Extending Cloud Build Systems to Eliminate Transitive Trust

Martin Schwaighofer
martin.schwaighofer@ins.jku.at
Johannes Kepler University Linz
Linz, Austria

Michael Roland
michael.roland@ins.jku.at
Johannes Kepler University Linz
Linz, Austria

René Mayrhofer
rm@ins.jku.at
Johannes Kepler University Linz
Linz, Austria

Abstract

Trusting the output of a build process requires trusting the build process itself, and the build process of all inputs to that process, and so on. Cloud build systems, like Nix or Bazel, allow their users to precisely specify the build steps making up the intended software supply chain, build the desired outputs as specified, and on this basis delegate build steps to other builders or fill shared caches with their outputs. Delegating build steps or consuming artifacts from shared caches, however, requires trusting the executing builders, which makes cloud build systems better suited for centrally managed deployments than for use across distributed ecosystems. We propose two key extensions to make cloud build systems better suited for use in distributed ecosystems. Our approach attaches metadata to the existing cryptographically secured data structures and protocols, which already link build inputs and outputs for the purpose of caching. Firstly, we include builder provenance data, recording which builder executed the build, its software stack, and a remote attestation, making this information verifiable. Secondly, we include a record of the outcome of how the builder resolved each dependency. Together, these two measures eliminate transitive trust in software dependencies, by enabling users to perform verification of transitive dependencies independently, and against their own criteria, at time of use. Finally, we explain how our proposed extensions could theoretically be implemented in Nix in the future.

CCS Concepts

• **Security and privacy** → **Software security engineering**; Trusted computing; • **Software and its engineering** → *Maintaining software*; Open source model.

Keywords

supply chain security; deterministic build; reproducible build; trustworthy build; verifiable build

ACM Reference Format:
Martin Schwaighofer, Michael Roland, and René Mayrhofer. 2024. Extending Cloud Build Systems to Eliminate Transitive Trust. In *Proceedings of the 2024 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3689944.3696169

SCORED '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1240-1/24/10
https://doi.org/10.1145/3689944.3696169

1 Introduction

1.1 Motivation

Securely building, deploying, and maintaining software packages, which practically always, but not always visibly, have deep hierarchies of dependencies, is not a solved issue. In these deep hierarchies we must not only consider direct dependencies, but also

- transitive dependencies, including build tools like compilers,
- as well as the hosts building these dependencies,
- and so forth, recursively.

This is necessary to guard against backdoored dependencies, including toolchains [19]. Sadly, in practice we often have to optimistically trust build hosts with dependency specification and resolution, and trust the software stack of the build hosts themselves. As a consequence, our ability to react to security issues deep within these hierarchies is severely lacking. See Figure 1 which illustrates the concept of transitive trust.

As part of our efforts to prevent a compromise in the supply chain we have to consider not only

- *intended dependencies*, meaning dependencies specific to the build step in question, which are deliberately declared or installed, and intentionally accessed as part of the build step, but also
- *inadvertent dependencies*, meaning the remaining software stack running on the build host, which is not supposed to influence the build output, but can absolutely do exactly that, potentially with malicious intent [2]. Examples of inadvertent dependencies include the operating system kernel, device drivers, or background services or developer tools that could potentially modify the source code or tamper with outputs.

Cloud build systems [15], like Nix [8], are conceptually powerful enough to keep track of build dependencies, like compilers, all the way back to a bootstrapping process [6]. Since they execute each build step in a hermetically isolated environment (see section 3.2), which only contains the intended dependencies, they segregate intended and inadvertent dependencies by design. Some cloud build systems were created and are deployed at a number of very large organizations, like Google's Bazel, while others are widely used by enthusiasts, for example the Nix [8] and Guix [7] open source projects.

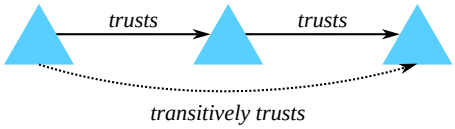

Figure 1: Transitive trust illustrated

The data structure at the heart of a cloud build system is a map, with each entry mapping an input hash, a hash that is specifically constructed to identify the set of build inputs (intended dependencies), to a content hash of the build output. This data structure, which we will call a *trace map*, enables the core feature of cloud build systems: sharing of build results between hosts, provided they have a trust relationship. This works well in centralized ecosystems, where users tend to trust one or at most a few central caches, which constitutes a verifiable build through a trusted build service [1, Chapter 14]. Some existing tools use cryptographic signatures [4, 17] or transparency log entries [21] as the cryptographic basis for this trust relationship, enabling users to consume build outputs from build hosts or binary caches they designate as trustworthy by public key. In more decentralized ecosystems, we would want users to be able to distribute trust among a set of independently maintained build infrastructure deployments. Not only do we want the ability to trust multiple parties, we want to be able to limit the degree of trust which needs to be placed in individual trusted parties. To achieve this, instead of just obtaining dependencies from any one of those deployments, if individual build steps are reproducible [11], we can mandate that a number of trusted build hosts agree about expected build outputs, as a mitigation against some of them being compromised [21]. This constitutes a verifiable build through a compromise between reproducibility and trust in the build service [1, Chapter 14].

However, cloud build systems do not record inadvertent dependencies in any way, which means trusting a specific build host implies trusting its software stack including its configuration, which is everything it trusts, still making trust a transitive relationship. This is not specific to cloud build systems; it is how build infrastructure behaves [12, 22], without serious effort and tools dedicated to transparency.

1.2 Contribution

We extend cloud build systems, as described in section 3, to eliminate transitive trust relationships by replacing them with verifiable evidence, in order to better support distributed ecosystems built on top of them. We do this by adding metadata as part of each cryptographically secured trace map entry, which already establish trust in the relationship between sets of build inputs and the build output produced from executing each build step. Aggregating and verifying this metadata by trace map entry can then serve as evidence for trusting or distrusting specific trace map entries and how they are linked.

Specifically, in section 4, we propose two significant extensions to this existing data structure:

build host provenance We initially propose a minimal addition, which effectively adds a few indicator values, which represent whether the signing entity claims to have built the output itself, and explain the advantages of this addition. We subsequently propose a much more elaborate scheme, where we add two things. First, a reference to the build step, which generated the software stack and configuration that the build host claims to have been booted with. Second, a remote attestation [5] which serves as verifiable evidence that this is indeed the case. We explain the additional benefits and implications of this more elaborate scheme.

dependency resolution Input hashes, depending on how they are constructed, effectively serve to record either the input or outcome of a kind of dependency resolution. We explain why to get rid of transitive trust it is always necessary to have a record of the outcome of this dependency resolution. We also explain why it can be practically useful to have a record of the input to dependency resolution and the ability to look up trace map entries by this key.

