SPAM: a Secure Package Manager

Fraser Brown* Ariana Mirian† Atyansh Jaiswal† Andres Nötzli* Deian Stefan†

*Stanford University †UC San Diego

Abstract

Uncurated package registries are extremely appealing attack vectors—why send out phishing emails when you can insert a backdoor into a popular library and target every user of the startups, banks, and government applications that rely on the library? Unfortunately, popular package management systems do very little to keep up with this threat; at best they host packages behind HTTPS. We argue for a more secure package manager, one that can cope with nation state adversaries (who have a history of infiltrating codebases). We describe the design of one such secure system—SPAM—that uses the new Stellar federated Byzantine fault tolerant system.

1 Introduction

Package managers present a security paradox: their job is to make it easy and secure to pull and install code from the internet. Uncurated package managers, package managers that do not impose restrictions on the packages that users upload, have an even tougher job precisely *because* of this lack of restriction. Uncurated managers have been wildly successful because they make it easy to install and publish packages, but they have also been subject to a number of security flaws. For example, in March of 2014, security researchers discovered a bug in npm that would allow an attacker complete control of the registry [1]. This attacker, for example, could replace packages with malicious counterparts or remove packages altogether. A similar vulnerability in RubyGems came to light when a user uploaded a proof-of-concept exploit that allowed remote code execution [2, 3].

Most existing uncurated systems focus on reducing risk by securing the connection between the client and the registry to prevent network attackers from tampering with registry data in transit; npm and pip/PyPI [4, 5] have both switched to making registry requests over `https` over the past few years. Similarly, most uncurated package managers check the hash of a package once it has been downloaded, a request (made over `https`) meant to ensure the integrity of the downloaded package [5]. These systems do *not* protect against registry compromise, but recent research addresses this threat. For example, the Diplomat system presents a key-based method for ensuring the integrity of packages in such an event [6].

These systems, though trying to address threats *outside* of the registry infrastructure, are not secure against internal threats: malicious or negligent maintainers and developers can compromise the registry ecosystem for all of its users. For example, in January of 2016, researchers described how evil npm software would propagate: once downloaded from the registry, this malware would add itself as a dependency to all of the infected developers' projects [7]. In a blog post, npm maintainers argue that this is a necessary side-effect of allowing arbitrary install scripts, and that "the utility of having installation scripts is greater than the risk of worms." Furthermore, "if a large number of users make a concerted effort to publish malicious packages to npm, malicious packages will be available on npm." However, embracing the dominant attitude of uncurated package managers, they also believe that "npm is largely a community of benevolent, helpful people, and so the overwhelming majority of software in the registry is safe and often useful" [7].

In contrast, other un- or lightly-curated systems have *not* found their communities to be full of benevolent, helpful people. The Google Play store struggles with data-stealing apps [8], rooting apps [9], and botnet apps [10]. Infection rates across Android app marketplaces ranged from 0.02%–0.47% in 2012, and researchers discovered multiple apps exploiting zero-day vulnerabilities during the course of this analysis [11]. The Chrome Extension Store faces similar problems: in 2014, Kapravelos et al. detected 130 malicious and 4,712 suspicious extensions [12]. Malware and adware creators have also purchased popular Chrome extensions and then pushed automatic, malicious updates to extension users [13].

Similarly, registry administrators may not always act—or may be compelled *not* to act—in the best interests of their users. For example, npm administrators have reuploaded packages after they are pulled by developers [14]. Many registries are also backed by for-profit companies[1]; trusting these registries to protect their users is like trusting Pinto-era Ford to protect its drivers—a gamble at best. Furthermore, even if npm is staffed by the opposite of Ford execs, governments may compel them to tamper with the registry anyway. At the request of the US government, for example, Yahoo! included custom backdoors to allow agencies to search user emails [15]. This surveillance shows no signs of slowing (e.g., see [16]).

In this paper, we describe the current state of uncurated package managers, using npm as a running example. Then, we present designs for a community repository and

[1] e.g. npm (npm, Inc.) and Yarn (Facebook)

package manager, SPAM, that limit the ramifications of malicious and negligent users and administrators.

2 Current package managers

This section presents an overview of uncurated package management systems and outlines several security challenges within these systems. We use the Node.js package manager npm as a running example because of its popularity. The security issues we describe are neither unique to npm nor specific to Node.js; most uncurated language package managers have similar drawbacks. To illustrate these common drawbacks, we first describe the workflow of a typical Node.js developer.

