
# in-toto: Providing farm-to-table guarantees for bits and bytes

Santiago Torres-Arias, *New York University;* Hammad Afzali, *New Jersey Institute of Technology;* Trishank Karthik Kuppusamy, *Datadog;* Reza Curtmola, *New Jersey Institute of Technology;* Justin Cappos, *New York University*

https://www.usenix.org/conference/usenixsecurity19/presentation/torres-arias

**This paper is included in the Proceedings of the 28th USENIX Security Symposium.**

**August 14–16, 2019 • Santa Clara, CA, USA**

978-1-939133-06-9

**Open access to the Proceedings of the 28th USENIX Security Symposium is sponsored by USENIX.**

# in-toto: Providing farm-to-table guarantees for bits and bytes

Santiago Torres-Arias[†], santiago@nyu.edu
Hammad Afzali[‡], ha285@njit.edu
Trishank Karthik Kuppusamy[*], trishank@datadog.com
Reza Curtmola[‡], crix@njit.edu
Justin Cappos[†], jcappos@nyu.edu

[†]New York University, Tandon School of Engineering
[*]Datadog
[‡]Department of Computer Science, New Jersey Institute of Technology

## Abstract

The software development process is quite complex and involves a number of independent actors. Developers check source code into a version control system, the code is compiled into software at a build farm, and CI/CD systems run multiple tests to ensure the software's quality among a myriad of other operations. Finally, the software is packaged for distribution into a delivered product, to be consumed by end users. An attacker that is able to compromise any single step in the process can maliciously modify the software and harm any of the software's users.

To address these issues, we designed `in-toto`, a framework that cryptographically ensures the integrity of the software supply chain. `in-toto` grants the end user the ability to verify the software's supply chain from the project's inception to its deployment. We demonstrate `in-toto`'s effectiveness on 30 software supply chain compromises that affected hundreds of million of users and showcase `in-toto`'s usage over cloud-native, hybrid-cloud and cloud-agnostic applications. `in-toto` is integrated into products and open source projects that are used by millions of people daily. The project website is available at: `https://in-toto.io`.

## 1 Introduction

Modern software is built through a complex series of steps called a *software supply chain*. These steps are performed as the software is written, tested, built, packaged, localized, obfuscated, optimized, and distributed. In a typical software supply chain, these steps are "chained" together to transform (e.g., compilation) or verify the state (e.g., the code quality) of the project in order to drive it into a *delivered product*, i.e., the finished software that will be installed on a device. Usually, the software supply chain starts with the inclusion of code and other assets (icons, documentation, etc.) in a version control system. The software supply chain ends with the creation, testing and distribution of a delivered product.

Securing the supply chain is crucial to the overall security of a software product. An attacker who is able to control any step in this chain may be able to modify its output for malicious reasons that can range from introducing backdoors in the source code to including vulnerable libraries in the delivered product. Hence, attacks on the software supply chain are an impactful mechanism for an attacker to affect many users at once. Moreover, attacks against steps of the software supply chain are difficult to identify, as they misuse processes that are normally trusted.

Unfortunately, such attacks are common occurrences, have high impact, and have experienced a spike in recent years [60, 129]. Attackers have been able to infiltrate version control systems, including getting commit access to the Linux kernel [58] and Gentoo Linux [76], stealing Google's search engine code [22], and putting a backdoor in Juniper routers [48, 96]. Popular build systems, such as Fedora, have been breached when attackers were able to sign backdoored versions of security packages on two different occasions [75, 123]. In another prominent example, attackers infiltrated the build environment of the free computer-cleanup tool CCleaner, and inserted a backdoor into a build that was downloaded over 2 million times [126]. Furthermore, attackers have used software updaters to launch attacks, with Microsoft [108], Adobe [95], Google [50, 74, 140], and Linux distributions [46, 143] all showing significant vulnerabilities. Perhaps most troubling are several attacks in which nation states have used software supply chain compromises to target their own citizens and political enemies [35,55,82,92,93,108, 127,128,138]. There are dozens of other publicly disclosed instances of such attacks [8,33,38,39,41,52,53,65,70,76,79,80, 83,95,107,113,115,118,119,122,130–132,134,139,141,146].

Currently, supply chain security strategies are limited to securing each individual step within it. For example, Git commit signing controls which developers can modify a repository [78], reproducible builds enables multiple parties to build software from source and verify they received the same result [25], and there are a myriad of security systems that protect software delivery [2, 20, 28, 100, 102]. These building blocks help to secure an individual step in the process.

Although the security of each individual step is critical, such efforts can be undone if attackers can modify the output of a step before it is fed to the next one in the chain [22, 47]. These piecemeal measures by themselves can not stop malicious actors because there is no mechanism to verify that: 1) the correct steps were followed and 2) that tampering did not occur in between steps. For example a web server compromise was enough to allow hackers to redirect user downloads to a modified Linux Mint disk image, even though every single package in the image was signed and the image checksums on the site did not match. Though this was a trivial compromise, it allowed attackers to build a hundred-host botnet in a couple of hours [146] due to the lack of verification on the tampered image.

In this paper we introduce `in-toto`, Latin for "as a whole," the first framework that holistically enforces the integrity of a software supply chain by gathering cryptographically verifiable information about the chain itself. To achieve this, `in-toto` requires a project owner to declare and sign a

*layout* of how the supply chain's steps need to be carried out, and by whom. When these steps are performed, the involved parties will record their actions and create a cryptographically signed statement — called *link metadata* — for the step they performed. The link metadata recorded from each step can be verified to ensure that all steps were carried out appropriately and by the correct party in the manner specified by the layout.

The layout and collection of link metadata tightly connect the inputs and outputs of the steps in such a chain, which ensures that tampering can not occur between steps. The layout file also defines requirements (e.g., Twistlock [30] must not indicate that any included libraries have high severity CVEs) that will be enforced to ensure the quality of the end product. These additions can take the form of either distinct commands that must be executed, or limitations on which files can be altered during that step (e.g., a step that localizes the software's documentation for Mexican Spanish must not alter the source code). Collectively, these requirements can minimize the impact of a malicious actor, drastically limiting the scope and range of actions such an attacker can perform, even if steps in the chain are compromised.

We have built a series of production-ready implementations of `in-toto` that have now been integrated across several vendors. This includes integration into cloud vendors such as Datadog and Control Plane, to protect more than 8,000 cloud deployments. Outside of the cloud, `in-toto` is used in Debian to verify packages were not tampered with as part of the reproducible builds project [25]. These deployments have helped us to refine and validate the flexibility and effectiveness of `in-toto`.

Finally, as shown by our security analysis of three `in-toto` deployments, `in-toto` is not a "lose-one, lose-all" solution, in that its security properties only partially degrade with a key compromise. Depending on which key the attacker has accessed, `in-toto`'s security properties will vary. Our `in-toto` deployments could be used to address most (between 83% - 100%) historical supply chain attacks.

## 2 Definitions and Threat Model

This section defines the terms we use to discuss the software supply chain and details the specific threat model `in-toto` was designed to defend against.

### 2.1 Definitions

The software supply chain refers to the series of *steps* performed in order to create and distribute a *delivered product*. A *step* is an operation within this chain that takes in *materials* (e.g., source code, icons, documentation, binaries, etc.) and and creates one or more *products* (e.g., libraries, software packages, file system images, installers, etc.). We refer to both materials and products generically as *artifacts*.

It is common to have the products of one step be used as materials in another step, but this does not mean that a supply chain is a sequential series of operations in practice. Depending on the specifics of a supply chain's workflow, steps may be executed in sequence, in parallel, or as a combination of both. Furthermore, steps may be carried out by any number of hosts, and many hosts can perform the same step (e.g., to test a step's reproducibility).

In addition to the materials and products, a step in the supply chain produces another key piece of information, *byproducts*. The step's byproducts are things like the `STDOUT`, `STDERR`, and return value that indicate whether a step was successful or had any problems. For example, a step that runs unit tests may return a non-zero code if one of the unit tests fails. Validating byproducts is key to ensuring that steps of the supply chain indicate that the software is ready to use.

As each step executes, information called *link* metadata that describes what occured, is generated. This contains the materials, products, and byproducts for the step. This information is signed by a key used by the party who performs the action, which we call a *functionary*. Regardless of whether the functionary commits code, builds software, performs QA, localizes documentation, etc., the same link metadata structure is followed. Sometimes a functionary's participation involves repeated human action, such as a developer making a signed git commit for their latest code changes. In other cases, a functionary may participate in the supply chain in a nearly autonomous manner after setup, such as a CI/CD system. Further, many functionaries can be tasked to perform the same step for the sake of redundancy and a minimum threshold of them may be required to agree on the result of a step they all carried out.

To tie all of the pieces together, the *project owner* sets up the rules for the steps that should be performed in a software supply chain. In essence, the project owner serves as the foundation of trust, stating which steps should be performed by which functionaries, along with specifying rules for products, byproducts, and materials in a file called the *layout*. The layout enables a *client* that retrieves the software to cryptographically validate that all actions were performed correctly. In order to make this validation possible, a client is given the *delivered product*, which contains the software, layout, and link metadata. The layout also contains any additional actions besides the standard verification of the artifact rules to be performed by the client. These actions, called *inspections*, are used to validate software by further performing operations on the artifacts inside the delivered product (e.g., verifying no extraneous files are inside a zip file). This way, through standard verification and inspections, a client can assure that the software went through the appropriate software supply chain processes.

### 2.2 Threat Model

The goal of `in-toto` is to minimize the impact of a party that attempts to tamper with the software supply chain. More specifically, the goal is to retain the maximum amount of security that is practical, in any of the following scenarios:

- Interpose between two existing elements of the supply chain to change the input of a step. For example, an attacker may ask a hardware security module to sign a malicious copy of a package before it is added to the repository and signed repository metadata is created to index it [27, 44, 51, 76, 107, 120, 120, 147].

- Act as a step (e.g., compilation), perhaps by compromising or coercing the party that usually performs that step [27, 57, 62, 64, 76, 81, 99, 112, 125]. For example, a hacked compiler could insert malicious code into binaries it produces [126, 136].
- Provide a delivered product for which not all steps have been performed. Note that this can also be a result of an honest mistake [37, 49, 56, 68, 73, 97, 142].
- Include outdated or vulnerable elements in the supply chain [59, 61, 91, 94, 117]. For example, an attacker could bundle an outdated compression library that has many known exploits.
- Provide a counterfeit version of the delivered product to users [8, 35, 66, 70, 71, 95, 118, 134, 135, 146]. This software product can come from any source and be signed by any keys. While `in-toto` will not mandate how trust is bootstrapped, Section 6 will show how other protocols such as TUF [28], as well as popular package managers [2] can be used to bootstrap project owner keys.

**Key Compromise.** We assume that the public keys of project owners are known to the verifiers and that the attacker is not able to compromise the corresponding secret key. In addition, private keys of developers, CI systems and other infrastructure public keys are known to a project owner and their corresponding secret keys are not known to the attacker. In section 5.2, we explore additional threat models that result from different degrees of attacker access to the supply chain, including access to infrastructure and keys (both online and offline).