The goal of these changes is that users can freely define who and what they consider trustworthy at verification time, independently of whom and what the build hosts upstream from them consider trustworthy. We present a threat model of the relevant threats, in section 5, to establish that our extensions mitigate them.

In section 6, we finally use Nix as an example to illustrate the application of what we outlined above, by showing how Nix and its existing built-in protocols and data structures for caching could be modified to incorporate our proposed extensions. Since we did not write an implementation of this proposal, we only discuss it on a theoretical basis.

2 Related Work

In this section we will introduce other build-process related supply chain security tools, in the form of Gitian and in-toto as a point of comparison. Instead of discussing existing cloud build systems as related work, we introduce them in detail in section 3.

2.1 Gitian

Gitian [10] tackles the problem of source to binary verification by making the build process of a software package reproducible. Until 2021 it was the build system of choice for the Bitcoin Core software, which powers the individual nodes making up the Bitcoin network. Gitian uses virtualization to provide a build environment that is tailored towards reproducibility. All the contents of the build environment are measured and recorded by hash as inputs to the build process into an `.assert` file. After the build process, the `.assert` file contains hashes of all inputs as well as a hash of the produced output. The `.assert` file can be signed using public key cryptography and published. An independent party would start the build process from a Git commit of the project source code, identified by its corresponding hash. Executing the same build process through Gitian, they should end up with the same output as well as the same `.assert` file, describing the same initial state and outcome, provided that the build process is actually reproducible. If the same input hashes lead to a different outcome, the build was not reproducible.

Gitian is limited in the sense that it only manages a single build step. The inputs that it measures are actually hashes of installed Debian packages, which need to be trusted, or audited, in order to trust the output produced by Gitian. Effectively, we have moved the problem of reproducibility upstream by one build step. We no longer need to trust the binary output, but we need to trust every binary that is part of the build environment. One standout feature of Gitian is that it executes the build step it manages inside a virtual machine. This means that a build process which wants

to compromise the host would have to break the isolation of the VM. In practice, this also increases the number of tools that we need to trust, in order to replace our trust in the build output with verification through reproducibility.

To move past the limitation of only being able to model one build step and depending on binary packages for dependencies, the Bitcoin Core project moved from Gitian to Guix [7, 9] with release 22.0 in 2021. As a cloud build system, originally forked from Nix, Guix makes it possible to reason about the reproducibility of multiple build steps (see section 4.2.3). Guix is the kind of system we propose to extend.

2.2 in-toto Framework

In-toto [20] is a supply chain security framework, which aims to verify a supply chain across a number of build and deployment steps. Initially, the project owner creates a software supply chain layout, which contains a machine-readable description of all the intended build steps to be verified. In-toto allows for modeling a distributed build and deployment pipeline, where specific parties are entrusted with performing different steps in the overall process. Verification in this context means verification that each build step was executed by some specific party, to which the permission to execute the build step was delegated in the layout. In order to create a record of build step execution, the executing party either invokes `in-toto-record start` and `in-toto-record stop` before and after a build step, or invokes the build step through `in-toto-run`. In either case, in-toto produces a cryptographically signed record which links inputs, outputs, as well as the executing party by their signing key. A layout can be verified by verifying that the inputs and outputs in the cryptographically signed records of build steps form a chain without any gaps, match up with the layout, and are signed with authorized keys. In contrast to Gitian, this approach does not place any constraints on the execution environments of build steps. This leads to greater flexibility in terms of being able to model build steps as they are currently executed in existing deployed environments.

Aside from all the build steps that are modeled in the layout the author could list everything that is present in these existing environments and could influence the production of outputs as additional inputs, to the desired degree. However, fitting a complete description of each build environment into an in-toto layout description ahead of time would be impractical. To close this gap, there is an in-toto Attestation Framework [3] which allows for adding arbitrary authenticated metadata to the description of individual build steps. This way, a more complete description of the build environment can be recorded and verified against policies, without making the layout itself impractically rigid. The in-toto Attestation Framework fills a similar role to what we propose in section 4.1.2. Overall, in-toto and cloud build systems with our proposed extensions use very different mechanisms to distribute the responsibility for various parts of the supply chain. While with in-toto the supplier exercises authoritative control over this distribution of responsibility, in our proposal consumers independently decide which parties they trust with no control over which party is allowed to take over which responsibility.

3 Cloud Build Systems

In this section we will introduce cloud build systems [15], based on their original description. We will introduce terms for some parts of their definition and introduce their defining characteristics. This will be useful in section 4, when we extend their definition to encompass more of what we would want from a supply chain security tool.

The class of build systems called cloud build system combines

- looking up build output by an identifying hash value, and
- hermetically isolating build steps so that only dependencies that are included in the identifying hash value can affect the build output.

Members of this category of build systems as originally identified [15] are: Bazel, CloudBuild, Cloud Shake, Buck and Nix.

3.1 Output Lookup by Hash

Like in other build tools, all the individual build steps that are required to produce a desired output form a tree that represents the dependency tree of the project. Instead of invalidating out-of-date artifacts produced by each build step, as for example Make does it, cloud build systems use specifically constructed hashes to identify build steps by an exhaustive list of their inputs.

Definition 1. *Cloud build systems construct a **dependency tree** in which each node is identified by a content or input hash.*

Definition 2. Terminal inputs, *which are leaves in the dependency tree, for example source files or binary blobs, are referred to by **content hash** (a hash of their contents).*

Definition 3. *The inner nodes of the dependency tree are build steps, which are always identified by **input hash** (a hash of their input set). Sometimes they are additionally also identified by a content hash to enable extra features (see note 2).*

Definition 4. *The **input set** of a build step consists of the build instructions that get executed during the specific step, including either*

(a) the content hashes of the outputs obtained by building all direct dependencies, or
(b) the input hashes of all dependencies, including transitive dependencies via recursion.

*The two tracing approaches are known as using **constructive traces** (a), and **deep constructive traces up to terminal inputs**[1] (b).*

Figure 2 illustrates how the terms defined above are related.

