



Policy Transparency: Authorization Logic Meets General Transparency to Prove Software Supply Chain Integrity

Andrew Ferraiuolo 
aferr@google.com
Google Research
United Kingdom

Tiziano Santoro 
tzn@google.com
Google Research
United Kingdom

Razieh Behjati
razieh@google.com
Google Research
United Kingdom

Ben Laurie 
benl@google.com
Google Research
United Kingdom

ABSTRACT

Building reliable software is challenging because today's software supply chains are built and secured from tools and individuals from a broad range of organizations with complex trust relationships. In this setting, tracking the origin of each piece of software and understanding the security and privacy implications of using it is essential. In this work we aim to secure software supply chains by using verifiable policies in which the origin of information and the trust assumptions are first-order concerns and abusive evidence is discoverable. To do so, we propose Policy Transparency, a new paradigm in which policies are based on *authorization logic* and all claims issued in this policy language are made transparent by inclusion in a transparency log. Achieving this goal in a real-world setting is non-trivial and to do so we propose a novel software architecture called PolyLog. We find that this combination of authorization logic and transparency logs is mutually beneficial – transparency logs allow authorization logic claims to be widely available aiding in discovery of abuse, and making claims interpretable with policies allows misbehavior captured in the transparency logs to be handled proactively.




CCS CONCEPTS

• **Security and privacy** → **Formal methods and theory of security**; **Software and application security**; **Cryptography**.

KEYWORDS

transparency, logic programming, policies, identity, authorization, authorization logic, supply chain security, reproducible builds, deterministic builds

ACM Reference Format:

Andrew Ferraiuolo , Razieh Behjati, Tiziano Santoro , and Ben Laurie . 2022. Policy Transparency: Authorization Logic Meets General Transparency to Prove Software Supply Chain Integrity. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem*

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SCORED '22, November 11, 2022, Los Angeles, CA, USA

© 2022 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-9885-5/22/11.

<https://doi.org/10.1145/3560835.3564549>

Defenses (SCORED '22), November 11, 2022, Los Angeles, CA, USA. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3560835.3564549>

1 INTRODUCTION

Following the SolarWinds [38] attack and the more recent Log4j vulnerability, software supply chain security has become a topic of more extensive research [28]. Attackers have plenty of opportunities to deliberately introduce vulnerabilities because a single piece of open-source software is often produced by a large collection of individuals, organizations, and tools. Constructing even one software binary involves the first-party software, the compiler and linker, software analysis tools, human code reviewers, the organizations that set best-practices for writing software, and all the similar software components and participants involved in constructing the dependencies. Each one of these represents a potential point of failure in the software supply chain and an opportunity for deliberate abuse by attackers.

Fundamentally, software supply chain attacks are possible because these trust relationships involved in constructing software are implicit and unclear, the terms under which these components are acceptable to software ecosystem participants are difficult to control, and even where the source of information (such as a human review) is known, it is difficult to hold them accountable. To even understand who is really involved in software authorship, a software consumer must manually audit a repository, its tools, and its dependencies and then keep up with near-constant changes to any of these. Controlling these trust relationships is another challenge as individual stakeholders may have differing needs (e.g., software consumers want strong assurance about dependencies whereas software authors have limited resources to offer assurance). Finally, holding abusers accountable is difficult as a bad actor could hide their abusive actions.

In this paper we present Policy Transparency, a novel framework in which the trust assumptions involved in constructing software are **understandable**, the criteria for trusting evidence for accepting software is **controllable** by each stakeholder, and abusers can be held **accountable**. Policy transparency is a new paradigm for open-source software authorship in which 1) policies specifying criteria for acceptance of software and evidence for satisfying these policies are both expressed as authorization logic, and 2) the policies and evidence are both retained in transparency logs to aid in accountability.

Accountability is crucial. Merely requiring a collection of human reviews and tools to run on source code is not enough – a malicious reviewer could approve of software with a backdoor and later hide their approval to avoid detection. Transparency logs [8] are a promising, practical, and widely-adopted approach to preventing these attacks. Transparency logs implement a tamper-evident append-only log of all claims – as long as consumers of this evidence always request a proof that this evidence is in the log, there is no way to hide malicious claims. Aside from detecting information hiding attacks, tracking the history of claims made by a claim issuer disincentivizes misbehavior because of the threat of reputation harm.

By representing the criteria for approving software as an authorization logic policy, the trust assumptions in this process are made clear. Authorization logic [9, 12, 15, 22] is a style of policy language well-suited to problems in supply-chain security because its language features focus on tracking the origins of information and making trust explicit in the policy code. Policies in authorization logic are based on logic programming such as Datalog or Prolog. Logic-based policies can govern a wide range of actions such as when to trust an imported library by specifying the needed evidence to take this action with *rules*. Importantly, all facts in authorization logic are attributed to an identity called a *principal* which is any entity that can take action or produce information [32, 34]. In practice principals can represent a human, tool, machine, or others. The only way for one principal to believe information produced by another is through a syntax for *delegation*. As a result, delegations programmatically express all trust assumptions.

Policy Transparency offers more than the sum of its parts. Authorization logic can produce proof trees that show what evidence was used to make a decision – this feature compliments transparency logs because each piece of evidence is discoverable on the log and cannot be removed, facilitating re-verification of policies. Because acceptance of software dependencies is controlled by policies, we can also use logs for proactive security by writing policies that prevent use of claims by misbehaving issuers.

To build a practical system for implementing Policy Transparency, we need to address a number of challenges:

- **Compatibility with the outside world:** We cannot expect immediate adoption of our policy language, and existing tools provide outputs in their own formats. We need to interpret these outputs in our policy language without losing the trust-tracking benefits of authorization logic.
- **Revocation:** Claims that are used as credentials in real-world policies need to be revocable, for example, when these credentials were abused. Similarly policies need to be changed to meet new regulations. Because transparency logs are append-only, revoked entries cannot be removed, so we need another solution.
- **Multi-issuer transparency:** Prior applications of transparency logs such as certificate transparency typically track claims by a single category of issuer (certificate authorities) and with a single purpose (binding keys to identities). Supply chain vulnerabilities could result from failures from a range of information sources including human reviews, automated tools, or dependencies. Therefore, we need logs that can support many kinds of issuers producing claims about many topics.

- **Multi-purpose monitors:** Generalized logging of claims requires generalized detection of misbehavior, so we need a way to write monitors that can report failures from many sources.

To address these challenges, we propose a novel software architecture, PolyLog, for realizing Policy Transparency. Importantly, PolyLog utilizes authorization logic as a *policy metasystem* [20] – rather than expecting all issuers of claims to adopt our policy language directly and immediately, we provide a framework in which users of our system can convert existing information sources into facts interpretable by authorization logic and made discoverable by inclusion in verifiable logs. Our architecture proposes *Trusted Wrappers* (which we sometimes call just “wrappers”) which use software constructed using a reproducible build system to convert information from the outside world into authorization logic claims. We propose *Revocation Monitors* which are a special case of Trusted Wrappers that detect misbehavior for a particular type of claim, and punish this misbehavior by revoking an issuer’s credentials for issuing claims.