### 2.3 Security Goals

To build a secure software supply chain that can combat the aforementioned threats, we envision that the following security goals would need to be achieved:

- **supply chain layout integrity:** All of the steps defined in a supply chain are performed in the specified order. This means that no steps can be added or removed, and no steps can be reordered.
- **artifact flow integrity:** All of the artifacts created, transformed, and used by steps must not be altered in-between steps. This means that if step A creates a file `foo.txt` and step B uses it as a material, step B must use the exact file `foo.txt` created by step A. It must not use, for example, an earlier version of the file created in a prior run.
- **step authentication:** Steps can only be performed by the intended parties. No party can perform a step unless it is given explicit permission to do so. Further, no delivered products can be released unless all steps have been performed by the right party (e.g., no releases can be made without a signoff by a release engineer, which would stop accidental development releases [68]).
- **implementation transparency:** `in-toto` should not require existing supply chains to change their practices in order to secure them. However, `in-toto` can be used to represent the existing supply chain configuration and reason about its security practices.
- **graceful degradation of security properties:** `in-toto` should not lose all security properties in the event of key compromise. That is, even if certain supply chain steps are compromised, the security of the system is not completely undermined.

In addition to these security goals, `in-toto` is also geared towards practicality and, as such, it should maintain minimal operational, storage and network overheads.

## 3 System overview

The current landscape of software supply chain security is focused on point-solutions that ensure that an individual step's actions have not been tampered with. This limitation usually leads to attackers compromising a weaker step in the chain (e.g., breaking into a buildfarm [115]), removing steps from the chain [68] or tampering with artifacts while in transit (i.e., adding steps to the chain [66]). As such, we identify two fundamental limitations of current approaches to secure the software supply chain:

1. Point solutions designed to secure individual supply chain steps cannot guarantee the security of the entire chain as a whole.
2. Despite the widespread use of unit testing tools and analysis tools, like fuzzers and static analyzers, software rarely (if ever) includes information about what tools were run or their results. So point solutions, even if used, provide limited protection because information about these tools is not appropriately utilized or even shown to clients who can make decisions about the state of the product they are about to utilize.

We designed `in-toto` to address these limitations by ensuring that all individual measures are applied, and by the right party in a cryptographically verifiable fashion.

In concrete terms, `in-toto` is a framework to gather and verify metadata about different stages of the supply chain, from the first step (e.g., checking-in code on a version control system) to delivered product (e.g., a .deb installable package). If used within a software supply chain, `in-toto` ensures that the aforementioned security goals are achieved.

### 3.1 `in-toto` parties and their roles

Similar to other modern security systems [101, 102, 121], `in-toto` uses security concepts like delegations and roles to limit the scope of key compromise and provide a graceful degradation of its security properties.

In the context of `in-toto`, a role is a set of duties and actions that an actor must perform. The use of delegations and roles not only provides an important security function (limiting the impact of compromise and providing separation of privilege), but it also helps the system remain flexible and usable so that behaviors like key sharing are not needed. Given that every project uses a very specific set of tools and practices, flexibility is a necessary requirement for `in-toto`. There are three roles in the framework:

- Project Owner: The project owner is the party in charge of defining the software supply chain layout (i.e., define

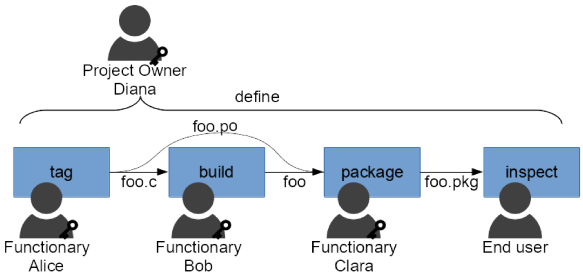


Figure 1: Graphical depiction of the software supply chain with `in-toto` elements added. The project owner creates a layout with three steps, each of which will be performed by a functionary. Notice how the tag step creates `foo.c` and a localization file `foo.po`, which are fed to different steps down the chain.

which steps must be performed and by who). In practice, this would be the maintainer of an open-source project or the dev-ops engineers of a project.

- Functionaries: Functionaries are the parties that perform the steps within the supply chain, and provide an authenticated record of the artifacts used as materials and the resulting products. Functionaries can be humans carrying out a step (e.g., signing off a security audit) or an automated system (e.g., a build farm).
- Client: (e.g., end user): The client is the party that will inspect and afterwards utilize a delivered product.

We will now elaborate on how these three parties interact with the components of `in-toto`.

### 3.2 `in-toto` components

`in-toto` secures the software supply chain by using three different types of information: the software supply chain layout (or layout, for short), link metadata, and the delivered product. Each of these has a unique function within `in-toto`.

#### 3.2.1 The supply chain layout

Laying out the structure of the supply chain allows the developers and maintainers of a project to define requirements for steps involved in source code writing, testing, and distribution within a software product's lifecycle. In the abstract sense, this supply chain layout is a recipe that identifies which steps will be performed, by whom, and in what order.

The supply chain layout defines a series of *steps* in the supply chain. These definitions are used to enforce measures on what artifacts should be used as *materials*. To ensure that only the intended parties execute the right steps, a public key is associated with each step. In order to ensure that the layout was created by the project owner, it is cryptographically signed with the project owner's private key.

The project owner will define this supply chain layout by setting different requirements for the project's steps. These requirements take the form of types of artifacts that can be produced (e.g., a localization step can only produce `.po` files), the expected return values, the type of host that can carry out this step and so forth. When consuming the delivered product, the client (end user) verifies that these requirements are satisfied.

In addition to defining supply chain steps, the layout will also specify a series of *inspection steps* (or inspections). These inspections will be performed by the verifier on the delivered product to draw further insight about its correctness. This is useful for complex supply chains in which the basic semantics of `in-toto` cannot describe their specific requirements. For example, an inspection step can be used to namespace restrict certain VCS-specific operations to specific functionaries such as making sure that only a QA team member merges code into the develop branch and that all commits are signed.

For example, as seen in Figure 1, a project owner can define a supply chain consisting of three steps: a tag, a build and a package step. With these definitions, the project owner also defines how the artifacts will flow through the supply chain (e.g., foo.c is used by build, yet foo.po is packaged directly from tag). Afterwards, the project owner can assign functionaries to carry out each of these steps and define an inspection so the end user can verify that foo was indeed created during build and that foo.po came from the tagged release.

**Layout creation tool.** We provide a web-based layout creation tool [12] to help project owners create `in-toto` layouts. The tool uses an intuitive, graphical interface to define: (1) the steps of the software supply chain (i.e., how is the source code managed? how is the software's quality verified? how is the software built? how is the software packaged?), (2) the actors (functionaries) who are allowed to perform different steps of the software supply chain. An `in-toto` layout is generated based on this information. In addition, the `in-toto` website [13, 15] provides several examples of layouts, which can serve as starting templates for project owners seeking to integrate `in-toto`.

#### 3.2.2 Link metadata

Verifying the actions carried out in the supply chain, requires information about all steps performed in creating the delivered product. Like a chain in real life, an `in-toto` supply chain consists of conjoined *links*, with each link serving as a statement that a given step was carried out.

Functionaries in charge of executing a step within the supply chain must share information about these links. Sharing such information as what *materials* were fed to the step, and what *product(s)* were created, can ensure no artifacts are altered in transit. To ensure that only the right functionaries performed this step, the piece of link metadata must be signed with the private key that corresponds to this functionary's key (as defined in the supply chain layout).

There is a one-to-one relationship between the step definitions in the supply chain layout and the link metadata. That is, each piece of link metadata gathered during each step within the supply chain must match what the requirements prescribe for that step. In order to ensure that the link metadata is generated by the intended entity, it must be cryptographically signed with one (or more, if there is a threshold higher than one defined) of the keys indicated in the requirements for that link.

When all the link metadata has been collected, and the supply chain has been properly defined, the supply chain layout and all the links can be shipped, along with the delivered product, to the end user for verification. We show

a minimal software supply chain, along with a graphical representation of an in-toto layout in Figure 1.

#### 3.2.3 The delivered product

The delivered product is the piece of software that the end user wants to install. In order to verify the delivered product, the end user (or client) will utilize the supply chain layout and its corresponding pieces of link metadata. The end user will use the link metadata to verify that the software provided has not been tampered with, and that all the steps were performed as the project owner intended. In Figure 1 the delivered product consists of the foo.pkg file.

### 3.3 in-toto usage lifecycle

The in-toto usage lifecycle encompasses the following overarching operations:

1. The project owner defines a supply-chain layout.
2. Each step is carried out as specified, and functionaries gather and sign link metadata.
3. A delivered product is shipped to the client, who verifies it upon installation by:
   - ensuring the layout provided was signed by the project owner and is not expired.
   - checking that all the steps defined have enough pieces of link metadata; that such links were signed by the indicated functionaries; and that all artifacts recorded flowed properly between the steps as indicated in the layout.
   - carrying out any inspection steps contained in the layout and making sure that all artifacts recorded match the flow described in the layout.

As seen in Figure 1 a project owner creates the layout to describe an overarching structure of the supply chain that the client can use to verify. Later, functionaries carry out their operations as usual, and submit link metadata to attest for the result of their operation. Finally, a client uses a delivered product, metadata links and a layout to verify the integrity of the delivered product and of the entire chain.

By following the chain of attestations in the link metadata, the client can reconstruct the operations described in Figure 1. Afterwards, the client can verify these attestations against the layout and execute any inspections to make sure everything is in order before consuming the delivered product.

## 4 in-toto internals

In order to avoid tampered, incomplete or counterfeit software, in-toto ensures the integrity and accuracy of all software supply chain operations. in-toto ensures supply chain integrity by the verifying the collected link metadata against a software supply chain layout file. This ensures that all operations were carried out, by the intended party and as the legitimate project owner intended.

Understanding how the system's metadata helps to ensure the integrity of the supply chain is critical to a deeper appreciation of how in-toto works. In this section, we will explore the specifics of the link metadata and the layout file to understand how in-toto operates.

For the context of this section, we will demonstrate the different features of in-toto using Figure 1 as an example. The project owner Diana will create a layout that describes three steps and three functionaries for each step. The first step, tag, will produce a file foo.c to be input into the build step, as well as a foo.po localization file. The second step, build, will use the foo.c file from the tag step and produce a foo binary. Finally, the package step will take the foo.po and foo files and produce a package installable by the end user.

For a more complete and thorough description of all the fields, signature schemes, implementations, a layout editing tool and more, refer to the resources on the project website: https://in-toto.io.

### 4.1 The supply chain layout

The supply chain layout explicitly defines the expected layout of the software supply chain. This way, end users can ensure that its integrity is not violated upon verification. To do this, the layout contains the following fields:

```
{ "_type" : "layout",
  "expires" : "<EXPIRES>",
  "readme": "<README>",
  "keys" : {"<KEYID>":"<PUBKEY_OBJECT>"...},
  "steps" : ["<STEP>", "..." ],
  "inspections" : ["<INSPECTION>","..." ]
}
```

Listing 1: The supply chain layout structure

The overarching architecture of the layout definition includes the following relevant fields:

- An expiration date: this will ensure that the supply chain information is still fresh, and that old delivered products can not be replayed to users.
- A readme field: this is intended to provide a human-readable description of the supply chain.
- A list of public keys: these keys belong to each functionary in the supply chain and will be assigned to different steps to ensure that only the right functionary performs a particular step in the supply chain.
- A list of steps: these are the steps to be performed in the supply chain and by who. Step definitions, described in depth in Section 4.1.1, will contain a series of requirements that limit the types of changes that can be done in the pipeline and what functionary can sign link metadata to attest for its existence.
- A list of inspections: these are the inspections to be performed in the supply chain. As described in depth in section 4.1.2, inspections are verification steps to be performed on the delivered product by the client to further probe into its completeness and accuracy.

