4% (3.7%)	0.0% (0.0%)	—
Public Key Revoked	20.1% (15.1%)	0.9% (2.2%)	0.0% (0.0%)	—
Bad Public Key	13.9% (2.7%)	4.6% (1.4%)	0.0% (0.0%)	—

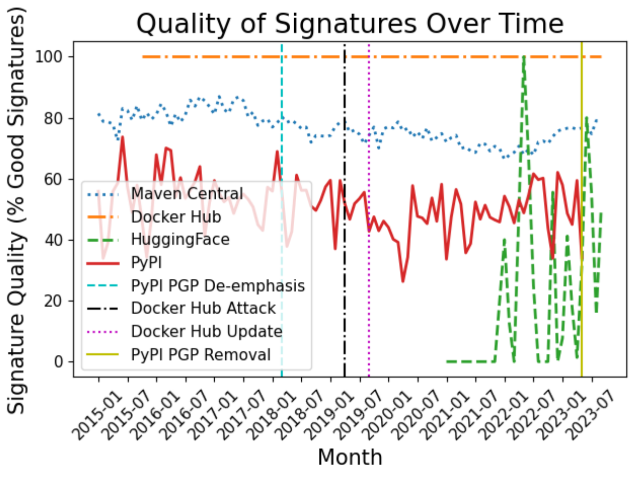

Figure 5. Quality of signed artifacts over time. X-axis shows time in monthly increments. Y-axis shows the percentage of signatures with a good status in each registry.

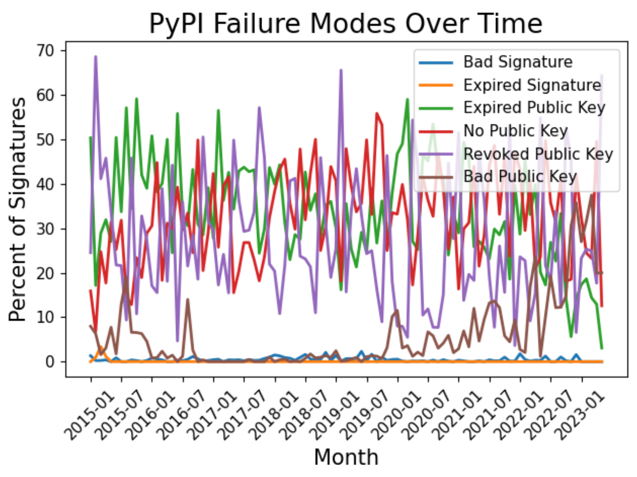

Figure 6. Failure modes of signatures on PyPI.

month. This figure shows how many such artifacts were signed, grouped by 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 nearly zero quantity of signed commits since its introduction. Even with its recent increase in popularity, signing has not been adopted by the community. 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.

7.2.2. Quality of Artifact Signatures. In Figure 5, 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 of quality in HuggingFace.

Next we consider the failure modes of signatures by registry. For PyPI, see Figure 6. There are several common failure modes. They vary in relative frequency. None dominates.

For the failure modes of signatures in Maven Central, see Figure 7. Before 2023, the most common failure mode was an expired public key. Although this doesn't necessarily mean that the public key was expired at the time of signing, it does mean that the public key was expired at the time of our analysis (indicating key or signature management issues). After 2023, the most common failure mode was a missing public key followed by a bad public key. This trend suggests that maintainers are getting worse at public key creation and distribution.