Our main novel contributions are as follows:

- Policy Transparency – a novel combination of authorization logic and general transparency that supports policies in which information sources and trust assumptions are clear and all evidence is discoverable.
- PolyLog – a novel software architecture for realizing Policy Transparency that has addressed the aforementioned challenges to make this practical.
- A prototype of a core subset of PolyLog that allows for expression of policies, making policy claims transparent, and interpreting information from the outside world as authorization logic claims. These core features can be extended to utilize the full PolyLog architecture.
- The first mapping of problems in supply-chain security into authorization logic, which is non-obvious.
- A case study that applies Policy Transparency in a context where the consumer of software release writes a policy to accept or deny this release. To make this decision, the policy uses evidence from software analysis tools, human reviews, and other policies written by standards writers.

The rest of the paper is outlined as follows: Section 2 gives background on related work, Section 3 describes the security goals PolyLog intends to satisfy, Section 4 describes the PolyLog architecture, Section 5 describes how our architecture meets these goals, Section 6 describes the core subset of PolyLog that we have prototyped, Section 7 describes our case study, and Section 8 concludes.

2 BACKGROUND AND RELATED WORK

Our software architecture relies on transparency logs, reproducible builds, authorization logic, and key management, so we briefly give background on these topics.

2.1 Transparency Logs

A transparency log, or a verifiable log, is a verifiable and tamper-evident data structure [8] that implements an append-only log in which all entries are permanently retained to detect misbehavior. Verifiable logs have been used for Certificate Transparency [30] where the entries are certificates issued by a Certificate Authority

(CA) that bind identities (such as website domain names) to asymmetric keys, and misbehavior is when a certificate is malicious or incorrect. The idea is that every issued certificate appears on the log and every consumer of certificates is shown the same log—this way, if the CA tries to show a malicious certificate to one consumer and a trustworthy certificate to another, it will be caught. The same idea has also been used to implement Binary Transparency [11, 26], in which the logged entries are metadata about software binaries. Cryptocurrency [33] can be viewed as applying transparency logs to the transfer of digital coins using *distributed* ledgers [14] (although cryptocurrencies usually also provide consensus whereas transparency logs in general might not). In general, transparency logs can support non-repudiation for arbitrary entries [2].

Importantly, verifiable logs aim to be *untrusted*, meaning that the user of the log does not need to trust the maintainer of the log, or the issuers of the claims, or trust that these parties won't collude together to tamper with the log. One way to achieve this is to implement verifiable logs using Merkle Trees [31]. Crucially, Merkle Trees support efficient *inclusion proofs* which show that an entry is a member of the log, and *consistency proofs* which give evidence that the log is append-only and has not omitted or modified entries over time. Inclusion proofs and consistency proofs can both be done efficiently [8].

Checking the correctness of the log requires a *full audit* which is expensive and requires access to the full contents of the log. As a result, many clients of the log cannot do this, and the ecosystem relies on additional parties called *auditors* that periodically perform full audits. To minimise the chances of unilateral tampering with the log, we assume the transparency log is public, and that there are sufficiently many independent auditors.

2.2 SBOM and Reproducible Builds

While open-sourcing the source code allows anyone to inspect it, most software is distributed pre-compiled with no method to confirm whether it corresponds to the source code allegedly used to build it. To address this, a "Software Bill Of Materials" (SBOM) [6] has emerged as a key building block in software supply chain security. An SBOM provides a description of how a binary was built and from which materials, for instance which revision of the source code. While an SBOM provides a correspondence between a distributed software binary and its sources, it cannot be trusted on its own. To be able to verify the claim provided by an SBOM, one needs to gather the sources, re-run the build steps, and verify that the resulting binary and the distributed binary are bit-for-bit identical, usually by computing and comparing their cryptographic hashes. This cannot be achieved unless the builds are intentionally made *reproducible*. More specifically, reproducible builds guarantee that for a given hardware architecture, re-running the build steps with identical input artifacts, at any time, results in an identical output. Making builds reproducible is not always straightforward because they may have non-determinism, depend on environment variables, or have incompletely specified dependencies. Despite this, the general trend is to make more software projects reproducibly buildable¹.

¹See <https://reproducible-builds.org/>.

2.3 Authorization Logic

Authorization logic is a style of policy language suitable for a broad class of problems involving authorization ("should this action be allowed?") and authentication ("does this public key belong to Bob?"). A number of authorization logics have been proposed in the literature [9, 12, 15, 16, 22]. Authorization logic is especially well-suited to addressing supply chain security because its key features are related to tracking the source of information and making trust assumptions clear. The key features of authorization logics are:

- *logic programming* – policies are based on logic programming languages such as Datalog or Prolog
- *decentralization* – policies are decentralized in the sense that the logical facts do not describe a single global truth, but instead, all statements are attributed to and express the beliefs of mutually distrusting actors called principals. A *principal* is any entity that can act or provide information [32, 34].
- *delegation* – the only way for a principal to believe statements made by another are through clear expressions in the language for doing delegation. As a result, delegations make the trust assumptions in the policy clear.
- *signing* – principals are associated with a public/private key pair and exported claims are digitally signed to prove the authenticity and integrity of claims. Because claims are signed, they can be used as unforgeable credentials in policies that govern access.

In our prototype of PolyLog we use an authorization logic based on SecPal [15], although in principle any language with these four key features would be suitable. Authorization logic has also been applied to operating system authorization [35, 37], privacy policies [17], networking [10], and remote attestation [36].

In a conventional logic programming language like Datalog, programs consist of logical facts and rules for deducing new logical facts from a set of input base facts. Facts are predicates with some number of arguments like `isSystemAdministrator("Alice", "FinancialDatabase")`. Predicate arguments surrounded in quotes like "Alice" are specific constants, whereas arguments without quotes like `user` are variables. Rules consist of one predicate on the left side of `:-`, and some number of other predicates on the right. The predicate on the left is proved and added as a new fact if all the predicates on the right have already been proven. For example, if we have the rule,

```
canAccess(file, user) :-
    isSystemAdministrator(user, server),
    isHostedOn(file, server).
```

in addition to the facts `isSystemAdministrator("Alice", "FinancialFileServer")` and `isHostedOn("PriceData", "FinancialFileServer")` we will also prove `canAccess("PriceData", "Alice")`.

Policies based on logic programming can be used to govern a wide range of problems related to software supply chain security by simply writing a piece of software that takes some action only when a particular goal predicate can be proven, and writing rules that prove the goal predicate only when the relevant evidence has been collected. For example, we can use this approach to write a policy that decides when to import a library or when to connect to a remote machine.

```

"FileAccessPolicy" says {
  canAccess(file, user) :-
    isSystemAdministrator(user, server),
    isHostedOn(file, server).

  "HR" canSay isSystemAdministrator(user, server).
  "FileRoutingDatabase" canSay isHostedOn(file, server).
}

```

Figure 1: An example of authorization logic code.

However, authorization and identification policies should be resilient to adversaries deliberately trying to gain access, perhaps by providing fraudulent credentials. For example, if "Mallory" is an attacker she might falsely claim `isSystemAdministrator("Mallory", "FinancialFileServer")` to gain access to the "PriceData" file.

Authorization logic is well-suited to dealing with problems in security because all facts and rules describe the claims and beliefs of mutually distrusting system participants called principals. A *principal* is broadly any entity that can take action such as humans, machines, tools, organizations, or even policies and protocols. All facts in authorization logic are prepended with a principal called the "speaker" using the syntax `<Principal> says <Predicate>` to indicate that this principal issues or equivalently believes the predicate. Rules are similarly prepended with a speaker.

