Finding Fixed Vulnerabilities with Off-the-Shelf Static Analysis

Trevor Dunlap, Seaver Thorn, William Enck, Bradley Reaves
North Carolina State University
Raleigh, North Carolina, USA
{*tdunlap, swthorn, whenck, bgreaves*}*@ncsu.edu*

***Abstract*—Software depends on upstream projects that regularly fix vulnerabilities, but the documentation of those vulnerabilities is often unreliable or unavailable. Automating the collection of existing vulnerability fixes is essential for downstream projects to reliably update their dependencies due to the sheer number of dependencies in modern software. Prior efforts rely solely on incomplete databases or imprecise or inaccurate statistical analysis of upstream repositories. In this paper, we introduce Differential Alert Analysis (DAA) to discover vulnerability fixes in software projects. In contrast to statistical analysis, DAA leverages static analysis security testing (SAST) tools, which reason over code context and semantics. We provide a language-independent implementation of DAA and show that for Python and Java based projects, DAA has high precision for a ground-truth dataset of vulnerability fixes — even with noisy and low-precision SAST tools. We then use DAA in two large-scale empirical studies covering several prominent ecosystems, finding hundreds of resolved alerts, including many never publicly disclosed. DAA thus provides a powerful, accurate primitive for software projects, code analysis tools, vulnerability databases, and researchers to characterize and enhance the security of software supply chains.**

1. Introduction

Open-source software (OSS) plays a significant role in the security of nearly all software used in production. In a recent report, 97% of codebases use OSS components [56]. Unfortunately, a shortcoming is the vulnerabilities within OSS dependencies [26]. There is a long-held assumption that the "many eyes" of developers working on OSS will have fewer vulnerabilities. Nevertheless, vulnerabilities are routinely discovered, even for high-profile projects (e.g., Log4j). These vulnerabilities may be announced through vulnerability databases [21], [43], [17], vendor mailing lists, changelogs, or simply not at all.

Regardless, with or without security advisories, developers and companies struggle to know if they need to update their dependencies [50]. While current advisory approaches bring awareness of security fixes, they often lack details on the vulnerability context. Developers need to know if their project is impacted in a way that matters, so they need insight into where the fix was and what type of fix took place in a security patch. Detailed fix information allows developers to better triage security patches within their software supply chain.

Limited work has been conducted on finding fixes in a code base. However, this prior work produces a binary classification of vulnerability fixes for commits, leaving context about the vulnerability unknown or the precise location [51], [61], [66], [42]. Additionally, once the fix is found, prior work relies on manual analysis to determine if project owners disclosed a fix. Closely related work on discovering vulnerabilities corresponding to a security fix assumes the existence of the fix is known [63], [37]. Therefore, we pose the broad research question: *How can developers find fixes, with context, when they may not be announced — or even known to the developers?*

In this paper, we answer this by introducing Differential Alert Analysis (DAA), a *language-agnostic* algorithm that uses the outputs of lightweight and imprecise off-the-shelf static analysis security tools (SAST) to discover resolved vulnerabilities in software projects without relying on an announcement. The key insight driving DAA is that when a fix is introduced, it will eliminate a SAST alert present in the prior version. We evaluate DAA against Python and Java to measure its detection performance of resolved vulnerabilities using a ground-truth database of 67 vulnerabilities from SafetyDB [48] and Project KB [46]. We show that when using a SAST tool with low precision, DAA had a precision of 98.08% and a recall of 64.56% across a ground-truth database. DAA serves as an accurate *language-independent* technique for identifying vulnerability fixes.

Next, we further demonstrate the value of DAA by performing large-scale breadth and depth studies of NPM, Go, PyPI, and Maven projects using SAST alerts provided by the LGTM [34] platform. Our breadth study considers the last ten commits of each project within those ecosystems. Our depth study is on the latest 1,000 commits for the 1,000 highest depended upon NPM, Go, PyPI, and Maven projects indexed by LGTM. In total, DAA tagged 284 projects with resolved alerts. We then developed an automated approach to determine if an associated security advisory was present for the resolved alert. We found 111 (52.86%) projects with resolved alerts that had no announcements. These findings confirm the necessity of fix discovery techniques independent of prior announcements.