[1]Technically, limited recursion depths of $n \in \mathbb{N} \mid n > 1$, can be used to create deep constructive traces up to depth n. Since this is not a popular design decision, we will not discuss such traces explicitly, but the term *deep constructive traces* includes them.

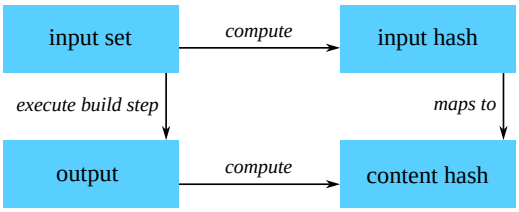

Figure 2: Cloud build system terms

Based on either of the two presented constructions,

- the input set of a build step contains all the information required to execute it and
- hashing the input set of a build step produces its input hash.

Cloud build systems always use hashes to look up and store outputs by the exact inputs that define the build step that produced them.

The supply chain security benefit from output lookups by hash is that they

- make modifications of build inputs inconsequential, because they also affect the input hash, and
- make modifications to produced and distributed artifacts detectable, by comparison against the expected build output as indicated by the lookup.

Note 1. *Most cloud build systems do not require build steps to be reproducible, so the execution of a build step with a specific input hash could lead to outputs with many different content hashes.*

3.2 Hermetic Isolation

In the previous section we established that cloud build systems use input and content hashes to store and look up outputs by the exact inputs that produced them. Many build tools, like Rust's `cargo`, also use this approach to store intermediary outputs in a local cache.

The distinguishing feature of cloud build systems compared to this is the possibility of relying on remote systems to perform parts of the build. Those specific parts are

(1) executing build steps on remote builders by submitting their input set as build jobs and
(2) obtaining outputs of build steps from remote caches by their input hash, which the requester generates from the input set.

This necessitates that users trust those remote systems or are able to verify their actions, so that attackers cannot tamper with other users' build outputs, for example to add a backdoor.

Definition 5. *In a cloud build system an attacker must not be able to bypass the isolation of executed build steps with adversarial inputs.*

We can achieve this by suitably isolating the execution of build steps from the system, each other, and the network. We will refer to such an isolation mechanism as ***hermetic isolation****.*

The addition of the word suitable in the above definition highlights that we need to be careful about sandbox escapes. The responsibility to uphold hermetic isolation lies with each individual builder. A builder which enforces hermetic isolation of build steps can be trusted by different users, because those users cannot interfere with each other except for denial of service and resource exhaustion. Cargo, for example, does not have this property, because it has an escape hatch out of its regular dependency management, in the form of `build.rs` files, which allow for arbitrary code execution without any sandboxing. Some might argue that the build steps for Rust code in Cargo are lightly sandboxed, with a compiler bug that leads from compiling Rust code to code execution constituting a sandbox escape. We do not consider this sufficient sandboxing for Cargo to support the sharing features described above, and meet the definition of a cloud build system. In supply chain security terms, hermetic isolation prevents tampering during the build.

Note 2. *Unrestricted network access could lead to the results of a given build steps being affected by its interactions with the network. This needs to be prevented to preserve the completeness of the input set, but not in any case. Downloading specific files can be allowed, if an extra content hash of the downloaded output ensures that their contents are treated as a terminal input by downstream build steps. This is the extra feature we had in mind in definition 3.*

4 Extending Cloud Build Systems

4.1 Builder Attribution and Provenance

Builders compute the build output produced from a given input set by executing the build step it describes.

Definition 6. *The central data structure underlying any cloud build system is a map from input hashes to build outputs. Often the build output is included in this map in the form of a content hash. We will refer to this data structure as a* ***trace map****.*

The reliability of the information of this map depends on hermetic isolation, which we cannot assume all builders to (successfully) enforce.

If a specific deployment of a cloud build system were using centralized infrastructure for builds and caching, we could choose to trust the maintainers of that infrastructure. We will instead consider a distributed scenario, where such infrastructure is deployed independently by various persons and organizations, and users pick and choose who they want to trust when obtaining cached outputs or delegating build steps.

To facilitate a decentralized ecosystem, users need to be able to attribute trace map entries to the parties which produced them by executing build steps, so that they can pick and choose which builders they trust to uphold hermetic isolation (see Threat 1). Any additional attributable evidence is also especially useful in such distributed ecosystems, where it is more difficult to obtain reliable information via informal channels.

Definition 7. *To attribute the outputs of build steps to builders we can*

- *supplement trace map with provenance data about the builder and a specific execution of a given build step, and*
- *protect the integrity of the resulting data with public key cryptography (signature or transparency log entry),*
- *with the persons and organizations responsible for the builder holding the private key.*

We will refer to such data as a ***provenance log****, with each entry consisting of input hash, content hash and provenance data.*

The user might be able to configure the cloud build system on the following basis. For a given build step that has been executed multiple times or by multiple builders:

- The system can choose to trust one of them (pick one provenance log entry for its trace map).
- The system can expect a certain subset of them to be in agreement (multiple provenance log entries lead to the same trace map entry).

Such a set of rules about who is trusted constitutes an example of a *trust model*. See section 4.2.1 for a general introduction of the term.

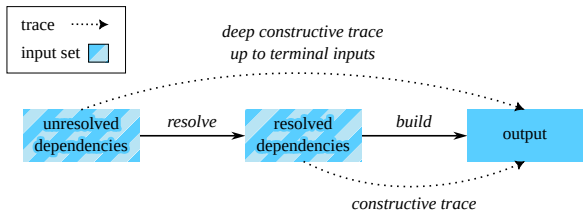

Figure 3: Tracing method comparison

Cloud build systems can verify the provenance data attached to each log entry according to their user's trust model and subsequently use successfully verified trace map entries.

Provenance logs should enable builders and the consumers of their outputs to have independent trust models. To enable this, we need to take a more detailed look at the content of the trace maps themselves first, before moving on to the other contents of provenance maps.

4.1.1 Comparing tracing methods. A trace map entry based on constructive trace determines the identity of direct dependencies by content hash, so that the corresponding provenance log entry is cryptographic evidence of exactly one specific build operation starting from its direct dependencies, which corresponds to one specific node in the dependency tree.

A trace map entry based on deep constructive traces up to terminal inputs only determines the identity of terminal inputs by content hash, while the identity of any intermediary outputs, including direct dependencies of the build step in question, is determined by its input hash. Which content hash the input hash of a direct dependency gets resolved to, ahead of the execution of a build step, depends on the trust model of the builder for both schemes. Figure 3 illustrates both tracing methods.

