Macaron: A Logic-based Framework for Software Supply Chain Security Assurance

Behnaz Hassanshahi
behnaz.hassanshahi@oracle.com
Oracle Labs
Brisbane, Australia

Trong Nhan Mai
trong.nhan.mai@oracle.com
Oracle Labs
Brisbane, Australia

Alistair Michael
alistair.michael@uq.edu.au
Oracle Labs
Brisbane, Australia

Benjamin Selwyn-Smith
benjamin.selwyn.smith@oracle.com
Oracle Labs
Brisbane, Australia

Sophie Bates
s.bates@uq.net.au
Oracle Labs
Brisbane, Australia

Padmanabhan Krishnan
paddy.krishnan@oracle.com
Oracle Labs
Brisbane, Australia

ABSTRACT

Many software supply chain attacks exploit the fact that what is in a source code repository may not match the artifact that is actually deployed in one's system. This paper describes a logic-based framework that analyzes a software component and its dependencies to determine if they are built in a trustworthy fashion. The properties that are checked include the availability of build provenances and whether the build and deployment process of an artifact is tamper resistant. These properties are based on the open-source community efforts, such as SLSA, that enable an incremental approach to improve supply chain security. We evaluate our tool on the top-30 Java, Python, and npm open-source projects and show that the majority still do not produce provenances. Our evaluation also shows that a large number of open-source Java and Python projects do not have a transparent build platform to produce artifacts, which is a necessary requirement to increase the trust in the published artifacts. We show that our tool fills a gap in the current software supply chain security landscape, and by making it publicly available the open-source community can both benefit from and contribute to it.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

supply chain security, program analysis, policies, logic programming, build integrity

ACM Reference Format:
Behnaz Hassanshahi, Trong Nhan Mai, Alistair Michael, Benjamin Selwyn-Smith, Sophie Bates, and Padmanabhan Krishnan. 2023. Macaron: A Logic-based Framework for Software Supply Chain Security Assurance. In *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED '23), November 30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3605770.3625213

1 INTRODUCTION

Over the past few years, software supply chain security has received much attention because of attacks targeting companies, like SolarWinds. Several standards and tools have emerged to address various challenges in supply chain security because attackers target various links in the software supply chain between source code and final artifacts. The rapidly evolving nature of software development lifecycle requires techniques that enable one to specify and verify new security properties. For this to become a reality, high-level specifications such as Supply-chain Levels for Software Artifacts (SLSA) [33], and CIS [4] have been proposed.

SLSA, which is getting traction in the community, is a supply chain security specification that provides guidelines to improve the build integrity of software artifacts. It mandates the production of authentic and verifiable provenance documents that describe the build process of a software artifact. It also requires the adoption of provenance generation by both open-source project maintainers and software package registries. An example of this is the npm public registry, which has added support for publishing SLSA Build Level 2 provenances [19].

Because the SLSA requirement of provenance generation is relatively new, few projects currently meet this requirement. Therefore, we need a solution that also reasons about the software components that do not generate provenances. Moreover, certain security properties cannot be verified solely based on the data collected in a provenance. For instance, the build scripts obtained from a source code repository that build an artifact might be contaminated by attackers [12] and need to be analyzed independently.

So far, we have considered a single software component. However, the security posture of a software system is not limited to the main component in question; it is also reliant on the security posture of the dependencies. Thus, any tool that analyzes software supply chain security issues, needs to reason about a software component and all its dependencies, recursively. This can be done by identifying the dependencies for a software component using existing Software Bill of Material (SBOM) generators, such as CycloneDX tools [7]. However, the produced SBOM documents often miss important properties, such as references to the source code

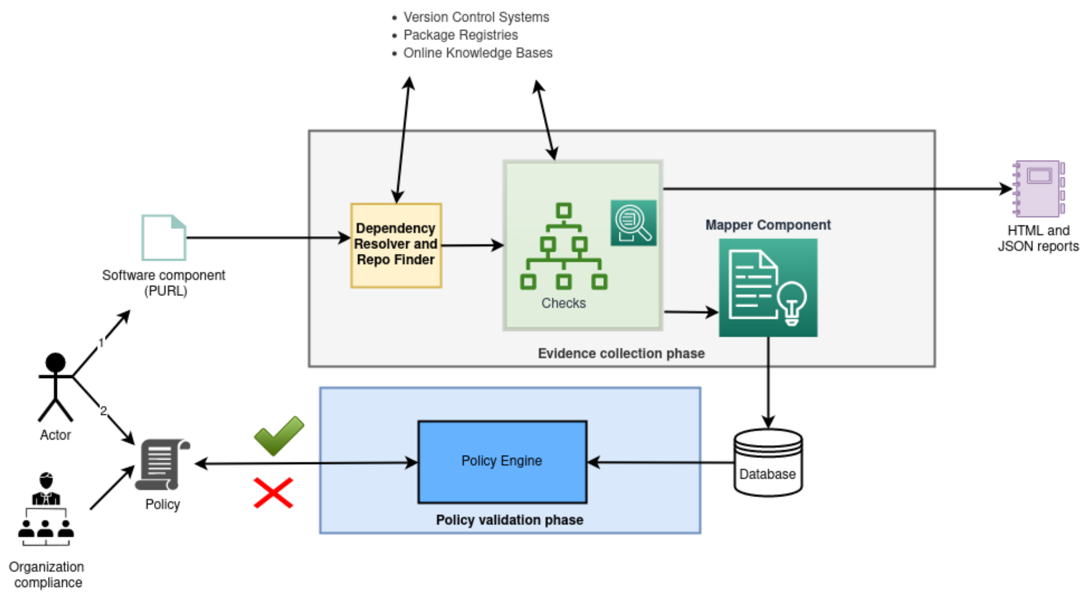

Figure 1: High-Level Architecture

repositories, which makes it difficult to analyze how the dependencies are generated. While a non-forgeable SLSA provenance can provide the URL to the source code repository reliably, unfortunately, the majority of packages still do not support SLSA (see Section 5). Therefore, one has to provide techniques to improve this.