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 modes. On Hugging Face, we cannot access the

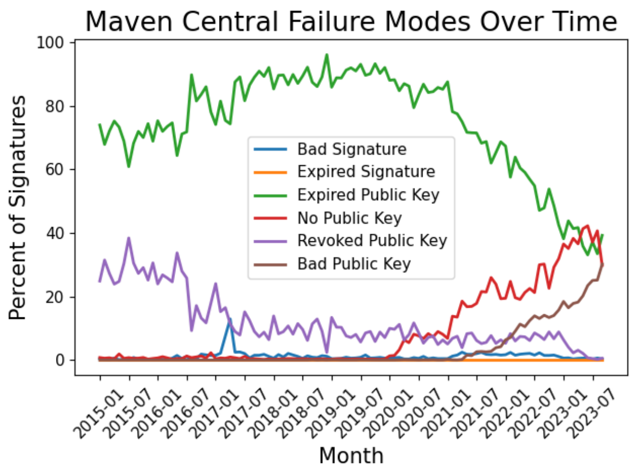

Figure 7. Failure modes of signatures on Maven Central.

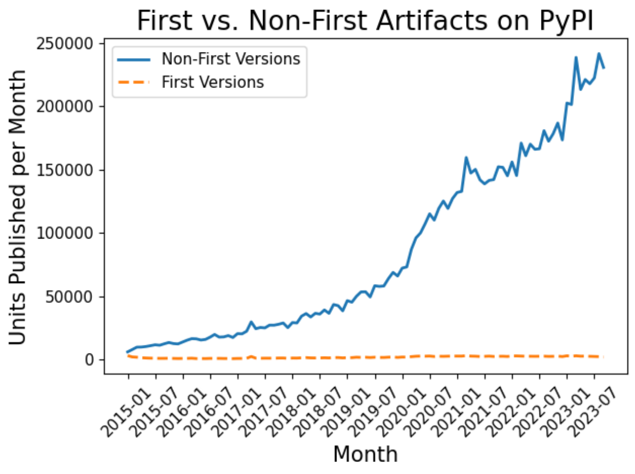

Figure 8. Number of first-versions and subsequent-versions of packages on PyPI.

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

7.2.3. No bias from new packages. Software package registries grow over time [87]. 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 8 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.

7.3. RQ3: Influence of Incentive

7.3.1. 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. 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 5 and appear as vertical lines in Figure 4.

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. Shortly after this, the quantity of signatures on PyPI began to decrease. This continued until 23 May 2023, when PyPI removed the ability to upload PGP signatures entirely. After that point, the quantity of signatures on PyPI dropped to zero.

Second, Docker Hub's quantity of signatures experienced an appreciable increase at the start of 2020. This increase follows a signature-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.

Third, the notable difference in signing quantity between Maven Central and the other registries suggests that mandatory signing policies encourage signing adoption. Since Maven Central has a mandatory signing policy, we expected, and observe, a high quantity of signatures.

7.3.2. 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, 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 Oct 2022 and Sep 2023Docker Hub's signing rate was only 1.4%.

However, we do find support for the quality aspect of H_2. In Figure 5, 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.

7.3.3. 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

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

the measurements shown in Figure 4 and Figure 5 from multiple registries. Table 5 lists several influential software supply chain attacks and cybersecurity events. However, a visual inspection of the trendlines suggests that none of these events had a significant impact on signing adoption.

We illustrate this with two cases. First, consider the registry-speciifc 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 8 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.

7.3.4. 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 7, 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 $\sim$40x. These changes suggest that the initial burden of signing is a significant barrier to adoption.

TABLE 7. 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	1.35%	44.43%
Maven Central	91.00%	96.22%
Docker Hub	2.86%	88.76%
HuggingFace	0.08%	4.16%

8. Discussion

We highlight three points for discussion.

First, our findings suggest that the long line of literature on the usability of signing tools (§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. However, 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-measurable 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 [96].

9. Threats to Validity

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

Construct: Our measurement 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. Our failure model omitted weak cryptography, and we relied on the correctness of specific (albeit widely used) tools to measure the signatures. We acknowledge that the limitations of the available data, such as limited access to corresponding public key registries for verifying the signatures, may potentially impact the robustness of our results. In addition, our insights into failure modes were limited by the characteristics of some 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 quasi-experiments instead. We assumed that there would be 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. As noted by Jiang *et al.*, Hugging Face is characterized by a "research to practice pipeline" more than traditional software package registries are [23]. 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 approach thus suggests 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 and organizational policies.