Developer Dan wants to reformat the columns of a CSV file. To get hip with the times, he decides to use JavaScript and Node.js, and begins by writing a helper function that inserts spaces on the left-hand side of a line. The function is elegant and useful; to share it with others, he adds testing and benchmarking (pulling in dependencies like the `tape` library) and creates a new package named `lpad`.

Dan decides to make the world a better place by publishing `lpad` on the npm registry. Any developer can install Dan's `lpad` package or depend on it within their own project. Some users may even download `lpad` without knowing it; as they install more popular packages that list `lpad` as a dependency, they also install `lpad` itself. Users may even end up with multiple versions of `lpad` or multiple copies of the same version (due to how npm resolves dependencies) [17, 18]. Eventually, fans of `lpad` start bombarding Dan with requests to update and extend his package, so he uploads it to GitHub and syncs his repository with the Travis continuous integration (CI) cloud service. Now, tests will run anytime someone creates a pull request or Dan pushes directly to the repository—`lpad` will only contain fast, functionally-correct code!

Using uncurated package managers, developers are able to build relatively complex software largely by repurposing existing projects (like Dan's `lpad`); the package manager and registry do not impose restrictions on uploaded code. This lax security, however, comes at a cost.

Malicious registry In most uncurated package management systems, all users that depend on a malicious registry are at its mercy. For example, a registry may serve malicious or vulnerable packages in place of packages the users request. A registry may behave maliciously even if its administrators are virtuous: admins may be compelled by the government, network attackers may tamper with packages via MITM attacks, etc. In fact, in writing this paper, we discovered a vulnerability in npm that allows a network attacker to intercept certain installs [19]; developers may explicitly link to their dependencies with `http` instead of `https` URLs, putting those who download their packages at risk. npm maintainers have not fixed this vulnerability. Instead, they warn developers to manually stick an "`s`" at the end of those links [20].

Malicious developer In addition to trusting the registry, developers must also trust the packages that they install, and those on which their installations depend on. This is deceptively difficult: even when developers rely on a handful of packages, their transitive dependency list is an order of magnitude larger—and that much more difficult to audit. We looked at the top 20 npm packages as of January 24, 2015, and found that the average npm package lists 50 dependencies, with a median of one, a minimum of zero, and a maximum of 626. In practice, this number is sometimes worse; for example, a popular dependency that many bootcamps include in their skeletons is `hackathon-starter` [21]. This project downloads 558 dependencies before bootcampers even write a line of app code. While NPM suggests installing only trusted packages, this also does not scale when popular packages like `lodash` release every thirteen days, on average [7, 22]. Furthermore, packages with many dependencies may also include vulnerable and out of date dependency versions. We analyzed[2] all packages by the top ten authors on npm—presumably trustworthy people—and out of 6,498 projects, we found that 1,693, or 26.1% have out of date dependencies.

Malicious integration services Along with registries and third-party code, developers also place implicit trust in integration tools like Travis [24]. This implicit trust is once again transitive: for example, when Dan shares his API keys with Travis, he—and anyone whose package depends on him—is trusting Travis not to publish any malicious package versions. Furthermore, Dan must properly encrypt his keys when he shares them with Travis, since continuous integrators are allowed to use plaintext API keys. Many developers accidentally publish their API keys publicly: we searched the first 1,000 GitHub results for "provider: npm api_key." and found that 174 of 527 unique GitHub developers uploaded a YAML file with unencrypted API keys.

3 Registry design

In this section, we describe our secure package manager (SPAM), which intends to prevent and contain damage even in the presence of malicious users, registry administrators, and third-party tools like Travis. SPAM consists of a distributed multi-node registry which manages package data and metadata and provides an interface to update and query this data. SPAM also includes a client tool that allows developers to communicate with the registry.

[2]The measurements and analyses described here are based on a clone of the npm registry; we fetched packages between December 6–15, 2016 using `registry-static` [23].

The registry At a high level, the goal of the registry is to maintain a ledger with information about users and packages. The ledger provides the client tools—and thus the developers who use them—with a way of verifying the authenticity of data and metadata about packages and other users. The ledger also serves to provide developers with a single mechanism for storing package metadata. We use a federated Byzantine agreement system (FBAS) to ensure that the ledger remains uncompromised in the presence of malicious actors (e.g., registry administrators or government mandates) [25].