Figure 1 shows an example of authorization logic code. Crucially, because "FileServer" is the speaker for the rule for proving `canAccess`, in order to prove

```
"FileAccessPolicy" says canAccess("PriceData", "Mallory")
```

using the rule in the code, we now need to prove

```
"FileAccessPolicy"
  says isSystemAdministrator("Mallory", "FinancialServer")
```

and the fact

```
"Mallory" says isSystemAdministrator("Mallory", "FinancialServer")
```

will not work. In order to use the rule set by the "FileAccessPolicy", all the relevant facts need to be proved to the "FileAccessPolicy".

For facts produced by one principal to be believed by another, authorization logic provides a syntax for *delegation*. In our authorization logic, delegations are expressed using the syntax, "P1" says "P2" canSay predicateName() which means that when "P2" says predicateName(), "P1" will also believe it. Delegations can also appear in the head of rules (with conditions on the right hand side) to specify more restricted conditions under which the speaker ("P1") delegates. Because the delegation syntax is the only way for one issuer to prove a fact to another principal, delegations make the trust assumptions clear in the code.

Principals in authorization logic are also associated with an asymmetric keypair. When authorization logic statements are exported, they are digitally signed using the private key, and when they are imported, the signature is checked against the public key to ensure the authenticity and integrity of these claims. Because authorization logic claims are digitally signed, they can be used as unforgeable access tokens. Signing also makes these claims non-repudiable.

2.4 Key Management

To both make use of the digital signature feature of authorization logic and to provide accountability for a wide range of claims, we

need to address two problems related to key management: 1) we need a trustworthy way to ensure that only these identities can gain access to these keys, and 2) we need a trustworthy way to associate public keys to these identities, and. Key management services such as AWS KMS [3] and Google Cloud Key Management [4] can help with the second problem.

Certificate authorities bind keys to public websites, but in our framework far more identities than websites will need keys including: automated tools, human reviewers, and policies mechanizing laws. Interestingly, authorization logic can help with this; it can bind keys to identities by expanding certificates into identity policies. For example, an X.509 certificate issued by a CA can equivalently be expressed as "CA" says PUB_KEY canActAs WEBSITE_NAME where criteria such as time ranges limit use of keys for identities can be added as conditions in the rule.

3 SECURITY GOALS

Before describing our architecture, we first describe the security goals our system is meant to satisfy. Our main security goal is to offer an open-source software authorship attribution framework in which the parties involved in constructing software are clear, consumers of software and other stakeholders have control over these trust relationships, and abusers of this trust can be held accountable. Together our sub-goals achieve this main goal and address other security-oriented barriers to practical adoption.

Discoverability of Misbehavior: It should be possible to detect misbehavior by generators of evidence, so they can be held accountable. In other words, claims entailing misbehavior should be available. For example, if a human reviewer was caught deliberately approving of software with backdoors, we should have a history of these malicious approvals.

Explicit Trust Assumptions: With conventional software ecosystems, who or what software must be held accountable and for what is unclear because the trust assumptions involved in software construction are implicit. Each step involved in generating a software release that is consumed by a client should instead be spelled out in a legible policy language, and these assumptions should be evident from these policies.

Stakeholder Control: Each of the stakeholders in our framework (OSS writers, OSS consumers, human reviewers, tool authors, standards authors) should have control over how policies they set are satisfied, including: what evidence must be gathered, the sources they trust to generate the policy, and the circumstances under which it can be used. Further, we need a way to express the distinct and competing needs and goals of these stakeholders, so that it is clear if and when they can be met.

A Single History of Prior Claims: There should be a single history of generated claims. If discrepancies in history are possible, for example if an approval of a backdoor was found in one history but not another, it is unclear if the human reviewer is at fault or if the software that maintains this history of claims is faulty. A single, trustworthy history prevents this potential failure. In other words, we must prevent *split-view attacks* in which a bad actor deliberately presents one history of their claims to a victim, while presenting another history to others to evade detection.

Minimal Disclosure: In some cases, the use of evidence is sensitive, and the potential for information leakage could dissuade participants from using our system in practice. For example, if the particular open-source packages used by an author of closed-source software are revealed, the outside world could learn about what the closed-source developer is building. Second, human reviewers acting on behalf of their employer may feel uncomfortable giving reviews under their individual name. Our system should minimize the need to reveal what evidence is used and how it is used where possible. We note this goal is at odds with transparency logs in which evidence is public.

Integrity of Evidence: It should not be possible to tamper with the evidence used to discharge a policy goal. We note this goal is at odds with the need for compatibility with existing tools and other sources outside of authorization logic; tampering should also be prevented if evidence is gathered from existing tools.

4 THE POLYLOG ARCHITECTURE

This section describes our PolyLog Architecture that realizes Policy Transparency. To do so, PolyLog supports: expression of arbitrary claims relevant to supply chain in authorization logic, making arbitrary claims in authorization logic transparent, policies that refer to information produced outside the policy language, revocation of claims, and punishment of misbehavior. In the following sections, we describe how this is done in more detail. We first describe the system participants in Section 4.1 and clarify interactions with policies and claims in Section 4.2 before going into details.

4.1 System Participants

Figure 2 shows the main participants in Policy Transparency including humans, organizations, and software components. We now describe these participants.

Software Release Consumers: Consumers of software releases set policies specifying the criteria for accepting open-source software releases according to their needs. Consumers of releases are generally developers of open-source or closed-source software. Because many software consumers will have the same needs, many will use policies written by policy standards writers rather write their own. However some projects may have specific needs requiring custom policies or extensions of standard ones. To dispatch their policies, they will collect evidence from the transparency log. To ensure any misbehavior by the evidence producers cannot be hidden, the policies set by consumers should check inclusion proofs for the evidence.

Policy Standards Writers: Organizations for setting policy standards write policies for broad issues in security, privacy, and software quality that are re-used for many software releases. For example, government agencies write policies for privacy (e.g., GDPR). Open-source software and security foundations might write policies about best practices for applied cryptography, or the circumstances under which a person can act as a credentialed cryptography reviewer. These policies will either be directly written in authorization logic, or other other parties interested in policy transparency will take existing natural language descriptions of these policies and translate them into authorization logic and make

these policies widely available by open source releases and publication on transparency logs. These policies are likely to be among the most complex, so we anticipate these will be standardized and used across many projects to amortize the effort involved in creating and vetting these machine-codified policies. Prior work has shown that it is possible to encode complex legal policies including HIPAA, GLBA, and COPAA using logic programming [13, 29]

Software Release Developers: Software release developers prepare open-source software to be consumed by software release consumers. They use normal open-source software processes and then run code analysis tools and solicit the aid of human reviewers to both gain more assurance about the code and to meet the policies of software release consumers.

Software Analysis Tools: Conventional software analysis tools such as fuzz tools, linters, static analyses, and analyses of dependencies run on the software under preparation for release. We need to interpret the results of these tools with authorization logic policies. However, many such tools already exist and we cannot expect them to produce authorization logic facts as outputs.