We make the following contributions:

- *We show that differential analysis over SAST alerts can efficiently identify vulnerability fixes in OSS.* DAA leverages the wealth of knowledge encoded into existing SAST tools and bypasses the intense manual collection of ground-truth datasets for training. We demonstrate that even noisy off-the-shelf SAST tools have valuable utility when repurposed with DAA.
- *We scale the analysis of silent fixes to tens of thousands of applications.* The largest dataset studied in

prior work is on the order of tens of applications, whereas we gracefully scale to thousands of applications. Additionally, we provide an automated approach for identifying announcement levels of resolved alerts. DAA is thus a practical mechanism for security vendors and project maintainers to inform developers of vulnerable dependencies.

- *We discover and validate silent fixes throughout multiple ecosystems.* Silent fixes were present in both the most widely used projects and in lesser-used projects for niche functions. We apply DAA to projects analyzed by LGTM and found 237 silent fixes, indicating that announcement agnostic techniques like DAA are essential to secure software supply chains.

We note that whether or not to release a security advisory for a patched vulnerability is a long-time debate for the software community. Some believe silently patching vulnerabilities is the safest option for the community [58]. For those that update software dependencies automatically, the security advisory would not matter. However, recent software supply chain attacks (e.g., SolarWinds [14]) have caused many software companies to reconsider updating dependencies automatically. Furthermore, software companies commonly maintain internal custom forks of open source projects, and importing upstream patches is a manual and time-consuming practice. We believe DAA is a valuable primitive for providing greater transparency for vulnerability fixes and, consequently, will lead to a renewed debate on the benefits of disclosing fixed vulnerabilities.

Availability: We have released our proof-of-concept implementation of DAA[1]. In addition to the software, the ground truth datasets used in the evaluation are available.

The remainder of this paper proceeds as follows. Section 2 motivates and defines our problem. Section 3 overviews our approach. Section 4 describes the design and theoretical intuition behind DAA. Section 5 evaluates DAA detection performance. Section 6 studies the LGTM ecosystem. Section 7 discusses DAA and responsible disclosure to the Global Security Database [9]. Section 8 overviews related work. Section 9 concludes.

2. When Fixes Are Silent

Software maintainers rely on security advisories to determine when to update third-party libraries and dependencies. Unfortunately, the developers of libraries and dependencies may not issue security advisories when they fix a vulnerability, leaving downstream projects unaware of the fix. This section will formally define this notion and other concepts fundamental to this work.

To make this discussion concrete, consider Django Tastypie and TensorFlow Models, two Python projects available in Python Package Index (PyPI), the official repository for Python projects. Both Django Tastypie version 0.9.9 (Figure 1) and TensorFlow Models version 2.4.0 (Figure 2) call `yaml.load()`, a deprecated function allowing arbitrary code execution. Instead, the PyYAML documentation recommends using `yaml.safe_load()` or a safe loader type. Both Djanjo TastyPie and TensorFlow Models fixed these vulnerabilities in subsequent versions, as shown in Figures 1 and 2.

1. https://github.com/s3c2/daa

```
def from_yaml(self, content):
    """
    Given some YAML data, returns a Python dictionary
         of the decoded data.
    """
    if yaml is None:
        raise ImproperlyConfigured("Usage of the YAML
             aspects requires yaml.")

-   return yaml.load(content)
+   return yaml.safe_load(content)
```

Figure 1: `django-tastypie` 0.9.9 to 0.9.10 patch

```
def read_yaml_to_params_dict(file_path):
  """Reads a YAML file to a ParamsDict."""
  with tf.io.gfile.GFile(file_path, 'r') as f:
-   params_dict = yaml.load(f, Loader=yaml.FullLoader)
+   params_dict = yaml.load(f, Loader=yaml.SafeLoader)
    return ParamsDict(params_dict)
```

Figure 2: `tf-models-official` 2.4.0 to 2.5.0 patch

While the patches are nearly identical, the two project maintainers took drastically different approaches to notify others about the fixed vulnerability. Django Tastypie released a security advisory urging users to upgrade to version 0.9.10 [29]. In contrast, a pull request was opened on January 25, 2021, to fix an arbitrary code execution within version 2.4.0 of TensorFlow Models (`tf-models-official`) due to the vulnerable use of `yaml.load()` [24]. An attached proof of concept shows how arbitrary code could be executed through a custom YAML file [28]. The same day, the pull request was merged into the main branch. The official Google team validated the request and awarded a small bug bounty to the issuer of the pull request. Almost four months after the pull request, version 2.5.0 of `tf-models-official` was released to PyPI with the patch. However, no official security advisory was released. These two instances provide clear examples of an *announced* and a *silent* fix, defined as follows.

