Mercury: Bandwidth-Effective Prevention of Rollback Attacks Against Community Repositories

Trishank Karthik Kuppusamy, Vladimir Diaz, and Justin Cappos, *New York University*

https://www.usenix.org/conference/atc17/technical-sessions/presentation/kuppusamy

This paper is included in the Proceedings of the
2017 USENIX Annual Technical Conference (USENIX ATC '17).

July 12–14, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-38-6

Mercury: Bandwidth-Effective Prevention of Rollback Attacks Against Community Repositories

Trishank Karthik Kuppusamy Vladimir Diaz Justin Cappos
New York University Tandon School of Engineering

Abstract

A popular community repository such as Docker Hub, PyPI, or RubyGems distributes tens of thousands of software projects to millions of users. The large number of projects and users make these repositories attractive targets for exploitation. After a repository compromise, a malicious party can launch a number of attacks on unsuspecting users, including rollback attacks that revert projects to obsolete and vulnerable versions. Unfortunately, due to the rapid rate at which packages are updated, existing techniques that protect against rollback attacks would cause each user to download 2–3 times the size of an average package in metadata each month, making them impractical to deploy.

In this work, we develop a system called Mercury that uses a novel technique to compactly disseminate version information while still protecting against rollback attacks. Due to a different technique for dealing with key revocation, users are protected from rollback attacks, even if the software repository is compromised. This technique is bandwidth-efficient, especially when delta compression is used to transmit only the differences between previous and current lists of version information. An analysis we performed for the Python community shows that once Mercury is deployed on PyPI, each user will only download metadata each month that is about 3.5% the size of an average package. Our work has been incorporated into the latest versions of TUF, which is being integrated by Haskell, OCaml, RubyGems, Python, and CoreOS, and is being used in production by LEAP, Flynn, and Docker.

1 Introduction

Community repositories, such as Docker Hub [25], Python Package Index (PyPI) [69], RubyGems [71], and SourceForge [80], provide an easy way for third party developers to distribute software to users. Unlike traditional repositories (e.g., Ubuntu, or the Apple App Store), community repositories allow any developer to immediately release new software without waiting for an administrator's approval. This distinctive feature has led to the tremendous popularity of these repositories, which have served billions of downloads to millions of users.

Unfortunately, their popularity also makes them attractive targets for attackers. Major repositories run by Adobe, Apache, Debian, Fedora, FreeBSD, Gentoo, GitHub, GNU Savannah, Linux, Microsoft, npm, Opera, PHP, RedHat, RubyGems, SourceForge, and WordPress have all been compromised at least once [2–4, 21–24, 28, 31, 33, 34, 36, 44, 54, 59, 63, 70, 72, 81, 82, 86–88, 93].

When a community repository is compromised, a number of attacks can be launched on unsuspecting users, including *rollback attacks*, where attackers revert the state of the repository to point to obsolete and vulnerable versions of software. Rollback attacks are trivial for attackers to perform: instead of tampering with signed software, they simply replace these software packages with older versions. It is equally trivial to prevent such attacks for software that is already installed by the user, because existing security systems can easily reject software older than what is already on disk. However, there may be tens of thousands of software projects on a repository, of which the user may install only a fraction. Unless the user keeps track of all projects, she is susceptible to a rollback attack on a project she might install at a much later date. Consequently, she would install authentic but obsolete software that contains known vulnerabilities. An attacker can later exploit these vulnerabilities to compromise her machine.

A solution to prevent rollback attacks needs to meet several important properties in order to be adopted:

- *No administrative overhead.* There must not be additional servers to manage. Many community repositories are managed by volunteers that infrequently interact with the repository, and so the administrative burden must remain low.

- *Simple client communications.* Retrieving a package should not require clients to gossip or communicate with third parties. This could create deployment issues and even security concerns (e.g., informing untrusted parties which security fixes are being requested [13]).

- *Low overhead.* Repositories often have large bandwidth costs and use mirrors or CDNs to offload this burden. A solution must not substantially increase this cost, even if the repository hosts a large number of projects that are rapidly updated.

In this paper, we describe Mercury, a bandwidth-efficient system that prevents security attacks, including rollback attacks, even if a community repository is compromised. This work is innovative in providing low-bandwidth rollback protection. However, the main contribution of this work is how the insights behind Mercury can be used by real-world community repositories to solve a widespread problem. Mercury has been incorporated into the latest versions of TUF [47, 73], which is being integrated by Haskell [94], OCaml [32], RubyGems [75–77], Python [45, 46], and CoreOS [66], and is being used in production by LEAP [49], Flynn [68], and Docker [64].

The key insight in Mercury is that the source of trust about which versions of projects are current can be safely shifted from developers to the repository. The repository uses online keys to sign and distribute the latest version numbers of projects as soon as they are updated. Although attackers can provide clients with incorrect version information when a repository is compromised, Mercury uses several techniques that can limit user susceptibility to rollback attacks even in this case. The key technique is that by always comparing the current list of version information signed by the repository to the previous list, these attacks are easily detected. Mercury is bandwidth-efficient with respect to rollback attacks because it downloads only the version numbers of all projects (instead of metadata about all packages), and uses delta compression [61, 62] to transmit only the differences between previous and current lists. While trusting the repository for version numbers opens users up to a new fast-forward attack, this can be mitigated by performing additional steps when revoking the repository key after a compromise.

In summary, our **contributions** are:

1. We find that existing security systems that prevent rollback attacks incur prohibitive bandwidth costs to do so when the number of projects, or the rate of project updates, is high (e.g., in popular community repositories).

2. We design and implement Mercury, a bandwidth-efficient system that prevents rollback attacks even though it depends on the repository to continually indicate the latest versions of projects.

3. We evaluate the effectiveness of Mercury using requests to PyPI. We find Mercury can prevent rollback attacks by having each user download metadata each month that is about 3.5% the size of an average package. Additionally, new users (or all users following a compromise) will download metadata 48% of the size of an average package (compared to two other systems with overheads of 1,152% and 3,092%).

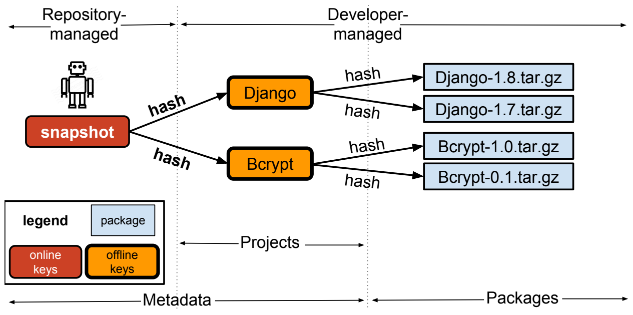


Figure 1: How software is organized by metadata about projects and packages on a community repository. Repositories and developers sign these metadata in order to prevent security attacks. See Section 2 for a detailed explanation.

2 Background

In order to better understand the design decisions behind Mercury, we provide some essential background information. First, we discuss how software is managed and distributed by community repositories. Then, we describe metadata used in an existing security system, TUF [47, 73], that we leverage in Mercury to protect these repositories from security attacks.

TUF is a framework that allows repositories to build different security models that provide varying degrees of security and usability. In this paper, we show that TUF has a severe performance drawback on popular community repositories. Thus, we devised Mercury, a more efficient variant of TUF that prevents rollback attacks using significantly lower bandwidth costs. As we stated earlier, Mercury has been incorporated into the latest versions of TUF.