Another challenge in analyzing a software system is that each of its dependent software components can have a different maturity level. For instance, the main software component and a few dependencies may meet SLSA Build Level three, but there can be dependencies that satisfy properties at lower build levels only. The question of whether or not this is acceptable depends on the organization consuming the main component. Hence, one needs a framework where such policies can be expressed and analyzed. Such policies can build upon the evidenece gathered from the provenances and the analysis of the build systems.

In this paper we describe our solution to the problems identified above. Our tool, called Macaron, is an extensible checker framework that is tightly integrated with a policy engine, allowing to validate relevant policies on the stored results. To enable flexibility and efficiency simultaneously, we distinguish between checks that gather evidence and the policies defined by the software consumer to verify claims [10]. This distinction will help the adoption of supply chain security specifications like SLSA. Our approach takes the non-trivial guidelines and translates them to enforceable policies.

The separation that is supported by Macaron enables the security analyst to write checks as part of trusted evidence gatherers in an imperative language, such as Python, without requiring them to make any assumptions about the actual policies specified by consumers. Figure 1 shows the high-level design of Macaron. The framework consists of four components: (1) The dependency resolver finds dependencies and the corresponding source code repositories. (2) The checks in Macaron implement the logic for different specifications. The extensible nature of Macaron helps keeping it current because it can handle the evolution of supply chain security requirements; (3) The mapper component transforms and persists the data produced by the checks as atomic predicates, which can be enforced by the policy engine; (4) The policy engine that accepts input policies expressed in Datalog and determines if the checks gathered in the first step meet those input policies.

As noted, the separation between the checks and the policies allows policy designers to write policies for security properties identified by a specification, without requiring them to know or implement the detection logic. We use Soufflé Datalog [29] because of its support for recursive rules, stratified negation, and data aggregation. Thus, policies involving transitive dependencies, as well as excluding certain dependencies from satisfying a requirement are easy to express. Also, Soufflé can generate proof trees [38], which can be used to understand why a software component satisfies an organization policy. Finally, because the evidence gathered by checks is persisted, it is possible to specify policies for different versions of the components. For example, one can write policies to detect regressions over time for a software component, which is crucial for supply chain security.

In summary, this paper makes the following contributions:

- A flexible and extensible framework for checking and validating policies for software artifacts, which is open sourced and available to the community. [1]
- Decoupling evidence collection from the policy validation phase that allows flexible policies.
- Persisting the collected evidence to support incremental analysis and detect regressions.

[1]https://github.com/oracle/macaron

- Macaron is able to reason about dependencies, some of which might not produce SLSA provenances.
- We have identified atomic predicates for build integrity using the SLSA specification, and designed two checks to reason about them.
- We have conducted an evaluation on the top-30 projects from three ecosystems: Java, Python, and npm.

Finally, to evaluate Macaron we ask the following research questions:

- RQ1: How effective is the check to detect hosted build platforms?
- RQ2: What is the SLSA provenance generation status for top open-source projects?
- RQ3: How effective is mapping dependencies to the source code repositories when SLSA provenances are not available?
- RQ4: Is it possible to define a policy for an organization that uses software components with different maturity levels?

2 RELATED WORK

Several supply chain security tools and frameworks are published with differing goals in the past three years. In this section, we provide an overview of the closely related works, focusing on checking software component analysis and properties, policy verification, and SBOM generation.

2.1 Security checkers and verifiers

Scorecard [24] and Legitify [16] are open-source tools that use the public information of a software repository, assess a number of security heuristics, and compute a score based on the results. The checks used in these tools are rather generic to be suitable for a wide range of repositories, and focus on the repository configurations, e.g., branch protection rules, rather than build-specific properties. Chainbench [3] is an auditing tool, similar to Scorecard, that runs health checks on a repository. It uses the CIS [4] framework as a guideline. OSSGadget [21], is another open-source project that provides a collection of tools, such as oss-health that calculates health metrics similar to Scorecard for a project, with difference in analyzing packages as well as repositories.

GUAC [13] aggregates software security metadata into a graph database. It collects data, such as VEX [35], SLSA, and SBOMs from multiple sources, and allows querying whether a software component and its transitive dependencies have any known vulnerabilities, SBOM, etc. GUAC complements Macaron as it can be a source of evidence collected for a software component and used by Macaron checks. Similar to GUAC, Macaron analyzes a software component and its dependencies to check certain metadata, such as SLSA provenances if available. However, we take a step further by analyzing the corresponding source code repository at a specific commit hash for various security properties. Macaron can also feed its results back to GUAC in future for software components that do not produce SLSA provenances for instance.