10. 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 valuable light on the relative weight of the factors we identified. Such studies could also expose new factors to be fed back into large-scale quantitative measurements.

Identifying Machine Learning (ML) Software and Pre-Trained Model Signing requirements: Signing adoption rates in the HuggingFace registry are substantially lower than in all other studied registries. We recommend further research into 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 [97].

Apply the results: As noted in §8, 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.

11. Conclusion

In this study, we assessed signing in four public software package registries (PyPI, Maven Central, Docker Hub, and Hugging Face). We provided comprehensive up-to-date measurements of signature adoption (quantity and quality) while proposing and evaluating a theory explaining maintainer's decision's to adopt signatures. We found that, aside from Maven Central, the quantity of signatures in package registries is low. In a similar vein, aside from Docker Hub, the quality of signatures in package registries is low. To explain these observations, we proposed and evaluated an incentive theory for signing adoption. We used quasi-experiments to test four hypotheses. We found that incentives do influence signing adoption. 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 to enhance the overall security of software systems. Our findings suggest specific incentives that could significantly improve software signing adoption rates.

Acknowledgments

This work was made possible by gifts from Google and Cisco as well as National Science Foundation (NSF) award #2229703.

References

[1] G. 18F, "18F: Digital service delivery | Open source policy," 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," 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. New York, NY, USA: Association for Computing Machinery, 2022, p. 15–24. [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,” Sonatype, Tech. Rep. 8th Annual, 2022, https://www.sonatype.com/state-of-the-software-supply-chain/open-source-supply-demand-security.

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

[10] 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.

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

[12] 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. New York, NY, USA: Association for Computing Machinery, 2022, p. 2203–2217. [Online]. Available: https://doi.org/10.1145/3548606.3559373

[13] M. Balliu, B. Baudry, S. Bobadilla, M. Ekstedt, M. Monperrus, J. Ron, A. Sharma, G. Skoglund, C. Soto-Valero, and M. Wittlinger, “Challenges of producing software bill of materials for java,” *arXiv preprint arXiv:2303.11102*, 2023.

[14] 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.

[15] T. Winters, T. Manshreck, and H. Wright, *Software engineering at Google: lessons learned from programming over time*, first edition ed. Beijing [China]: O’Reilly Media, 2020.

[16] 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)*. Santa Clara, CA: USENIX Association, Mar. 2016, pp. 567–581, https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/kuppusamy.

[17] W. Woodruff, “PGP signatures on PyPI: worse than useless,” May 2023, https://blog.yossarian.net/2023/05/21/PGP-signatures-on-PyPI-worse-than-useless.

[18] M. Felderer and G. H. Travassos, Eds., *Contemporary empirical methods in software engineering*. Cham: Springer, 2020.

[19] W. Jiang, N. Synovic, M. Hyatt, T. R. Schorlemmer, R. Sethi, Y.-H. Lu, G. K. Thiruvathukal, and J. C. Davis, “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. IEEE Press, 2023, p. 2463–2475. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00206

[20] SourceForge, “Compare, Download & Develop Open Source & Business Software - SourceForge,” https://sourceforge.net/.

[21] GitLab, “The DevSecOps Platform,” https://about.gitlab.com/.

[22] “GitHub Packages,” https://github.com/features/packages, accessed: 2023-12-06.

[23] W. Jiang, N. Synovic, R. Sethi, A. Indarapu, M. Hyatt, T. R. Schorlemmer, G. K. Thiruvathukal, and J. C. Davis, “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. New York, NY, USA: Association for Computing Machinery, 2022, p. 105–114. [Online]. Available: https://doi.org/10.1145/3560835.3564547

[24] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, “Small world with high risks: A study of security threats in the npm ecosystem,” in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 995–1010.

[25] 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.