2.1 Community repositories

A *community repository* is a single server that hosts and distributes third-party software. Three groups of people interact with the repository. *Administrators*, who are usually volunteers, manage the repository software and hardware. *Developers* upload software to the repository, which administrators publish as soon as possible, for users to download. *Users* download, validate, and

Django project metadata file

```
{
  “signatures”: {
    “keyid”: “4445d918dfcf1af804b749eeee4835dccfd27c06b6828533be827473ff6343
              9f”,
    “method”: “ed25519”,
    “sig”: “190f4b6228f2f72b3cbafa3446e032c9eaed03b055acfe8a9d3c445060b47d1b3
            ccd5c1ed9a9367a53d21e1d265f453996268dfeb1f005e530a025b0676ec720”
  },
  “signed”:{
    “packages”: {
      “Django-1.7.tar.gz”: “0654407104e420508cf5be04fb85a066131df3117117dbaca09
                            5e9a248949359”
      “Django-1.8.tar.gz”: “066bad42cb4c66944e7efcf7304d3d17f7b0eb222e53958cdd8
                            66420d2e8b412”
    },
    "expires": "2015-03-21T00:00:00Z",
    “version”: 2
  }
}
```

Figure 2: An example of a project metadata file for the repository in Figure 1, explained in Section 2.2.

Snapshot metadata file

```
{
  “signatures”: {
    “keyid”: “16a0eeb0791b6c92451fd284dd9f599e0a7dbe7f6ebea6e2d2d06c7f74aec1
              12”,
    “method”: “ed25519”,
    “sig”: “7a7e4858a2f86f740c2a9d8627df4cda92f7b4b8e600ea596ffa3623ca31b0e7b
            0e59c3bd601645e03ae5ba0581d2c31a8ce3a879d34afdf09dc3040339bfac”
  },
  “signed”:{
    “projects”: {
      “Django.json”: “1919ff5cc47994470e539169db049f61ff133538ea1b935484e9819e
                      00beb9d6”
      “Bcrypt.json”: “bd162a5385407e07e0e67310d8ebc60abe759d8937bf72ad125802
                      4dff6f561a”
    },
    "expires": "2014-03-29T09:44:10Z"
  }
}
```

Figure 3: An example of a snapshot metadata file for the repository in Figure 1, explained in Section 2.2.

install software with a *package manager* that may download software through middlemen, such as content delivery networks (CDNs) and/or mirrors. These middlemen allow the repository to reduce bandwidth costs.

The software uploaded by developers is organized as follows. A developer registers a *project* with a unique name, such as `Django` or `Bcrypt`. When a specific version of the software for that project is ready to be released, the software is built into a *package* (e.g., `Django-1.7.tar.gz`), and the developer uploads that package to a community repository. A project may make multiple packages available at any time. For example, in Figure 1, even though `Django-1.8.tar.gz` may be the latest package of the `Django` project, `Django-1.7.tar.gz` is still available to users who request it.

2.2 Project and snapshot metadata

Appropriately structured and signed *metadata* can be used to prevent security attacks when a repository is compromised [15, 47, 73]. These metadata are used by package managers to tell whether attackers have tampered with projects, or reverted projects to obsolete versions. In this paper, we focus on two types of metadata.

Project metadata is the manifest of all packages released by a project [47]. It lists the cryptographic hashes for available packages, and includes an expiration date as well as a *version number* for the metadata file itself. In Figure 2, version 2 of the `Django` project metadata lists the hashes for the `Django-1.7.tar.gz` and `Django-1.8.tar.gz` packages, and an expiration date of March 21st 2015. Developers use *offline keys* (or private keys stored off the repository) to sign project metadata, so that attackers cannot modify it without being detected.

Snapshot metadata is the manifest of all project metadata currently available on the repository. Following common practice in traditional repositories [15], snapshot metadata binds the location (e.g., relative path) of every project metadata file to the cryptographic hash of the file [73]. In Figure 3, the snapshot metadata file lists the hashes for the `Django` and `Bcrypt` project metadata files. Since packages and project metadata are continually updated (as often as every few minutes [47]) and made available to users immediately, community repositories use *online keys* (or private keys stored on the repository) to sign snapshot metadata [47]. Because the snapshot key is stored on the repository, an attacker who compromises the repository can sign maliciously generated metadata with that key. In the next few sections, we discuss how Mercury deals with this scenario.

3 Threat model

In this paper, we are concerned with a scenario where attackers have compromised a community repository. Our threat model then assumes that:

1. Attackers can compromise a running repository, and tamper with any files and keys stored on the repository.

2. Developers store their keys off the repository, so that attackers cannot compromise these keys. Project metadata, which is managed and signed by the developers of each project, are not under the control of the attacker.

3. Attackers have access to any file that was previously published on the repository.

4. Attackers are aware of vulnerabilities in outdated packages, and are able to exploit them. These vulnerabilities can be found by looking at security an-

nouncements, or changes in source code repositories such as GitHub. However, attackers do not know of zero-day flaws in packages.

We leverage pre-existing techniques from TUF and other software security systems to provide effective protection against a wide array of other attacks [14, 16, 17, 47, 73]. As a result, our system can recover from key compromises [47, 73] and resist malicious man-in-the-middle attackers or mirrors [16]. Note that these techniques are orthogonal to Mercury, and do not interfere with its evaluation.

This work focuses on *rollback attacks* that cause package managers to install obsolete packages containing known vulnerabilities. A rollback attack happens when a package manager accepts a project that is older than the version at the last time the user visited the repository.

4 Analysis of the limitations of existing systems

A motivation for our work is that existing security systems that can be deployed on community repositories fall short for one of two reasons. They either do not prevent rollback attacks, or require prohibitive bandwidth costs to defend against such attacks.

4.1 Systems that are insecure

Many of the popular community repositories use either HTTPS or package signing (e.g., GPG or RSA) to ensure packages are not tampered with. This system does prevent rollback attacks on projects already installed by the user, because the package manager will not accept a project metadata file with a version number lower than in the previous copy of the file.

However, it suffers from a subtle but serious security problem. The package manager does not know about the version number of project metadata files for packages that are not requested by the user. If the repository is subsequently compromised, then attackers can execute rollback attacks on projects *yet to be installed* by the user. Hence, when an attacker compromises the repository, they can provide package managers out-of-date versions of packages that have known vulnerabilities.

4.2 Systems that are bandwidth-inefficient

As discussed in Section 2, Mercury is a variant of TUF [73], a security system deployed [47] by some community repositories. TUF protects users from rollback attacks by downloading developer-signed project metadata for all projects. This way, if a repository is compromised, the attacker cannot provide forged project metadata. To avoid detection, the package manager must be given project metadata that is at least as current as the previous project metadata downloaded by the package manager.

This system prevents rollback attacks on projects yet to be installed by the user, but has high bandwidth costs in two cases. First, for any new user (i.e., a user that has no previous project metadata), the package manager must download all project metadata files on the repository. This may be large since there may be tens of thousands of projects and hundreds of thousands of packages. Second, projects are continually created or updated on community repositories. Thus, returning users will download significant amounts of metadata to update to the latest version. As a result, this security system can be costly. However, the bandwidth cost for TUF is low should users need to recover from a repository compromise. Since the developer signs all of the project metadata, it need not be revoked if an attacker controls the repository. So, recovery from compromise is inexpensive, while normal operation is costly.

While it is not used in practice, for comparison purposes we also propose TUF-version, a variant of TUF where a project developer separately signs a *project-version metadata* file that simply contains the version number of her project. Then, the package manager downloads all project-version metadata files, but only the project metadata file for the package to be installed. The number of signatures is a significant cost for the project-version metadata. Thus, as we will see later in Section 6, this variant incurs between 37–53% of the cost of TUF, but is still too expensive for community repositories.

5 Mercury: a new security system

To address the limitations of existing security systems, we present Mercury, a security system for community repositories that can prevent rollback attacks while using a reasonable amount of bandwidth. Mercury retains security even if a potentially compromised repository signs version information on behalf of all projects. This is due to its slightly more complicated functionality when recovering from a repository compromise. Thus the "rare case" of recovering from a compromise is less straightforward, but the "common case" of distributing version information requires much less bandwidth. In this section, we discuss how and why package managers using Mercury will be protected from rollback attacks.