Though its structure is quite simple, the layout actually provides a detailed description of the supply chain topology. It characterizes each of the steps, and defines any possible requirements for every step. Likewise, it contains instructions for local inspection routines (e.g., verify that every file in a tar archive was created by the right party in the supply chain), which further ensure the delivered product has not been

tampered with. As such the layout allows the project owner to construct the necessary framework for a secure supply chain.

For our example supply chain, Diana would have to list the public keys as described on Listing 2, as well as all the steps.

```
{ "_type" : "layout",
  "expires" : "<EXPIRES>",
  "readme": "foo.pkg supply chain",
  "keys" : {"<BOBS_KEYID>":"<PUBKEY>",
            "<ALICES_KEYID":"<PUBKEY>",
            "<CLARAS_KEYID":"<PUBKEY>"},
  "steps" : [{"name": "tag", "..."},
             {"name": "build", "..."},
             {"name": "package", "..."} ],
  "inspections" : ["{"name": "inspect", "..."}]
}
```

Listing 2: The supply chain for our example

As described, the layout file already limits all actions to trusted parties (by means of their public keys), defines the steps that are carried out (to limit the scope of any step) and specifies verification routines that are used to dive into the specifics of a particular supply chain. We will describe the latter two fields in depth now.

#### 4.1.1 Step definition

```
{ "_name": "<NAME>",
  "threshold": "<THRESHOLD>",
  "expected_materials": [["<ARTIFACT_RULE>"], "..."],
  "expected_products": [["<ARTIFACT_RULE>"], "..."],
  "pubkeys": ["<KEYID>", "..." ],
  "expected_command": "<COMMAND>"
}
```

Listing 3: A supply chain step in the supply chain layout

Every step of the supply chain contains the following fields:

- **name**: A unique identifier that describes a step. This identifier will be used to match this definition with the corresponding pieces of link metadata.
- **expected_materials**: The materials expected as input `ARTIFACT_RULES` as described in Section 4.1.3. It serves as a master reference for all the artifacts used in a step.
- **expected_products**: Given the step's output information, or *evidence*, what should be expected from it? The expected products also contains a list of `ARTIFACT_RULES` as described in section 4.1.3.
- **expected_command**: The command to execute and any flags that may be passed to it.
- **threshold**: The minimum number of pieces of signed link metadata that must be provided to verify this step. This field is intended for steps that require a higher degree of trust, so multiple functionaries must perform the operation and report the same results. For example, if the threshold is set to $k$, then at least $k$ pieces of signed link metadata need to be present during verification.
- **a list of public keys id's**: The id's of the keys that can be used to sign the link metadata for this step.

The fields within this definition list will indicate requirements for the step identified with that name. To verify these requirements, these fields will be matched against the link metadata associated with the step. The `expected_materials` and `expected_products` fields will be used to compare against the materials and products reported in the link metadata. This ensures that no disallowed artifacts are included, that no required artifacts are missing, and the artifacts used are from allowed steps who created them as products. Listing 4 contains the step definition for the build step for our example Layout above.

```
{ "_name": "build",
  "threshold": "1",
  "expected_materials": [
    ["MATCH", "foo.c", "WITH",
     "PRODUCTS", "FROM", "tag"]
  ],
  "expected_products": [["CREATE", "foo"]],
  "pubkeys": ["<BOBS_PUBKEY>"],
  "expected_command": "gcc foo.c -o foo"
}
```

Listing 4: The build step in our example layout

#### 4.1.2 Inspection definition

Inspection definitions are nearly identical to step definitions. However, since an inspection causes the verifier on the client device to run a command (which can also generate artifacts), there cannot be a threshold of actions. The other fields are identical to the link metadata generated by a step.

#### 4.1.3 Artifact rules

Artifact rules are central to describing the topology of the supply chain by means of its artifacts. These rules behave like firewall rules and describe whether an artifact should be consumed down the chain, or if an artifact can be created or modified at a specific step. As such, they serve two primary roles: to limit the types of artifacts that a step can create and consume; and to describe the flow of artifacts between steps.

For the former, a series of rules describes the operation within the step. A rule, such as `CREATE`, indicates that a material must not exist before the step is carried out and must be reported as a product. Other rules, such as `MODIFY`, `DELETE`, `ALLOW` and `DISALLOW` are used to further limit what a step can register as artifacts within the supply chain. These rules are described in Grammar 5 (full definition in Appendix A). An example of a simple `CREATE` rule can be seen on the step definition in Listing 4.

```
[CREATE|DELETE|MODIFY|ALLOW|DISALLOW] artifact_pattern
```

Grammar 5: Grammar for operations within a step. artifact_pattern is a regular expression for the paths to artifacts.

For the latter, the `MATCH` rule is used by project owners to describe the flow of artifacts *between steps*. With it, a project owner can mandate that, e.g., a buildfarm must only use the sources that were created during a tag-release step or that only the right localization files are included during a localization step. Compared to the rules above, the `MATCH` rule has a richer syntax, as it needs to account for artifacts relocated during steps (e.g,. a packaging step moving .py files to `/usr/lib/pythonX.X/site-packages/` or a build step moving artifacts to a build directory) using the `IN` clause. Grammar 6 describes this rule and the Match function

describes the algorithm for processing it during verification. An example of a `MATCH` rule used to match the `foo.c` source from tag into the build step is shown in Listing 4.

```
MATCH source_pattern [IN prefix]
  WITH <MATERIALS|PRODUCTS> [IN prefix] FROM step_name
```

Grammar 6: The match rule grammar. The IN clauses are optional and source_pattern is a regular expression

**function MATCH**
**Input:** source_artifacts; destination_artifacts, rule
**Output:** *result: (SUCCESS/FAIL)*

```
1:  // Filter source and destination materials using the rule's patterns
2:  source_artifacts_filtered = filter(rule.source_prefix + rule.source_pattern,
    source_artifacts)
3:  destination_artifacts_filtered = filter(rule.destination_prefix +
    rule.destination_pattern, destination_artifacts)
4:  // Apply the IN clauses, to the paths, if any
5:  for artifact in source_artifacts_filtered do
6:    artifact.path -= rule.source_in_clause
7:  for artifact in destination_artifacts_filtered do
8:    artifact.path -= rule.destination_in_clause
9:  // compare both sets
10: for artifact in source_artifacts_filtered do
11:   destination_artifact = find_artifact_by_path(destination_artifacts,
    artifact.path)
12:   // the artifact with this path does not exist?
13:   if destination_artifact == NULL then
14:     return FAIL
15:   // are the files not the same?
16:   if destination_artifact.hash != artifact.hash then
17:     return FAIL
18: // all of the files filtered by the source materials exist
19: return SUCCESS
```

### 4.2 Link metadata files

Link metadata serves as a record that the steps prescribed in the layout actually took place. Its fields show *evidence* that is used for verification by the client. For example, the *materials* and *products* fields of the metadata are used to ensure that no intermediate products were altered in transit before being used in a step.

In order to determine if the executed step complies with its corresponding metadata, several types of information need to be gathered as evidence. A link includes the following fields:

```
{ "_type" : "link",
  "_name" :  "<NAME>",
  "command" : "<COMMAND>",
  "materials": {"<PATH>": "<HASH>", "..." : "..." },
  "products": {"<PATH>": "<HASH>", "..." : "..." },
  "byproducts": { "stdin": "", "stdout": "",
        "return-value": "" },
  "environment": {"variables": "<ENV>",
        "filesystem": "<FS>",  ...}
}
```

Listing 7: Link metadata format

- Name: This will be used to identify the step and to match it with its corresponding definition inside the layout.
- Material(s): Input file(s) that were used in this step, along with their cryptographic hashes to verify their integrity.
- Command: The command run, along with its arguments.
- Product(s): The output(s) produced and its corresponding cryptographic hash.
- Byproduct(s): Reported information about the step. Elements like the standard error buffer and standard output buffer will be used.
- Signature: A cryptographic signature over the metadata.

Of these fields, the `name`, `materials`, and `products` fields are the counterpart of the fields within the layout definition. This, along with a cryptographic signature used to authenticate the functionary who carried out the step can be used to provide a baseline verification of the supply chain topology (i.e., the steps performed and how do they interrelate via their materials and products). For example, the build step metadata described in Listing 8 shows the file `foo.c` used as a material and the product `foo` as created in the build step.

The *byproducts* field is used to include other meaningful information about a step's execution to further introspect into the specifics of the step that was carried out. Common fields included as byproducts are the standard output, standard error buffers and a return value. For example, if a command exited with non-zero status, then the byproduct field be populated with a value such as `return-value: "126"`. In this case, inspections can be set up to ensure that the return value of this specific command must be 0.

```
{ "_type": "link",
  "name": "build",
  "command": ["gcc", "foo.c", "-o", "foo" ],
  "materials": {"foo.c": { "sha256": "bff95e..."}},
  "products": {"foo": {"sha256": "25c696..."}}
  "byproducts": { "return-value": 0,
        "stderr": "", "stdout": ""},
  "environment": {},
}
```

Listing 8: The link for the build step

Having a software supply chain layout along with the matching pieces of link metadata and the delivered product are all the parts needed to perform verification. We will describe verification next.

### 4.3 Verifying the delivered product

Verification occurs when the link metadata and the layout are received by the client and upon installing the delivered product. A standalone or operating-system tool will perform the verification, as described in the function Verify_Final_Product. To do this, the user will need an initial public key that corresponds to the supply chain layout, as distributed by a trusted channel or as part of the operating system's installation [106].

The end user starts the verification by ensuring that the supply chain layout provided was signed with a trusted key (lines 2-3) and by checking the layout expiration date to make sure the layout is still valid (lines 5-6). If these checks pass, the public keys of all the functionaries are loaded from the layout (line 8). With the keys loaded, the verification routine will start iterating over all the steps defined in the layout and make sure there are enough pieces of link metadata signed by the right functionaries to match the threshold specified for that role (lines 10-20). If enough pieces of link metadata

```
function VERIFY_FINAL_PRODUCT
Input: layout; links; project_owner_key
Output: result: (SUCCESS/FAIL)
 1: // verify that the supply chain layout was properly signed
 2: if not verify_signature(layout, project_owner_key) then
 3:     return FAIL
 4: // Check that the layout has not expired
 5: if layout.expiration < TODAY then
 6:     return FAIL
 7: // Load the functionary public keys from the layout
 8: functionary_pubkeys = layout.keys
 9: // verify link metadata
10: for step in layout.steps do
11:     // Obtain the functionary keys relevant to this step and its corresponding
        metadata
12:     step_links = get_links_for_step(step, links)
13:     step_keys = get_keys_for_step(step, functionary_pubkeys)
14:     // Remove all links with invalid signatures
15:     for link in step_links do
16:         if not verify_signature(link, step_keys) then
17:             step_links.remove(link)
18:     // Check there are enough properly-signed links to meet the threshold
19:     if length(step_links) < step.threshold then
20:         return error("Link metadata is missing!")
21: // Apply artifact rules between all steps
22: if apply_artifact_rules(steps, links) == FAIL then
23:     return FAIL
24: // Execute inspections
25: for inspection in layout.inspections do
26:     inspections.add(Run(inspection))
27: // Verify inspections
28: if apply_artifact_rules(steps + inspections, links) == FAIL then
29:     return FAIL
30: return SUCCESS
```

could be loaded for each of the steps and their signatures pass verification, then the verification routine will apply the artifact rules and build a graph of the supply chain using the artifacts recorded in the link metadata (lines 22-23). If no extraneous artifacts were found and all the `MATCH` rules pass, then inspections will be executed[1] (line 25-26). Finally, once all inspections were executed successfully, artifact rules are re-applied to the resulting graph to check that rules on inspection steps match after inspections are executed, because inspections may produce new artifacts or re-create existing artifacts (lines 28-29). If all verifications pass, the function will return `SUCCESS`.