For deep constructive traces up to terminal inputs only the input of this dependency resolution process is represented in the trace map entry, for constructive trace map entries only the outcome is represented. As a consequence assembling a dependency tree from deep constructive traces can lead to Frankenbuilds [15], where a direct dependency of a cached output, obtained from another cache, only equals the direct dependency that was used during the build in terms of input hash, but not in terms of content hash. One of the possible reasons for such a difference is that the trust model of the incorporated builder is incompatible. To exclude this possibility, we have to rely on constructive traces to validate that every link in the dependency tree satisfies the trust model of the verifying party. If we want to eliminate transitive trust, we can not place trust in the mappings from deep constructive traces to constructive traces or content hashes created by other parties.

Systems built on constructive traces have the opposite problem in relation to transitive trust. Since they fetch dependencies based on the result of dependency resolution, which is necessarily trust-model specific, differences in trust model lead to requesting different dependencies, which can lead to cache misses. The end result of dependency resolution needs to be a single result, but there might be different specific outputs which would fulfill all criteria that are mandated by a specific trust model.

We therefore propose to use any suitable lookup mechanism to discover provenance log entries, including (untrusted) deep constructive traces up to terminal inputs. The provenance log entries are filtered down to only those who satisfy the trust model. Constructive trace based trace map entries are then extracted from the provenance log entries, and used to construct one dependency tree from all possible links. See section 6.3.2 for a proposal of how this could work to mitigate Threat 3 in Nix. As usual with cloud build systems, if building locally is allowed, any build steps where no suitable cached output is available are performed locally.

4.1.2 Provenance data. The role of the provenance data and the public key cryptography ensuring its integrity and binding to a specific trace map (entry) is to increase the trustworthiness of the trace map (entry), so that its consumers do not have to blindly trust, but can evaluate or verify claims about builders in accordance with the criteria they define, in addition to their trust in the signing keys of builders.

Just like the implementation of existing cloud build systems assumes responsibility over accurately mapping inputs to outputs, an implementation of the extensions described in this section should assume responsibility over accurately reporting provenance data out of the box, so that false provenance data would constitute either an implementation error or a malicious change to the cloud build system.

Other than this goal of accurate data reporting out of the box, it is not necessary to standardize or predefine the contents of the provenance data. Instead, builders can innovate about what additional data to include and how to make it trustworthy, while at the same time users decide which elements they verify, and how. Ideally provenance logs are verifiable using remote attestation [5]. See section 6.3.3 for a proposal of how this could work to mitigate Threat 4 in Nix.

4.2 Verifying Build Steps

The definition of verifiable builds [1, Chapter 14] bases them on trusted builders, reproducibility, or a mix of those two concepts. For each build step a provenance log entry associates

- a trusted origin system, as identified by a public key, with
- provenance data that constitutes potentially verifiable evidence, moving this trust relationship towards verification, and
- a trace map entry from input hash to content hash.

4.2.1 Trust Model. We always define verification in terms of some trust model, which so far we have not defined rigorously. A trust model is defined at minimum by a set of trusted public keys and in which combinations they are trusted (e.g., requiring agreement from multiple specific keys or accepting any one key from a set). The Trustix project [21] already provides this functionality. We extend this concept, so that a trust model might additionally include a number of constraints on the provenance data attached to each provenance log entry that is signed with such a public key. Such constraints might for example require an attached remote attestation to be verifiable in a specific way, or that a specific package in a specific version is not present on the builder. Some of these criteria might be verified recursively for a builder, the packages

installed on said builder, the builders those packages were built on, and so forth. One example where this could be useful is for excluding builders, which depend on libraries with high impact security issues discovered after deployment, like some recent versions of XZ Utils which had been backdoored [13]. Eliminating transitive trust also means that the trust model can be updated at any time, based on new information, and compliance to the new trust model can be re-evaluated (see Threat 5).

4.2.2 Reproducibility. We choose not to define reproducibility as the successful (on-the-fly) repetition of a build process with the same outcome, which is quite a fleeting observation. Instead, we assume provenance log entries always act as a persisted and shareable intermediary format, which decouples building, a computationally expensive, trust-model independent process, from making the computationally cheap determination that this process has been repeated sufficiently in accordance with a trust model.

Provided the cloud build system records repeated execution of build steps in the provenance log, this definition works locally as well as for large numbers of potentially trustworthy builders (see Threat 2).

4.2.3 Bootstrappability. Since verifiability, through reproducibility or other means, is determined on a build step level, it can be applied to each step in the dependency tree, covering the tools and dependencies which make up the build environment, all the way up the tree to a set of bootstrapping binaries. In the case of verification through reproducibility, verification of each step ensures that the build process is considered a bootstrappable build [6]. While this concept is known, our other proposals make its verification more efficient.

5 Threat Model

In this section we will demonstrate how our proposed extensions mitigate a number of threats to users of cloud build systems. Broadly speaking, threats can be mitigated using our proposed extensions by users moving towards

- distributing trust among a set of trusted signers[2], in an open ecosystem which also contains untrustworthy signers,
- relying on signatures by builders instead of caches in order to establish build provenance and extend transport security end-to-end, and
- verifying instead of trusting builders.

Threats are numbered T1 to T5, and their mitigations are numbered M1 to M5. For this threat model we will take the perspective of a specific user, who builds a specific project, giving the cloud build system the opportunity to obtain the outputs of some build steps from caches. We assume that the cloud build system is used in such a way that the checked-in source code of the project identifies specific versions of each dependency back to a set of bootstrapping binaries.[3] We assume that the user invokes a trusted instance of the cloud build system in a trusted environment and configures it with a trust model that reflects their trust relationships. Outputs produced based on an incompatible trust model must be assumed to not be trustworthy. On other trustworthy builders, hermetic isolation ensures that the invocation of malicious build instructions can have no side effects, outside the output of the specific build step. This and the fact that our stated goal is to build a project, not run it, places the source code and binary blobs that are part of the project build out of scope of the threat model. The cloud build system itself and other software running on builders might also be vulnerable or malicious. While the discovery of such issues is out of scope of our threat model, Threat 4 and Threat 5 aim to prevent and allow recovery from placing trust in untrustworthy builders.