5.1 Insight: shifting trust from developers to the repository

Existing systems (Section 4.2) are expensive because they were designed with the assumption that there is no

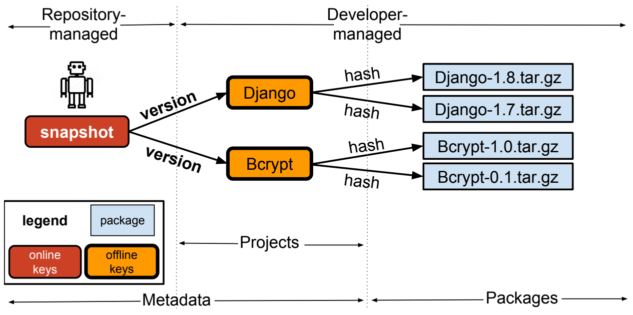
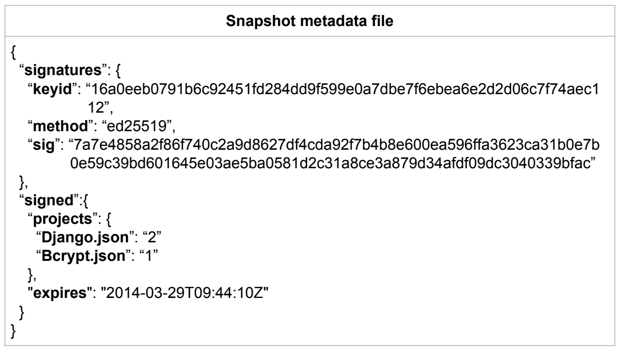


Figure 4: In Mercury, the snapshot metadata binds the location of every project metadata file to the *version number* instead of the hash of the file.


Snapshot metadata file

```
{
 "signatures": {
  "keyid": "16a0eeb0791b6c92451fd284dd9f599e0a7dbe7f6ebea6e2d2d06c7f74aec1
            12",
  "method": "ed25519",
  "sig": "7a7e4858a2f86f740c2a9d8627df4cda92f7b4b8e600ea596ffa3623ca31b0e7b
          0e59c39bd601645e03ae5ba0581d2c31a8ce3a879d34afdf09dc3040339bfac"
 },
 "signed":{
  "projects": {
   "Django.json": "2"
   "Bcrypt.json": "1"
  },
  "expires": "2014-03-29T09:44:10Z"
 }
}
```

Figure 5: An example of a Mercury snapshot metadata file for the repository illustrated in Figure 4. See Section 5.1 for details.

trusted party (e.g., hardware or administrators) on the repository that can always correctly indicate the version numbers of project metadata files. In the absence of this trusted party, package managers have relied on project metadata files signed by developers to learn about version numbers, even though it has meant downloading all new files.

Our key insight is that by handling key revocation in a different manner (Section 5.3), a repository can securely distribute the version numbers of project metadata files in the snapshot metadata. In Mercury, the snapshot metadata binds the location of every project metadata file to its *version number* instead of the hash of the file (as illustrated in Figure 4 and Figure 5). Now, the snapshot metadata informs the package manager not only about which projects on the repository are new or updated, but also gives the version numbers of their corresponding project metadata files. By shifting the source of trust from developers to the repository, Mercury allows the package manager to save bandwidth as long as it: (1) has access to a previous snapshot metadata file that was signed by the repository, and (2) always verifies the current snapshot metadata file as follows.

Suppose the user wishes to install a `Django` package. The package manager begins by downloading the difference between the previous and current snapshot metadata files, s_{prev} and s_{curr}, respectively. Next, the package manager must verify that the version number b of every project metadata file in s_{curr} is greater than or equal to the version number a of the same project metadata file in s_{prev}. If this verification step passes, then it sets s_{prev} to s_{curr}. Finally, the package manager downloads only the `Django` project metadata file, and ensures that the version number c in this file is indeed equal to the version number b for this file in s_{curr}.

There are two reasons why this saves bandwidth cost. First, the package manager downloads only a new project metadata file for the package to be installed, as opposed to all new project metadata files. Second, in Mercury there is a single signature (from the repository) in a snapshot metadata file. With TUF / TUF-version, the package manager downloads all new or updated project / project-version metadata files (and hence metadata about their packages).

5.2 Security analysis

A primary strength of Mercury is that an attacker who compromises the repository cannot rollback projects to versions that existed before the last time the user visited it. This is because whenever the user installs a package, the package manager always compares the current snapshot metadata file s_{curr} to the previous copy s_{prev}. The package manager would detect a rollback attack, and refuse to install the package, if: (1) the version number b of any project metadata file in s_{curr} is lower than the version number a of the same project metadata file in s_{prev}, or (2) the version number c of the project metadata file for the requested package is lower than the version number b for this project metadata file in s_{curr}.

As with existing systems [47,73], the attacker can rollback projects to versions that were added after the last time the user visited the repository. However, unlike existing systems, Mercury provides a stronger method for imposing stringent limits on these attacks (Section 5.4).

Attackers can deny the installation of packages by executing *fast-forward attacks*, where they arbitrarily increase the version numbers of project metadata files in the snapshot metadata. In a sense, fast-forward attacks are the opposite of rollback attacks. In this attack, the version number b of at least one project metadata file in s_{curr} is greater than the version number a of the same project metadata file in s_{prev}. However, this version number b is also greater than the actual version number c contained within the project metadata file itself. Thus, the package manager would refuse to install a package from this project.

Fast-forward attacks are not nearly as severe a threat as rollback attacks because they simply block a package from being installed. Since the attacker has multiple ways to achieve the same goal (the simplest of which is to refuse to serve anything), fast-forward attacks do not present a major threat so long as it is possible to recover from them securely.

5.3 Recovering from a repository compromise

As discussed earlier, attackers who compromise a repository may launch fast-forward attacks that prevent the user from installing newer versions of existing software. This problem can be addressed by replacing the package manager's copy of the snapshot metadata. To do so, administrators must use an offline backup [47] to restore all project metadata and packages to a verifiable point before the compromise. Then, the online keys used to sign snapshot metadata can be revoked and replaced with new keys.

The process for distributing and revoking these keys is borrowed from TUF [47, 73]. The repository signs root-of-trust metadata using a quorum of offline keys. The root-of-trust metadata indicates which keys can be trusted for verifying metadata files, including snapshot metadata files. This leads to a seamless and automatic recovery from fast-forward attacks after a repository compromise.

5.4 Securing out-of-date package managers

The security of a Mercury user relies on her package manager possessing version numbers that are relatively recent. Users who have never visited the repository before are protected against rollback attacks by bundling the latest root-of-trust and snapshot metadata with the package manager. Nevertheless, a package manager using Mercury is vulnerable to rollback attacks against software released after the last time the package manager was updated. (Note that this limitation also applies to TUF for the same reason.) To combat this, a repository can choose to periodically sign a version of the snapshot metadata using offline keys (see s_{mid} in Figure 6). For example, if the repository administrator commits to signing snapshot metadata with offline keys at least every month, then the package manager can first retrieve that snapshot metadata, and verify that it was signed within the last month. Then, it verifies that all version numbers in the snapshot metadata signed with the online keys are later than or equal to those signed with the offline keys. This

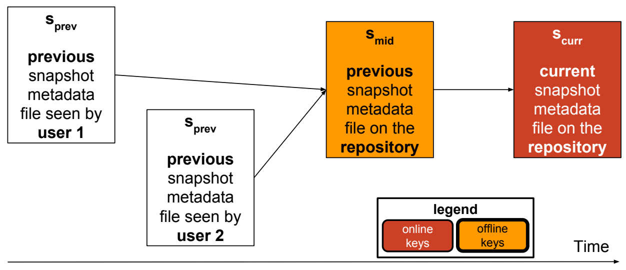

s_{prev}
previous snapshot metadata file seen by user 1
s_{prev}
previous snapshot metadata file seen by user 2
s_{mid}
previous snapshot metadata file on the repository
s_{curr}
current snapshot metadata file on the repository
legend
online keys
offline keys
Time

Figure 6: In order to help outdated package managers catch up to the latest snapshot metadata signed before the repository is compromised, administrators may periodically sign a copy s_{mid} of the latest snapshot metadata using offline keys. See Section 5.4 for details.

prevents attackers who compromised the repository from blocking packages that were released in the last month.