With this verification in place, the user is sure that the integrity of the supply chain is not violated, and that all requirements made by the project's maintainers were met.

### 4.4 Layout and Key Management

A layout can be revoked in one of two ways, the choice being up to the project owner. One is by revoking the key that was used to sign the layout, the other is by superseding/updating the layout with a newer one. To update a layout, the project owner needs to replace an existing layout with a newer layout. This can be used to deal with situations when a public key of a misbehaving functionary needs to be changed/revoked, because when the project owner publishes a newer layout without that public key, any metadata from such misbehaving functionary is automatically revoked. Updating a layout can also be used to address an improperly designed initial layout. The right expiration date for a layout depends on the operational security practices of the integrator.

[1] Inspections are executed only after all the steps are verified to avoid executing an inspection on an artifact that a functionary did not create.

## 5 Security Analysis

`in-toto` was designed to protect the software supply chain as a whole by leveraging existing security mechanisms, ensuring that they are properly set up and relaying this information to a client upon verification. This allows the client to make sure that all the operations were properly performed and that no malicious actors tampered with the delivered product.

To analyze the security properties of `in-toto`, we need to revisit the goals described in Section 2. Of these, the relevant goals to consider are *supply chain layout integrity*, *artifact flow integrity*, and *step authentication*. In this section, we explore how these properties hold, and how during partial key compromise the security properties of `in-toto` degrade gracefully.

`in-toto`'s security properties are strictly dependent on an attacker's level of access to a threshold of signing keys for any role. These security properties degrade depending on the type of key compromise and the supply chain configuration.

### 5.1 Security properties with no key compromise

When an attacker is able to compromise infrastructure or communication channels but not functionary keys, `in-toto`'s security properties ensure that the integrity of the supply chain is not violated. Considering our threat model in Section 2, and contrasting it to `in-toto`'s design which stipulates that the supply chain layout and link metadata are signed, we can assert the following:

- An attacker cannot interpose between two consecutive steps of the supply chain because, during verification, the hash on the products field of the link for the first step will not match the hash on the materials field of the link for the subsequent step. Further, a completely counterfeit version of the delivered product will fail validation because its hash will not match the one contained in the corresponding link metadata. Thus, **artifact flow integrity holds**.
- An attacker cannot provide a product that is missing steps or has its steps reordered because the corresponding links will be missing or will not be in the correct order. The user knows exactly which steps and in what order they need to be performed to receive the delivered product. As such, **supply chain layout integrity holds**.
- Finally, an attacker cannot provide link metadata for which he does not have permission to provide (i.e., their key is not listed as one that can sign link metadata for a certain step). Thus, **step authentication holds**.

However, it is important to underline that this threat model requires that the developer's host systems are not compromised. Likewise, it assumes that there are no rogue developers wishing to subvert the supply chain. For practical purposes, we consider a rogue functionary to be equivalent

to a functionary key compromise. Hence this section frames attacks from the standpoint of a key compromise, even when the issue may be executed as a confused deputy problem or a similar issue with equivalent impact.

### 5.2 Security properties if there is a key compromise

`in-toto` is not a "lose-one, lose-all" solution, in that its security properties only partially degrade with a key compromise. Depending on which key the attacker has accessed, `in-toto`'s security properties will vary. To further explore the consequences of key compromise, we outline the following types of attacks in the supply chain:

- Fake-check: a malicious party can provide evidence of a step taking place, but that step generates no products (it can still, however, generate byproducts). For example, an attacker could forge the results of a test suite being executed in order to trick other functionaries into releasing a delivered product with failing tests.
- Product Modification: a malicious party is able to provide a tampered artifact in a step to be used as material in subsequent steps. For example, an attacker could take over a buildfarm and create a backdoored binary that will be packaged into the delivered product.
- Unintended Retention: a malicious party does not destroy artifacts that were intended to be destroyed in a step. For example, an attacker that compromises a cleanup step before packaging can retain exploitable libraries that will be shipped along with the delivered product.
- Arbitrary Supply Chain Control: a malicious party is able to provide a tampered or counterfeit delivered product, effectively creating an alternate supply chain.

#### 5.2.1 Functionary compromise

A compromise on a threshold of keys held for any functionary role will only affect a specific step in the supply chain to which that functionary is assigned to. When this happens, the **artifact flow integrity** and **step authentication** security properties may be violated. In this case, the attacker can arbitrarily forge link metadata that corresponds to that step.

The impact of this may vary depending on the specific link compromised. For example, an attacker can fabricate an attestation for a step that does not produce artifacts (i.e., a fake-check), or create malicious products (i.e., a product modification), or pass along artifacts that should have been deleted (i.e., an unintended retention). When an attacker creates malicious products or fails to remove artifacts, the impact is limited by the usage of such products in subsequent steps of the chain. Table 1 describes the impact of these in detail from rows 2 to 5 (row 1 captures the case when the attacker does not compromise enough keys to meet the threshold defined for a step). As a recommended best practice, we assume there is a "`DISALLOW *`" rule at the end of the rule list for each step.

It is of note from Table 1 that an attacker who is able to compromise crucial steps (e.g., a build step) will have a greater impact on the client than one which, for example, can only alter localization files. Further, a compromise in functionary keys that do not create a product is restricted to a fake check attack (row two). To trigger an unintended retention, the first step must also have rules that allow for some artifacts before the `DELETE` rule (e.g., the `ALLOW` rule with a similar artifact pattern). This is because rules behave like artifact rules, and the attacker can leverage the ambiguity of the wildcard patterns to register an artifact that was meant to be deleted. Lastly, note that the effect of product modification and unintended retention is limited by the namespace on such rules (i.e., the `artifact_pattern`).

| Type of Key Compromise | Compromised Step Rule | Subsequent Step Rule | Impact |
|---|---|---|---|
| Under threshold | Regardless of rule | Regardless of rule | None |
| Step | None | Regardless of rule | Fake-check |
| Step | `ALLOW pattern1`<br>`DELETE pattern2` | `MATCH pattern*` | Unintended Retention |
| Step | `[ALLOW \| CREATE \| MODIFY] pattern` | `MATCH pattern` | Product Modification |
| Layout | N/A | N/A | Arbitrary Supply Chain Control |

Table 1: Key compromise and impact based on the layout characteristics.

**Mitigating risk.** As discussed earlier, the bar can be raised against an attacker if a role is required to have a higher threshold. For example, two parties could be in charge of signing the tag for a release, which would require the attacker to compromise two keys to successfully subvert the step.

Finally, further steps and inspections can be added to the supply chain with the intention of limiting the possible transformations on any step. For example, as shown in Section 6, an inspection can be used to dive into a Python's wheel and ensure that only Python sources in the tag release are contained in the package.

#### 5.2.2 Project owner compromise

A compromise of a threshold of keys belonging to the project owner role allows the attacker to redefine the layout, and thereby subvert the supply chain completely. However, like with step-level compromises, an increased threshold setting can be used to ensure an attacker needs to compromise many keys at once. Further, given the way `in-toto` is designed, the layout key is designed to be used rarely, and thus it should be kept offline.

### 5.3 User actions in response to `in-toto` failures

Detecting a failure to validate `in-toto` metadata involves making a decision about whether verification succeeded or whether it failed and, if so, why. The user's device and the reason for failure are likely to be paramount in the user's decision about how to respond. If the user is installing in an ephemeral environment on a testing VM, they may choose to ignore the warning and install the package regardless. If the user is installing in a production environment processing PCI data, the failure to validate `in-toto` metadata will be a serious concern. So, we expect users of `in-toto` will respond in much the same way as administrators do today for a package that is not properly signed.

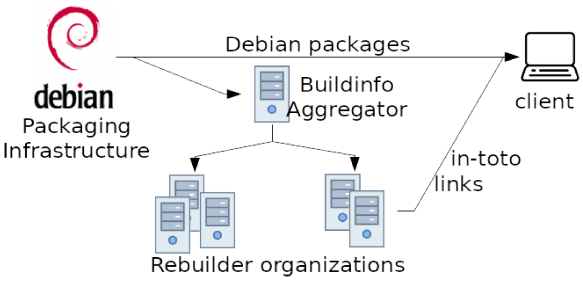


Figure 2: The rebuilder setup.

## 6 Deployment

in-toto has about a dozen different integrations that protect software supply chains for millions of end users. This section uses three such integrations to examine how threshold signing, metadata generation, and end-to-end verification function in practical deployments of in-toto.

### 6.1 Debian rebuilder constellation

Debian is one of the biggest stakeholders in the reproducible builds project [26], an initiative to ensure bit-by-bit deterministic builds of a source package. One of the main motivations behind reproducible builds is to avoid backdooring compilers [136] or compromised toolchains in the Debian build infrastructure. in-toto helps Debian achieve this goal via its step-thresholding mechanism.

The apt-transport [16] for in-toto verifies the trusted rebuilder metadata upon installing any Debian package. Meanwhile, various institutions (that range from private to non-profit and educational) run rebuilder infrastructure to rebuild Debian packages independently and produce attestations of the resulting builds using in-toto link metadata. This way, it is possible to cryptographically assert that a Debian package has been reproducibly built by a set of $k$ out of $n$ rebuilders. Figure 2 shows a graphical description of this setup.

By using the in-toto verifiable transport, users can make sure that no package was tampered with unless an attacker is also able to compromise at least $k$ rebuilders and the Debian buildfarm. Throughout this deployment, we were able to test the thresholding mechanism, as well as practical ways to bootstrap project owner signatures through the existing package manager trust infrastructure [32, 34].

**Deployment insight.** Through our interaction with reproducible builds, we were able to better understand how the thresholding mechanism can be used to represent concepts such as a build's reproducibility and how to build in-toto into operating-system tools to facilitate adoption.

### 6.2 Cloud native builds with Jenkins and Kubernetes

"Cloud native" is used to refer to container-based environments [3]. Cloud native ecosystems are characterized by rapid changes and constant re-deployment of the internal components. They are generally distributed systems, and often managed by container orchestration systems such as Kubernetes [23] or Docker Swarm [6]. Thus, their pipelines are mostly automated using pipeline managers such as Jenkins [18] or Travis [137]. In this type of ecosystems, a host- and infrastructure-agnostic, automated way to collect supply-chain metadata is necessary not only for security, but also for auditing and analyzing the execution of build processes that led to the creation of the delivered product.

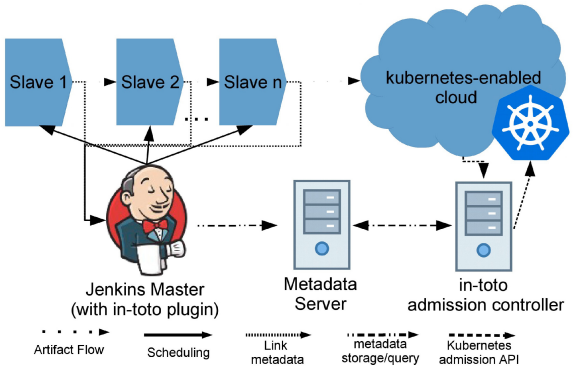


Figure 3: The kubesec supply chain.