Code Reviewers: Human reviewers review the source code under preparation for release. Reviewers may either perform general-purpose software reviews (e.g., to confirm that the code makes effective use of the language in which it is written), or reviewers may be credentialed experts in some domain like cryptography; privacy; or formal methods, and review the code for best practices within this domain. We need to use these reviews as evidence in authorization logic policies. While it is possible for code reviews to be written directly in authorization logic, requiring reviewers to learn logic programming is a barrier to practical adoption we aim to avoid.

Trusted Wrappers: Trusted wrappers are software components that bridge the gap between authorization logic and evidence produced by tools and reviewers. Trusted wrappers comprise 1) code written in a general-purpose programming language that converts evidence from its original format into authorization logic and 2) security features and practices that prevent tampering with this code or evidence (described in Section 4.4). Tool results may come from command-line interfaces, SaaS workflows, or serialization formats (e.g. JSON/protobuf). Human reviews might be entered in IDE tools (e.g. VSCode plugins) or using code review features from source code hosting platforms (e.g. GitHub). However, reviews need to be constrained into predicates rather than natural language prompts, and suitable interfaces reviews are an area of future work.

Auditors: An auditor is a software service that checks that the transparent log is consistent by conducting *full audits* [8], which entail downloading the entire contents of the log and enumerating all entries. During the full audits by these auditors the entries are completely opaque and they have no additional meaning to the auditors.

Monitors: Monitors are software services or human ecosystem participants that scan the logs to find claims that are misbehavior. Unlike auditors, monitors do interpret the specific meaning of claims. What entails misbehavior depends on the type of claim, and so we anticipate different forms of monitors for different kinds of claims. For example, a monitor for detecting faulty fuzz tools might scan the log for claims made by fuzz tools and re-run the tool

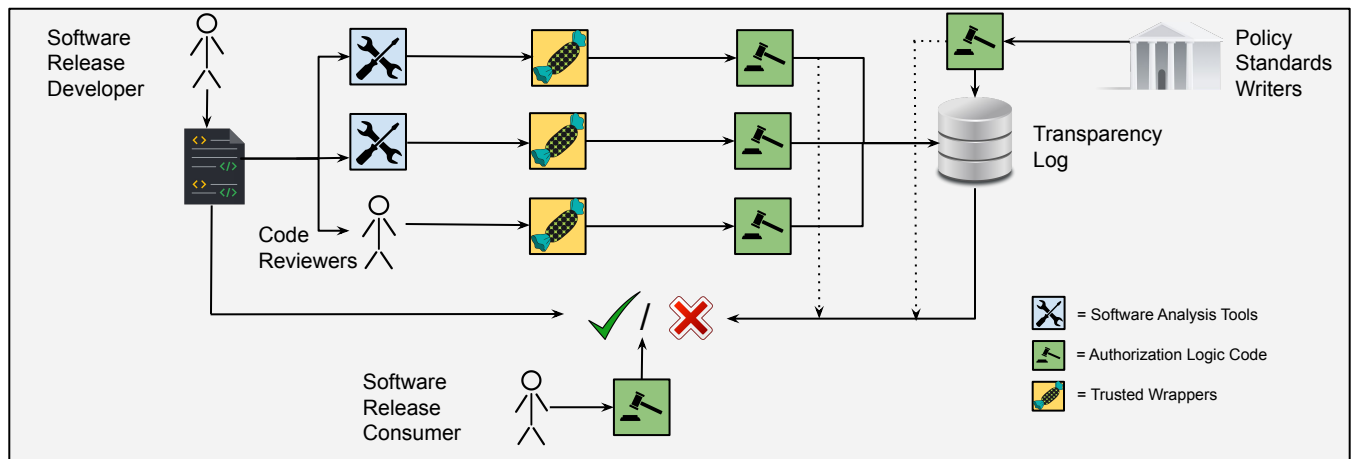


Figure 2: Human, organizational, and software participants of policy transparency.

based on the parameters in the claim to determine if result is repeatable. To detect deliberate approval of software with backdoors by a human reviewer, a monitor would be a human that searches the log for malicious approvals. In PolyLog, we support revocation monitors that can be used proactively by revoking credentials as described in Section 4.5. When a monitor detects misbehavior, it submits a claim that a particular principal has issued a faulty claim. Both transparency logs and authorization logic aid in the detection of misbehavior – transparency logs ensure that all evidence is available and authorization logic can be used to produce proof trees showing which evidence was used to make faulty decisions.

4.2 Claims and Policies

Here we clarify the distinction and interactions between claims and policies and discuss other details about how these are distributed and used in our system.

Any entity, including but not limited to software analysis tools, code reviewers, policy standards writers, auditors, and monitors can issue claims. A *claim* is any piece of information that can be used as evidence in support of a policy and is made transparent by inclusion in the log. The producer of this claim is the *issuer*, and when interpreting claims as authorization logic, the issuer is the principal used as the speaker. Importantly, although many claims in this system will be about the software in preparation for release, claims can also be about other system participants (e.g. a claim made by a policy standard writer could say that a software tool is approved for fuzz testing).

A *policy* is authorization logic code that governs some action (e.g. approval of a software release or assuming an identity) by describing claims needed as evidence to take this action. When combined with appropriate evidence, the action governed by the policy is taken.

Policies use delegations to specify the circumstances under which claims are trusted. In particular, delegations specify which principals are trusted by the policy author to produce which claims. The policy author can make these trust relationships narrower by adding rules to these delegations (e.g. that make them expire after some time or if misbehavior has happened).

Notably, the same artifact can be both a policy and a claim. Both of them are statements attributed to speakers in authorization logic and either can be included in a transparency log. One policy might be used as evidence (a claim) in support of checking another larger policy. The policies written by policy standards writers will often be used as claims by software consumers. For example, a standard policy for describing when a person can act as a cryptography reviewer would be used as a claim in a broader policy set by a consumer stating that approval is needed by a cryptographer. In this case, the consumer delegates to the standard policy to decide who canActAs a "Cryptographer" and only delegates to human reviewers that can assume this identity.

Because authorization logic policies are decentralized, the policy entry points emerge in parallel from two sides: 1) a policy about the open-source software describing the assurance it offers (often a collection of facts produced by ordinary tools that have been fed into wrappers), and 2) a policy that the consumer of open-source packages writes to describe the requirements of packages they consume. These two policies will evolve in a kind of "negotiation" – providers of OSS will make their software compatible with as many analysis tools and human audits as possible to both meet the requirements of many consumers and also offer as much assurance as they can; consumers of OSS will write requirements based on their needs, but may also attenuate their requirements based on what assurance is commonly available from open-source packages. Consumers of OSS may make their policies open-source to make their requirements understood by software producers. The policies written by standards writers will often serve as a kind of entry point because they will likely be used across many projects, however, we note that within the PolyLog architecture, use of standard policies happens only through explicit delegations by policy consumers, so they are optional. Policies by both standards writers and consumers will often be published and distributed on transparency logs – doing so can both aid in making these policies discoverable and in catching policies with bugs.

4.3 Transparency for Claims and Policies

To support transparency for both authorization logic claims and policies, we describe minimal requirements for logging. We need a serialization format that includes the issuer (the principal making the claim/policy) and the body of the claim which can be an arbitrary list of rules and predicates. As described in Section 4.4, some claims are generated by Trusted Wrappers, and to support this we need to include the hash of the Trusted Wrapper code. All artifacts on the transparency log must be digitally signed, and to be compatible with the authorization logic signing feature, the key used must be bound to the principal. So the serialization of the claim/policy is signed with the key of the speaker principal (which is the issuer). The same key can be bound to more than one principal name though, for example to support modularity in policy code. By signing the serialization of the claim/policy, the same signature can be used by both the authorization logic compiler when importing the statement and by the transparency log (because the signature covers the artifact used for both of these cases).