However, this functionality is not used in production by current users of Mercury. This is largely due to two concerns. First, the management overhead of having separate keys stored securely offline was deemed high for this use case. Second, there was some concern that the administrator would forget to sign an update with the offline keys within the prescribed period, and that this would cause users to incorrectly deduce an attack was underway. Hence, Docker [64] and Flynn [68] do not use this feature of Mercury in their deployments.

5.5 Deleting projects from snapshot metadata

It is fairly common practice for community repositories to allow projects to be deleted. However, deleting projects can make it harder for Mercury to defend against rollback attacks. Suppose that the package manager naively drops the version information for projects deleted from the snapshot metadata file. This would enable an attacker who controls the repository to reset known version numbers. Therefore, to better secure a repository using Mercury, projects should not be deleted from snapshot metadata. This is the route that Docker [64] and Flynn [68] chose with their deployments.

5.6 Protection against malicious mirrors

Some community repositories, such as Docker [64], use mirrors to serve metadata and packages to users, which opens users to malicious mirrors that may be able to tamper with some files. Specifically, consider a scenario outside of our threat model, where malicious mirrors do not have access to snapshot metadata keys, but have access to a few keys used to sign some project metadata files. These mirrors cannot tamper with the snapshot metadata.

However, they can substitute a few original project metadata files with malicious project metadata files. These malicious project metadata files contain version numbers identical to original project metadata files, but point to malicious instead of original packages. Mercury cannot detect these substitutions, because there is only information about the version numbers of project metadata files in its snapshot metadata.

In order to address this problem, we propose Mercury-hash, a variant of Mercury where the snapshot metadata contains both version numbers and hashes of all project metadata files. This prevents a malicious mirror from substituting project metadata files without being detected. As we will see in Section 6, this variant incurs 7x the cost of Mercury, which may be acceptable for community repositories where preventing this problem is important.

5.7 Implementation

Our reference implementation of Mercury is written in Python. It includes: command-line tools [89, 90] that help administrators and developers create, sign, and validate metadata (4,661 SLOC); integration libraries that package managers can use to download and verify metadata as well as packages (1,218 SLOC); unit and integration tests (6,247 SLOC); documentation such as specifications, and example metadata.

6 Evaluation of bandwidth costs

In the previous section, we discussed how Mercury is designed to prevent rollback attacks, even if the repository is compromised. In this section, we show that very same design is also efficient with respect to bandwidth cost. Using a log of package downloads from PyPI, the Python community repository, we compare Mercury to existing security systems, and answer the following questions:

1. What is the bandwidth overhead needed by each security system to prevent rollback attacks on PyPI? (Section 6.2)
2. How does the bandwidth overhead change as the number of projects on PyPI is varied? (Section 6.3)
3. How does the bandwidth overhead change as the rate of project updates on PyPI is varied? (Section 6.4)

6.1 Experimental setup

To answer these questions, we obtained an anonymized log of package downloads from PyPI for the month between March 21st and April 19th, 2014. This log contains 69,890,162 package requests by 1,175,625 users (identified by anonymized IP addresses). These users downloaded 46TB of packages, and the average downloaded package size was 660KB. We elected to use the average downloaded package size as a basis of comparison for metadata overhead, because it is the average expenditure when obtaining new or updated software packages. As such, it serves as a logical frame of reference in determining whether metadata overhead is reasonable or excessive.

To measure the cost for a package request in the download log, we must know the file sizes of packages and their corresponding metadata. To obtain package file sizes, we copied all packages hosted on PyPI at the time of writing. (Thus, these are approximations of the file sizes of packages available that month.) To obtain metadata file sizes, we produced 10,981 *releases* of metadata as follows. The first release contains snapshot, project-version, and project metadata about all packages that were available at the beginning of the month. Then, we produced a new release whenever a project was created or updated during the month. The first and last releases describe 58,328 and 59,486 projects, respectively. When computing the cost for a request, we compared using compression, delta encoding [61, 62], or delta compression [41, 42], and chose the most cost-efficient method.

We compare Mercury to four security systems that have been, or can be, deployed by community repositories (Section 4). One security system does not prevent rollback attacks (Section 4.1). In many deployments of this system, used by community repositories such as PyPI and RubyGems, developers use GPG or RSA to sign project metadata. Thus, as a useful abbreviation, we will call this security system GPG/RSA. We also compare Mercury against a variant called Mercury-hash (Section 5.6). Finally, we compare Mercury against TUF [47, 73] and a variant called TUF-version (Section 4.2).

The source code and data for these experiments are freely available at `https://theupdateframework.com/`. Unfortunately, the download log is not publicly available, because it may inadvertently compromise the privacy of PyPI users.

6.2 Bandwidth overhead by security system

The initial benchmark required in our study was the bandwidth cost for all five systems. This was measured by looking at the cost per user. A user may incur three different types of costs. First, a new user who just installed the PyPI package manager incurs an *initial cost* to download its first copy of metadata. Second, a user

	Initial cost	Recurring cost	Recovery cost
GPG/RSA	0.6KB (0.1%)	0.02KB (0.003%)	N/A
Mercury	319KB (48%)	23KB (3.5%)	320KB (48%)
Mercury-hash	2.4MB (360%)	156KB (24%)	2.4MB (361%)
TUF-version	7.6MB (1,152%)	1.1MB (171%)	2.3MB (350%)
TUF	20MB (3,092%)	2.1MB (320%)	2.3MB (350%)

Table 1: The overhead for a user incurred by each security system. The user incurs an initial cost when she contacts PyPI for the first time, a recurring cost when she returns to an uncompromised PyPI after the month of the download log, and a recovery cost when she returns to PyPI after it has recovered from a compromise. The percentages indicate the overhead relative to the average downloaded package size.

	Total initial costs of new users
Packages	2.2TB
GPG/RSA	0.005TB (0.2%)
Mercury	0.4TB (17%)
Mercury-hash	2.8TB (125%)
TUF-version	8.9TB (396%)
TUF	23.9TB (1,067%)

Table 2: The overhead to PyPI incurred by each security system for new users. We consider every IP address that appears for the first time in the download log as a new user. The percentages indicate the overhead relative to the total size of all packages downloaded by these new users.

who returns to an uncompromised PyPI incurs a *recurring cost* to update the metadata (which is the common case). Third, a user who returns to PyPI after it has recovered from a compromise incurs a *recovery cost* to re-download metadata to deal with the compromise.

Table 1 lists these costs. The first column shows the initial cost. The second column shows the recurring cost after the month represented by our download log. We chose this period in order to study the greatest recurring cost that can be measured with the available data. The third column shows the recovery cost.

If she is a new user, then GPG/RSA incurs the lowest initial cost (0.1%) relative to the average downloaded package size. (The cost is equal to the average size of project metadata files available in the last release.) This is because it downloads only the project metadata file for the requested package. However, the user is left vulnerable to rollback attacks against all other projects. Mercury incurs a larger initial cost (48%), because it must also download snapshot metadata about all projects. However, this is a one-time cost, and protects the user from rollback attacks against all known projects. Mercury-hash incurs an even larger initial cost (360%), because its snapshot metadata also contains the hashes of all project metadata files. In contrast, TUF-version (1,152%) and TUF (3,092%) incur significantly higher initial costs for the same protection. This is because they must download all project-version and project metadata, respectively. TUF-version is still bandwidth-inefficient compared to Mercury and even Mercury-hash, because it downloads incompressible signatures for nearly sixty thousand projects.

If she is returning to an uncompromised PyPI after the month, then GPG/RSA again incurs the lowest recurring cost (0.003%), because it downloads only the difference to the project metadata file for the requested package. (This recurring cost is equal to the average size of differences to project metadata files between the first and last releases.) Now, Mercury incurs significantly less recurring cost (3.5%), because it needs to download only the difference to snapshot metadata over the month. Mercury-hash incurs a larger recurring cost (24%), because it needs to also download hashes in its snapshot metadata. In contrast, TUF-version (171%) and TUF (320%) still incur a recurring cost greater than the average downloaded package, because they must download project-version and project metadata, respectively, about all projects created or updated over the month.