[26] 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*. Los Angeles CA USA: ACM, Nov. 2022, pp. 15–24. [Online]. Available: https://dl.acm.org/doi/10.1145/3560835.3564556

[27] D. Kumar, Z. Ma, Z. Durumeric, A. Mirian, J. Mason, J. A. Halderman, and M. Bailey, “Security challenges in an increasingly tangled web,” in *Proceedings of the 26th International Conference on World Wide Web*, 2017, pp. 677–684.

[28] Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Wu, and Y. Liu, “An empirical study of usages, updates and risks of third-party libraries in java projects,” in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Adelaide, Australia: IEEE, Sep. 2020, p. 35–45. [Online]. Available: https://ieeexplore.ieee.org/document/9240619/

[29] 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.

[30] 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.

[31] 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.

[32] ——, “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.

[33] 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.

[34] 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.

[35] W. Jiang, N. Synovic, M. Hyatt, T. R. Schorlemmer, R. Sethi, Y.-H. Lu, G. K. Thiruvathukal, and J. C. Davis, “An empirical study of pre-trained model reuse in the hugging face deep learning model registry,” in *IEEE/ACM 45th International Conference on Software Engineering (ICSE’23)*, 2023.

[36] D.-L. Vu, F. Massacci, I. Pashchenko, H. Plate, and A. Sabetta, “Lastpymile: identifying the discrepancy between sources and packages,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 780–792.

[37] 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. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/torres-arias

[38] 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.

[39] J. Hejderup, “On the Use of Tests for Software Supply Chain Threats,” 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. [Online]. Available: https://dl.acm.org/doi/10.1145/3560835.3564557

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

[41] D.-L. Vu, F. Massacci, I. Pashchenko, H. Plate, and A. Sabetta, “LastPyMile: identifying the discrepancy between sources and packages,” 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. [Online]. Available: https://dl.acm.org/doi/10.1145/3468264.3468592

[42] “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

[43] 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. [Online]. Available: https://doi.org/10.1145/3407023.3409183

[44] R. Shirey, “Internet security glossary, version 2,” https://www.rfc-editor.org/rfc/rfc4949.html, Network Working Group, RFC 4949, August 2007, fYI: 36. Obsoletes: 2828. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4949.html

[45] S. T. A. Group, “Software Supply Chain Best Practices,” Cloud Native Computing Foundation, Tech. Rep., May 2021. [Online]. Available: https://project.linuxfoundation.org/hubfs/CNCF_SSCP_v1.pdf

[46] “Supply-chain Levels for Software Artifacts.” [Online]. Available: https://slsa.dev/

[47] D. Cooper, A. Regenscheid, M. Souppaya, C. Bean, M. Boyle, D. Cooley, and M. Jenkins, “Security considerations for code signing,” National Institute of Standards and Technology, Ft. George G. Meade, Maryland, NIST Cybersecurity White Paper, January 2018. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/CSWP/NIST.CSWP.01262018.pdf

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

[49] T. P. o. t. G. Project, “The GNU Privacy Guard,” Apr. 2023, publisher: The GnuPG Project. [Online]. Available: https://gnupg.org/

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

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

[56] 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. New York, NY, USA: Association for Computing Machinery, Apr. 2006, pp. 199–203, https://dl.acm.org/doi/10.1145/1132736.1132768.

[57] 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. [Online]. Available: https://dl.acm.org/doi/10.1145/2501604.2501609

[58] A. Reuter, K. Boudaoud, M. Winckler, A. Abdelmaksoud, and W. Lemrazzeq, “Secure Email - A Usability Study,” in *Financial Cryptography and Data Security*, ser. Lecture Notes in Computer Science, M. Bernhard, A. Bracciali, L. J. Camp, S. Matsuo, A. Maurushat, P. B. Rønne, and M. Sala, Eds. Cham: Springer International Publishing, 2020, pp. 36–46.

[59] 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. New York, NY, USA: Association for Computing Machinery, Jul. 2012, pp. 1–17, https://dl.acm.org/doi/10.1145/2335356.2335371.