4.4 Trusted Wrappers

To be useful, authorization logic policies need to refer to information generated outside of the authorization logic policy language. For example, to check the time, read a vulnerability database, or check the output of a static analysis tool. As a related problem, relevant claims may be issued in existing standards, such as X.509 certificates. We could try to evangelize yet another standard format for claims, but adopting new standards is a time-consuming process requiring many stakeholders to be convinced of the value of this change and for much pre-existing software to be rewritten. Even if we could cause authorization logic to be adopted as a standard way of expressing claims, standards are ultimately imperfect, and eventually yet another new standard will be desired.

Instead of trying to standardize authorization logic as a way of expressing claims, we use authorization logic as a *metalayer* that can interpret information from many sources. To do so, we use Trusted Wrappers which convert arbitrary information from the outside world into claims interpretable as authorization logic. Trusted Wrappers are similar to *oracles* [18, 19, 23] from the cryptocurrency domain, however, Trusted Wrappers convert information into authorization logic instead of smart contracts.

However, wrappers introduce a new trust assumption – if there is a flaw within the wrapper code, it could invalidate the authorization logic claim. Likewise, the signing feature of authorization logic no longer gives us assurance about the source of information since the signatures generated and checked by the authorization logic compiler only covers the authorization logic claim and not the original source of information or the wrapper code that transforms the information. Trusted wrappers need to prevent tampering with the evidence gathered from the outside world.

To permit compatibility without enabling tampering, trusted wrappers comprise these components:

- Conversion code written in a general-purpose programming language (e.g. python) that reads the evidence in its original source and outputs an equivalent authorization logic fact,
- A reproducible build artifact corresponding to this code, and which contains the hash of this code

- A principal that represents the code; this is the speaker to which the emitted statements are attributed
- The inclusion of the hash of the wrapper code in the signature that covers the serialization of facts produced by the wrapper
- An authorization logic syntax for indicating that that a principal corresponds to a wrapper with a particular hash. When facts are deserialized from a principal bound to a hash, the serialization is checked for this hash (in addition to the usual signature check over the claim).

Because trusted wrappers are general-purpose code, they place few assumptions on the source of evidence. Tampering with the wrapper code would be detected because changes to the code will result in changes to the hash. Generally, wrapper code will be open-source, and a claim made by a trusted build artifact that this wrapper has a particular hash will be included on the transparency log. Tampering with the evidence read by the wrapper is mitigated by auditing the open-source wrapper code.

4.5 Revocation Monitors

One of the primary motivations for systems that use transparency logs is that the activity stored in the log is tracked permanently and broadcast widely, making it possible to detect misbehavior. Previous systems for transparency such as Contour [11] make use of *monitors*, which are applications that scan the transparency log to find bad behavior and punish this misbehavior. How misbehavior is defined, detected, and punished depends on the application of the log. Monitors are reactive rather than proactive because they can only detect misbehavior rather than preventing it.

In our system, arbitrary claims are made transparent, so we need monitors that can catch arbitrary misbehavior. At the same time, we can make monitors proactive by using policies. To achieve both goals, we propose *revocation monitors*. Revocation monitors are implemented as a special case of Trusted Wrappers– they scan the contents of the log and run arbitrary other code to determine if the log entries contain bad behavior. If bad behavior is detected, the monitors emit an authorization logic fact that can be understood by our policies. As with other wrappers, monitors are built using a reproducible build system that produces a hash of the monitor and ties this hash to claims made by the monitor.

The claims issued by a revocation monitor can be used to revoke other claims, by writing the claim to be revoked in a particular way. To allow a predicate $\text{predX}(\text{arg1}, \dots, \text{argn})$ to be revoked by another predicate $\text{predY}(\text{arg1}, \dots, \text{argn})$, any rule that can prove predX (i.e., rules of the form $\text{predX}(\dots) :- \text{predX1}(\dots), \dots, \text{predXn}(\dots)$) should be extended to include predY negated on the right-hand side (that is $\text{predX}(\dots) :- \text{predX1}(\dots), \dots, \text{predXn}(\dots), \text{!predY}(\text{arg1}, \dots, \text{argn})$).

Figure 3 shows an example of a revocation monitor that revokes acceptance of a library when a monitor finds it in a vulnerability database.

Because revocations can cause proof goals that were previously provable to become false, we need a way for the consumers of these facts (the proof goals) to know when relevant predicates have been revoked and policy checks should re-run. To do this, we use: 1) a publish/subscribe system in which claim consumers "subscribe" to a set of predicates that might cause their policies to be revoked and

```

"SoftwareMaintainer" says {
  "TransparencyLog" canSay someApp hasPublishedHash(hashX).
  "VuInDBRevocationMonitor" canSay appearsInVuInDB(someApp).
  accept(softwareLibrary) :-
    softwareLibrary hasPublishedHash(hashX),
    !appearsInVuInDB(someApp).
}

```

Figure 3: Revocation monitor code.

monitors "publish" these predicates, and 2) a compiler pass that runs on the consumer's policy to populate the list of predicates to which the consumer should subscribe. Note that when a proof goal is revoked, the claims that could previously be used as evidence for this proof are still available on the transparency log in case they are needed for forensics.

Notably, revocation is also future-proof. A client relying on a revocation monitor to revoke a credential can allow the monitor to iterate on the terms for doing revocation by doing an unrestricted delegation about the predicate that will be used for revocation as in:

```

"Client" says
  "Revoker" canSay revokeCredentials(userX)}.

```

With this policy, the "Revoker" can set and change the rules for `revokeCredentials` arbitrarily. The client can also limit changes to revocation terms by adding conditions to this delegation as in

```

"Client" says
  "Revoker" canSay revokeCredentials(userX) :- !inAllowList(userX)}.

```

Because policies are all transparent and available on the log, clients wanting to ensure a policy to which they delegate can be revoked have the opportunity to inspect the policy and ensure it has revocation terms they are willing to trust before relying on it.

4.6 Summary Claims

Because wrappers are arbitrary code, getting results from them can make policies slower to check than if they were purely written in authorization logic. Another issue is that software consumers will often need inclusion proofs from the remote transparency log server which requires an internet connection, and consumers may need to check policies offline. To address both problems, we propose *summary claims*, which are just a special case of wrappers that input an authorization logic policy which may be complex and require checking evidence produced by other wrappers, and outputs a simpler claim which may be just one base fact. Checking the outputs of wrappers producing summary claims can both be done offline and efficiently because it caches proof results.

5 SECURITY ANALYSIS

We now describe how the architecture presented in Section 4 satisfies the security goals described in Section 3.

Discoverability of Misbehavior: Inclusion of claims on transparency logs facilitates detection of misbehavior, because the logs provide a history of all claims made by a particular issuer. For example, if a human reviewer of code is accused of deliberately approving code with backdoors, the log offers a complete history of code the reviewer has approved to aid in determining if this reviewer was malicious. Logs are similarly useful for identifying

misbehavior by faulty tools, or policy failures (e.g. if a policy used to produce a summary claim had a bug in it).