If she is returning to PyPI after it has recovered from a compromise, then she may incur a recovery cost for re-downloading metadata. GPG/RSA does not have a recovery cost (N/A), because there is no recovery procedure to reset the version numbers of projects after a repository compromise. On the other hand, Mercury has a recovery cost (48%) that is dominated by snapshot metadata (for the reason explained in Section 5.3). Similarly, the recovery cost (350%) for TUF-version and TUF are dominated by snapshot metadata. The snapshot metadata in both systems is always a few times larger than in Mercury, because it lists random hashes, instead of version numbers. However, note that, in this instance, the recovery cost for Mercury-hash is the largest, because its snapshot metadata contains both hashes and version numbers.

Finally, for the sake of completeness, we also look at the cost to PyPI incurred by "new" users in this month, or users who appear for the first time in our download log. We do not have ground truth about the number of new versus returning users, since that information cannot be determined from our download log. However, we can get a conservative estimate of this size, by assuming all users are new. While this may overestimate the cost to PyPI in this month, this is accurate for many virtualized environments, such as continuous integration / deployment systems. Table 2 lists these costs. New users downloaded 2.2TB of packages, and GPG/RSA adds to this an overhead of 0.005TB (0.2%) in project metadata. Mercury and Mercury-hash add an overhead of 0.4TB (17%) and 2.8TB (125%), mostly due to snapshot meta-

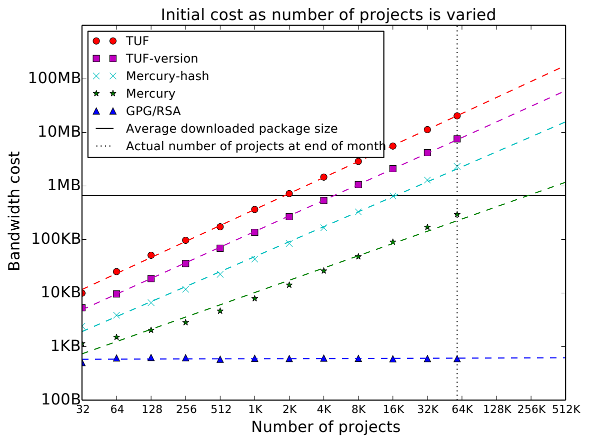


Figure 7: The initial cost for a user incurred by each security system, depending on the number of projects. The dashed lines show the regression lines based on the observed data (points) for each system. The horizontal line marks the average downloaded package size, whereas the vertical line marks the actual number of projects on PyPI at the end of the month. Points on the x and y axes have been plotted on the log-2 and log-10 scales, respectively.

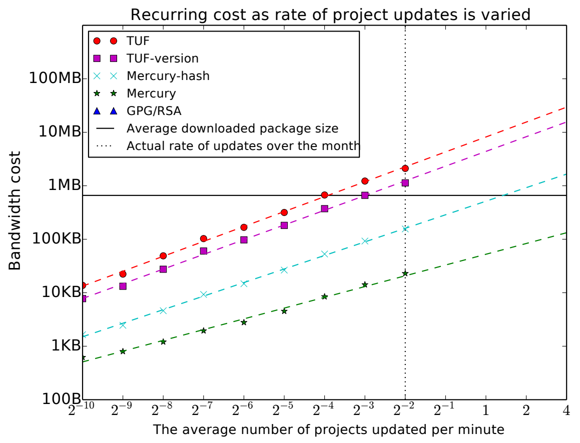


Figure 8: The recurring cost for a user incurred by each security system, if she returns to an uncompromised PyPI after the month of the download log, depending on the rate of project updates. The dashed lines show the regression lines based on the observed data (points) for each system. The horizontal line marks the average downloaded package size, whereas the vertical line marks the actual rate of project updates on PyPI over the month. Points on the x and y axes have been plotted on the log-2 and log-10 scales, respectively. Note that GPG/RSA is not represented in this figure, because its average recurring cost is effectively zero, since most projects were not updated during that month.

data. TUF-version adds an overhead of 8.9TB (396%), more than two-thirds of which is due to project version metadata. TUF adds an overhead of 23.9TB (1,067%), eighty-nine percent of which is due to project metadata. It is fair to conclude that it is not practical for community repositories to deploy TUF, or even TUF-version, especially when Mercury, and even Mercury-hash, can prevent rollback attacks just as well, but at a fraction of the cost.

6.3 Bandwidth versus number of projects

This subsection focuses on how the bandwidth costs would vary for a repository that has fewer or more projects than PyPI did at the end of the month. To answer this, we looked at how the initial cost for a new user would change as the number of projects in the last release (which contains the largest number of projects) is varied. (We focused on this cost because changing the number of projects on the repository would affect new users the most.) To study the cost if the number of projects is smaller than in the last release, we produced a new release based on a random sample of projects. On the other hand, to study the cost if the number of projects is larger than in the last release, we used linear regression to extrapolate the costs for this number based on the costs for smaller numbers of projects.

Figure 7 shows these costs. The vertical line marks the number of projects at the end of the month. The bandwidth overhead at that number of projects for all security systems is similar to the first column of Table 1. The horizontal line marks the average downloaded package size.

The initial cost for GPG/RSA changes little as the number of projects is varied, because its cost depends only on the size of the average project metadata file. In contrast, the initial costs for Mercury, Mercury-hash, TUF-version, and TUF grow linearly with the number of projects. With Mercury, this cost is dominated by the snapshot metadata. It outgrows the average downloaded package if the number of projects on PyPI grows larger by more than 4x (256K). The cost for Mercury-hash is also dominated by the snapshot metadata. However, it outgrows the average downloaded package if the number of projects on PyPI is nearly 3.4x smaller (17K).

Unlike Mercury, the costs for TUF-version and TUF are dominated by project-version and project metadata files, respectively. In fact, the cost for TUF-version is already greater than the average downloaded package if the number of projects on PyPI is nearly 12x smaller (5K), and for TUF if this number is more than 29x smaller (2K).

6.4 Bandwidth versus rate of project updates

This last subsection focuses on how the bandwidth costs would vary for a repository that has a lower or higher rate of project updates than PyPI did over the month. To answer this, we looked at how the recurring cost for a returning user would change as the rate of project updates varies between the beginning and end of the month. (We focused on this cost because changing this rate would principally affect users who are returning to an uncompromised repository.) Between these two points, 3,612 projects were created or updated 10,980 times. To study the cost if this rate is decreased, we artificially decreased it by increasing the time interval between the first and any subsequent release. For example, say that there are only three releases, and that the second and last releases were produced n and $2n$ minutes, respectively, after the first release. Since we assume that this user returns to the repository at the end of the month (say, at $2n+1$ minutes), her security system would download metadata from the last release. To artificially slow down the rate of project updates by half, the second and last releases would arrive $2n$ and $4n$ minutes, respectively, after the first release. Now, her security system would download metadata from the second release instead of the last one. On the other hand, to study the cost if this rate is increased, we used linear regression to extrapolate the costs for larger rates based on the costs for smaller rates.

Figure 8 shows these costs. The vertical line marks the actual rate of project updates at the end of the month (2^{-2} projects per minute, or a project every 4 minutes). The bandwidth overhead then for all security systems is identical to the second column of Table 1. The horizontal line marks the average downloaded package size. Note that GPG/RSA is not represented in this figure, because its average recurring cost is effectively zero, since most projects were not updated during that month.

When the rate of project updates is varied, the cost for Mercury and Mercury-hash are determined by the differences to snapshot metadata as projects are created or updated. However, the cost for Mercury remains well under the average downloaded package even if the rate of project updates is 16x higher than on PyPI (4 projects per minute, or 16 projects every 4 minutes). The cost for Mercury-hash is only greater than the average downloaded package when the rate of project updates is nearly 5.7x higher than on PyPI (1.4 projects per minute, or 5.6 projects every 4 minutes). In contrast, the cost for TUF-version is already greater than the average downloaded package when the rate of project updates is 2x lower than on PyPI (2^{-3} projects per minute, or a project every 8 minutes), and for TUF when this rate is 4x lower (2^{-4} projects per minute, or a project every 16 minutes).