SLSA Verifier [27] verifies SLSA provenances that are generated by SLSA compliant tools, such as the GitHub SLSA provenance generator [26]. It verifies a signed provenance against an artifact by looking up the corresponding Rekor [23] log entry and verifying the signature; verifying the builder identity based on the signing certificate; and checking that the provenance information matches the provided source code repository. Macaron invokes SLSA verifer as part of the checks to analyze provenances, and transforms the results to atomic predicates to be used by the policy engine.

2.2 Policy engines

Rego [22] is a policy language inspired by Datalog that allows queries on the data stored in Open Policy Agent (OPA). However, unlike Datalog, Rego has limited support for transitive rules, which is especially necessary for analyzing transitive dependencies. Moreover, Macaron makes a distinction between software consumer policies and the checks that collect evidence. This distinction allows software consumers to focus on the compliance rules in a project, delegating the detection logic to security analysts who design checks.

The in-toto framework is designed to verify that each task in the software supply chain is carried out by authorized personnel only, and that the product is not tampered with in transit [14]. It also defines a standard format for attestations, which is used in SLSA provenances [33]. in-toto requires a project owner to create a layout, consisting of the sequence of steps of the software supply chain, and the functionaries authorized to perform these steps. in-toto gathers information about the build execution in a `link` metadata file and validates it against the steps defined in the layout. In comparison, Macaron provides a logic-based framework that gathers various types of evidence, including existing attestations, and maps them to atomic predicates to be used in expressive policies that allow recursion for dependencies.

PolyLog [10] is the closest work to Macaron. The purpose of PolyLog is to reason about authorisation in the context of limited trust for supply chain security. It uses an authorization logic based on SecPal [1]. However, we do not fix any particular logic. Our policy language, i.e., Soufflé Datalog, is more general and any set of rules that can be expressed in Datalog is permitted. Note that PolyLog [10] translates the policies to Soufflé Datalog ultimately. Moreover, PolyLog needs wrappers for different tools to collect the information and evidence (e.g., provenance expressed in the in-toto format [14]), while Macaron achieves a clean separation by decoupling the checker framework from the policy engine.

2.3 Provenance generation

Witness [37] wraps a build command and records various types of information in a provenance document as the build execution happens. It also enables the enforcement of policies using Rego [22]. Macaron can be extended to verify provenances generated by Witness, while getting benefit from other build-specific checks.

The GitHub SLSA provenance generator [26] provides a set of tools that can produce SLSA provenances for projects hosted on GitHub. The tools provide trusted builders as well as a generic provenance generator to produce a provenance for a given artifact. While the generic provenance generator does not give any guarantees about the security posture of the build commands, it establishes a non-forgeable link from an artifact to the source code repository and the pipeline that has produced it. The checks in Macaron considers these provenance generators trusted and verifies the security properties accordingly.

2.4 SBOM generation

An SBOM is a formal, machine-readable inventory of software components and dependencies, information about those components, and their hierarchical relationships [20]. CycloneDX [7] and SPDX [30] are the two widely known standards, providing specifications and tools to assist generating SBOMs. Macaron uses the CycloneDX tools to automatically generate SBOMs (if not available). However, generating an SBOM can be challenging because reproducing the build environment in a generic way is not always feasible. Macaron takes a best-effort approach to generate SBOMs.

Unfortunately, SBOMs do not always contain the necessary metadata and information, making it challenging to detect dependencies of a software component and map them to the source code repositories. Recently, Synk has open sourced Parlay [28] to add additional information to an existing SBOM. It obtains the additional information from existing services, such as Ecosyste.ms [9] that index repositories and artifacts. Parlay also relies on an external service, and the tool becomes non-functional if the service is not available. Macaron, on the other hand analyzes the configuration files, such as `pom.xml` to map artifacts to source code repositories. In future, we plan to use tools like Parlay as a fall-back solution to add additional information to SBOMs.

3 APPROACH

Figure 1 shows the architecture diagram of Macaron, which consists of two main phases: evidence collection and policy validation. In the evidence collection phase, given a software component as input, Macaron first determines its dependencies by either running an existing SBOM generator [7], or taking an SBOM document as input. Next, the checks will collect evidence for the security properties of interest and provide the outputs to the mapper component. The mapper transforms the collected data into atomic predicates and persists them to a database.

In the second phase, the policy engine verifies the policy specified in the Datalog logic programming language over the persisted atomic predicates and reports whether it is satisfied or not. Such policies can be provided by the software consumer or a compliance organization. The overhead introduced by the policy engine is relatively small because it uses the persisted data and does not need to analyze the build system.

We describe the main components in Figure 1 in more detail in the rest of this section.

3.1 Dependency resolution and finding source code repositories

By default Macaron analyzes the main software component as well as its *direct* dependencies. Note that if enough resources are available, Macaron can be run on all dependencies. SBOM generators are integrated to automatically detect the dependencies. To run an SBOM generator plugin, Macaron identifies the build directory inside a repository at a specific commit hash. If there are multiple build directories in a repository, the dependency resolver aggregates the dependencies from all the SBOM files. It is also possible to provide an SBOM document as input if it is trusted. The dependency resolver component parses the SBOM file, detects direct dependencies, and maps them to the source repository, using `externalReferences` attributes. It also performs validation on the URLs, such as checking whether the `host` part is within an allow list.