The ledger consists of different messages that record user information (e.g., name registrations and proofs of identity) and package metadata (e.g., package release information). Each entry e_n in the ledger contains the entry number n (a monotonically increasing number), a new client message m, and a hash of the entry number, message and previous ledger entry—i.e., $H(n\|m\|e_{n-1})$. Therefore, each entry in the ledger securely refers to the previous state of the ledger. The message m describes an action (e.g., update package) that the client has requested of the registry and contains that client's signature.

For example, to create a new account, a developer executes the `spam new-user` command. The SPAM client creates a new user public key pair and sends the `register_user` message to the registry; this message contains the user's public key and proposed name, signed by their corresponding private key. If that name is not already taken—if it has not already been recorded in the ledger—the registry will add the `register_user` request to the ledger. This record acknowledges that the new name is now associated with the user's key.

There are ten different messages that the SPAM client can send to the registry after interacting with users on the command line that lead to ledger modifications. Table 1 lists these messages, most of which are self explanatory, with the exception of the `prove_identity` and `extensible` messages—the latter of which we describe later on. The `prove_identity` message allows a user to associate their private key with their public social networks, therefore tying their SPAM identity to an external identity. For example, a developer will associate his SPAM key with his Twitter `@handle` by publishing a signed Tweet. This allows other users to verify the authenticity of the developer's packages; furthermore, it allows other users to install his packages *only if* they trust him and he has not revoked his key.

User and package keys SPAM manages all keys automatically to prevent key overuse and compromise. For example, SPAM creates a new user key when a developer creates a new account. SPAM also generates a new key each time the user creates a new project, and *only* that key may sign project updates—the user key cannot sign them. Users can run other SPAM commands that create additional keys associated with a project, allowing them to release projects from multiple machines or integrate with CI tools like Travis. Compromise of a project key is not equivalent to compromise of a user's account; a malicious, key-stealing Travis can only push updates to a single project. Moreover, users can revoke project keys if they have been compromised.

To revoke a project key, the user runs the `spam proj revoke-key` command, which sends a message, signed with the user key, to the registry. This message indicates which project key should be revoked. At this point, the user must approve or flag any past changes to the compromised project (such as collaborator additions). In practice, this amounts to choosing an entry number to demarcate the point of compromise. The SPAM client displays a list of project changes and asks the user to flag the first suspicious change, if any. All subsequent changes are automatically flagged as suspicious. Finally, the SPAM client sends these flag messages to the registry.

To revoke and replace a user key, the user runs the `spam user replace-key` command, which consists of several steps. First, the cli tool prompts the user to update the majority of their external identity proofs with a new key (which is equivalent to sending a number of `prove_identity` messages, atomically). The cli tool will then notify the registry that the old key is invalid and that the new key is associated with the same user. By updating the ledger with the `replace_user_key` message, the registry asserts that it has verified the new key with the user's external identities (e.g., Twitter). Then, the SPAM client asks the user to demarcate a point of compromise, following similar steps to those in the project-key compromise.

Distributed registry Our registry consists of several (at least four) top-level mirrors, or nodes that use a consensus protocol—specifically, the Stellar Consensus Protocol (SCP)—to agree on ledger entries [25]. Like traditional, centralized Byzantine fault tolerant agreement (e.g., PBFTA [26]), SCP can guarantee safety and liveness by relying on a quorum of nodes to come to an agreement; for example, in an arrangement with four top-level nodes, our system tolerates one failure. Users can configure the SPAM cli tool to talk to any one of the top-level nodes, and users can trust its results as long as they configure it to receive at least two signed replies from distinct nodes.

SCP, in contrast to PBFTA, allows nodes to choose which other nodes to listen to. In combination with other design choices, this allows a node to detect and attribute attacks in which multiple registry mirrors are colluding. This is because SCP does not require a majority of nodes to compose a quorum; instead, nodes choose one of more *quorum slices* [25]. Every node n's quorum slices must be non-empty and contain n itself. For n to agree on ledger entry e, every node (including n) in *one* of n's quorum slices must communicate that they believe e to be true.