7 Related work

In this section, we survey some prior work that is related to Mercury.

Accountability systems. Accountability systems, such as PeerReview [37], CATS [95], and CloudProof [67], provide a way to detect a subclass of Byzantine failures in distributed systems. All of these systems can detect rollback attacks after they happen, but, unlike Mercury, they are not designed to prevent such attacks before they occur.

Security systems for software repositories. Previous work have shown software updaters to be prone to security problems such as rollback attacks [5, 15]. Popular Linux package managers use a security architecture that protects against malicious mirrors or CDNs [15]. But, unlike Mercury, it will not necessarily withstand a compromise of the original repository [47, 73].

Revere [53] uses a self-organizing, peer-to-peer (P2P) overlay network to deliver security updates. However, a P2P setup would increase the complexity of deploying a community repository, and as such, was deemed impractical by the administrators we have been working with. Since Mercury does not require a P2P setup, it is an easier system to put in place.

File systems for untrusted storage servers. In this subsection, we will discuss a number of file systems that are inherently designed to detect whether attackers have tampered with packages. The biggest difference is that Mercury is not a file system, which means that repositories are free to use any file system that they like. Mercury works on top of existing file systems, and requires repositories only to add a layer of signed metadata, and modifying package managers to verify these metadata before installing packages.

ECFS [6] and TCFS [19], both of which are based on the Cryptographic File System (CFS) [7], allow developers to share files with users by offering the option of not encrypting files. However, ECFS does not appear to prevent rollback attacks on files not yet read by the user, whereas TCFS does not prevent rollback attacks at all, because unencrypted files are not protected with digital signatures. By providing security without the need to encrypt, Mercury offers a more accessible alternative.

To guarantee freshness, SiRiUS [35] requires every project developer to sign a hash tree of metadata files. This signature expires quickly, and so a software agent acting on behalf of the developer must renew it every few seconds or minutes. Unfortunately, this would not work on community repositories that provide rarely updated projects which are still heavily used, but no longer actively maintained by developers [47]. Unlike SiRiUS, Mercury does not require developers to quickly renew signatures on project metadata.

SNAD [60] can prevent rollback attacks against all projects by using a certificate object, which serves a similar purpose to the snapshot metadata in Mercury. However, SNAD is computationally expensive for community repositories, because all files must be encrypted, even though these repositories have no need for encryption.

The Protected File System (PFS) [84] records hashes of file blocks, where each hash is parameterized not only with the file block itself, but also a secret key kept on trusted storage. This prevents attackers from tampering with blocks. However, like Iris [83], PFS assumes that both developers and users would share the same secret key to read and write files. Sharing this secret key only makes sense when the users share the same computer, as in PFS, or the same organization, as in Iris. Mercury does not require developers or users to share a secret key, which means that they do not have to share the same computer or organization.

Security systems with different trust assumptions. SUNDR [51, 57] is a file system designed for software repositories. Unlike Mercury, SUNDR can prevent rollback attacks as well as detect forking attacks [8–12, 78] despite using a single untrusted server. However, the price of this is that SUNDR requires clients to trust that other clients would honestly report whether the repository has performed a forking attack. The problem is that a single faulty or malicious client could accidentally or deliberately frame an honest repository.

Depot [55] is a file system that eliminates trust for safety, and minimizes trust for liveness and availability. Unlike Mercury, Depot not only detects forking attacks, but can continue functioning despite these attacks. However, the price is potentially high bandwidth costs, because Depot is essentially a replication protocol that requires clients to continually exchange updates about all read and write operations with servers or other clients.

In the most popular method used in file systems to provide file integrity, a trusted party signs a Merkle hash tree [58] over a set of files [27,29,30,35,39,40,56,65,83, 92]. Unfortunately, there is no such trusted party on community repositories. Community repositories must use online keys instead to sign the root of this tree, because packages are continually updated, and must be published as soon as possible (Section 2.2). Unfortunately, attackers who compromise the repository can use these online keys to sign new hash trees that point to obsolete project metadata files. Mercury does not use hash trees, and addresses this problem by distributing version numbers of all project metadata files using the snapshot metadata, which help to prevent rollback attacks.

Other systems, such as Proof of Freshness [92], A2M [20], and TrInc [50], assume that there is trusted hardware (such as a Trusted Platform Module [91] chip). Unfortunately, except in limited settings [85], such trusted hardware is generally not available on commodity cloud servers that community repositories may use to host packages [1]. Mercury does not need trusted hardware, which greatly increases where it can be deployed.

Byzantine fault-tolerant security systems. Byzantine fault-tolerant (BFT) systems use many replicas instead of a single server to execute operations [18, 26, 38, 43, 52, 74, 79]. Unfortunately, PBFT requires administrators to manage $3f+1$ independent replicas instead of a single server [48], where f is the maximum number of repositories whose compromise can be tolerated. This significantly increases administrative burden. Mercury can work using only a single server, making it less expensive, and more easily deployable.

8 Conclusions

As community repositories continue to grow in popularity, so does the need for reliable and economically-feasible security systems to protect users from a number of possible attacks. Solutions that require developers to indicate the latest version number are too costly to be used in practice. In this paper, we present Mercury, a security system that instead uses the community repository to indicate the latest version numbers of projects. Although attackers can compromise the repository, Mercury always prevents rollback attacks, and its recovery mechanism helps users recover from fast-forward attacks. Using a key on the repository to sign the version number for every project allows Mercury to efficiently use bandwidth to prevent rollback attacks.

The Mercury source code and standards documents are freely available at `https://theupdateframework.com/`.

Acknowledgements

We thank our shepherd, Eric Eide, and the anonymous reviewers for their valuable comments. We would also like to thank Lois Anne DeLong for her efforts on this paper, as well as the Docker, CoreOS, Flynn, Haskell, LEAP, OCaml, Python, Ruby, and Square communities for their collaboration. Our work on Mercury was supported by U.S. National Science Foundation grants CNS-1345049 and CNS-0959138.

References

[1] ACHEMLAL, M., GHAROUT, S., AND GABER, C. Trusted Platform Module as an Enabler for Security in Cloud Computing. In *Network and Information Systems Security (SAR-SSI), 2011 Conference on* (May 2011), pp. 1–6.

[2] Apache Infrastructure Team. apache.org incident report for 8/28/2009. `https://blogs.apache.org/infra/entry/apache_org_downtime_report`, 2009.

[3] Apache Infrastructure Team. apache.org incident report for 04/09/2010. `https://blogs.apache.org/infra/entry/apache_org_04_09_2010`, 2010.

[4] Arkin, B. Adobe to Revoke Code Signing Certificate. `https://blogs.adobe.com/conversations/2012/09/adobe-to-revoke-code-signing-certificate.html`, 2012.

[5] Bellissimo, A., Burgess, J., and Fu, K. Secure Software Updates: Disappointments and New Challenges. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Security* (Berkeley, CA, USA, 2006), HOTSEC'06, USENIX Association, pp. 7–7.

[6] Bindel, D., Chew, M., and Wells, C. Extended Cryptographic File System, 1999. Unpublished. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.4339`.

[7] Blaze, M. A Cryptographic File System for UNIX. In *Proceedings of the 1st ACM Conference on Computer and Communications Security* (New York, NY, USA, 1993), CCS '93, ACM, pp. 9–16.

[8] Cachin, C., and Geisler, M. *Applied Cryptography and Network Security: 7th International Conference, ACNS 2009, Paris-Rocquencourt, France, June 2-5, 2009. Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, ch. Integrity Protection for Revision Control, pp. 382–399.

[9] Cachin, C., Keidar, I., and Shraer, A. Fork sequential consistency is blocking. *Information Processing Letters 109*, 7 (2009), 360–364.