Definition 1 (Vulnerability Fix $\Delta_{\bar{v}}$)**.** Let P be a software project, and P' be the subsequent version of P. A *vulnerability fix* $\Delta_{\bar{v}}$ has occurred if there exists a vulnerability v in P that is not in P'.

Definition 2 (Announced and Silent Fixes)**.** Let $\Delta_{\bar{v}}$ be the vulnerability fix for a vulnerability v that appears in P but not P'. We say that $\Delta_{\bar{v}}$ is an *announced fix* if there exists a security advisory for P' of v or $\Delta_{\bar{v}}$. Security advisories should be in a location where it would be reasonable for a developer using the project to check for updates, including the information listed in a changelog, a news post owned by the project, a repository-integrated security advisory, or a CVE. Otherwise, we consider $\Delta_{\bar{v}}$ to be a *silent fix* if no mention in a changelog (e.g., notes identifying version changes or making use of release/tags tab within GitHub) or a CVE is available. Because it is unreasonable to expect a developer to read *every* commit in *every* dependency, a commit-log message alone is not a vulnerability announcement.

This paper identifies silent fixes by first identifying vulnerability fixes and then determining whether or not the fix was announced or silent. We discover vulnerability fixes by leveraging the wealth of existing tools to discover security

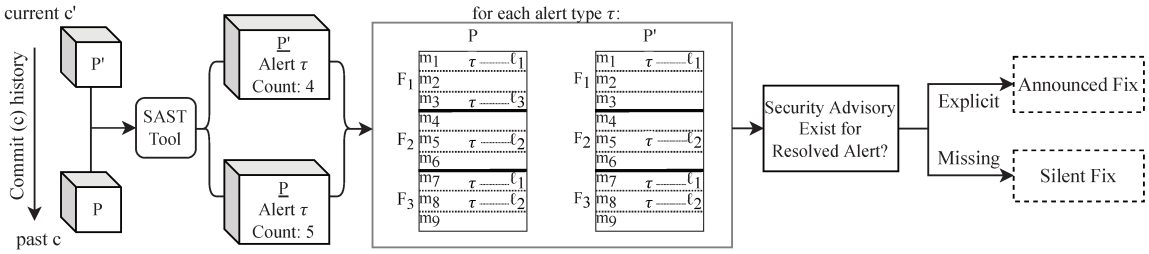

Figure 3: A high-level architecture of our approach.

vulnerabilities. For this paper, we limit our discussion to static analysis security testing (SAST) tools. However, we expect the methodology can be expanded to other types of vulnerability detection tools. Our evaluation in Sections 5 uses the popular tools Bandit [47] and CodeQL [16] for Python, and CodeQL and SpotBugs [54] for Java.

Definition 3 (Alert)**.** Running a SAST tool on a software project P produces a set of alerts A. Each alert $a \in A$ indicates a possible security vulnerability. For most SAST tools, an alert a is a tuple $(\tau, line, file)$ that defines the alert type τ, as well as line location $line$ and filename $file$ where the SAST tool indicates a security vulnerability.

For example, $a = (\texttt{yaml\_load}, 8, \texttt{serializers.py})$ occurs when running a SAST tool on version 0.9.9 of Django Tastypie, shown in Figure 1. The set of alerts produced by running the same SAST tool on version 0.9.10 of Django Tastypie does not include alert a.

Definition 4 (Resolved Alerts $\bar{A}$)**.** Let P be a software project, and P' be the subsequent version of P. Let A and A' be the sets of alerts generated by running a SAST tool on P and P', respectively. The set of resolved alerts $\bar{A} = \{a | a \in A \wedge a \notin A'\}$.

Conceptually, a resolved alert $a \in \bar{A}$ suggests the existence of a vulnerability fix. However, as we discuss in the next section, there are several reasons why resolved alerts may not be a vulnerability fix.

3. Overview

This paper introduces new techniques to identify vulnerability fixes and then determines if they were announced. In this section, we provide an overview of the approach before discussing details in subsequent sections. Our techniques must address the following issues:

- *Identifying vulnerabilities is inherently difficult.* All SAST tools face fundamental limits in detecting vulnerabilities without runtime context, and there are classes of vulnerabilities some or all tools cannot detect. SAST tools can also produce an overwhelming number of false positives [25], [8].
- *Code refactoring complicates identifying and localizing vulnerability fixes.* Software changes disturb line numbering, which SAST tools use to localize and distinguish specific vulnerabilities. Changes can also reorganize logic at the subroutine, class, or file level, which moves the vulnerability and changes an alert between versions. Figure 3 demonstrates alert type, τ, resolving in the third method m_3 of file one F_1 in P'. Alert localization ability disappears when using coarse granularity levels for alert analysis (e.g., project).
- *There is no single standard location or format for fix announcements.* Developers are free to announce fixes in any place they choose in unstructured natural language. The most common locations are changelogs, news posts, and through CVEs.