[60] S. Ruoti, J. Andersen, T. Hendershot, D. Zappala, and K. Seamons, “Private Webmail 2.0: Simple and Easy-to-Use Secure Email ,” Oct. 2015, http://arxiv.org/abs/1510.08435.

[61] 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. USA: USENIX Association, Jul. 2015, pp. 69–88, https://www.usenix.org/system/files/conference/soups2015/soups15-paper-atwater.pdf.

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

[63] M. Green, “What’s the matter with PGP?” Aug. 2014, https://blog.cryptographyengineering.com/2014/08/13/whats-matter-with-pgp/.

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

[65] D. Cooper, Andrew Regenscheid, Murugiah Souppaya, Christopher Bean, Mike Boyle, Dorothy Cooley, and Michael Jenkins, “Security Considerations for Code Signing,” *NIST Cybersecurity White Paper*, Jan. 2018, https://csrc.nist.rip/external/nvlpubs.nist.gov/nistpubs/CSWP/NIST.CSWP.01262018.pdf.

[66] 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. [Online]. Available: https://dl.acm.org/doi/10.1145/3548606.3560596

[67] 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.

[68] H. Malvai, L. Kokoris-Kogias, A. Sonnino, E. Ghosh, E. Oztürk, K. Lewi, and S. Lawlor, “Parakeet: Practical key transparency for end-to-end encrypted messaging,” *Cryptology ePrint Archive*, 2023.

[69] 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.

[70] 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

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

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

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

[74] 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.

[75] M. Baddeley, *Behavioural Economics: A Very Short Introduction*. Oxford University Press, 01 2017. [Online]. Available: https://doi.org/10.1093/actrade/9780198754992.001.0001

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

[77] 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. [Online]. Available: http://www.jstor.org/stable/2586896

[78] 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

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

[80] W. Jiang, N. Synovic, M. Hyatt, T. R. Schorlemmer, R. Sethi, Y.-H. Lu, G. K. Thiruvathukal, and J. C. Davis, "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. IEEE Press, 2023, p. 2463–2475. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00206

[81] D. K. Remler and G. G. Van Ryzin, *Research methods in practice: strategies for description and causation*, second edition ed. Los Angeles: SAGE, 2015.

[82] C. Wohlin, *Experimentation in software engineering*. New York: Springer, 2012.

[83] J. M. Wooldridge, *Introductory econometrics: a modern approach*, 5th ed. Mason, OH: South-Western Cengage Learning, 2013, oCLC: ocn827938223.

[84] 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.

[85] S. Cass, "The top programming languages 2023," https://spectrum.ieee.org/the-top-programming-languages-2023, August 2023, accessed: 2023-12-07. [Online]. Available: https://spectrum.ieee.org/the-top-programming-languages-2023

[86] 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.

[87] Open Collective, "Ecosyste.ms," 2023, https://ecosyste.ms/.

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

[89] ——, "Removing PGP from PyPI - The Python Package Index," May 2023. [Online]. Available: https://blog.pypi.org/posts/2023-05-23-removing-pgp/

[90] S. D. Relations, "PGP vs. sigstore: A Recap of the Match at Maven Central," 2023. [Online]. Available: https://blog.sonatype.com/pgp-vs.-sigstore-a-recap-of-the-match-at-maven-central

[91] Docker, "Content trust in Docker," 0200, https://docs.docker.com/engine/security/trust/. [Online]. Available: https://docs.docker.com/engine/security/trust/

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

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

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

[95] W. Jiang, N. Synovic, P. Jajal, T. R. Schorlemmer, A. Tewari, B. Pareek, G. K. Thiruvathukal, and J. C. Davis, "PTMTorrent: A Dataset for Mining Open-source Pre-trained Model Packages," *Mining Software Repositories*, 3 2023, https://figshare.com/articles/dataset/PTMTorrent_A_Dataset_for_Mining_Open-source_Pre-trained_Model_Packages/22009880.

[96] 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.

[97] 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.