As noted earlier, not all generated SBOMs have the external references. For such cases, the dependency resolver finds the source code repository associated with the software component. A software component is identified by a Package URL (PURL) string. For instance, by obtaining the `group`, `artifact`, and `version` of a particular Java dependency PURL string, it is possible to retrieve the POM file from a package registry (e.g., Maven Central). POMs discovered in this way can then be analyzed to extract the source control management (SCM) metadata, which often contains the source repository URLs. Macaron currently uses the latest commit in the corresponding repository for a dependency. In future, we plan to find the exact commit used to create an artifact.

To prevent the repeated polling of a package registry, the discovered URLs are added to the Macaron's database, allowing faster retrieval of missing URLs for artifacts.

3.2 Collecting evidence through checks

Macaron provides an extensible checker platform to facilitate collecting evidence for a specific security property. Before running the checks, it creates abstractions and intermediate representations for the build-related code obtained from the mapped repository as follows:

Defining a new check. The extensible design of Macaron makes it easy to define a new check. The platform provides a callback method with the representations constructed from the initial processing tasks, such as callgraph, as a parameter. The security analyst can use an imperative language like Python, to specify the logic of the check, or call other tools and online knowledge bases that provide evidence. For instance, a check can download an artifact and its corresponding provenance document from a GitHub release, and run the SLSA Verifier tool [27] to verify that the artifact meets the `isolated` and `non-forgeable` SLSA properties.

Moreover, a check can define a relationship with another check to skip running or run only if the other check has failed or passed. Such a dependency relation is especially helpful for specifications, such as SLSA, that have maturity levels, with some properties being stronger than others. For instance, if a check at level three is passed, certain checks at lower levels can be skipped to avoid repetition and improve performance. Consider, as an example, the provenance verification check. If this check passes successfully, another check can compare expected values against the verified provenance and report if the artifact is built as expected.

3.3 Mapping evidence to atomic predicates

The evidence gathered by the checks can be in any format and is not directly consumable by a policy engine. The mapper component in Macaron decouples the checks from policy engine by transforming the check results and evidence to atomic predicates. Moreover, it persists the predicates in a local database, which has several benefits:

- **Incremental analysis**: Because a software component that is analyzed once does not need to be analyzed again, the results can be reused when the component is used as a dependency of another software component. Note that the granularity level for persisted results can be adjusted to atomic predicates in principle.

- **Temporal behavior**: The policies can check for behavior over time, such as regression over a given time interval.
- **Performance of policy engine**: The policy verification can be used even when performance is critical. This is because, once the checks are run, the policy engine can use the locally cached atomic properties thereby reducing the runtime overhead.

3.4 Enforcing policies using logical rules

Specifying policies using a logic programming language, such as Datalog is not new [8, 10]. Datalog programs are sufficiently expressive with precise semantics to support a wide variety of policies. Datalog is monotonic, i.e., if a fact is derivable, it will continue to be derivable after the addition of new rules. It also supports stratifiable negation which is essential for specifying exceptions in policies. For instance, a rule in a policy might not be needed for a certain software component, while other rules should be checked over multiple components and dependencies. In practice, such policies need to be refined based on the collected evidence to exempt a dependency from a requirement using negations, which cannot be known ahead of time.

While our policy engine is agnostic to the actual algorithm used to execute the Datalog rules, Soufflé's bottom-up computation is suitable for our approach. In general bottom-up computation is shown to perform well compared to top-down [34]. Moreover, if users know ahead of time that certain predicates are not necessary to be computed, our design allows checks to be run selectively on a software component and its dependencies in the evidence collection phase. Therefore, the bottom-up evaluation can be guided towards the solution of the original problem if necessary.

It is possible to write expressive Datalog policies provided that the associated facts are generated. The example below shows a policy that ensures all the software components are hosted on `github.com`.

```
Policy("hosted-on-github", repo) :-
   repository_attribute(repo,"remote_path",url),
   match("^https://github.com.*$",url).

apply_policy_to("hosted-on-github", repo) :-
   is_repo(repo).
```

The policy relies on Macaron generating a fact associated with the location of the repository, which is called `remote_path`. The value associated with this attribute is then checked to make sure it is on `github.com`. The rule `apply_policy_to` states that the policy "hosted-on-github" should apply to all source code repositories of software components (i.e., any item marked as a `repo` should satisfy the policy).

4 DESIGNING CHECKS AND POLICIES FOR SLSA SPECIFICATION

In this section, we describe two checks designed to analyze the build integrity properties of a software component. This is based on two main requirements from SLSA v1.0 [33], namely, hosted build platform, and availability of provenances. We also describe a policy (expressed in Datalog) that uses the results of these two checks.

4.1 Detecting hosted build platform

One of the important prerequisites for build integrity and transparency is to have a hosted and automated build and deployment platform [33], which corresponds to SLSA level two. Note that these two properties are the basis of increasing the trustworthiness of the built artifacts, where the build and deployment process is carried out by a build service, with no manual inputs or intervention. Such an *isolated* build process prevents separate runs, even within the same project, from influencing each other. Note that hardened build platforms are required to satisfy level three properties; but that is not the focus of this check.