Our broad approach to addressing the first two challenges is to use Differential Alert Analysis (DAA) with SAST tools. Conceptually, DAA operates by determining the set of resolved alerts $\bar{A}$ from Definition 4. Our work has two key observations. First, DAA eliminates many of the false positives common to static analysis because developers tend not to fix false positives, so alerts remain in later versions. However, as we gained experience with this approach, we discovered that DAA can still have false positives when the code causing a SAST false positive is refactored or deleted. When DAA sees the false positive alerts disappear, it incorrectly assumes the initial positive was correct and marks the removal as a fix. Fortunately, prior work suggests that vulnerability fix commits rarely change the underlying semantics of the program and often change a few lines [35]. To handle code refactoring instances, counting SAST alerts at several levels (i.e., project, file, function, and line level) of the project can eliminate false positives. While the lower granularity levels (e.g., function or line) enable resolved *alert localization*. Figure 3 shows the overall flow of our approach.

Step 1 — Generating Alerts: DAA leverages a SAST tool to generate initial security-related alerts on projects. DAA tracks four levels of granularity for alerts: (1) project, (2) file, (3) function, and (4) line level. Tools commonly emit alerts at the fine-grained level of the line number (i.e., line level). To increase the granularity of the alert location to its function, we use an AST of the file to extract the coarse-grained code elements (e.g., classes and functions). The SAST alerts are then tagged with the coarse-grained code elements for the associated version of the project. Once alerts have been generated, we can then start to identify differences.

Step 2 — Identifying Alert Differences: We rely on DAA to identify resolved alerts. The high-level concept is to extract alerts present in P and not in P'. The four levels of analysis allow for *alert localization* of resolved alerts while still maintaining high precision. Alert counts are taken at each level to determine when alerts are in P and not P'. The project level analysis drives the overall recall of DAA, but without file, function, and line level information, the approach loses the ability to identify the exact resolved vulnerability.

Step 3 — Identifying Silent Fixes: An automated process is then used to determine the announcement level of the resolved alert. In prior work, determining the announcement level relied on a fully manual process [61], [66]. Automating the announcement level process is a multi-step which requires extracting commit metadata, locating the changelog (e.g., notes identifying version changes or making use of the release/tags tab within GitHub), parsing the changelog, and checking the text of the changelog for security related terms. Additionally, advisories exist in security advisory databases (e.g., OSV or GHSA), which we automatically search. Automating the announcement level is critical for scaling DAA to determine if resolved alerts are announced.

4. DAA for Vulnerability Fixes

Identifying vulnerability fixes is a prerequisite to identifying silent fixes. Our Differential Alert Analysis (DAA) methodology identifies vulnerability fixes by running a SAST tool on a software project P and its subsequent version P'. Conceptually, the set of resolved alerts $\bar{A}$ suggests the existence of vulnerability fixes. This section describes how our DAA algorithm works. It then provides a theoretical intuition on DAA accuracy in practice.

4.1. DAA Algorithm

DAA takes as input a SAST tool and two subsequent versions of a software project P and P'. It runs the SAST tool on each commit to collect corresponding alerts. Next, it uses static program analysis to extract additional program context for each alert (e.g., class and function name). This additional function name is needed when performing fine-grained alert localization. DAA then combines the four levels of analysis to produce the set of resolved alerts $\bar{A}$. We note that this section describes the process from the perspective of only *two* commits of a program. In Section 6, the algorithm expands to inspect hundreds of commits for a single program.

4.1.1. Differential Alert Analysis. The high-level logic for DAA is in Algorithm 1. P' is the child commit of P. Using a time-based approach on commits is incorrect due to merges and thus requires a parent-child relationship to capture the linear progression of code changes. We consider the alert type, τ, unique to the desired SAST tool and do not change the context for the tool between runs. Finally, DAA considers resolved alerts at various levels to handle code refactoring and alert localization: project, file, function, and line levels.