[10] Cachin, C., Keidar, I., and Shraer, A. Fail-Aware Untrusted Storage. *SIAM Journal on Computing 40*, 2 (2011), 493–533.

[11] Cachin, C., and Ohrimenko, O. *Principles of Distributed Systems: 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings*. Springer International Publishing, Cham, 2014, ch. Verifying the Consistency of Remote Untrusted Services with Commutative Operations, pp. 1–16.

[12] Cachin, C., Shelat, A., and Shraer, A. Efficient Fork-linearizable Access to Untrusted Shared Memory. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2007), PODC '07, ACM, pp. 129–138.

[13] Cappos, J. Avoiding Theoretical Optimality to Efficiently and Privately Retrieve Security Updates. In *Financial Cryptography and Data Security: 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers* (Berlin, Heidelberg, 2013), A.-R. Sadeghi, Ed., Springer Berlin Heidelberg, pp. 386–394.

[14] Cappos, J., Baker, S., Plichta, J., Nyugen, D., Hardies, J., Borgard, M., Johnston, J., and Hartman, J. H. Stork: package management for distributed VM environments. In *The 21st Large Installation System Administration Conference, LISA'07* (2007).

[15] Cappos, J., Samuel, J., Baker, S., and Hartman, J. H. A look in the mirror: Attacks on package managers. In *Proceedings of the 15th ACM conference on Computer and communications security* (2008), ACM, pp. 565–574.

[16] Cappos, J., Samuel, J., Baker, S., and Hartman, J. H. Package management security. Tech. Rep. TR-08-02, University of Arizona, 2008.

[17] Capppos, J. *Stork: Secure Package Management for VM Environments*. Dissertation, University of Arizona, 2008.

[18] Castro, M., and Liskov, B. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 1999), OSDI '99, USENIX Association, pp. 173–186.

[19] Cattaneo, G., Catuogno, L., Sorbo, A. D., and Persiano, P. The Design and Implementation of a Transparent Cryptographic File System for UNIX. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2001), USENIX Association, pp. 199–212.

[20] Chun, B.-G., Maniatis, P., Shenker, S., and Kubiatowicz, J. Attested Append-only Memory: Making Adversaries Stick to Their Word. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 189–204.

[21] CORBET, J. An attempt to backdoor the kernel. http://lwn.net/Articles/57135/, 2003.

[22] CORBET, J. The cracking of kernel.org. http://www.linuxfoundation.org/news-media/blogs/browse/2011/08/cracking-kernelorg, 2011.

[23] DEBIAN. Debian Investigation Report after Server Compromises. https://www.debian.org/News/2003/20031202, 2003.

[24] DEBIAN. Security breach on the Debian wiki 2012-07-25. https://wiki.debian.org/DebianWiki/SecurityIncident2012, 2012.

[25] DOCKER INC. Docker Hub. https://hub.docker.com/.

[26] DOUCEUR, J. R., AND WATTENHOFER, R. P. Optimizing file availability in a secure serverless distributed file system. In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems* (2001), pp. 4–13.

[27] DUCHAMP, D. A Toolkit Approach to Partially Connected Operation. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 1997), USENIX Association, pp. 23–23.

[28] FRIELDS, P. W. Infrastructure report, 2008-08-22 UTC 1200. https://www.redhat.com/archives/fedora-announce-list/2008-August/msg00012.html, 2008.

[29] FU, K., KAASHOEK, M. F., AND MAZIÈRES, D. Fast and Secure Distributed Read-only File System. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4* (Berkeley, CA, USA, 2000), OSDI'00, USENIX Association.

[30] FU, K. E. Group sharing and random access in cryptographic storage file systems. Master's thesis, Massachusetts Institute of Technology, 1999.

[31] GENTOO LINUX. rsync.gentoo.org: rotation server compromised. https://security.gentoo.org/glsa/200312-01, 2003.

[32] GESBERT, L., AND MEHNERT, H. Signing the OPAM repository. https://opam.ocaml.org/blog/Signing-the-opam-repository/, 2015.

[33] GITHUB, INC. Public Key Security Vulnerability and Mitigation. https://github.com/blog/1068-public-key-security-vulnerability-and-mitigation, 2012.

[34] GNU SAVANNAH. Compromise2010. https://savannah.gnu.org/maintenance/Compromise2010/, 2010.

[35] GOH, E.-J., SHACHAM, H., MODADUGU, N., AND BONEH, D. SiRiUS: Securing Remote Untrusted Storage. In *The 10th Annual Network and Distributed System Security Symposium* (San Diego, CA, Feb. 2003), The Internet Society, pp. 131–145.

[36] GOODIN, D. Attackers sign malware using crypto certificate stolen from Opera Software. http://arstechnica.com/security/2013/06/attackers-sign-malware-using-crypto-certificate-stolen-from-opera-software/, 2013.

[37] HAEBERLEN, A., KOUZNETSOV, P., AND DRUSCHEL, P. PeerReview: Practical Accountability for Distributed Systems. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 175–188.

[38] HO, C., VAN RENESSE, R., BICKFORD, M., AND DOLEV, D. Nysiad: Practical Protocol Transformation to Tolerate Byzantine Failures. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2008), NSDI'08, USENIX Association, pp. 175–188.

[39] KALLAHALLA, M., RIEDEL, E., SWAMINATHAN, R., WANG, Q., AND FU, K. Plutus: Scalable Secure File Sharing on Untrusted Storage. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2003), FAST '03, USENIX Association, pp. 29–42.

[40] KEYBASE. Introducing the Keybase filesystem, 2016. https://keybase.io/docs/kbfs.

[41] KORN, D., MACDONALD, J., MOGUL, J., AND VO, K. The VCDIFF Generic Differencing and Compression Data Format. RFC 3284 (Proposed Standard). https://tools.ietf.org/html/rfc3284, June 2002.

[42] KORN, D. G., AND VO, K.-P. A Generic Differencing and Compression Data Format. Tech. Rep. HA1630000-021899-02TM, AT&T Labs - Research, February, 1999.

[43] KOTLA, R., ALVISI, L., DAHLIN, M., CLEMENT, A., AND WONG, E. Zyzzyva: Speculative Byzantine Fault Tolerance. In *Proceedings of Twenty-first*

ACM SIGOPS Symposium on Operating Systems Principles (New York, NY, USA, 2007), SOSP '07, ACM, pp. 45–58.

[44] KUHN, B. M. News: IMPORTANT: Information Regarding Savannah Restoration for All Users. https://savannah.gnu.org/forum/forum.php?forum_id=2752, 2003.

[45] KUPPUSAMY, T. K., DIAZ, V., STUFFT, D., AND CAPPOS, J. PEP 458 – Securing the Link from PyPI to the End User. https://www.python.org/dev/peps/pep-0458/, 2013.

[46] KUPPUSAMY, T. K., DIAZ, V., STUFFT, D., AND CAPPOS, J. PEP 480 – Surviving a Compromise of PyPI. https://www.python.org/dev/peps/pep-0480/, 2014.

[47] KUPPUSAMY, T. K., TORRES-ARIAS, S., DIAZ, V., AND CAPPOS, J. Diplomat: Using Delegations to Protect Community Repositories. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, Mar. 2016), USENIX Association, pp. 567–581.

[48] LAMPORT, L., SHOSTAK, R., AND PEASE, M. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst. 4*, 3 (July 1982), 382–401.

[49] LEAP ENCRYPTION ACCESS PROJECT. New releases for a new year - LEAP. https://leap.se/en/2014/darkest-night, 2014.

[50] LEVIN, D., DOUCEUR, J. R., LORCH, J. R., AND MOSCIBRODA, T. TrInc: Small Trusted Hardware for Large Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2009), NSDI'09, USENIX Association, pp. 1–14.

[51] LI, J., KROHN, M., MAZIÈRES, D., AND SHASHA, D. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association, pp. 9–9.

[52] LI, J., AND MAZIÉRES, D. Beyond One-third Faulty Replicas in Byzantine Fault Tolerant Systems. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2007), NSDI'07, USENIX Association.