Client message	Description
$\langle$register_user : $U, U_{pk}\rangle_{U_{sk}}$	Register a new user U and public key U_{pk} signed with the user's private key U_{sk}.
$\langle$prove_identity : $p, U_{pk}\rangle_{U_{sk}}$	Associate an external identity with user key U_{pk}; the proof of identity p contains URL(s) to signed social network post(s). When a registry node records this message in its ledger, it asserts to have verified the signed post(s).
$\langle\langle$register_package : $P, P_{pk}\rangle_{P_{sk}}\rangle_{U_{sk}}$	Register a new package name P and project keys (P_{pk}, P_{sk}).
$\langle\langle$replace_project_key : $P, P_{pk}\rangle_{P_{sk}}\rangle_{U_{sk}}$	Revoke existing project key and associate the new key pairs (P_{pk}, P_{sk}) with project P.
$\langle$replace_user_key : $U, U_{pk}, p\rangle_{U_{sk}}$	Replace user U's existing key with with key-pair (U_{pk}, U_{sk}). Here, p contains proofs of identity (as above) for a majority of the U's external identities.
$\langle\langle$register_project_key : $P, P'_{pk}\rangle_{P'_{sk}}\rangle_{P_{sk}}$	Add a new collaborator/device key to P by registering their project key P'_{pk}. P_{sk} must correspond to the project key of the owner or another collaborator/device.
$\langle$revoke_project_key : $P, P'_{pk}\rangle_{P_{sk}}$	Revoke a collaborator/device key P'_{pk}. P_{sk} must correspond to the project key of the owner or another collaborator. If P'_{pk} is the owner's project key, P_{sk} must be the corresponding secret key.
$\langle$release_package : $P, v, pkg\rangle_{P_{sk}}$	Release a new package version v for project P. The package data pkg contains the actual package tarball (and other metadata). The registry nodes record a similar message; instead of the tarball, however, they record the URL(s) where the tarball can be downloaded from.
$\langle$flag_package : $P, v, f\rangle_{P_{sk}}$	Flag version v of package P as f (suspicious or approved), if not already flagged.
$\langle$extensible : $data\rangle_{X_{sk}}$	Add application-specific $data$ to the ledger signed by either project or user key X_{sk}.

Table 1: The different messages supported by our registry. We omit the messages that clients use to retrieve data from the ledger.

Now, in order for e to be appended to the ledger, every other node must be convinced by one of its own quorum slices. Reaching consensus is a multi-phase process, the details of which we omit; we refer the reader to the SCP paper and Stellar core for details [25, 27].

In our design, the top-level mirror nodes may employ other second-level nodes as members of other slices. Top-level mirror nodes maintain both the ledger and a mirror of the registry contents; they track package data and metadata. Second-level nodes, in contrast, do *not* store package data. Rather, they simply maintain a copy of the ledger. Second-level nodes can arbitrarily join the network (by considering at least two top level nodes to be in their quorum slice) to add additional oversight.

This additional oversight is not provided by second-level nodes referring only to the ledger. Rather, all nodes also keep track of a "rollback safety" data structure that contains the previous round of signed (SCP-level) messages from nodes in its quorum slices. Since these messages contain a hash of the ledger entry histories, a signed message from a node indicates that this node agrees to the entire history of the ledger. If top-level nodes collude to rollback the ledger history, second-level nodes will be able to detect this; they will be able to prove collusion by producing two signed, contradicting messages from a culprit node. One of these messages comes from the rollback safety buffer and one is the most recent message with a backdated entry number.

The registry threat model In the absence of second-level nodes, our system can only provide strong guarantees if the majority of top-level nodes are well-behaved. Under this assumption, we get availability directly from SCP's liveness guarantees [25]. SCP also ensures the safety of the ledger, which itself contains signed messages; together, this ensures the integrity and authenticity of packages. Every time a developer downloads a package, the SPAM cli tool verifies that package's signature. Moreover, the SPAM cli tool verifies that that package has not been flagged (e.g., due to key compromise).

During the time when key is compromised but not yet revoked, our system cannot protect the developers that download malicious packages signed under the compromised key. However, SPAM's separation of project and user keys and tie in with external services—e.g., using Twitter for identity verification—make it possible to recover from key compromises without having to create a new identify. Even if an attacker steals a user key, they cannot prevent the user from revoking the key unless they compromise a majority of that user's external identities (GitHub, Twitter, StackOverflow, etc.).

In general, our system cannot provide guarantees against users running malicious code. However, we can attribute malicious packages back to their authors—and, to a certain extent, to those authors' identities. Furthermore, users can configure the SPAM cli tool to only install packages authored by developers that the user explicitly trusts—a restriction that may be imposed on sub-dependencies as well. Though this simple white-list policy is easy to ship in an early release of SPAM, we envision eventually extending the cli tool with more interesting policies. For example, we can take advantage of the fact that SPAM users are already connected to social graphs (e.g., Twitter) to compute a level of trustworthiness. Even if Eve creates fake

Twitter and GitHub identities, she will have to cultivate real friendships in order to appear legitimate [3].