The DAA algorithm begins by running the SAST tool on both P and P', producing corresponding sets of alerts A and A'. Conceptually, to determine if an alert a in A is not in A', we need a way to correlate alerts in A to alerts in A'. However, the alert information provided by most SAST tools (Definition 3) only provides the filename and line location. DAA must also extract the function name for the code triggering the alert. Specifically, it uses $FindCodeContext(\cdot)$ (Algorithm 2) to identify the function, method, class, or file scope most appropriate for considering each alert in A and A', adding this context to the alert data structure. For more detail, we describe $FindCodeContext(\cdot)$ in Section 4.1.2.

Algorithm 1: DAA to produce $\bar{A}$ from P to P'

```
   Input: P; P';
   Output: Resolved alerts Ā
1  DAA(P, P'):
2      Ā, A, A' = ∅, SAST(P), SAST(P')
3      foreach a ∈ A do
4          a.function = FindCodeContext(a, P)
5      end foreach
6      foreach a' ∈ A' do
7          a'.function =
             FindCodeContext(a', P')
8      end foreach
9      foreach a ∈ A do
10         τ = a.τ
11         if count(τ, A) > count(τ, A')
             ∧ count(τ, A.file) > count(τ, A'.file)
             ∧ count(τ, A.function) >
             count(τ, A'.function)
             ∧ count(τ, A.line) > count(τ, A'.line)
             then
12             Ā.append(a)
13         end if
14     end foreach
15     return Ā
```

In the final phase, the DAA algorithm determines if the disappearance of the alert results from a vulnerability fix or a code refactor. Recall from Definition 4 that $\bar{A} = \{a | a \in A \wedge a \notin A'\}$. This definition makes sense if alerts in A can be matched directly to alerts in A'. In Figure 3, an alert resolved from P to P'. The alert, τ, is resolved in the third method m_3 of the first file F_1. Considering the project alerts at the coarsest level (i.e., project) will lose the ability to identify which alert resolved. To address this, DAA takes the intersection of four levels of analysis from the set of alerts: project, file, function, and line. The groupings count how many specific vulnerability types are within each level. For example, if A has n amount of $a.\tau$ and A' has $n-1$ amount of $a.\tau$, then a vulnerability must have been resolved. DAA then uses the finer-grain levels (i.e., file, function, and line) to maintain alert localization capabilities. Conceptually, granularity-based matching improves the overall precision of DAA while maintaining the ability to localize the resolved alert down to the line level.

DAA cannot account for the case when one vulnerability resolves, but the same type of vulnerability is re-introduced somewhere else. For example, the vulnerable code for alert a was in function `foo` in P and resolved, but function `bar` in P' introduces the same type of vulnerability. Our DAA algorithm will not include a in $\bar{A}$, resulting in a false negative. While we do not anticipate this to be a common scenario in practice, as few lines typically change during a patch [35], it is nevertheless a possibility. We dive deeper into the state space of such scenarios in Section 4.2.

4.1.2. Function Identification. As discussed in Section 4.1.1, SAST tools commonly only output the file and line numbers of the code related to each alert. Since

Algorithm 2: Extract code context names (e.g., function F, class C, method M) from a project P for a given alert a.

Input: Alert a; Project P;
Output: Alert code context $context$

```
FindCodeContext(a, P):
    name = a.file
    F = ∅; C = ∅; M = ∅
    foreach node ∈ ParseAST(P, a.file) do
        if node.type = function then
            F.append(node)
        else if node.type = class then
            C.append(node)
            foreach child ∈ node.children do
                M.append(child)
            end foreach
        end if
    end foreach
    foreach e ∈ F ∪ M ∪ C do
        if e.line_start ≤ a.line ≤ e.line_end
         then
            name = e.name
            break
        end if
    end foreach
    return name
```

line numbers can easily change, DAA needs to extract appropriate code context for the line producing the alert. Ideally, this context is the name of a function or method within a class. However, not all alerts triggered are within functions, classes, or methods. In such cases, the file is the location of the alert. We note that as long as this name is determined consistently for both P and P', DAA only needs to retain a string for the name.

Algorithm 2 shows how $FindCodeContext(\cdot)$ produces a context name from an alert a and project P. Since the SAST identifies the filename, the algorithm only needs to consider that file from the project. The first step is to extract the file's abstract syntax tree (AST). $FindCodeContext(\cdot)$ then iterates through all nodes, identifying functions, classes, and member functions. The second step compares the line number of the alert with the line number of each identified node. Note that the union of $F \cup M \cup C$ retains the order such that $FindCodeContext(\cdot)$ prioritizes functions over methods over classes over files. The loop breaks when a match occurs, and the resolved name is returned. The filename is returned by default if no line number match is identified.