Explicit Trust Assumptions: By expressing the criteria for accepting a binary as authorization logic policies, the rules describe the evidence that must be gathered. Because all claims are authorization logic statements attributed to a principal, the source of the claim is clear. The only way for a policy set by one party to accept evidence produced by another is through delegations. As a result, the delegations spell out the circumstances under which each party is trusted to produce what information. Authorization logic also lacks escape hatches such as side effects and foreign function interfaces that are present in most general-purpose programming languages. As a result, these absent features cannot circumvent delegations as the only mechanism for crossing the trust boundary.

Stakeholder Control: Stakeholder control is also provided by delegations. Delegations describe the sources a stakeholder trusts for a particular piece of evidence. A policy author can also add rules to a delegation to further limit when evidence can be used, for example by adding expiration and revocation timestamps.

A Single History of Prior Claims: To ensure a single history of claims is provided to every client (thereby preventing "split-view" attacks), policies that consume these claims also request an inclusion proof ensuring the claim is logged. As long as clients have inclusion proofs in their policies, an attacker attempting to omit a malicious claim from the log will be caught because there will be no way to provide the requested inclusion proof. The full audits by the auditors also ensure the log is consistent. Another way to prevent split-view attacks would be to use a blockchain as the transparency log, as is done with Contour [11] – a blockchain uses a consensus protocol to ensure that distributed copies of the log are the same.

Minimal Disclosure: The expression of claims as authorization logic predicates, the use of roles in authorization logic, and the signing of authorization logic claims all support minimal disclosure. Because claims are predicates, we can have predicates that express the minimum information needed to satisfy a policy; for example rather than writing a reviewer's exact birth date, a predicate appearing on the log can check if they are over an age.

Similarly, code reviewers working on behalf of an organization may prefer to hide their individual name. To do so, we can use authorization logic policies expressing roles. A software consumer delegates to reviews from the abstract role principal "EFFReviewer", and another policy expresses the criteria for occupying this role, (e.g., "Alice" `canActAs` "EFFReviewer").

Finally, signing of credentials supports minimal disclosure about the use of evidence even when we check the authenticity of the evidence. Because authorization logic claims are signed using a key mapped to each principal, the originator of a claim does not need to be notified to be assured that the originator authored it.

Integrity of Evidence: For claims made by ordinary principals (by contrast to trusted wrappers), integrity of claims is provided by signature checking. Each principal has an asymmetric key pair and whenever a claim produced by one principal is consumed by another, the claim is serialized and accompanied by an ECDSA signature over the serialization of the claim. Because ECDSA covers the hash of the claim, if the claim is tampered with, this tampering will be detected when the hash check fails.

For claims generated by trusted wrappers, the claim was initially generated from some source other than authorization logic and it is converted using code in a general-purpose language. In this case, either the source claim or the conversion code could be tampered with. To prevent tampering with source claims: trusted wrappers should be made open-source so that they can be inspected to ensure they faithfully convert source claims. To prevent tampering with wrapper code, claims made by trusted wrappers include a hash of the wrapper code. This hash is checked when deserializing these claims which will detect any tampering with the wrapper code.

6 PROTOTYPE IMPLEMENTATION

To examine the practicality of PolyLog, we prototyped its core features. To build our prototype, we used a number of open-source projects as core components. We use the public instance of Rekor [5], maintained by Sigstore [24], as the transparency log. Rekor is built on top of Trillian [2], which implements a Merkle tree. When publishing entries to Rekor, a LogEntry containing an inclusion proof for the entry is provided, allowing it to be used as summary proof of inclusion. We assume regular consistency proofs are done.

Additionally, our prototype implementation builds on top of SLSA (Supply chain Levels for Software Artifacts) [21], the industry's leading software supply chain best practices framework. SLSA provides a series of controls and recommendations that aim to secure the integrity of every link in a given software supply chain. Based on this, SLSA has provided a SLSA provenances specification which describe how a software artifact or set of artifacts was produced. The SLSA provenance specification extends the in-toto statement standard [27] (an open standard for specifying metadata about binaries). We also use a custom extension of the in-toto statement standard to express our claims.

6.1 Authorization Logic Compiler

We prototyped ² an authorization logic language and compiler based on SecPal [15]. In principle, any variant of authorization logic with the key features of principals and delegation could be used to implement Policy Transparency and PolyLog. We chose a language based on SecPal because SecPal is decidable and translatable to Datalog. Our prototype compiles from our policy language into Souffle Datalog [7] and relies on the Souffle compiler to check queries. The compiler itself is written in Rust. Our prototype compiler supports generation/verification of ECDSA signatures when statements are exported/imported. The signatures cover a serialized object representing the claim. We have language features for binding keys to principals and for importing/exporting claims from files containing these serializations. When exporting, a signature is generated over the claim using the private key bound to the speaker of the exported claim. When importing, the signature is checked using a public key bound to the speaker (principal) of the imported claim. We note that authorization logic policies are also useful for binding principals (identities) to public keys as described in Section 2.4 and by Abadi et al. [9]. At time of writing, the compiler totals 2309 lines of code and an additional 827 lines for tests.

²<https://github.com/google-research/raksha/tree/rust-auth-logic-scored/rust/tools/authorization-logic>

6.1.1 Negation, Non-monotonicity, and Decidability. To support revocation monitors, our language must support negation, which introduces non-monotonic reasoning which can make a logic undecidable. Our language supports negation while remaining decidable. To do so, our language extends SecPal with stratifiable negation, similar to Datalog with stratifiable negation. Negations may appear on the right hand side of rules as long as other restrictions are met – it must be possible to order the predicates into strata. A negated base fact such as $\neg \text{foo}(a, b, c)$ is proved when the positive version of the fact $(\text{foo}(a, b, c,))$ is not proved. With these language restrictions, a standard decision procedure can be used to solve programs as described by Green et al. in Section 2.2 [25].

6.1.2 Handling of Universes. The language restrictions Datalog places to enable decidable bottom-up solving had a complicated interaction with the logic of belief semantics of authorization logic. Datalog requires all variables used as arguments in the LHS of a rule to occur in a predicate on the RHS of the rule which is in a stratum ordered before the stratum on the LHS [25]. Essentially, this keeps the rules finite allowing bottom-up solving to terminate.

A common pattern used to meet these obligations when writing ordinary Datalog is to: 1) write relations describing the universe for each type of variable, 2) extend rules that otherwise would not meet the stratification requirements to refer to this relation on the RHS of the rule. In the following abstract example, the rule defining `some_predicate` would not be stratifiable without adding `is_app` which requires the application to be in the universe of applications. The code on the following line populates this universe with a particular application.

```
some_predicate(appX, hashX) :-some_fact(appX), is_app(appX) .
is_app("SpecificApp") .
```

However, this pattern cannot be used with authorization logic's belief semantics in which all facts are attributed to a principal, because there is no way to communicate membership in universes across principals. Consider the following example, in which the software consumer would like to delegate to the "TrustedBuilder" the right to declare *any* hash as the expected one for "Application".

```
"SoftwareConsumer" says {
  "TrustedBuilder" expected_hash("Application", some_hash) :-
    is_hash(some_hash).}
```

For this rule to be used, the "SoftwareConsumer" also needs to be convinced the hash is a member of the universe. However, the set of hashes that needs to be described is only known to the trusted builder. We could imagine the consumer delegating the builder the right to claim any hash is a member of the universe, however this rule would not be stratifiable!