There is no technical distinction between the capabilities of different actors in the modeled distributed system. Users are running an instance of the cloud build systems, trustworthy builders are running the same cloud build system, possibly with minor distinctions. Caches can be passive, effectively HTTP proxies, as long as they do not produce their own signatures, and have to run the cloud build system code that enables signing otherwise. We are taking on the perspective of the user, as the user of one instance of the cloud build system, trying to establish trustworthiness of other signers across intermediaries.

The asset under consideration are trace map and provenance log entries, which protect the integrity of the relationship between input sets and outputs. Availability and secrecy of build outputs are placed out of scope. Availability of build outputs can be regained via rebuilding[4], which is in fact how participants in the ecosystem can migrate to stricter trust models over time to protect integrity. Secrecy of input sets is poorly supported in cloud build systems, and while it might warrant further discussion it is a concern which is completely orthogonal to the extensions we propose. Availability of input sets is not a goal of cloud build systems.

T1 Even if a trusted cache signs a trace map entry, this does not prove to the user which entity executed the build step, and therefore who the user trusts with upholding hermetic isolation. The original builder might not be trustworthy, or cache contents might have been tampered with before signing. **M1:** Through the provenance log entry a signer can claim to have executed the build step and this signature can be passed on to caches. The user can decide to trust the builder instead of or in combination with the cache, adopting a stricter trust model.

> Note 3. *The cache might adopt a stricter trust model, by only signing artifacts that are already signed by an origin that is on the cache's allow list. This is however only a variation of the exact problem we are trying to resolve, as the cache's allow list reflects its trust model, which then leaks into the users trust model. The significant issue with such a setup, which we are trying so solve, is that there is no way to later revoke trust in specific members of the allow list. We think caches should either only distribute builder signatures, or use cryptography and protocols, which allow for revocation.*

[2] Builders and caches can both act as signers, but only builders have first-hand knowledge of the build process.

[3] One way to achieve this is to use a cloud build system which supports `.lock` files, like for example Nix with the `flakes` experimental feature.

[4] This has been demonstrated in practice using Nix [14], with caveats around bit-for-for bit reproducibility, which would often take more upfront effort to achieve.

T2 Since builders might trust each other directly or transitively[5], two trusted builders signing the same trace map entry does not prove that the corresponding build step has been executed by both. This makes it impossible to formulate trust models where builders are only trusted if they directly produce the same output. **M2:** Through the provenance log entry, both builders can claim to have executed the build step themselves.

T3 If the cloud build system is based on deep constructive traces, an output produced by a builder with an incompatible trust model might incorporate untrusted dependencies. Since trace map entries do not contain resolved dependencies, it is impossible for the user to detect this issue. **M3:** The builder can add the content hashes of all direct dependencies to the provenance log entry. The cloud build system of the user can then require the dependency tree of the project to be valid on this basis, which is a constructive trace basis, eliminating the issue for trustworthy builders with differing trust models.

T4 A builder might be breaking hermetic isolation, by running insecurely configured or vulnerable software, which could allow others[6] to bypass hermetic isolation or exfiltrate the signing key. A compromised builder might also be dishonest about provenance data. **M4:** The builder can include a link to the booted software state in the provenance data, make this information verifiable to the user as part of the dependency tree and keep the signing key in secure hardware. See Section section 6.3 for details.

T5 Over time the users trust model might become outdated, and they might want to adapt it. Reasons for this might be the discovery that a particular builder, or software or hardware component used by the builder, is vulnerable or compromised or simply no longer judged to be trustworthy. **M5:** The user can update their trust model accordingly. As long as the compromised component is reflected in the provenance data verified by the users trust model, for any output, we can identify if there still exists a trustworthy path from the set of bootstrapping binaries to said output. Missing links in the path can be handled via rebuilds from source, or changing the build instructions to depend on earlier or newer uncompromised versions of inputs, which are/were derived by verifiable uncompromised instances of the cloud build system running on uncompromised builders.

6 Applying Extensions to Nix

In this section we are first going to explain how caching of build outputs and signing works in Nix. Then we will outline how our proposed extensions would apply to the integrity verification of build outputs that are cached by the Nix package manager[7].

[5]Nix is one example of a cloud build system which allows this, and we make the same assumption in this threat model.

[6]This could be anyone on the web, anyone running other build jobs, or an administrator of the builder.

[7]We assume that Nix was installed using the installer available at https://zero-to-nix.com/start/install, because this installer enables the `nix-command` and `flakes` experimental features out of the box, which includes a CLI that offers better reproducibility.

6.1 Build Outputs and Caching

As an illustrative example of how caching works in Nix, we will look at obtaining a specific version of the GNU Hello program packaged using the Nix language.

The exact version of the package is defined by a specific git commit hash in the https://github.com/NixOS/nixpkgs repository on GitHub, which contains the `hello` package and all of its dependencies back to a set of bootstrapping binaries[8].

We can build this exact version of the package with the following command that includes a unique prefix of the desired git commit hash.

```
$ nix build nixpkgs/4f807e8940284ad7#hello
```

When this operation successfully terminates a symlink with the name `result` linking to the build results will appear in the current directory.

```
$ readlink result
/nix/store/yb84nwgvixzi9sx9nxssq581pc0cc8p3-hello
    ↪ -2.12.1
```

This symlink points to the build output, stored at a read-only path inside the `/nix/store` directory. Nix incorporates input hashes into the storage paths of build results, using a hashing scheme that is based on deep constructive traces up to terminal inputs.

Note 4. *Derivation is the Nix-specific term for the input set which defines a build step. Since content and input hashes determine storage paths in Nix, they are also referred to as addressing schemes. More specifically, those schemes are called input addressing and content addressing. Since build recipes in Nix always refer to their inputs by store path, switching the inputs of derivations to content addressing is the same as changing the hashing scheme to one based on constructive traces. There is an experimental feature in Nix called* `ca-derivations` *which does this.*

As a cloud build system Nix might have produced the build output in various ways, depending on which caches it considers trustworthy. It might have re-built the full dependency tree up until and including the `hello` package from source. When terminal inputs were required, it would have placed them within subdirectories that were named based on their content hash. Alternatively, it might have successfully looked up just the `hello` package and its runtime dependencies from https://cache.nixos.org, which is the cache configured by default, or from any other cache. A mix between the two options, where some build outputs were obtained from a cache and others were built locally, might have also occurred. In any case, as long as none of the involved caches are malicious, the build results have been obtained by executing the same build step and are in that sense semantically equivalent and Nix therefore considers them semantically equivalent. This kind of equivalence however cannot distinguish between a legitimately produced output and one where a transitively trusted malicious builder has led to a malicious direct dependency being incorporated into the build

[8]The file which defines the `hello` package at the exact commit in question is available on GitHub at https://github.com/NixOS/nixpkgs/blob/4f807e8940284ad7925ebd0a0993d2a1791acb2f/pkgs/by-name/he/hello/package.nix. The dependency tree of the `hello` package consists of 93 build steps and 155 terminal inputs.

undetected, and is therefore too weak for our purposes (see Threat 3).