We describe the check, whether a project has a hosted deployment workflow, using Horn clauses [17], although in principle the functionality can be implemented in any language. The check, shown in Listing 1, is based on the following facts that can be generated by analyzing the source code of the repository:

- `InvokeCommand`: Identifies the command running a build tool, such as Gradle, Maven, Pip, and Poetry, and the acceptable command arguments for deployment of an artifact, which is called from a shell script. This information is obtained by examining the source files in a repository that are used for building and deploying a project, such as `pom.xml`, and `pyproject.toml`, and the command is determined by parsing the shell script.
- `InvokeScript`: Identifies whether a shell script calls another shell script, which can be inlined as part of a CI workflow step, or a bash script file.
- `InvokeWorkflow`: Identifies the relevant CI workflow file, such as GitHub Actions workflow, or GitLab CI file calling another CI workflow file.
- `InvokeWorkflowScript`: Is related to the above CI workflow and determines if there are calls to an inlined shell script as a workflow step.

Using these facts, we define clauses that determine if a deployment command is reachable from the workflow entrypoint. More specifically, the `Reachable` clause computes a callgraph and determines if a node, which can be a CI workflow, or a shell script is reachable from another node. The `ReachableDeploy` rule determines if a reachable node invokes a deployment command, hence indicating that the software component is built and deployed using a hosted build platform.

Finally, the result computed by the `ReachableDeploy` clause is transformed by the mapper component described in 3.3 into an atomic predicate (`BuildPlatform`) with `"passed"` or `"failed"` as the `status`, and made available to the policy engine along with some of the facts as justifications for the result. The details of the mapping are omitted for brevity.

4.2 Discovering and verifying SLSA provenances

The SLSA specification [33] demands that a build process has to generate a provenance. This is a fundamental requirement and is

Listing 1: The predicates computed for the build platform check.

```
// Facts.
InvokeCommand(script, command)
InvokeScript(caller_script, callee_script)
InvokeWorkflow(caller_workflow, callee_workflow)
InvokeWorkflowScript(caller_workflow, callee_script)

// Clauses to compute the check results.
InvokeNode(caller, callee) ⟵
  InvokeScript(caller, callee) ∨
  InvokeWorkflow(caller, callee) ∨
  InvokeWorkflowScript(caller, callee)

Reachable(caller, deploy_node) ⟵
  Reachable(caller, node) ∧
  InvokeNode(node, deploy_node)
Reachable(caller, deploy_node) ⟵
  InvokeNode(caller, deploy_node)

ReachableDeploy(caller, deploy_command) ⟵
  Reachable(caller, script) ∧
  InvokeCommand(script, deploy_command)
```

Listing 2: The predicates computed for the provenance check.

```
// Facts.
Artifact(name, release_url, digest, digest_algorithm,
      software_component)
Provenance(name, digest, digest_algorithm, software_component)

// The clause to compute the check results.
ProvenanceVerified(component) ⟵
  Artifact(name, release_url, digest, digest_algorithm,
        component) ∧
  Provenance(name, digest, digest_algorithm, component)
```

independent of the actual maturity level. Depending on how the provenance is created and what it contains, it can be compliant with different SLSA levels. For instance, if an artifact is built in an isolated environment that the project maintainer cannot influence, and the provenance is generated and signed by a control plane, which is also isolated from the build environment, it reaches Build Level three. We have designed a check in Macaron that searches for a provenance of an artifact (e.g., in the corresponding GitHub release assets) and analyzes it to determine its level.

The check downloads the artifact and its provenance if available. Similar to the check in Section 4.1, Listing 2 shows the logic of this check using Horn clauses. The `ProvenanceVerified` rule determines if a provenance associated with an artifact can be verified using the output of the SLSA Verifier tool [27]. SLSA Verifier verifies the provenance's cryptographic signatures and makes sure it was created by the expected builder.

Once the check collects the results, the mapper component, described earlier, transforms the result to `ProvenanceAvailable` atomic predicate, and persists it along with the facts to the database to be used by the policy engine.

Listing 3: Example policy for SLSA expressed in Datalog.

```
Policy("SLSA2-transitive",parent) :-
  Dependency(parent, child),
  SLSA2(parent),
  SLSA2(child)

SLSA2(component) :-
  ProvenanceAvailable(component, "SLSA2"),
  BuildPlatform(component, "passed")

apply_policy_to("SLSA2-transitive", component) :-
  is_component(component).
```

4.3 Example policy for SLSA checks

An example policy based on the results of the two checks described in this section is to verify that a software component and all its dependencies either produce non-forgeable and verifiable provenances or pass the hosted build platform. Because Macaron provides all the necessary predicates as to the policy engine automatically, the final policy can be specified in few lines of code expressed in Datalog as shown in Listing 3. In this policy, `BuildPlatform` and `ProvenanceAvailable` correspond to the derived predicates described in Sections 4.1 and 4.2, respectively.

5 IMPLEMENTATION AND EVALUATION

Macaron is primarily written in Python and the checks are written as Python modules. The mapper component uses SQLAlchemy's ORM mappings [31] to transform the results of the checks and evidence to atomic predicates. These atomic predicates are then stored in a SQLite [32] database to be used by the policy engine, which uses the Soufflé Datalog engine to determine if the Datalog policies are satisfied. Macaron is open sourced and available on GitHub.[2] We use version 0.2.0 of Macaron for the evaluations in this section.