To solve this problem, we implemented a compiler pass that automatically: 1) modifies otherwise unstratifiable rules to include conditions that check that variables are in universe relations 2) populates universes for each principal. To do so, the syntax is also extended with type declarations so we know which universe in which to place each variable. Rules are extended by adding a condition to the RHS for each ungrounded variable on the LHS that places the variable in a universe based on the variable's type in the declaration of the relation on the LHS. Universes are populated by finding the set of all constant principal names and the set of pairs of

```
"SoftwareConsumerPolicy" says {
  "TrustedBuildPolicy" canSay someHashX canActAs "MLInferenceApp".
  "SoftwareConsumerSourceCriteria" canSay
    acceptSource("MLInferenceApp").
  accept(someHashX) :-someHashX canActAs "MLInferenceApp",
    acceptSource("MLInferenceApp").
}
```

Figure 4: Policy set by software consumer to decide if they will accept a software release identified using a hash.

other kinds of constants and their types referenced in the program and adding base facts wherein the constant principals state the other constants are in the relevant universes. As a side-effect, this makes writing programs more convenient because developers do not have to deal with universes.

6.2 Trusted wrappers

At present, we can translate the claims (written using an in-toto format) on the transparency log into authorization logic by using wrappers. Eventually, we intend to use this same in-toto format as a serialization format used by the authorization logic compiler, so that the signature checking done by the transparency log and the authorization logic compiler interoperate seamlessly.

We have prototyped³ trusted wrappers that can interpret SLSA provenance files, check the system time, and compare it to the release time of a binary specified in a specific type of claim (which we call "endorsements"). However, we have not prototyped a few trusted wrappers needed to make our case study complete. We do not yet support external reviewers that publish their reviews, or integration with static analysis tools.

We have not yet prototyped reproducible builds for other kinds of wrappers, a language feature for binding principals to hashes of trusted wrappers, the integration of revocation monitors with a publish/subscribe system, or a compiler pass for identifying which predicates need re-evaluation post-revocation. However, none of these changes impact the expressiveness of our policies.

7 POLICY EXAMPLE

This section gives a comprehensive example of using Policy Transparency and the PolyLog architecture. Here, a software consumer writes a policy describing the criteria under which they will accept an open-source software release, "MLInferenceApp" identified by its hash. This policy makes use of human reviews, software analysis tools, trusted wrappers, and other policies written by policy standard writers as evidence to dispatch this policy.

Figure 4 shows the policy used by the software consumers to decide if they will accept the hash. A particular hash sha256:0x... is accepted if the evidence gathered from the transparency log is sufficient to prove `accept("sha256:0x...")`. This policy checks two things: 1) that the hash really belongs to the application source, and 2) that the application source has a number of properties outlined by the software consumers. Establishing that the hash belongs to a particular application is a general problem applicable to many applications, so the software consumers delegate to a policy written by policy standards writers called "TrustedBuildPolicy". The criteria for accepting the application source is also written by

³<https://github.com/project-oak/transparent-release/tree/main/experimental/auth-logic>

```
"TrustedBuildPolicy" says {
  hashX canActAs appX :-
    hasAcceptableBuilder(appX),
    hashX hasExpectedHash(appX),
    appX hasReleaseTime(releaseTime),
    currentTimeIs(currentTime),
    currentTime > releaseTime,
    % 2592000000 is 30 days in milliseconds
    currentTime < releaseTime + 2592000000.

  "Provenance" canSay appX hasExpectedHash(hashX).
  "Provenance" canSay appX hasBuilderId(hashX).
  "Endorsement" canSay hasReleaseTime(timeX).

  "UnixEpochTime" canSay curentTimeIs(timeX).

  hasAcceptableBuilder(appX) :-appX hasBuilderId(
    "https://github.com/Attestations/GitHubHostedActions@v1").
  hasAcceptableBuilder(appX) :-appX hasBuilderId(
    "https://cloudbuild.googleapis.com/GoogleHostedWorker@v1").
}
```

Figure 5: Policy by standards writers expressing criteria for associating a hash with an application.

```
"SoftwareConsumerSourceCriteria" says {
  acceptSource("MLInferenceApp") :-
    "MLInferenceApp"
    adheresToStandard("CryptographyGuidelines:1.52"),
    "MLInferenceApp" passes("FuzzTesting"),
    !appearsInVulnDB("MLInferenceApp").

  reviewerX canSay appX
    adheresToStandard("CryptographyGuidelines:1.52") :-
    reviewerX canActAs "CryptographyReviewer",
    !isBannedReviewer(reviewerX).

  "CryptographyReviewCertificationStandard" canSay
    reviewerX canActAs "CryptographyReviewer".

  "TrustedFuzzTool" canSay appX passes("FuzzTesting").

  "ReviewerBanMonitor" canSay isBannedReviewer(reviewerX).
  "VulnerabilityDBMonitor" canSay appearsInVulnDB(appX).
}
```

Figure 6: Criteria for approving the application source.

the software consumers in "SoftwareConsumerSourceCriteria", and a delegation is used just to make this code modular.

Figure 5 shows the policy for establishing that an application has a particular hash. This policy relies on a release system built on top of in-toto and SLSA. The release system generates two types of in-toto statements about the binary (parsed into authorization logic using trusted wrappers). First, a SLSA provenance describes how the binary was built from its sources. The second is provided by the software release developer to endorse this release until an expiration time that serves as a passive revocation. A trusted wrapper is used to check the time in support of this revocation. This wrapper uses a system utility for the time, however for other threat models, more trustworthy sources of time such as RoughTime [1] may be desired; because checking the time happens through delegation, the source of time is clear in the policy. The policy lists GitHub Actions and Google Cloud Build as trusted builders that can build the binary to generate its SLSA provenances.

Figure 6 shows the criteria for the application source code. These criteria entail: checking that uses of cryptography adhere to a standard, that the application passes fuzz testing, and that the application is not in a database of known vulnerabilities. The policy

```

"CryptographyReviewCertificationStandard" says {
  reviewerX canActAs "CryptographyReviewer" :-
    reviewerX hasFormalTraining("cryptography"),
    reviewerX hasPerformedGithubReviews(q), q > 1000.

  "Coursera" canSay
    reviewerX hasFormalTraining("cryptography").
  "ABETAccreditedUniversity" canSay
    reviewerX hasFormalTraining("cryptography").
  "ABETUniversityList" canSay
    universityX canActAs "ABETAccreditedUniversity".

  "GitHubCrawlingTool" canSay reviewerX
    hasPerformedGithubReviews(numReviews).
  "GitHubCrawlingTool" canSay reviewerX
    hasReviewedInLanguage(languageX).
}

```

Figure 7: Policy set by standards writers for granting human reviewers the right to act as cryptography reviewers.

delegates to reviewers to check adherence to cryptography guidelines, but only if the reviewers have a credential for cryptography. The policy delegates to another policy in Figure 7 to check this credential. The policy also delegates to a trusted wrapper called "TrustedFuzzTool" for the fuzz tool. Two revocation monitors are used: one revokes credentials of reviewers, for example, if they deliberately approve of malicious software; another rejects the applications in vulnerability databases.