In the context of cloud native applications, in-toto is used by Control Plane to track the build and quality-assurance steps on kubesec [19], a Kubernetes resource and configuration static analyzer. In order to secure the kubesec supply chain, we developed two in-toto components: a Jenkins plugin [11] and a Kubernetes admission controller [7, 17]. These two components allow us to track all operations within a distributed system, both of containers and aggregate in-toto link metadata, to verify any container image before it is provisioned. Figure 3 shows a (simplified) graphical depiction of their supply chain.

This deployment exemplifies an architecture for the supply chains of cloud native applications, in which new container images, serverless functions and many types of deployments are quickly updated using highly-automated pipelines. In this case, a pipeline serves as a coordinator, scheduling steps to worker nodes that serve as functionaries. These functionaries then submit their metadata to an in-toto metadata store. Once a new artifact is ready to be promoted to a cloud environment, a container orchestration system queries an in-toto admission controller. This admission controller ensures that every operation on this delivered product has been performed by allowed nodes and that all the artifacts were acted on according to the specification in the in-toto layout.

**Deployment insight.** Our interaction with kubesec forced us to investigate other artifact identifiers such as container images (in addition to files). While in-toto can be used today to track container images, the ability to point to an OCIv2 [21] image manifest can provide a more succinct link metadata representation and will be part of future work.

### 6.3 Datadog: E2E verification of Python packages

Datadog is a monitoring service for cloud-scale applications, providing monitoring of servers, databases, tools, and services, through a software-as-a-service-based data analytics platform [5]. It supports multiple cloud service providers, including Amazon Web Services (AWS), Microsoft Azure,

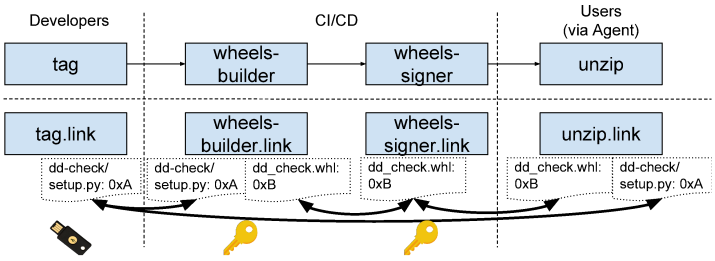


Figure 4: The simplified Datadog agent integrations supply chain. There are three steps (`tag` step, `wheels-builder` step, `wheels-signer` step), and one inspection. Arrows denote `MATCH` rules, the tag step is signed using a hardware dongle whereas the CI system uses online keys.

Google Cloud Platform, and Red Hat OpenShift. At the time of writing, it has over 8,000 customers, and collects trillions of monitoring record points per day.

The Datadog *agent* is software that runs on hosts. It collects events and metrics from hosts and sends them to Datadog, where customers can analyze their monitoring and performance data. The agent *integrations* are plug-ins that collect metrics from services running on customer infrastructure. Presently, there are more than one hundred integrations that come installed out-of-the-box with the Agent.

Datadog developers wanted an ability to automatically build and publish new or updated integrations independently of agent releases. This is so interested users can try new or updated integrations as they become available, and test whether they are applicable to their needs.

This section will cover how Datadog built the first tamper-evident pipeline using `in-toto` and how it leveraged TUF to safely bootstrap key distribution and provide replay-protection and freshness guarantees to in-toto metadata.

**End-to-end verification with `in-toto`.** The Datadog agent integrations supply chain, shown in Figure 4, has three steps:

1. The first `tag` step outputs Python source code as products. Every integration consists of Python source code and several YAML [133] configuration files. The link for this step is signed using a Yubikey hardware dongle [29]
2. In the second `wheels-builder` step, the pipeline must receive the same source code from the `tag` step and produce a Python wheel [24], as well as its updated Python metadata. Each wheel is a ZIP file and its metadata is an HTML file that points to all the available *versions* of an integration.
3. In the third `wheels-signer` step, the pipeline must receive, as materials, the same products as the `wheels-builder` step. This steps signs for all wheels using the system described in the next subsection. It can be dangerous packaging Python source code, because arbitrary code can be executed during the packaging process, which can be inserted by compromising the GitHub repository. Therefore, this step is separate from the `wheels-builder` step, so that a compromise of the former does not yield the signing keys of this step.

Finally, there is one inspection, which first ensures that a given wheel matches the materials of the `wheels-signer` step. It then extracts files from the wheel and checks that they correspond to exactly the same Python source code and YAML configuration files as the products of the `tag` step. Thus, this layout provides end-to-end verification: it prevents a compromised pipeline from causing users to trust wheels with source code that was never released by Datadog developers.

**Transport with The Update Framework (TUF).** Whereas `in-toto` provides end-to-end verification of the Datadog pipeline, it does not solve a crucial problem that arises in practice: How to securely distribute, revoke, and replace the public keys used to verify the `in-toto` layout. This mechanism must be *compromise-resilient* [100–102, 121], and resistant to a compromise of the software repository or server used to serve files. While SSL / TLS protects users from man-in-the-middle (MitM) attacks, it is not compromise-resilient, because attackers who compromise the repository can simply switch the public keys used to verify `in-toto` layout undetected, and thus defeat end-to-end verification. Likewise, other solutions, such as X509 certificates do not support necessary features such as in-band key revocation and key rotation.

The Update Framework (TUF) [100–102, 121] provides precisely this type of compromise-resilient mechanism, as well as in-band key revocation and key rotation. To do so, TUF adds a higher layer of signed metadata to the repository following two design principles that inspired the `in-toto` design. The first is the use of *roles* in a similar fashion to `in-toto`, so that a key compromise does not necessarily affect all *targets* (i.e., any Python wheels, or even `in-toto` metadata). The second principle is minimizing the risk of a key compromise using *offline* keys, or signing keys that are kept off the repository and pipeline in a cold storage mechanism, such as safe deposit boxes, so that attackers who compromise the infrastructure are unable to find these keys.

TUF is used within the Datadog integrations downloader to distribute, in a compromise-resilient manner, the: (1) root of trust for all wheels, TUF and `in-toto` metadata, (2) `in-toto` layout, and (3) public keys used to verify this layout. TUF also guarantees that MitM attackers cannot tamper with the consistency, authenticity, and integrity of these files, nor rollback or indefinitely replay `in-toto` metadata. This security model is simplified because it ignores some considerations that are out of the scope of this paper.

In summary, the Datadog pipeline uses TUF to appropriately bootstrap the root of the trust for the entire system, and `in-toto` to guarantee that the pipeline packaged exactly the source code signed by one of the Datadog developers inside universal Python wheels. By tightly integrating TUF and `in-toto`, Datadog's users obtain the compromise resilience of both systems combined.

**Deployment insight.** Through the Datadog deployment, we learned how to use other last-mile systems like TUF to provide not only compromise-resilience, but also replay-protection, freshness guarantees, and mix-and-match protection for in-toto metadata.

## 7 Evaluation

We evaluated in-toto's ability to guarantee software supply chain integrity on two fronts: efficiency and security. We set off to answer the following questions:

- Does in-toto incur reasonable overheads in terms of bandwidth, storage overhead and verification time?
- Can in-toto be used to protect systems against real-life supply chain compromises?

In order to answer the first question, we explored in-toto as used in the context of Datadog for two reasons: Datadog offers more than 111 integration packages to verify with in-toto, and its data and source code is publicly available. Furthermore, it is a production-ready integration that can be used by Datadog's more than 8,000 clients today [31]. Their clients include major companies like Twitter, NASDAQ and The Washington Post [4].

Then, we surveyed historical supply chain compromises and catalogued them. We evaluated these compromises against the in-toto deployments described in Section 6, accounting for their supply chain configuration, and including the actors involved and their possible key assignments. By studying the nature of each compromise, we were able to estimate what degree of key compromise could hypothetically happen and, with it, the consequences of such a compromise on these supply chains when in-toto is in place.

### 7.1 in-toto's overhead in the Datadog deployment

In the three major costs that in-toto presents are the storage, transfer and verification cost. In order to explore these costs, we set out to use the publicly available Datadog agent integration client and software repository. From this data, we can derive the cost of storing in-toto metadata in the repository, the cost of transferring the in-toto metadata for any given package and the verification times when installing any package.

**Storage overhead.** In order to understand the storage overhead, we mirrored the existing agent integrations Python package repository. Then, we inspected the package payloads and the repository metadata to show the cost breakdown of the repository as a whole. Table 2 depicts the cost breakdown of the Datadog repository, as mirrored on February 8 of 2019.

| Type | total package | Python metadata | TUF | in-toto links | in-toto Layout |
|---|---|---|---|---|---|
| RSA 4096 | 74.02% | 0.83% | 5.51% | 16.75% | 2.89% |
| DSA 1024 & ed25519 | 74.48% | 0.84% | 5.54% | 16.35% | 2.79% |
| optimized | 79.56% | 0.90% | 5.92% | 10.65% | 2.97% |

Table 2: Storage overhead breakdown for a in-toto enabled package repository. All metadata is compressed using zlib.

Table 2 shows that in-toto takes up about 19% of the total repository size, and thus makes it a feasible solution in terms of storage overhead. In addition, compared to its co-located security system TUF, the cost of using in-toto is higher, with almost four times the metadata storage cost. Further, the breakdown also indicates that the governing factor for this storage overhead are the in-toto links, rather than the layout file, with a layout being approximately 6 to 3 times smaller than the links (42 KB in comparison of the 148KB for all the link metadata).

There are two main reasons that drive this cost. First and foremost, is the engineering decision to track all the files within the pipeline (including Python metadata). Although these are not required to be tracked with in-toto, for the sake of security (as this type of metadata is being protected by TUF), it eases the implementation at a manageable cost. The second is that of signatures: the signatures used within the Datadog deployment come from either 4096-bit openpgp keys on a Yubikey, or 4096-bit PEM keys. These alone account for almost half of the in-toto metadata size.

For this reason, it is possible to further reduce the size of the metadata to 13% of the total repository size. Rows two and three of Table 2 represent the repository overhead when limiting the amount of files tracked to only Python sources and packages and using a DSA1024 as the signing algorithm.

**Network overhead.** Similar to storage overhead, the network overhead incurred by clients when installing any integration is of utmost importance. To explore this cost, we investigate the raw package sizes and contrast it with the size of the package-specific metadata size. It is of note though, that the size of in-toto metadata does not scale with the size of the package, but rather the number of files inside of it. This is because most of the metadata cost is taken by pieces of link metadata, of which the biggest three fields are the signature, expected_materials and expected_products. Figure 5 shows both the distribution of package sizes and the distribution of metadata sizes in increasing order.

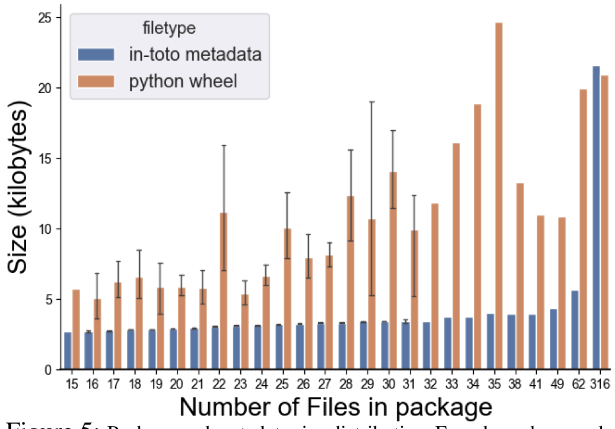


Figure 5: Package and metadata size distribution. Error bars show packages with the same number of files but different sizes.