To evaluate Macaron, we have collected the top 90 open-source projects for Maven, Python, and npm ecosystems, containing 30 projects each. The first 15 projects are collected from the Census II dataset [2], published by Harvard Laboratory for Innovation Science (LISH) and the Open Source Security Foundation (OpenSSF) for the top open-source libraries usage in production applications. The second 15 projects are collected from the Criticality Score project [6].

5.1 RQ1: How effective is the check to detect hosted build platforms?

Table 2 shows the evaluation results for the hosted build platform check described in Section 4.1. Currently, this check supports only Java and Python build systems. Hence the evaluation is conducted on the public GitHub Python and Java repositories in our dataset. Among the 60 projects, Macaron correctly reports that 17 projects have a hosted build platform for building and deploying artifacts, while 32 projects do not have a transparent build platform. Macaron does not produce any false positives. It produces false negatives for 11 projects which are due to a lack of support for specific package and environment management systems, such as Conda [5]

[2] https://github.com/oracle/macaron

Table 1: Finding dependencies and their corresponding source code repositories in Java projects. Note that if a row contains zero, it is due to the SBOM generator failing to generate an SBOM.

	External references in SBOM		Repo Finder’s additional findings
	Repo found	Repo not found	
Alluxio/alluxio	79	48	25
apache/camel	227	259	143
apache/cloudstack	68	62	30
apache/commons-io	9	0	0
apache/commons-lang	7	1	0
apache/commons-logging	1	4	0
apache/flink	94	55	25
apache/hadoop	103	72	31
apache/httpcomponents-core	9	2	2
apache/kafka	0	0	0
apache/maven	27	15	13
apereo/cas	0	0	0
eclipse/jetty.project	78	56	29
FasterXML/jackson-annotations	1	0	0
FasterXML/jackson-core	5	0	0
FasterXML/jackson-databind	7	1	0
google/gson	8	1	0
google/guava	10	5	0
hibernate/hibernate-orm	16	26	18
junit-team/junit4	2	0	0
mockito/mockito	2	6	2
neo4j/neo4j	74	22	8
OpenAPITools/openapi-generator	50	63	58
qos-ch/logback	21	5	1
qos-ch/slf4j	3	4	0
quarkusio/quarkus	193	105	80
raphw/byte-buddy	9	16	4
spring-projects/spring-framework	0	0	0
spring-projects/spring-security	0	0	0
Total	**1103**	**828**	**469**

Table 2: Results for the hosted build platform check on public Github Python and Java repositories.

	PASSED		FAILED	
	TP	FP	TN	FN
Python repos	9	0	17	4
Java repos	8	0	15	7
Total	**17**	**0**	**32**	**11**

for Python, which we plan to add in future. Another source of false negatives is the partial support for Jenkins CI services [15] compared to GitHub Actions CI in Macaron. Overall, Macaron achieves high precision and reports that a large number of popular projects do not have a transparent hosted build platform, which is necessary to adopt SLSA.

5.2 RQ2: What is the SLSA provenance generation status for top open-source projects?

To understand the current state of SLSA provenance generation in open-source projects, we run the check described in Section 4.2 on the 90 projects in our benchmark. We observed that two Python projects generate SLSA provenances using GitHub SLSA provenance generator [26], which are Flask [11] and MarkupSafe [18].

Even though the npm public registry has added support for publishing SLSA Build Level 2 provenances [19], only one npm project (Semver [25]) in our dataset, is generating provenances using this feature.[3]

5.3 RQ3: How effective is finding source code repositories when SLSA provenances are not available?

To check and verify certain security properties of a software component, we need to find the source code repository at a specific commit

[3]When npm projects are built on GitHub Actions, they can enable generating and publishing provenances on the npm registry by adding –provenance flag to the npm publish command.

hash from which the component is built. A SLSA provenance links a software component to the corresponding commit hash reliably. However, as observed in Section 5.2, the adoption of SLSA is still in early stages and we need alternative solutions meanwhile. We evaluate the effectiveness of the technique described in Section 3.1 in this section on Java projects in our dataset. Support for other ecosystems will be added in the future.

Table 1 shows the results for finding source code repositories for software components. The "External references in SBOM" column shows the number of dependencies in the generated SBOM that had external references to repositories ("Repo found") or missed references ("Repo not found"). The last column shows the additional repositories found by Macaron using the technique described in Section 3.1. Overall, we were able to find 469 additional repositories for software components. Note that if a row contains zero, it is due to the SBOM generator failing to generate an SBOM. By default Macaron looks for the `bom.json` files generated by the SBOM generator. If a project is configured to generate SBOMs with custom names, we manually rename the output files to help Macaron discover them.

In terms of execution time, when a source code repository for a software component is missing in the SBOM, it takes Macaron 1.43 seconds on average to find it, which is mostly due to the overhead of API call to the package registry. Once the repository is found and stored in the database, the overhead decreases to two milliseconds on average.

Finally, it is worth noting that because this technique relies on the Source Control Management (SCM) metadata provided by the projects' maintainers in their project configurations, they need to be verified further to ensure they are reliable.

5.4 RQ4: Is it possible to define a policy for an organization that uses software components with different maturity levels?

In this section, we present a case study to use Macaron as a checker and policy engine to help an organization determine its supply chain security posture and determine if its policies are satisfied. The goal of this case study is to understand if Macaron allows flexible policies for a complex software system in an organization. We use the SLSA for organizations recommendations [33] as the reference guideline in this case study.