ACKNOWLEDGMENTS

We thank Maria Schett, Geta Sampemane, and the anonymous reviewers for their insightful feedback.

8 CONCLUSION

We presented Policy Transparency which controls supply chain integrity with policies that make trust assumptions explicit and a transparency log that allows arbitrary policy statements be discoverable, thereby enabling detection of misbehavior. We support interoperability with other sources of information relevant to controlling the supply chain such as outputs from automated tools by using trusted wrappers. We have shown that our architecture and policy language can be used to check a complex policy involving human reviews, automated tools, and mechanized rules for generating credentials for human reviewers. A more complete implementation and evaluation of the PolyLog architecture is left for future work.

REFERENCES

- [1] [n.d.]. . <https://developers.cloudflare.com/time-services/rougtime>
- [2] 2018. Trillian: General Transparency. <https://github.com/google/trillian>.
- [3] 2022. *AWS Key Management Service*. Retrieved July 21, 2022 from <https://docs.aws.amazon.com/kms/latest/developerguide/overview.html>
- [4] 2022. *Google Cloud Key Management*. Retrieved July 21, 2022 from <https://cloud.google.com/security-key-management>
- [5] 2022. *Rekor: Software Supply Chain Transparency Log*. <https://github.com/sigstore/rekor>
- [6] 2022. *Software bill of materials*. Retrieved July 21, 2022 from <https://www.ntia.gov/SBOM>
- [7] 2022. *Souffle: Logic Defined Static Analysis*. Retrieved July 21, 2022 from <https://souffle-lang.github.io/>
- [8] A. Cutter A. Eijdenberg, B. Laurie. 2015. Verifiable Data Structures. <https://continusec.com/static/VerifiableDataStructures.pdf>.
- [9] Martin Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. 1993. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 15, 4 (1993).
- [10] Martin Abadi and Boon Thau Loo. 2007. Towards a Declarative Language and System for Secure Networking.. In *NetDB*.
- [11] Mustafa Al-Bassam and Sarah Meiklejohn. 2018. Contour: A Practical System for Binary Transparency. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*.
- [12] Andrew W Appel and Edward W Felten. 1999. Proof-carrying authentication. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*.
- [13] Adam Barth, Anupam Datta, John C Mitchell, and Helen Nissenbaum. 2006. Privacy and contextual integrity: Framework and applications. In *2006 IEEE symposium on security and privacy (S&P'06)*. IEEE.
- [14] Dave Bayer, Stuart Haber, and W Scott Stornetta. 1993. Improving the efficiency and reliability of digital time-stamping. In *Sequences II*. Springer.
- [15] M Becker, C Fournet, and A Gordon. [n.d.]. Design and Semantics of a Decentralized Authorization Language, CSF'07: Proc. In *20th IEEE Computer Security Foundations Symposium*.
- [16] Moritz Y Becker. 2009. Specification and analysis of dynamic authorisation policies. In *2009 22nd IEEE Computer Security Foundations Symposium*. IEEE.
- [17] Moritz Y Becker, Alexander Malkis, Laurent Bussard, et al. 2010. S4P: A generic language for specifying privacy preferences and policies. *Microsoft Research* 167 (2010).
- [18] Lorenz Breidenbach, Christian Cachin, Benedict Chan, Alex Coventry, Steve Ellis, Ari Juels, Farinaz Koushanfar, Andrew Miller, Brendan Magauran, Daniel Moroz, et al. 2021. Chainlink 2.0: Next steps in the evolution of decentralized oracle networks. *Chainlink Labs* (2021).
- [19] Lorenz Breidenbach, Christian Cachin, Alex Coventry, Ari Juels, and Andrew Miller. 2021. Chainlink off-chain reporting protocol. URL: <https://blog.chainlink.com/off-chain-reporting-live-on-mainnet> (2021).
- [20] Kim Cameron. 2005. The laws of identity. *Microsoft Corp* 12 (2005).
- [21] Supply chain Levels for Software Artifacts (SLSA). 2022. *Safeguarding artifact integrity across any software supply chain*. Retrieved July 21, 2022 from <https://slsa.dev/>
- [22] John DeTreville. 2002. Binder, a logic-based security language. In *Proceedings 2002 IEEE Symposium on Security and Privacy*. IEEE.
- [23] Ethereum. 2022. *Oracles*. Retrieved July 21, 2022 from <https://ethereum.org/en/developers/docs/oracles/>
- [24] The Linux Foundation. 2022. *A new standard for signing, verifying and protecting software*. Retrieved July 21, 2022 from <https://www.sigstore.dev/>
- [25] Todd J Green, Shan Shan Huang, Boon Thau Loo, Wenchao Zhou, et al. 2013. Datalog and recursive query processing. *Foundations and Trends® in Databases* (2013).
- [26] Juan Guarnizo, Bithin Alangot, and Pawel Szalachowski. 2020. SmartWitness: A Proactive Software Transparency System Using Smart Contracts. In *Proceedings of the 2nd ACM International Symposium on Blockchain and Secure Critical Infrastructure (BSCI '20)*. Association for Computing Machinery, New York, NY, USA. <https://doi.org/10.1145/3384943.3409428>
- [27] in-toto Authors and The Linux Foundation. 2022. *A framework to secure the integrity of software supply chains*. Retrieved July 21, 2022 from <https://in-toto.io/>
- [28] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. 2022. Taxonomy of Attacks on Open-Source Software Supply Chains. <https://doi.org/10.48550/ARXIV.2204.04008>
- [29] Peifung E Lam, John C Mitchell, and Sharada Sundaram. 2009. A formalization of HIPAA for a medical messaging system. In *International Conference on Trust, Privacy and Security in Digital Business*. Springer.
- [30] Ben Laurie, Adam Langley, and Emilia Kasper. 2013. Certificate Transparency. RFC 6962. <https://doi.org/10.17487/RFC6962>
- [31] Ralph C. Merkle. 1980. Protocols for Public Key Cryptosystems. <https://doi.org/10.1109/SP.1980.10006>
- [32] Andrew C Myers and Barbara Liskov. 1997. A decentralized model for information flow control. *ACM SIGOPS Operating Systems Review* (1997).
- [33] Satoshi Nakamoto and A Bitcoin. 2008. A peer-to-peer electronic cash system. *Bitcoin*.—URL: <https://bitcoin.org/bitcoin.pdf> (2008).
- [34] Fred Schneider. 2019. . <https://www.cs.cornell.edu/courses/cs5430/2019sp/paper.chptr01.pdf>
- [35] Alan Shieh, Dan Williams, Emin Gün Sirer, and Fred B Schneider. 2005. Nexus: a new operating system for trustworthy computing. In *Proceedings of the twentieth ACM symposium on Operating systems principles*.
- [36] Emin Gün Sirer, Willem de Bruijn, Patrick Reynolds, Alan Shieh, Kevin Walsh, Dan Williams, and Fred B Schneider. 2011. Logical attestation: an authorization architecture for trustworthy computing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*.
- [37] Edward Wobber, Martin Abadi, Michael Burrows, and Butler Lampson. 1994. Authentication in the Taos operating system. *ACM Transactions on Computer Systems (TOCS)* (1994).
- [38] Evan D Wolff, KM Growley, MG Gruden, et al. 2021. Navigating the SolarWinds Supply Chain Attack. *The Procurement Lawyer* 56, 2 (2021).