In Figure 5 we can see that, for most packages, the metadata size cost is below 44% of the package size. In fact for the 90th percentile, the metadata cost approaches a costly 64%, to a worst case of 103%. However, upon inspecting these cases, we found that the issue is that these are cases in which *link metadata is tracking files not contained in the delivered product*. Indeed, in-toto is tracking files, such as test suites, fixtures and even iconography that does not get packaged on the integrations Python wheel. The worst case scenario is in fact cisco_aci, which only packages 12 files out of 316 contained in the tag step metadata.

**Verification overhead.** Finally, to draw insight from the computation time required to verify each package, we ran a series of micro-benchmarks on a laptop with an Intel i7-6500U processor and 8GB of RAM. In this case, we ran an iterated verification routine with the packages already fetched and instrumented the Datadog agent installer to measure the installation time with and without `in-toto` verification.

From this experiment, we conclude that `in-toto` verification adds less than 0.6 seconds on all cases. This is mostly dominated by the signature verification, and is thus bounded by the number of links to verify (i.e., the number of steps times the threshold).

### 7.2 Supply chain data breaches

We surveyed 30 major different supply chain breaches and incidents occurring from January 2010 to January 2019 (this list of historical attacks is included in Appendix B). These historical incidents cover a variety of software products and platforms, such as Apple's Xcode [113], Android GTK [8], MeDoc financial software [35], Adobe updater [95], PHP PEAR repository [33], and South Korean organizations [138].

Studying these historical attacks identified the type of access that the attacker had (or was speculated to have) and identified three categories: the attacker had control of infrastructure (but not functionary keys), the attacker had control over part of the infrastructure or keys of a specific functionary, and the attacker was able to control the entire supply chain by compromising a project owner's infrastructure (including their signing key).

For the historical attacks in Appendix B, we determined whether an attack used a compromised key, and then labeled those attacks with "Key Compromise". We also determined the degree of access in the attack (all the way to the possible step) and labeled each attack with an "Access Level" that indicates the step in the chain where the attack took place.

We now analyze how these compromises could affect the three supply chains where `in-toto` was deployed (as described in Section 6). Our analysis indicates that the majority of attacks (23 out of 30) took place without any key compromise. In those cases, none of the three `in-toto` deployments would have been affected since the client inspection (as described in Sec. 4.3) could detect extraneous artifacts or malicious delivered products.

Out of the 30 studied incidents, 7 involved a key compromise. We summarize our analysis of these attacks in Table 3. One attack, Keydnap [71], used a stolen Apple developer certificate to sign the malicious software package. Therefore, this attack would not have affected any `in-toto` deployments, because `in-toto` would detect that an unauthorized functionary signed the link metadata. Another attack used the developer's ssh key to upload a malicious Python package on PyPI [52]. All `in-toto` deployments could have detected this attack since files extracted from the malicious package would not exactly match the source code as the products of the first step in the supply chain.

The remaining five attacks involving a key compromise were recent sophisticated incidents that affected many clients and companies. The CCleaner [126] and RedHat [125] attacks are not effective against the Reproducible Builds deployment (RB) and Datadog (DD), as the former implements a threshold mechanism in the build step and the latter does not build binaries in their infrastructure. In a similar flavor, three attacks (Operation Red [138], NotPetya [35], and KingSlayer [118]) would not affect the Datadog deployment, as it implements a threshold mechanism in the packaging step. The Cloud Native deployment, on the other hand, would detect none of these five attacks, as it does not employ thresholds. To conclude, the `in-toto` deployments would detect most of the historical attacks based on the general `in-toto` design principles. For those attacks that involve key compromises, our analysis shows that `in-toto`'s use of thresholds is an effective mechanism.

| Attack Name | DD | RB | CN |
|---|---|---|---|
| Keydnap [71] | ✓ | ✓ | ✓ |
| backdoored-pypi [52] | ✓ | ✓ | ✓ |
| CCleaner Atatck [126] | ✓ | ✓ | ✗ |
| RedHat breach [125] | ✓ | ✓ | ✗ |
| *NotPetya [35] | ✓ | ✗ | ✗ |
| Operation Red [138] | ✓ | ✗ | ✗ |
| KingSlayer [118] | ✓ | ✗ | ✗ |

Table 3: The impact of the historical attacks on the three `in-toto` deployments: Datadog (DD), Reproducible Builds (RB), Cloud Native (CN). Out of the 30 historical attacks, 23 did not involve a key compromise, so none of the deployments would have been affected. This table shows the remaining attacks which involved a key compromise. In one attack, marked with a star (*), it is unknown if a key compromise took place. We assumed that was the case. A ✓ indicates that the deployment could have detected the attack.

**Key Takeaway.** Cloud Native (83%) and reproducible builds (90%) integrations of `in-toto` would prevent most historical supply chain attacks. However, integration into a secure update system as was done by Datadog (100%) provides further protection.

## 8 Related Work

To the best of our knowledge, work that attempts to use an automated tool to secure the supply chain is scarce. However, there has been a general push to increase the security of different aspects within the supply chain, as well as to tighten the binding between neighboring processes within that chain. In this section, we mention work relevant to supply chain security, as some of it is crucial for the success of `in-toto` as a framework. We also list work that can further increase the security guarantees offered by `in-toto`.

**Automated supply chain administration systems.** Configuring and automating processes of the supply chain has been widely studied. Works by Bégin et al. [45], Banzai et al., [43] and Andreetto et al. [36] focus on designing supply chains that automatically assign resources and designate parties to take part in different processes to create a product. This work is similar to `in-toto` in that it requires a supply chain topology to carry out the work. However, none of these projects were focused on security. Instead, they deal with adaptability of resources and supply chain automation.

Perhaps most closely related to in-toto is the Grafeas API [9] released by Google. However, Grafeas's focus is on tracking and storing supply chain metadata rather than security. Grafeas provides a centralized store for supply chain metadata, which can be later queried by verification tools such as Google's Binary Authorization [84]. Grafeas does not provide a mechanism to describe what steps should be performed, validate performed steps, or even support cryptographic signatures [1]. Finally, in-toto is architecture agnostic, while Grafeas is mostly cloud-native; in-toto was geared to represent supply chains whether they are cloud-native, off-cloud or hybrid-cloud. We are collaborating with the Grafeas team to natively support in-toto link metadata within Grafeas [10].

**Software supply chain security.** In addition, many software engineering practices have been introduced to increase the security of the software development lifecycle [42, 104, 105, 111, 116]. Additional work by Devanbu et al. [67] has explored different techniques to "construct safe software that inspires trust in hosts." These techniques are similar to in-toto in that they suggest releasing supply chain information to the end users for verification.

Though none of these proposals suggest an automated tool to ensure the integrity of the supply chain, they do serve as a helpful first step in designing in-toto. As such, their practices could be used as templates for safe supply chain layouts.

Finally, there have been hints by the industry to support features that could be provided by in-toto [90, 114, 145]. This includes providing certificates noting the presence of a process within the supply chain and providing software transparency through mechanisms such as reproducible builds.

**Source control security.** The source code repository is usually the first link in the supply chain. Early work in this field has explored the different security properties that must be included in software configuration management tools [63]. Version control systems, such as Git, incorporate protection mechanisms to ensure the integrity of the source code repository, which include commit hash chaining and signed commits [77, 78].

**Buildsystem and verification security.** The field of automated testing and continuous integration has also received attention from researchers. Recently, self-hosted and public automated testing and continuous integration systems have become popular [54, 72, 137]. Work by Gruhn et al. [85] has explored the security implications of malicious code running on CI systems, showing that it is possible for attackers to affect other projects being tested in the same server, or the server itself. This work, and others [69] serve as a motivation for in-toto's threat model.

Further work by Hanawa et al. [87] explores different techniques for automated testing in distributed systems. The work is similar to in-toto in that it allocates hosts in the cloud to automatically run tests for different environments and platforms. However, in-toto requires such systems to provide certification (in the form of link metadata) that the tests were run and the system was successful.

Subverting the development environment, including subverting the compiler, can have a serious impact on the software supply chain [135]. Techniques such as Wheeler's diverse double-compiling (DDC) [144] can be used to mitigate such "trusting trust" attacks. In the context of reproducible builds project, DDC can also be used for multi-party verification of compiler executables.

**Verifying compilers, applications and kernels.** Ongoing work on verifying compilers, applications and kernels will provide a robust framework for applications that fully comply with their specification [88, 98]. Such work is similar to in-toto in that a specification is provided for the compiler to ensure that their products meet stated requirements. However, in contrast to our work, most of this work is not intended to secure the origin of such specification, or to provide any proof of the compilation's results to steps further down the supply chain. Needless to say, verifying compilers could be part of a supply chain protected with in-toto.

Furthermore, work by Necula et al. introduces proof-carrying code [109, 110], a concept that relies on the compiler to accompany machine code with proof for verification at runtime. Adding to this, industry standards have included machine code signing [40] to be verified at runtime. This work is similar to in-toto in that compilers generate information that will be verified by the end user upon runtime. Although these techniques are more granular than in-toto's (runtime verification vs verification upon installation), they do not aim to secure the totality of the supply chain.

**Package management and software distribution security.** Work by Cappos et al. has been foundational to the design of in-toto's security mechanisms [46, 102, 121]. The mechanisms used to secure package managers are similar to in-toto in that they rely on key distribution and role separation to provide security guarantees that degrade with partial key compromise. However, unlike in-toto, these systems are restricted to software updates, which limit their scope. Concepts from this line of work could be overlaid on in-toto to provide additional "last mile" guarantees for the resulting product, such as package freshness or protection against dependencies that are not packaged with the delivered product.

## 9 Conclusions and future work

In this paper, we have described many aspects of in-toto, including its security properties, workflow and metadata. We also explored and described several extensions and implications of using in-toto in a number of real-world applications. With this we have shown that protecting the entirety of the supply chain is possible, and that it can be done automatically by in-toto. Further, we showed that, in a number of practical applications, in-toto is a practical solution to many contemporary supply chain compromises.

Although plenty of work needs to be done in the context of the in-toto framework (e.g., decreasing its storage cost), tackling the first major limitations of supply chain security will increase the quality of software products. We expect that, through continued interaction with the industry and

elaborating on the framework, we can provide strong security guarantees for future software users.


### Acknowledgments

We would like to thank the USENIX reviewers and Luke Valenta for reviewing this paper. We would also like to thank Lukas Pühringer and Lois DeLong from the `in-toto` team; Holger Levsen, Chris Lamb, kpcyrd, and Morten Linderud from Reproducible Builds; the Datadog Agent Integrations (especially Ofek Lev) and Product Security teams; as well as Andrew Martin and Luke Bond from Control Plane for their valuable work towards integrating `in-toto` in all these communities. This research was supported by the NSF under Grants No. CNS 1801430 and DGE 1565478.


### References


[1] Add Signature message to v1beta common.proto. #253. https://github.com/grafeas/grafeas/pull/253.

[2] Apt. https://wiki.debian.org/Apt.

[3] Cloud native computing foundation. https://www.cncf.io/.

[4] Customers | Datadog. https://www.datadoghq.com/customers/.

[5] Datadog: Modern monitoring & analytics. https://www.datadoghq.com/.

[6] Docker Swarm overview. https://docs.docker.com/swarm/overview/.

[7] Dynamic admission control. https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/.