According to the guidelines, the organization should choose a target SLSA level, and select tools that support the desired SLSA level. We choose level two for the build track, and assume the project produces SLSA provenances using the trusted, off the shelf, generator [26]. We also use Macaron to check and verify the build pipeline.

The organization in this study involves both a producer of software artifacts and a software consumer that uses different software artifacts. We assume that they rely on both open-source and proprietary dependencies. The software producer must build the artifacts on a hosted build platform that generates signed provenances, while the software consumer, demands that all dependencies should meet SLSA Build Level two. Therefore, they should use a hosted build platform that produces provenances.

Listing 4: Example transitive Datalog policy for SLSA with an exception rule.

```
Policy("SLSA2-transitive",parent) :-
  Dependency(parent, child),
  SLSA2(parent)
  !violate_SLSA2(parent).

// Detect dependencies that violate SLSA2 rule.
.decl violate_SLSA2(parent: SoftwareComponent)
violate_SLSA2(parent) :-
    Dependency(parent, child),
    !SLSA2(child),
    !exception_dependencies(dependency).

// Exceptions for violating dependencies.
.decl exception_dependencies(dependency: SoftwareComponent)
exception_dependencies(dependency) :-
  Component(dependency, "A").
exception_dependencies(dependency) :-
  Component(dependency, "B").

// SLSA Build Level 2 rules.
.decl SLSA2(component: SoftwareComponent)
SLSA2(component) :-
  ProvenanceAvailable(component, "SLSA2").
SLSA2(component) :-
  !ProvenanceAvailable(component, "SLSA2"),
  BuildPlatform(component, "passed").

apply_policy_to("SLSA2-transitive", component) :-
  is_component(component).
```

The first step in this process is to specify the above policy that permits the consumption of the artifact produced by the main project if it reaches SLSA Build Level two. Listing 3 in Section 4.3 presents the policy using Datalog. While this policy allows reducing the attack surface by ensuring that all the software components have the same level of maturity, in reality not all dependencies satisfy the SLSA level two rule. Hence the consumer can add exceptions. This is possible because we have broken down the SLSA requirements to atomic predicates. This enables the specification of fine-grained policies including the addition of exceptions at the atomic predicate level for a subset of dependencies. For instance, Listing 4 shows a more flexible policy, where dependencies *A* and *B* do not need to satisfy the SLSA2 rule. Macaron then runs the checks that collect evidence for the main project and the dependencies and executes the policy engine with the specified policy.

Detecting regressions. Another useful policy is to detect regressions over time for a software component, which is crucial for supply chain security. Because the evidence gathered by the checks is persisted, it is possible to specify policies for "temporal" behavior. We can use the policy engine to make sure the checks that passed historically do not fail on later analyses. Listing 5 presents a policy that detects such regressions for the `ProvenanceAvailable` checks. Given a reference timestamp, if the `ProvenanceAvailable` check fails on any subsequent analysis, the policy fails.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we have presented Macaron to analyze software components for software supply chain security issues. Macaron uses well-defined frameworks, such as SLSA, to check that key properties of both the main component and all its dependencies

Listing 5: Example transitive Datalog policy for SLSA with an exception rule.

```
Policy("no-regression", repo) :-
   reference_timestamp = REFERENCE_TIMESTAMP,
   analysis(reference_timestamp, ref_component),
   // There are no ProvenanceAvailable checks that passed at
         the refernce timestamp that did not pass subsequently.
   0 = count : {
        repository_component(repo, component),
        ProvenanceAvailable(ref_component, "SLSA2"),
        !ProvenanceAvailable(component, "SLSA2")
    }.

apply_policy_to("no-regression", repo) :-
   reference_timestamp = REFERENCE_TIMESTAMP,
   // Enforce the policy any time we analyze the same
         repository in a subsequent analysis.
   analysis(reference_timestamp, ref_component), // Reference
         analysis.
   repository_component(repo, ref_component),
   analysis(timestamp, component), // Other analysis.
   repository_component(repo, component),
   greater(timestamp, reference_timestamp).
```

are satisfied. Macaron also supports analysis and validation of organization-level supply chain security policies. We have evaluated our tool on the top-30 Java, Python, and npm open-source projects. Our results show that the majority still do not produce provenances, while a large number of open-source Java and Python projects do not have a transparent build platform to produce artifacts.

Currently, only two checks related to build integrity are supported. We plan to extend Macaron with more properties that are identified by SLSA and other frameworks. Furthermore, as noted, Macaron needs to be extended with better support for different build and packaging systems.

While our approach is designed to analyze software components and their corresponding source code, some components cannot be linked to the source code accurately. When a provenance is not present or if a commit hash is not provided, we are currently using the latest commit in the corresponding repository. In future, we plan to find the exact commit used to create an artifact. We are also assuming that the metadata, such as source code repository URLs provided by the package developers is correct. However, that might not always be the case [36] and provenances are required to verifiably link an artifact to its source. We are planning to extend Macaron with the extra analyses required for this verification.

Macaron prioritizes verifiable provenances to determine SLSA levels. When such provenances are not present, the checks perform static analysis and use heuristics to infer security properties. Such checks can have false positives and false negatives. The predicates derived by checks are labelled explicitly to help the policy designer interpret the results and make an informed decision. As future work, we plan to improve the labels by providing confidence scores. Moreover, given that Macaron performs static analysis on build-related systems, well understood static analysis techniques, such as dataflow analysis for build systems, package managers and deployment techniques need to be developed in future.