Without second-level nodes, a quorum of top-level mirror nodes can collude to carry out a number of attacks. For example, they can revoke and replace a user's key by appending a `replace_user_key` message to the ledger, signing it with a new, fake user-key. This is possible even without compromising the user's Twitter or GitHub accounts, because the onus of verifying identity is on the registry itself. Colluding nodes can also roll back ledger history to, for example, conceal backdoors.

With second-level nodes, on the other hand, our system is able to detect rollback attacks and attribute them to malicious nodes. As long as a first- or second-level node is not colluding, this node can prove that other nodes have colluded. Using the rollback safety mechanism, the honest node identifies any contradictions in any lying node's version of history. Even if all nodes except for one are colluding, honest, second-tier nodes will still see externalized messages which can be used to prove that the colluding first-level nodes in their quorum slices have said contradicting things. This alone, however, does not prevent key revocation attacks.

We can extended our system via the `extensible` message to both detect key revocation attacks and attacks wherein colluding nodes upload backdoored packages only to remove them from their registries shortly after. If third-party services (e.g., the Internet Archive, ACLU, or EFF) are willing to commit to hosting signed packages and copies of signed social network messages, they can use the `extensible` message to state their purpose in the ledger. The ACLU, for example, might say that it is willing to host this data for a certain period of time. Then, every time a package is released or an identity is proven, the ACLU will store the package or verification data on its servers and use the `extensible` message to state that it has done so. Such services enable second-level nodes to verify that the ledger's claims about key revocations and the contents of packages are replicated on the ACLU service. If registry nodes collude to revoke someone's key, the ACLU will not be able to replicate any social network data, since this data does not exist. If the SPAM cli tool does not see a such an identity proof nor a (verifiable) confirmation of revocation from ACLU, it can be configured to treat the new key as compromised: it will refuse to trust anything that the key has signed. A similar process can be used to ensure that registry nodes are not colluding to conceal backdoored packages.

Limitations Our system design, naturally, comes with limitations. For example, our design does not prevent front-running attacks. These attacks occur when, for example, a developer tries to register a user name but is communicating with a malicious registry node. The malicious node will receive the user request but instead request the user's chosen name on behalf of an attacker. Once the attacker has successfully registered the name, the malicious node will handle the developer's request as usual—ensuring its denial. This problem is not an artifact of our design [28].

Our design also explicitly does not allow users to transfer user or project names to others (without essentially giving up on their social network identities as well). Though we believe that such extensions are possible, providing such "features" in the presence of nation-state adversaries and key continuity—i.e., the ability to easily replace compromised keys—remains an open problem.

[3] Here again we can leverage the idea of quorum intersection from SCP to ensure that even if Eve creates thousands of followers, if none intersect with other peoples' graphs, she will still appear suspicious.

4 Related work

Cappos et al. [29] identified the dangers of compromised or malicious package managers and registries. The Update Framework (TUF) [30] and the Diplomat [6] package managers—related work closest to ours—follow up on [29], borrowing ideas from Tor's update framework Thandy [31]. TUF guarantees package integrity by (1) using multiple keys to sign any package data and (2) assigning roles to keys so that one key cannot be used to sign off on different kind of data (e.g., one project's key cannot be used to sign another's releases). Diplomat further extends TUF with a key hierarchy, maintained by the registry, to address an important concern—how to retrofit existing package manager design with security. We adopt the idea that user keys and project keys should be separated from these frameworks. Unlike these systems, however, we do not assume registry administrators to be trusted—our attacker model is considerably stronger. This naturally comes at the cost of not being easy to retrofit on top of existing, insecure systems.

Keybase [32] pioneered the idea of tying public keys with online identities. Our system uses this same idea to provide key continuity. Though we do not rely the PGP web of trust that Keybase builds on—our design is considerably simpler [33]—we believe that a similar idea of following or endorsing users on SPAM can potentially prove useful in assigning a level of (dis)trust to downloaded packages. For example, if a developer installs a package that is signed by a user that is not followed by anybody that the developer follows/endorses, the SPAM cli can warn before continuing with the install—potentially preventing typosquatting attacks.

References

[1] NPM, "Newly paranoid maintainers." `http://blog.npmjs.org/post/80277229932/newly-paranoid-maintainers`, March 2014.

[2] Evan, "Data verification." http://blog.rubygems.org/2013/01/31/data-verification.html, January 2013.

[3] postmodern, "Add safe_load #119." https://github.com/ruby/psych/issues/119#issuecomment-12875715, January 2013.