[8] ExpensiveWall: A Dangerous Packed Malware On Google Play. https://blog.checkpoint.com/2017/09/14/expensivewall-dangerous-packed-malware-google-play-will-hit-wallet/.

[9] Grafeas. https://grafeas.io/.

[10] Grafeas + in-toto. https://github.com/in-toto/totoify-grafeas.

[11] in-toto Jenkins plugin. https://plugins.jenkins.io/in-toto.

[12] in-toto layout creation tool. https://in-toto.engineering.nyu.edu.

[13] in-toto Metadata Examples. https://in-toto.github.io/metadata-examples.html.

[14] in-toto Specification: Version 0.9. https://github.com/in-toto/docs/blob/v0.9/in-toto-spec.md.

[15] in-toto Specifications. https://in-toto.github.io/specs.html.

[16] in-toto transport for apt. https://github.com/in-toto/apt-transport-in-toto.

[17] in-toto-webhook. https://github.com/SantiagoTorres/in-toto-webhook.

[18] Jenkins: Build great things at any scale. https://jenkins.io/.

[19] Kubesec.io: Quantify risk for kubernetes resources. https://kubesec.io/.

[20] Notary. https://docs.docker.com/samples/library/notary/.

[21] Oci image format specification. https://github.com/opencontainers/image-spec.

[22] Operation Aurora. https://en.wikipedia.org/wiki/Operation_Aurora.

[23] Production-Grade Container Orchestration. https://kubernetes.io/.

[24] Python Wheels. https://pythonwheels.com/.

[25] Reproducible builds. https://reproducible-builds.org/.

[26] Reproducible builds: Who is involved? https://reproducible-builds.org/who/.

[27] Some Debian Project machines compromised. https://www.debian.org/News/2003/20031121.

[28] The Update Framework (TUF). https://theupdateframework.github.io/.

[29] The YubiKey. https://www.yubico.com/products/yubikey-hardware/.

[30] Twistlock: Cloud Native Security for Docker, Kubernetes and Beyond. https://www.twistlock.com/.

[31] Forbes Cloud 100: #19 Datadog, 2018. https://www.forbes.com/companies/datadog/?list=cloud100#3cad45279e03.

[32] in-toto at the reproducible builds summit-paris 2018, 2019. https://ssl.engineering.nyu.edu/blog/2019-01-18-in-toto-paris.

[33] *PHP PEAR Software Supply Chain Attack*, 2019. https://blog.dcso.de/php-pear-software-supply-chain-attack/.

[34] Reproducible builds: Weekly report #196, 2019. https://reproducible-builds.org/blog/posts/196/.

[35] A. Cherepanov. *Analysis of TeleBots' cunning backdoor*. https://www.welivesecurity.com/2017/07/04/analysis-of-telebots-cunning-backdoor.

[36] P. Andreetto, S. Andreozzi, G. Avellino, S. Beco, A. Cavallini, M. Cecchi, V. Ciaschini, A. Dorise, F. Giacomini, A. Gianelle, et al. The glite workload management system. In *J. of Physics: Conf. Series*, volume 119, page 062007. IOP Publishing, 2008.

[37] Andy Greenberg. *MacOS Update Accidentally Undoes Apple's "Root" Bug Patch*. https://www.wired.com/story/macos-update-undoes-apple-root-bug-patch/.

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

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

[40] Apple Computers. iOS Security Guide, 2016. https://www.apple.com/business/docs/iOS_Security_Guide.pdf.

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

[42] R. Bachmann and A. D. Brucker. Developing secure software. *Datenschutz und Datensicherheit*, 38(4):257–261, 2014.

[43] T. Banzai, H. Koizumi, R. Kanbayashi, T. Imada, T. Hanawa, and M. Sato. D-cloud: Design of a software testing environment for reliable distributed systems using cloud computing technology. In *Proc. of the 10th IEEE/ACM CCGrid*, 2010.

[44] Barb Darrow. *Adobe source code breach; it's bad, real bad*. https://gigaom.com/2013/10/04/adobe-source-code-breech-its-bad-real-bad.

[45] M.-E. Bégin, G. D.-A. Sancho, A. Di Meglio, E. Ferro, E. Ronchieri, M. Selmi, and M. Żurek. Build, configuration, integration and testing tools for large software projects: Etics. In *Rapid Integration of Software Engineering Techniques*, pages 81–97. Springer, 2006.

[46] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman. A look in the mirror: Attacks on package managers. In *Proc. of the 15th ACM CCS*, pages 565–574, 2008.

[47] S. Checkoway, S. Cohney, C. Garman, M. Green, N. Heninger, J. Maskiewicz, E. Rescorla, H. Shacham, and R.-P. Weinmann. A systematic analysis of the juniper dual ec incident. Cryptology ePrint Archive, Report 2016/376, 2016. http://eprint.iacr.org/.

[48] S. Checkoway, J. Maskiewicz, C. Garman, J. Fried, S. Cohney, M. Green, N. Heninger, R. P. Weinmann, E. Rescorla, and H. Shacham. A Systematic Analysis of the Juniper Dual EC Incident. In *Proc. of ACM CCS '16*, 2016.

[49] R. Chirgwin. *Microsoft deletes deleterious file deletion bug from Windows 10*. https://www.theregister.co.uk/2018/10/10/microsoft_windows_deletion_bug/.

[50] A. Chitu. The Android Bug 8219321. https://googlesystem.blogspot.com/2013/07/the-8219321-android-bug.html#gsc.tab=0, 2013.

[51] Christian Nutt. *Cloud source host Code Spaces hacked, developers lose code*. http://www.gamasutra.com/view/news/219462/Cloud_source_host_Code_Spaces_hacked_developers_lose_code.php.

[52] C. Cimpanu. *Backdoored Python Library Caught Stealing SSH Credentials*, 2018. https://www.bleepingcomputer.com/news/security/backdoored-python-library-caught-stealing-ssh-credentials/.

[53] C. Cimpanu. *Microsoft Discovers Supply Chain Attack at Unnamed Maker of PDF Software*, 2018. https://www.bleepingcomputer.com/news/security/microsoft-discovers-supply-chain-attack-at-unnamed-maker-of-pdf-software/.

[54] Codeship. Continuous Delivery with Codeship: Fast, Secure, and fully customizable. https://codeship.com/.

[55] Context Threat Intelligence. *Threat Advisory: The Monju Incident*, 2014. https://paper.seebug.org/papers/APT/APT_CyberCriminal_Campagin/2014/The_Monju_Incident.pdf.

[56] M. Coppock. *Windows Update not working after October 2018 patch? Here's how to fix it* . https://www.digitaltrends.com/computing/windows-update-not-working/.

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

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

[59] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti. A large-scale analysis of the security of embedded firmwares. In *Proc. of the 23rd USENIX Security Symposium*, pages 95–110, 2014.

[60] CrowdStrike. *Securing the supply chain*. https://www.crowdstrike.com/resources/wp-content/brochures/pr/CrowdStrike-Security-Supply-Chain.pdf.

[61] A. Cui, M. Costello, and S. J. Stolfo. When firmware modifications attack: A case study of embedded exploitation. In *NDSS*, 2013.

[62] Dan Goodin. Kernel.org Linux repository rooted in hack attack. http://www.theregister.co.uk/2011/08/31/linux_kernel_security_breach/.

[63] David A. Wheeler. Software Configuration Management Security. http://www.dwheeler.com/essays/scm-security.html.

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

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

[66] Dennis Fisher. *Researcher Finds Tor Exit Node Adding Malware to Binaries*. https://threatpost.com/researcher-finds-tor-exit-node-adding-malware-to-binaries/109008/.

[67] P. T. Devanbu, P. W. Fong, and S. G. Stubblebine. Techniques for trusted software engineering. In *Proceedings of the 20th international conference on Software engineering*, pages 126–135. IEEE Computer Society, 1998.

[68] Dona Sarkar. *A note about the unintentional release of builds today*. https://blogs.windows.com/windowsexperience/2017/06/01/note-unintentional-release-builds-today/.

[69] P. M. Duvall, S. Matyas, and A. Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[70] Edward Iskra. *Vulnerable Wallets and the Suspicious File*, 2017. https://bitcoingold.org/vulnerable-wallets/.

[71] ESET Research. *OSX/Keydnap spreads via signed Transmission application*. https://www.welivesecurity.com/2016/08/30/osxkeydnap-spreads-via-signed-transmission-application/.

[72] B. Fitzgerald and K.-J. Stol. Continuous software engineering and beyond: trends and challenges. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, pages 1–9. ACM, 2014.

[73] A. Forums. Numix gnome 3.20. https://bbs.archlinux.org/viewtopic.php?id=211164.

[74] J. Freeman. Yet Another Android Master Key Bug. http://www.saurik.com/id/19, 2014.

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

[76] Gentoo Linux. *Incident Reports/2018-06-28 Github*. https://wiki.gentoo.org/wiki/Project:Infrastructure/Incident_Reports/2018-06-28_Github.

[77] M. Gerwitz. A Git Horror Story: Repository Integrity With Signed Commits. http://mikegerwitz.com/papers/git-horror-story.

[78] Git SCM. Signing your work. https://git-scm.com/book/en/v2/Git-Tools-Signing-Your-Work.

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

[80] GNU Savannah. Compromise2010. https://savannah.gnu.org/maintenance/Compromise2010/.

[81] D. Goodin. Fedora servers breached after external compromise. http://www.theregister.co.uk/2011/01/25/fedora_server_compromised/.

[82] D. Goodin. *Meet "Great Cannon", the man-in-the-middle weapon China used on GitHub*. https://arstechnica.com/security/2015/04/meet-great-cannon-the-man-in-the-middle-weapon-china-used-on-github/.

[83] D. Goodin. 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.

[84] Google. Binary Authorization. https://cloud.google.com/binary-authorization/.

[85] V. Gruhn, C. Hannebauer, and C. John. Security of public continuous integration services. In *Proc. of the 9th ACM International Symposium on Open Collaboration*, page 15, 2013.

[86] Hackread. *Proton malware*. https://www.hackread.com/hackers-infect-mac-users-proton-malware-using-elmedia-player.

[87] T. Hanawa, T. Banzai, H. Koizumi, R. Kanbayashi, T. Imada, and M. Sato. Large-scale software testing environment using cloud computing technology for dependable parallel and distributed systems. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 428–433. IEEE, 2010.

[88] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In *Proc. of the 11th USENIX OSDI*, pages 165–181, 2014.

[89] Idrees Patel. *Janus Vulnerability*. https://www.xda-developers.com/janus-vulnerability-android-apps.

[90] ISO/IEC JTC 1/SC 27 IT Security techniques. *ISO/IEC 27034:2011 Information technology – Security techniques – Application security*. https://www.iso.org/standard/44378.html?browse=tc.

[91] Jane Silber. Notice of Ubuntu Forums breach. https://blog.ubuntu.com/2016/07/15/notice-of-security-breach-on-ubuntu-forums.

[92] Jared Newman. Gauss Malware: What You Need to Know. https://www.pcworld.com/article/260735/gauss_malware_what_you_need_to_know.html.

[93] Jeff Erickson. Inside OilRig – Tracking Iran's Busiest Hacker Crew On Its Global Rampage. https://www.forbes.com/sites/thomasbrewster/2017/02/15/oilrig-iran-hackers-cyberespionage-us-turkey-saudi-arabia/#5415a493468a.

[94] Jensen Beeler. Millions of Motorcyclists Hacked in VerticalScope Breach. https://www.asphaltandrubber.com/news/verticalscope-hack/.