6.2 Unaltered Signature Verification

For each cache lookup, like for the `hello` package itself, all configured caches are queried by input hash, by looking up a specific URL[9].

In case of a cache hit, Nix finds the metadata from Listing 1 about the build output in question in the `.narinfo` file available at this URL. A NAR-File (Nix ARchive) is a Nix-specific archive file format, which is used to transmit the actual build output. It is available at the URL listed in the `.narinfo` file. The NAR format also serves as the basis for the content hashing scheme that Nix uses. Effectively, NAR-Hashes are the specific kind of content hash that Nix relies on.

Many of the metadata from the `.narinfo` file (Listing 1) are combined in the detached signature scheme Nix uses to protect its trace map entries. Each entry in the map is secured by a detached signature over a fingerprint, containing various data. Listing 2 shows the fingerprint calculation.

The store path represents the input hash which identifies the build step, the NAR-Hash represents the corresponding content hash of the produced build output. The reference set is the set of runtime dependencies, meaning all other store paths that are contained in the output as strings. Additionally, the leading 1 allows for changes to the fingerprinting scheme, which we will propose.

In conclusion, we note that the signature does not contain information about the signing party and the signing key is also not associated with this kind of information in any other way.

6.3 Proposed Changes

As mentioned above, we can create a newer version of the fingerprinting scheme by changing the leading 1 to a 2. Since the `.narinfo` file format supports a list of signatures, we can add one signature using the new scheme alongside another signature using the old scheme, for backwards compatibility with existing Nix installations.[10] In the new scheme, we can then add additional data to the updated fingerprint. Effectively this gives us the opportunity to add more data which the producer and consumer of a cached output have to agree about. We can communicate the expected values for these additional fields as additions to the `.narinfo` file. In line with our more generic proposals we want to do this with two pieces of data. This additional data turns our signed trace map entries into provenance log entries.[11]

6.3.1 Claimed origin information. First, we add a single `origin` enumeration with a number of predetermined values to the fingerprint.

Those values in increasing order of trustworthiness are:

"unknown" The signer did not build this output itself, so it has no first-hand knowledge of its origin. The signature provides transport security starting at the signer. Note that a different signature by the original builder indicating first-hand knowledge may be passed along in addition to this one.

"trusted" Additionally to the properties described in **unknown** the signer considers this output trusted.

"builder-according-to-db" The Nix database of the signing builder indicates that it has built the output itself (this uses existing database records of build results for legacy support).

"builder-signature" This signature was created directly by the builder immediately after the build.

This additional information makes two things possible, which were not possible before:

(1) We can choose to only trust build outputs for which we can obtain a signature that states that it was built directly by a trusted builder (see Threat 1). This guarantee is still weak, because we have no evidence of the software stack of the builder, but at least the claimed origin is clear.
(2) We can determine that a build step is reproducible, if we can obtain two different signatures of the same trace map entry, which are both marked as **builder-signatures** by two different keys we consider trustworthy (see Threat 2).

6.3.2 Verify constructive traces. Nix uses deep constructive traces up to terminal inputs. The input address inside the unmodified signatures is therefore ambiguous about the identity of direct dependencies in terms of content hash, since they are only identified by their own input hashes (see Threat 3). To resolve this ambiguity we add a list with the content hashes of direct dependencies to both the fingerprint and `.narinfo` file. We define the order of the list in a way that allows us to match up corresponding input hashes and content hashes, for example by picking the same order of appearance that is used to compute the input hash.

This additional information makes it possible to reject cached outputs during dependency resolution, if the content hashes listed for all direct dependencies do not match the content hashes of trusted outputs, ensuring that the full dependency tree is valid in terms of constructive traces and therefore does not introduce implicit transitive trust in potentially untrusted third parties.

6.3.3 Evidence of origin. We can additionally add a reference to the build step, which builds the software stack that is running on the builder itself, to each provenance log entry. This reference cannot simply be an input hash, because it has to contain enough information to reconstruct the dependency tree of the builder's software stack. A reachable URL, pointing to a git repository, with a suffix that identifies a specific commit and build step, for potential verification, is suitable. This can use the same format as the `hello` reference we used above,[12] with the produced output being a bootable system configuration. Producing references to system configurations for the NixOS Linux distribution is already common practice in Nix, while at the same time we are proposing extensions to Nix which do not rely specifically on NixOS and Nix is capable of assembling images of other kinds of systems this way [16, 18].

[9] For our example the URL Nix looks up in order to determine a cache hit is https://cache.nixos.org/yb84nwgvixzi9sx9nxssq581pc0cc8p3.narinfo.
[10] We have not tested this.
[11] As the name suggests, a transparency log is one promising way to implement such a data structure. Nonetheless, to limit the scope of the proposed changes for this paper, we decided to propose extending the existing scheme that is based on signatures. Some technical details of our proposal are described in the following, at the time of writing open, issue we created: https://github.com/NixOS/nix/issues/9644.
[12] This format is called a *flake reference*.