[4] Python Software Foundation, "PyPI - the Python package index." https://pypi.python.org/pypi, January 2017.

[5] NPM, "npm registry is now fully HTTPS!." http://blog.npmjs.org/post/142077474335/npm-registry-is-now-fully-https, April 2016.

[6] T. K. Kuppusamy, S. Torres-Arias, V. Diaz, and J. Cappos, "Diplomat: Using delegations to protect community repositories," in *NSDI*, March 2016.

[7] NPM, "Package install scripts vulnerability." http://blog.npmjs.org/post/141702881055/package-install-scripts-vulnerability, March 2016.

[8] S. Evans, "Data-Stealing Malicious Apps Found in Google Play Store." https://www.infosecurity-magazine.com/news/malicious-apps-found-in-google, September 2016.

[9] D. Goodin, "Godless apps, some found in Google Play, can root 90Android phones." http://arstechnica.com/security/2016/06/godless-apps-some-found-in-google-play-root-90-of-android-phones/, June 2016.

[10] A. Menczer and A. Lysunets, "DressCode Android Malware Discovered on Google Play." http://blog.checkpoint.com/2016/08/31/dresscode-android-malware-discovered-on-google-play, August 2016.

[11] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets.," in *NDSS*, February 2012.

[12] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson, "Hulk: Eliciting malicious behavior in browser extensions," in *USENIX Security*, August 2014.

[13] L. Constantin, "Spammers buy Chrome extensions and turn them into adware." http://www.pcworld.com/article/2089580/spammers-buy-chrome-extensions-and-turn-them-into-adware.html, January 2014.

[14] NPM, "kik, left-pad, and npm." http://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm, March 2016.

[15] M. Sullivan, "Yahoo helps the government read your emails. just following orders, they say.." https://www.washingtonpost.com/lifestyle/style/yahoo-helps-the-government-read-your-emails-just-following-orders-they-say/2016/10/05/05648894-8b01-11e6-875e-2c1bfe943b66_story.html, October 2016.

[16] D. J. Trump. https://twitter.com/realdonaldtrump/status/392990408843984897, October 2013.

[17] npm, "npm v3 dependency resolution." https://docs.npmjs.com/how-npm-works/npm3, December 2015.

[18] npm, "npm v2 dependency resolution." https://docs.npmjs.com/how-npm-works/npm2, December 2015.

[19] D. Stefan, "npm shrinkwrap allows remote code execution." https://hackernoon.com/npm-shrinkwrap-allows-remote-code-execution-63e6e0a566a7#.e7an55fo2, December 2016.

[20] Seldo, "Avoid http urls in shrinkwrap files." http://blog.npmjs.org/post/154400916805/avoid-http-urls-in-shrinkwrap-files, December 2016.

[21] S. Yalkabov, "Hackathon starter - a kickstarter for nodejs web applications." https://github.com/sahat/hackathon-starter, January 2017.

[22] C.-A. Staicu, M. Pradel, and B. Livshits, "Understanding and automatically preventing injection attacks on node. js," Tech. Rep. TUD-CS-2016-14663, TU Darmstadt, Department of Computer Science, November 2016.

[23] davglass, "registry-static." https://www.npmjs.com/package/registry-static, January 2017.

[24] Travis CI, "Teamwork makes Travis CI possible." https://travis-ci.com/about, January 2017.

[25] D. Mazieres, "The stellar consensus protocol: A federated model for internet-level consensus." https://www.stellar.org/papers/stellar-consensus-protocol.pdf, February 2016.

[26] M. Castro, B. Liskov, *et al.*, "Practical byzantine fault tolerance," in *OSDI*, February 1999.

[27] stellar, "stellar-core." https://github.com/stellar/stellar-core, January 2017.

[28] I. Security and S. A. Committee, "Advisory on domain name front running." https://www.icann.org/en/system/files/files/sac-022-en.pdf, October 2007.

[29] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman, "A look in the mirror: Attacks on package managers," in *CCS*, ACM, 2008.

[30] J. Samuel, N. Mathewson, J. Cappos, and R. Dingledine, "Survivable key compromise in software update systems," in *CCS*, ACM, October 2010.

[31] N. Mathewson, "Thandy: Secure update for tor." https://opensource.googleblog.com/2009/03/thandy-secure-update-for-tor.html, March 2009.

[32] Keybase, "What keybase is really doing." https://keybase.io/docs/server_security, January 2017.

[33] T. Unangst, "signify: Securing OpenBSD from us to you," in *BSDCan*, 2015.