[95] Jeremy Kirk. *New malware overwrites software updaters*, 2010. https://www.itworld.com/article/2755831/security/new-malware-overwrites-software-updaters.html.

[96] Juniper. 2015-12 Out of Cycle Security Bulletin: ScreenOS: Multiple Security issues with ScreenOS (CVE-2015-7755, CVE-2015-7756). https://kb.juniper.net/InfoCenter/index?page=content&id=JSA10713, Dec. 15.

[97] G. Kelly. *Apple iOS 12.1.4 Fails To Fix Cellular, WiFi Problems*. https://www.forbes.com/sites/gordonkelly/2019/02/10/apple-ios-12-1-4-problem-iphone-cellular-data-wifi-upgrade-ipad/.

[98] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. sel4: Formal verification of an os kernel. In *Proc. of the 22nd ACM SOSP*, pages 207–220, 2009.

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

[100] T. K. Kuppusamy, A. Brown, S. Awwad, D. McCoy, R. Bielawski, C. Mott, S. Lauzon, A. Weimerskirch, and J. Cappos. Uptane: Securing software updates for automobiles. *14th ESCAR Europe*, 2016.

[101] T. K. Kuppusamy, V. Diaz, and J. Cappos. Mercury: Bandwidth-effective prevention of rollback attacks against community repositories. In *Proc. of the 2017 USENIX Annual Technical Conference*, 2017.

[102] T. K. Kuppusamy, S. Torres-Arias, V. Diaz, and J. Cappos. Diplomat: using delegations to protect community repositories. In *proc. of the 13th USENIX NSDI*, pages 567–581, 2016.

[103] Martin Brinkmann. *Attention: Some Fosshub downloads compromised*. https://www.ghacks.net/2016/08/03/attention-fosshub-downloads-compromised/.

[104] M. S. Merkow and L. Raghavan. Secure and resilient software: Requirements, test cases, and testing methods. 2011.

[105] Microsoft. Microsoft secure development lifecycle. https://www.microsoft.com/en-us/sdl/default.aspx.

[106] Microsoft. Microsoft Trusted Publishers Certificate Store. https://msdn.microsoft.com/en-us/library/windows/hardware/ff553504(v=vs.85).aspx.

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

[108] Naked Security. Flame malware used man-in-the-middle attack against Windows Update. https://nakedsecurity.sophos.com/2012/06/04/flame-malware-used-man-in-the-middle-attack-against-windows-update/.

[109] G. C. Necula. Proof-carrying code. In *Proceedings of the ACM SIGPLAN*, 1997.

[110] G. C. Necula. *Proof-carrying code. design and implementation*. Springer, 2002.

[111] I. S. Organization. Information technology – security techniques – application security – part 1: Overview and concepts. http://www.iso.org/iso/catalogue_detail.htm?csnumber=44378.

[112] Patrick Gray. Gentoo Linux server compromised. https://www.zdnet.com/article/gentoo-linux-server-compromised/, 2003.

[113] D. Pauli. icloud phishing attack hooks 39 ios apps and wechat. theregister, 2015. https://www.theregister.co.uk/2015/09/21/icloud_phishing_attack_hooks_39_ios_apps_most_popular_message_client/.

[114] S. Quirolgico, J. Voas, T. Karygiannis, C. Michael, and K. Scarfone. *Vetting the Security of Mobile Applications*. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-163.pdf.

[115] Red Hat, Inc. Infrastructure report, 2008-08-22 UTC 1200. https://rhn.redhat.com/errata/RHSA-2008-0855.html, 2008.

[116] J. Robbins. Adopting open source software engineering (OSSE) practices by adopting OSSE tools. *Perspectives on free and open source software*, pages 245–264, 2005.

[117] RODRIGO ARANGUA. The security flaws at the heart of the Panama Papers. https://www.wired.co.uk/article/panama-papers-mossack-fonseca-website-security-problems.

[118] RSA Research. *Kingslayer-A Supply Chain Attack*. https://www.rsa.com/content/dam/premium/en/white-paper/kingslayer-a-supply-chain-attack.pdf.

[119] RubyGems.org. Data Verification. http://blog.rubygems.org/2013/01/31/data-verification.html, 2013.

[120] Ryan Naraine. *Open-source ProFTPD hacked, backdoor planted in source code*. http://www.zdnet.com/article/open-source-proftpd-hacked-backdoor-planted-in-source-code.

[121] J. Samuel, N. Mathewson, J. Cappos, and R. Dingledine. Survivable key compromise in software update systems. In *Proc. of the 17th ACM CCS*, pages 61–72. ACM, 2010.

[122] Slashdot Media. phpMyAdmin corrupted copy on Korean mirror server. https://sourceforge.net/blog/phpmyadmin-back-door/, 2012.

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

[124] Steve Klabnik. *Security advisory for crates.io, 2017-09-19*. https://users.rust-lang.org/t/security-advisory-for-crates-io-2017-09-19/12960.

[125] Steven J. Vaughan-Nichols. *Red Hat's Ceph and Inktank code repositories were cracked*. http://www.zdnet.com/article/red-hats-ceph-and-inktank-code-repositories-were-cracked.

[126] Swati Khandelwal. CCleaner Attack Timeline–Here's How Hackers Infected 2.3 Million PCs. https://thehackernews.com/2018/04/ccleaner-malware-attack.html, 2018.

[127] Symantec. W32.Duqu: The precursor to the next Stuxnet. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_duqu_the_precursor_to_the_next_stuxnet.pdf.

[128] Symantec. W32.Stuxnet Dossier. https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_

dossier.pdf.
[129] Symantec Corporation. Internet threat security report, 2018. https://www.symantec.com/content/dam/symantec/docs/reports/istr-23-2018-en.pdf.
[130] The FreeBSD Project. FreeBSD.org intrusion announced November 17th 2012. http://www.freebsd.org/news/2012-compromise.html, 2012.
[131] The PHP Group. php.net security notice. http://www.php.net/archive/2011.php#id2011-03-19-1, 2011.
[132] The PHP Group. A further update on php.net. http://php.net/archive/2013.php#id2013-10-24-2, 2013.
[133] The YAML Project. The Official YAML Web Site. https://yaml.org/, 2019.
[134] Thomas Reed. *HandBrake hacked to drop new variant of Proton malware*. https://blog.malwarebytes.com/threat-analysis/mac-threat-analysis/2017/05/handbrake-hacked-to-drop-new-variant-of-proton-malware/.
[135] Thomas Reed. XcodeGhost malware infiltrates App Store. https://blog.malwarebytes.com/cybercrime/2015/09/xcodeghost-malware-infiltrates-app-store/.
[136] K. Thompson. Reflections on Trusting Trust. http://cm.bell-labs.com/who/ken/trust.html.
[137] Travis CI. Travis CI – test and deploy your code with confidence. https://travis-ci.org/.
[138] Trend Micro Cyber Safety Solutions Team. *Supply Chain Attack Operation Red Signature Targets South Korean Organizations*, 2018. https://blog.trendmicro.com/trendlabs-security-intelligence/supply-chain-attack-operation-red-signature-targets-south-korean-organizations/.
[139] Trend Micro Cyber Safety Solutions Team. *New Magecart Attack Delivered Through Compromised Advertising Supply Chain*, 2019. https://blog.trendmicro.com/trendlabs-security-intelligence/new-magecart-attack-delivered-through-compromised-advertising-supply-chain/.
[140] W. Verduzu. Xposed Patch for Master Key and Bug 9695860 Vulnerabilities. https://www.xda-developers.com/xposed-patch-for-master-key-and-bug-9695860-vulnerabilities/, 2013.
[141] L. Voss. Newly Paranoid Maintainers. http://blog.npmjs.org/post/80277229932/newly-paranoid-maintainers, 2014.
[142] T. Warren. *Major new iOS bug can crash iPhones*. https://www.theverge.com/2018/2/15/17015654/apple-iphone-crash-ios-11-bug-imessage.
[143] F. Weimer. CVE-2013-6435. https://access.redhat.com/security/cve/CVE-2013-6435, 2013.
[144] D. A. Wheeler. Fully countering trusting trust through diverse double-compiling. *arXiv preprint arXiv:1004.5534*, 2010.
[145] Yan/Bcrypt. Software transparency: Part 1. https://zyan.scripts.mit.edu/blog/software-transparency/.
[146] Zack Whittaker. Hacker explains how he put 'backdoor' in hundreds of linux mint downloads. http://www.zdnet.com/article/hacker-hundreds-were-tricked-into-installing-linux-mint-backdoor.
[147] K. Zetter. *'Google' Hackers had ability to alter source code'*. https://www.wired.com/2010/03/source-code-hacks.

## A `in-toto` artifact rule definition

The following artifact rule definition is taken from the `in-toto` specification v0.9 [14].

- `ALLOW`: indicates that artifacts matched by the pattern are allowed as materials or products of this step.
- `DISALLOW`: indicates that artifacts matched by the pattern are not allowed as materials or products of this step.
- `REQUIRE`: indicates that a pattern must appear as a material or product of this step.
- `CREATE`: indicates that products matched by the pattern must not appear as materials of this step.
- `DELETE`: indicates that materials matched by the pattern must not appear as products of this step.
- `MODIFY`: indicates that products matched by the pattern must appear as materials of this step, and their hashes must not by the same.
- `MATCH`: indicates that the artifacts filtered in using `source-path-prefix/pattern` must be matched to a "MATERIAL" or "PRODUCT" from a destination step with the "`destination-path-prefix/pattern`".

## B Surveyed Attacks

| Attack Name | Key Compromise | Access Level |
|---|---|---|
| *NotPetya [35] | ✓ | PI |
| CCleaner Attack [126] | ✓ | BS, PI |
| Operation Red [138] | ✓ | PI |
| KingSlayer [118] | ✓ | PI |
| RedHat breach [125] | ✓ | BS |
| keydnap [71] | ✓ | PI |
| backdoored-pypi [52] | ✓ | PI |
| PEAR breach [33] | ✗ | PI |
| Monju Incident [55] | ✗ | PI |
| Janus Vulnerability [89] | ✗ | PI |
| Rust flaw [124] | ✗ | PI |
| XcodeGhost [113] | ✗ | BS |
| Expensive Wall [8] | ✗ | BS |
| WordPress breach [107] | ✗ | CR |
| HandBrake breach [134] | ✗ | PI |
| Proton malware [86] | ✗ | PI |
| FOSSHub breach [103] | ✗ | PI |
| BadExit Tor [66] | ✗ | PI |
| Fake updater [95] | ✗ | PI |
| Bitcoin Gold breach [70] | ✗ | PI |
| Adobe breach [44] | ✗ | CR |
| Google Breach [147] | ✗ | CR |
| ProFTPD breach [120] | ✗ | CR |
| Kernel.org breach [62] | ✗ | CR |
| Hacked Linux Mint [146] | ✗ | PI |
| Code Spaces breach [51] | ✗ | CR |
| Unnamed Maker [53] | ✗ | PI |
| Gentoo backdoor [76] | ✗ | CR |
| Buggy Windows [68] | ✗ | PI |
| Buggy Mac [37] | ✗ | PI |

Table 4: Summary of surveyed supply chain attacks. CR, BS and PI stand for Code Repository, Build System and Publishing Infrastructure, respectively. A ✓indicates that the attack involved a key compromise. In one attack, marked with a star (*), it was unknown if a compromised key was involved. We assumed that was the case.