REFERENCES

[1] Moritz Y. Becker, Cédric Fournet, and Andrew D. Gordon. 2010. SecPAL: Design and Semantics of a Decentralized Authorization Language. *J. Comput. Secur.* 18, 4 (2010), 619–665.
[2] Census II dataset 2022. https://project.linuxfoundation.org/hubfs/LFResearch/HarvardCensusIIofFreeandOpenSourceSoftware-Report.pdf
[3] ChainBench 2023. Retrieved June 2023 from https://github.com/aquasecurity/chain-bench
[4] CIS 2023. Center for Internet Security (CIS). Retrieved June 2023 from https://www.cisecurity.org/benchmark/Software-Supply-Chain-Security
[5] Conda package and environment management system 2023. Retrieved June 2023 from https://docs.conda.io
[6] Criticality Score Project 2022. Retrieved November 2022 from https://github.com/ossf/criticality_score
[7] CycloneDX 2023. Retrieved June 2023 from https://cyclonedx.org/
[8] J. DeTreville. 2002. Binder, a logic-based security language. (2002), 105–113. https://doi.org/10.1109/SECPRI.2002.1004365
[9] ecosyste.ms 2023. Retrieved June 2023 from https://ecosyste.ms/
[10] 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 2022ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED).* 3–13. https://doi.org/10.1145/3560835.3564549
[11] Flask: a WSGI web application framework 2023. Retrieved June 2023 from https://github.com/pallets/flask/releases/tag/2.3.2
[12] Pronnoy Goswami, Saksham Gupta, Zhiyuan Li, Na Meng, and Daphne Yao. 2020. Investigating The Reproducibility of NPM Packages. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME).* 677–681. https://doi.org/10.1109/ICSME46990.2020.00071
[13] GUAC: Graph for Understanding Artifact Composition 2023. Retrieved June 2023 from https://github.com/guacsec/guac
[14] in-toto Attestation Framework 2023. Retrieved June 2023 from https://in-toto.io/
[15] Jenkins automation server for building, testing, and delivering or deploying software 2023. Retrieved June 2023 from https://www.jenkins.io/
[16] Legitify 2023. Retrieved June 2023 from https://github.com/Legit-Labs/legitify
[17] J.A. Makowsky. 1987. Why Horn formulas matter in computer science: Initial structures and generic examples. *J. Comput. System Sci.* 34, 2 (1987), 266–292.
[18] MarkupSafe 2023. Retrieved June 2023 from https://github.com/pallets/markupsafe/releases/tag/2.1.3
[19] npm Package Provenance 2023. Retrieved June 2023 from https://github.blog/2023-04-19-introducing-npm-package-provenance/
[20] NTIA: SBOM at a Glance 2023. Retrieved June 2023 from https://www.ntia.doc.gov/files/ntia/publications/sbom_at_a_glance_apr2021.pdf
[21] OSS Gadget 2023. Retrieved June 2023 from https://github.com/microsoft/OSSGadget
[22] Rego policy language 2023. Retrieved June 2023 from https://www.openpolicyagent.org/docs/latest/policy-language/
[23] Rekor 2023. Retrieved June 2023 from https://github.com/sigstore/rekor
[24] Scorecard 2023. Retrieved June 2023 from https://github.com/ossf/scorecard
[25] Semver: The semantic versioner for npm 2023. Retrieved June 2023 from https://www.npmjs.com/package/semver/v/7.5.3
[26] SLSA GitHub Generator 2023. Retrieved June 2023 from https://github.com/slsa-framework/slsa-github-generator
[27] SLSA Verifier 2023. Retrieved June 2023 from https://github.com/slsa-framework/slsa-verifier
[28] Snyk. 2023. Retrieved June 2023 from https://github.com/snyk/parlay
[29] Soufflé: a logic programming language inspired by Datalog 2023. Retrieved June 2023 from https://souffle-lang.github.io/
[30] SPDX: The Software Package Data Exchange 2023. Retrieved June 2023 from https://spdx.dev/
[31] SQLAlchemy 2023. Retrieved June 2023 from https://docs.sqlalchemy.org
[32] SQLite 2023. Retrieved June 2023 from https://www.sqlite.org
[33] Supply-chain Levels for Software Artifacts (SLSA) 2023. Retrieved June 2023 from https://slsa.dev
[34] J. D. Ullman. 1989. Bottom-up Beats Top-down for Datalog. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.* New York, NY, USA, 140–149. https://doi.org/10.1145/73721.73736
[35] VEX: Vulnerability Exploitability eXchange 2023. Retrieved June 2023 from https://www.cisa.gov/sites/default/files/publications/VEX_Use_Cases_Document_508c.pdf
[36] Duc-Ly Vu, Fabio Massacci, Ivan Pashchenko, Henrik Plate, and Antonino Sabetta. 2021. LastPyMile: Identifying the Discrepancy between Sources and Packages. In *ESEC/FSE.*
[37] Witness: Secure Your Supply Chain 2023. Retrieved June 2023 from https://github.com/testifysec/witness
[38] David Zhao, Pavle Subotić, and Bernhard Scholz. 2020. Debugging Large-scale Datalog: A Scalable Provenance Evaluation Strategy. *ACM Trans. Program. Lang. Syst.* (2020).