Listing 1: .narinfo file

```
StorePath: /nix/store/yb84nwgvixzi9sx9nxssq581pc0cc8p3-hello-2.12.1
URL: nar/1a2rz1s7r6b7zy25dbrjxgkhpgkd4cybqch6nh56l63m9fcn4zzm.nar.xz
Compression: xz
FileHash: sha256:1a2rz1s7r6b7zy25dbrjxgkhpgkd4cybqch6nh56l63m9fcn4zzm
FileSize: 50364
NarHash: sha256:0982m132as66yjs4jcddjks1r7g9r6x50x90bim1vfr4wca3hacb
NarSize: 226560
References: 3dyw8dzj9ab4m8hv5dpyx7zii8d0w6fi-glibc-2.39-52 yb84nwgvixzi9sx9nxssq581pc0cc8p3-hello-2.12.1
Deriver: crmj28zg09517n5sskml9fmy2c6r3rsr-hello-2.12.1.drv
Sig: cache.nixos.org-1:DIsZWyTRIauN6NyAhtGxQgwo2fF3IMrAf5T+
    ↪ W1PyPZYuyO5rh4ZCEJwZWq2fNavzJYOLUcR3pC2s8NUHymikDg==
```

Listing 2: excerpt from libstore/path-info.cc file

```
std::string ValidPathInfo::fingerprint(const Store & store) const
{
    if (narSize == 0)
        throw Error("cannot calculate fingerprint of path '%s' because its size is not known",
            store.printStorePath(path));
    return
        "1;" + store.printStorePath(path) + ";"
        + narHash.to_string(HashFormat::Base32, true) + ";"
        + std::to_string(narSize) + ";"
        + concatStringsSep(",", store.printStorePathSet(references));
}
```

With this additional information the decision whether to trust a specific builder can take its claimed software stack into account.[13]

In order to make this information trustworthy, to mitigate Threat 4, we can implement a scheme for remote attestation of the builder. As a prerequisite this requires that we can derive the expected hash values for measured boot from the built OS image[14]. We also need to have some reason to trust the hash values lower in the boot chain, for example by knowing they are derived from unmodified, up-to-date hardware which we consider secure.

There are various suitable, standardized mechanisms available for different hardware platforms[15], which generate a key in secure hardware and can provide a remote attestation, which

- proves that the key was generated in secure hardware,
- incorporates measurement values from measured boot, which correspond to the built and booted OS image and
- uses the trace map entry and a monotonically increasing counter as the nonce[16].

The resulting remote attestation can then be incorporated into the provenance data of the provenance log entry in question and signed with the attested key.

When the abovementioned requirements are fulfilled and satisfy the trust model of the verifier, which may change over time to mitigate Threat 5. The verifier should then be able to

(1) verify the claimed software configuration of the builder against the verifiers trust model, which may or may not mandate reproducibility,
(2) derive the expected measurement values for measured boot from the built software configuration,
(3) verify the signature on the provenance log entry,
(4) verify from the contents of the remote attestation, that
   (a) the included trace map entries match up,
   (b) the upper level measurement values from measured boot match up, the lower level values are trusted,
   (c) the attestation itself is valid, and
   (d) the attestation is about the intended signing key.

If all of these properties can be verified, this establishes a correspondence between the booted system and the configuration it was built from.

The verifier needs to additionally determine that the configuration of the builder, which we have established as booted at the time, is trustworthy. While this might sound like another daunting task, verifiers might whitelist specific configurations, and configurations can be maintained and audited in public.

Note 5. *One important prerequisite before such a scheme could be deployed in production is that the build sandbox Nix uses for hermetic isolation has to hold. Since, if hermetic isolation is broken, the builder can make false statements about the relationship between inputs and*

[13] Build steps, including those involved in building builders, can be verified to meet specific criteria this way, including reproducibility as we define it.
[14] NixOS itself does not support measured boot yet. If the OS image can be built reproducibly we can demand that multiple builders agree about the output of each build step.
[15] Examples are a standard TPM 2.0 in a server setting, or Key Attestation on Android.
[16] The benefit of the counter is the ability to record multiple builds on the same host, even if they are reproducible. A liveness check should not be required in this application. A repeated nonce value is not a security threat in this setting, because it would just re-state the same statement.

outputs. We assume that the build sandbox would have to expertly integrate a component with a proven track record to suitably enhance sandboxing, in order to enable this use case in production.

6.4 Limitations and Adoption

We have aimed to present our approach clearly and concisely so far, leaving some open and broader questions for this section. We will mention such topics here, in a way that gives a clearer picture of how an implementation of our extensions aimed at adoption in Nix might be tackled.

The adoption of Nix and other cloud build systems, is a different issue, which we will not get into.

6.4.1 Implement provenance log entries. Builders should sign packages per default.[17]

To add provenance data, adding a single field with a base64 encoded JSON object would probably be preferable to extending `.narinfo` files with additional fields, because it makes it possible to handle unknown provenance data. This makes it easier for competing additions to provenance data to coexist, facilitating innovation and an open ecosystem.

It is not clear which cryptographic mechanism should protect the integrity of provenance log entries. The available options are the existing signing scheme, as discussed in section 6.3, or a transparency log, specifically Trustix [21]. Both options offer different security properties, because a transparency log is a complete record of executed build steps, which potentially reveals more information than a signature based design. The issue of key lifecycle and revocation might provide another relevant point of comparison for both options.

6.4.2 Changes to dependency resolution. Nix's existing dependency resolution would need to be adapted, so that it can take provenance data into account. In section 6.3.2 we assumed that missing dependency information would be added to input-addressed derivations. Another option would be building on content addressed derivations, from the experimental `ca-derivations` feature, instead. Since this feature uses constructive traces it might be the better approach. One not yet implemented, but originally proposed [8], aspect of `ca-derivations` allows Nix to rewrite references inside build outputs to increase cache efficiency. This would also have to be taken into account when reasoning about trust.

The implementation of Nix has recently added a number of internal interfaces, in order to make the project more modular. We think that a generic interface that identifies the trustworthy subset from a set of provenance log entries would be beneficial, to encourage the implementation of different verification mechanisms. We are not sure how such an interface would incorporate the references to the source code of a builder. It is our assumption, that the basic claim to be the builder would also be implemented in Nix directly, to ensure accuracy out of the box.

6.4.3 Implement trustworthy attestation. Putting the generic support described above in place could encourage various parties to implement and later open source more complex validation mechanisms, including attestation. We assume that doing attestation in a generic way that works on a wide range of deployed systems would be exceedingly difficult, because of all the different possible hardware and system configurations involved. Initial implementations will most likely be developed for in-house use inside very specific and controlled environments, with excellent support for hardware attestation, with solutions with broad hardware support arriving much later.

6.4.4 Open questions about usage. Other open questions exist around the usage of such a system

(1) What does its bootstrapping look like?
(2) How costly is the generation and validation of evidence computationally?
(3) How can the sandbox providing hermetic isolation be improved to the required degree?