[53] LI, J., REIHER, P., AND POPEK, G. J. Resilient self-organizing overlay networks for security update delivery. *Selected Areas in Communications, IEEE Journal on 22*, 1 (2004), 189–202.

[54] MAGNUSSON, H. The PHP project and Code Review. http://bjori.blogspot.com/2010/12/php-project-and-code-review.html, 2010.

[55] MAHAJAN, P., SETTY, S., LEE, S., CLEMENT, A., ALVISI, L., DAHLIN, M., AND WALFISH, M. Depot: Cloud Storage with Minimal Trust. *ACM Trans. Comput. Syst. 29*, 4 (Dec. 2011), 12:1–12:38.

[56] MAHESHWARI, U., VINGRALEK, R., AND SHAPIRO, W. How to Build a Trusted Database System on Untrusted Storage. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4* (Berkeley, CA, USA, 2000), OSDI'00, USENIX Association.

[57] MAZIÈRES, D., AND SHASHA, D. Building Secure File Systems out of Byzantine Storage. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing* (New York, NY, USA, 2002), PODC '02, ACM, pp. 108–117.

[58] MERKLE, R. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology — CRYPTO '87*, C. Pomerance, Ed., vol. 293 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1988, pp. 369–378.

[59] MICROSOFT, INC. Flame malware collision attack explained. http://blogs.technet.com/b/srd/archive/2012/06/06/more-information-about-the-digital-certificates-used-to-sign-the-flame-malware.aspx, 2012.

[60] MILLER, E., LONG, D., FREEMAN, W., AND REED, B. Strong security for distributed file systems. In *Performance, Computing, and Communications, 2001. IEEE International Conference on.* (2001), IEEE, pp. 34–40.

[61] MOGUL, J., KRISHNAMURTHY, B., DOUGLIS, F., FELDMANN, A., GOLAND, Y., VAN HOFF, A., AND HELLERSTEIN, D. Delta encoding in HTTP. RFC 3229 (Proposed Standard). https://tools.ietf.org/html/rfc3229, Jan. 2002.

[62] MOGUL, J. C., DOUGLIS, F., FELDMANN, A., AND KRISHNAMURTHY, B. Potential Benefits of Delta Encoding and Data Compression for HTTP. In *Proceedings of the ACM SIGCOMM '97 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 1997), SIGCOMM '97, ACM, pp. 181–194.

[63] MULLENWEG, M. Passwords Reset. https://wordpress.org/news/2011/06/passwords-reset/, 2011.

[64] MÓNICA, D., AND DOCKER, INC. Introducing Docker Content Trust. https://blog.docker.com/2015/08/content-trust-docker-1-8/, 2015.

[65] OPREA, A., AND REITER, M. K. Integrity Checking in Cryptographic File Systems with Constant Trusted Storage. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium* (Berkeley, CA, USA, 2007), SS'07, USENIX Association, pp. 13:1–13:16.

[66] PHILIPS, B. Evaluate The Update Framework. https://github.com/appc/spec/issues/211, 2015.

[67] POPA, R. A., LORCH, J. R., MOLNAR, D., WANG, H. J., AND ZHUANG, L. Enabling Security in Cloud Storage SLAs with CloudProof. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2011), USENIXATC'11, USENIX Association.

[68] PRIME DIRECTIVE, INC. Development - Flynn. https://flynn.io/docs/development, 2015.

[69] PYTHON SOFTWARE FOUNDATION. PyPI - the Python Package Index: Python Package Index. https://pypi.python.org/pypi.

[70] RED HAT, INC. Infrastructure report, 2008-08-22 UTC 1200. https://rhn.redhat.com/errata/RHSA-2008-0855.html, 2008.

[71] RUBYGEMS.ORG. RubyGems.org — your community gem host. https://rubygems.org/.

[72] RUBYGEMS.ORG. Data Verification. http://blog.rubygems.org/2013/01/31/data-verification.html, 2013.

[73] SAMUEL, J., MATHEWSON, N., CAPPOS, J., AND DINGLEDINE, R. Survivable key compromise in software update systems. In *Proceedings of the 17th ACM conference on Computer and communications security* (2010), ACM, pp. 61–72.

[74] SCHNEIDER, F. B. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv. 22*, 4 (Dec. 1990), 299–319.

[75] SHAY, X., AND SQUARE, INC. Securing RubyGems with TUF, Part 1. https://goo.gl/XO4AHu, 2013.

[76] SHAY, X., AND SQUARE, INC. Securing RubyGems with TUF, Part 2. https://goo.gl/yymo8z, 2013.

[77] SHAY, X., AND SQUARE, INC. Securing RubyGems with TUF, Part 3. https://goo.gl/pPKcgB, 2013.

[78] SHRAER, A., CACHIN, C., CIDON, A., KEIDAR, I., MICHALEVSKY, Y., AND SHAKET, D. Venus: Verification for Untrusted Cloud Storage. In *Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop* (New York, NY, USA, 2010), CCSW '10, ACM, pp. 19–30.

[79] SINGH, A., FONSECA, P., KUZNETSOV, P., RODRIGUES, R., AND MANIATIS, P. Zeno: Eventually Consistent Byzantine-fault Tolerance. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2009), NSDI'09, USENIX Association, pp. 169–184.

[80] SLASHDOT MEDIA. About. http://sourceforge.net/about.

[81] SLASHDOT MEDIA. phpMyAdmin corrupted copy on Korean mirror server. https://sourceforge.net/blog/phpmyadmin-back-door/, 2012.

[82] SMITH, J. K. Security incident on Fedora infrastructure on 23 Jan 2011. https://lists.fedoraproject.org/pipermail/announce/2011-January/002911.html, 2011.

[83] STEFANOV, E., VAN DIJK, M., JUELS, A., AND OPREA, A. Iris: A Scalable Cloud File System with Efficient Integrity Checks. In *Proceedings of the 28th Annual Computer Security Applications Conference* (New York, NY, USA, 2012), ACSAC '12, ACM, pp. 229–238.

[84] STEIN, C. A., HOWARD, J. H., AND SELTZER, M. I. Unifying File System Protection. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2001), USENIX Association, pp. 79–90.

[85] TECTONIC/COREOS, INC. Tectonic with Distributed Trusted Computing, 2016. https://tectonic.com/trusted-computing/.

[86] THE FREEBSD PROJECT. FreeBSD.org intrusion announced November 17th 2012. http://www.freebsd.org/news/2012-compromise.html, 2012.

[87] THE PHP GROUP. php.net security notice. http://www.php.net/archive/2011.php#id2011-03-19-1, 2011.

[88] THE PHP GROUP. A further update on php.net. http://php.net/archive/2013.php#id2013-10-24-2, 2013.

[89] THE UPDATE FRAMEWORK. Developer Tools. https://github.com/theupdateframework/tuf/blob/develop/tuf/README-developer-tools.md, 2014.

[90] THE UPDATE FRAMEWORK. Repository Management. https://github.com/theupdateframework/tuf/blob/develop/tuf/README.md, 2014.

[91] TRUSTED COMPUTING GROUP. Trusted Platform Module (TPM), 2016. https://www.trustedcomputinggroup.org/work-groups/trusted-platform-module/.

[92] VAN DIJK, M., SARMENTA, L. F., O'DONNELL, C. W., AND DEVADAS, S. Proof of freshness: How to efficiently use an online single secure clock to secure shared untrusted memory. Tech. Rep. CSG Memo 496, Massachusetts Institute of Technology, 2006.

[93] VOSS, L. Newly Paranoid Maintainers. http://blog.npmjs.org/post/80277229932/newly-paranoid-maintainers, 2014.

[94] WELL-TYPED LLP. Improving Hackage security. http://www.well-typed.com/blog/2015/04/improving-hackage-security/, 2015.

[95] YUMEREFENDI, A. R., AND CHASE, J. S. Strong Accountability for Network Storage. *Trans. Storage 3*, 3 (Oct. 2007).