7 Conclusion

Our introduction into cloud build systems provides the necessary terminology to discuss how they function in general and in terms of specific design considerations like trust. We analyze what data cloud build systems can use to establish trust in cached outputs, and how this process can involve transitive trust in third parties. With the specific use case of distributing trust across an open ecosystem in mind, we

- eliminate implicit transitive trust relationships that exist in systems based on deep constructive traces,
- link provenance data to build steps, in order to attribute their outputs to those build hosts, which originally executed them,
- make implicit transitive trust in the software stack of build hosts explicit and independently verifiable, via remote attestation, instead.

Our proposed scheme adds what we call a *provenance log*, which

- is motivated by the included threat model, and
- decouples the creation of provenance data and its verification as much as possible, to facilitate the creation of an open and evolving ecosystem, which can accommodate participants with a diverse set of trust models.

Finally, we explain how what we propose could be implemented in the Nix package manager in the future.

We do not see this work only as an innovation in the space of cloud build systems, but as a potential path forward for supply chain security, building and managing provenance data into one unified tool, which could offer quite a principled and elegant solution for real problems that exist in our current software supply chains. The open questions about the implications of this warrant both further research and the creation of testable implementations, and therefore, we hope that the build system and supply chain security communities will help us answer them.

Acknowledgments

We want to thank Linus Heckemann for first pointing out to us the gaps around the outcome of dependency resolution, which exist in Nix, and which we address in this paper.

[17] As of right now they do not, since signatures are primarily used as a mechanism to communicate that a cache considers a trace map entry trustworthy and provide transport security, not to attribute packages to the original builder. At the time of writing, there is an open issue about generating signatures per default: https://github.com/NixOS/nix/issues/3023.

This work has been carried out within the scope of Digidow, the Christian Doppler Laboratory for Private Digital Authentication in the Physical World and has partially been supported by the LIT Secure and Correct Systems Lab. We gratefully acknowledge financial support by the Austrian Federal Ministry of Labour and Economy, the National Foundation for Research, Technology and Development, the Christian Doppler Research Association, 3 Banken IT GmbH, ekey biometric systems GmbH, Kepler Universitätsklinikum GmbH, NXP Semiconductors Austria GmbH & Co KG, Österreichische Staatsdruckerei GmbH, and the State of Upper Austria.

References

[1] Heather Adkins, Betsy Beyer, Paul Blankinship, Piotr Lewandowski, Ana Oprea, and Adam Stubblefield. 2020. *Building Secure and Reliable Systems.* O'Reilly Media.

[2] Rahaf Alkhadra, Joud Abuzaid, Mariam AlShammari, and Nazeeruddin Mohammad. 2021. Solar Winds Hack: In-Depth Analysis and Countermeasures. In *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)* (Kharagpur, India). IEEE, 1–7. https://doi.org/10.1109/ICCCNT51525.2021.9579611

[3] Attestation Project Contributors. 2024. attestation: in-toto Attestation Framework. https://github.com/in-toto/attestation 2024-07-03.

[4] Attic Project Contributors. 2024. attic: Multi-tenant Nix Binary Cache. https://github.com/zhaofengli/attic

[5] Henk Birkholz, Dave Thaler, Michael Richardson, Ned Smith, and Wei Pan. 2023. RFC 9334: Remote ATtestation procedureS (RATS) Architecture. https://doi.org/10.17487/RFC9334

[6] Bootstrappable Builds. 2024. Bootstrappable Builds Website. https://bootstrappable.org/

[7] Ludovic Courtès. 2022. Building a Secure Software Supply Chain with GNU Guix. *The Art, Science, and Engineering of Programming* 7, 1, Article 1 (June 2022). https://doi.org/10.22152/programming-journal.org/2023/7/1

[8] Eelco Dolstra. 2006. *The Purely Functional Software Deployment Model.* Ph. D. Dissertation. Utrecht University.

[9] Carl Dong. 2019. Bitcoin Build System Security. Talk at Breaking Bitcoin 2019 Amsterdam. https://www.youtube.com/watch?v=I2iShmUTEl8 (accessed 2024-04-12).

[10] Gitian-Builder Contributors. 2024. gitian-builder: Build packages in a secure deterministic fashion inside a VM. https://github.com/devrandom/gitian-builder

[11] Chris Lamb and Stefano Zacchiroli. 2021. Reproducible Builds: Increasing the Integrity of Software Supply Chains. *IEEE Software* 39, 2 (2021), 62–70. https://doi.org/10.1109/MS.2021.3073045

[12] Clement Lefebvre. 2016. Beware of hacked ISOs if you downloaded Linux Mint on February 20th! The Linux Mint Blog. https://blog.linuxmint.com/?p=2994

[13] Mario Lins, René Mayrhofer, Michael Roland, Daniel Hofer, and Martin Schwaighofer. 2024. On the critical path to implant backdoors and the effectiveness of potential mitigation techniques: Early learnings from XZ. arXiv:2404.08987 [cs.CR]

[14] Julien Malka, Stefano Zacchiroli, and Théo Zimmermann. 2024. Reproducibility of Build Environments through Space and Time. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results* (Lisbon, Portugal) *(ICSE-NIER'24)*. ACM, New York, NY, USA, 97–101. https://doi.org/10.1145/3639476.3639767

[15] Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. 2018. Build Systems à La Carte. *Proc. ACM Program. Lang.* 2, ICFP, Article 79 (July 2018), 29 pages. https://doi.org/10.1145/3236774

[16] Nix-Openwrt-Imagebuilder Project Contributors. 2024. nix-openwrt-imagebuilder: Build OpenWRT images in Nix derivations. https://github.com/astro/nix-openwrt-imagebuilder

[17] Nix-Serve Project Contributors. 2024. nix-serve: A standalone Nix binary cache server. https://github.com/edolstra/nix-serve

[18] Robotnix Project Contributors. 2024. robotnix - Build Android (AOSP) using Nix. https://github.com/nix-community/robotnix

[19] Ken Thompson. 1984. Reflections on trusting trust. *Commun. ACM* 27, 8 (1984), 761–763. https://doi.org/10.1145/358198.358210

[20] Santiago Torres-Arias. 2020. *In-toto: Practical Software Supply Chain Security.* Ph. D. Dissertation. New York University Tandon School of Engineering.

[21] Trustix Project Contributors. 2024. Trustix: Distributed trust and reproducibility tracking for binary caches. https://github.com/nix-community/trustix

[22] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. 2009. When private keys are public: results from the 2008 Debian OpenSSL vulnerability. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement* (Chicago, Illinois, USA) *(IMC '09)*. ACM, New York, NY, USA, 15–27. https://doi.org/10.1145/1644893.1644896