4.2. DAA Intuition

Previously we discussed the design of DAA; now, we discuss the intuition of how DAA works. Consider the underlying SAST tool's outputs on a single project P. SAST tools either emit alerts (positive) or do not emit alerts (negative). True positives (TP) are alerts of true vulnerabilities. False positives (FP) are alerts on non-vulnerable code. False negatives (FN) are vulnerabilities that the SAST tool misses; thus, it does not emit any alerts. True negatives (TN) are correct rejections to alert on non-vulnerable code. Precision is the proportion of true positives. That is, $\texttt{precision} = \frac{TP}{TP+FP}$. Recall is the proportion of vulnerabilities detected. That is, $\texttt{recall} = \frac{TP}{TP+FN}$.

DAA has outputs of type positive (true positive and false positive) and type negative (true negative and false negative) alerts. For DAA to produce an alert, a SAST alert transition of type positive to type negative must occur between P to P'. For example, a $\texttt{TP}_\texttt{P}$ is patched in P and becomes a $\texttt{TN}_{\texttt{P}'}$ in P', representing a $\texttt{DAA}_\texttt{TP}$. We refer to these as *alert transitions*.

Figure 4 demonstrates the alert transitions to generate a $\texttt{DAA}_\texttt{TP}$ and a $\texttt{DAA}_\texttt{FP}$. $\texttt{DAA}_\texttt{TP}$ was previously discussed; it is the alert transition of $\texttt{TP}_\texttt{P}$ to $\texttt{TN}_{\texttt{P}'}$ (solid arrow 4 in Figure 4). This alert transition represents a true resolved vulnerability. A $\texttt{DAA}_\texttt{FP}$ can be obtained through three alert transitions. The first is a $\texttt{TP}_\texttt{P}$ to $\texttt{FN}_{\texttt{P}'}$ (dotted arrow 2 in Figure 4). This alert transition is a resolved vulnerability and only appears to be fixed. An example would be a code refactor that altered the vulnerability from P to P' so that it is still present but not detected. The second $\texttt{DAA}_\texttt{FP}$, is an alert transition from $\texttt{FP}_\texttt{P}$ to $\texttt{FN}_{\texttt{P}'}$ (dotted arrow 6 in Figure 4). These alert transitions represent a new vulnerability in P', but one the underlying SAST would miss, appearing as a resolved vulnerability. The final $\texttt{DAA}_\texttt{FP}$, is an alert transition from $\texttt{FP}_\texttt{P}$ to $\texttt{TN}_{\texttt{P}'}$ (dotted arrow 8 in Figure 4). By construction, this alert transition is true — an FP is not a vulnerability — but through some alteration of the project, the developer removed the alert in P'.

Paths to generate recall of DAA can be seen in Figure 5 based on the underlying $\texttt{SAST}_\texttt{recall}$ and paths to determine a $\texttt{DAA}_\texttt{TP}$ and a $\texttt{DAA}_\texttt{FN}$. We previously discussed the $\texttt{DAA}_\texttt{TP}$ and now will discuss the intuition of a $\texttt{DAA}_\texttt{FN}$. A $\texttt{DAA}_\texttt{FN}$ consists of three paths. The first, a $\texttt{TP}_\texttt{P}$ to $\texttt{FP}_{\texttt{P}'}$ (dotted arrow 3 in Figure 5). This alert transition represents a resolved vulnerability, but the alert still appears in the P'. The second, a $\texttt{FN}_\texttt{P}$ to $\texttt{FP}_{\texttt{P}'}$ (dotted arrow 7 in Figure 5). This alert transition represents a resolved vulnerability not initially detected by the underlying SAST tool but now emits an alert in P'. The third, $\texttt{FN}_\texttt{P}$ to $\texttt{TN}_{\texttt{P}'}$ (dotted arrow 8 in Figure 5). This alert transition represents a resolved vulnerability the underlying SAST tool missed and would not emit an alert in P'. We consider the $\texttt{SAST}_\texttt{recall}$ the probability out of all vulnerabilities within P to be how many are of condition true positive or false negative.

DAA alert transition types are complex, but the probabilities of those transitions are crucial to how well DAA performs. Because a $\texttt{DAA}_\texttt{FP}$ will be relatively rare (especially compared to the SAST FP rate), the $\texttt{DAA}_\texttt{precision}$ ratio approaches 1. This provides an intuition for the findings in Table 2 that DAA can have high precision even with low precision tools. We show these probabilities within our evaluation in the next section and that it is unlikely developers resolve SAST false positives. We also note that the underlying SAST tool will determine the maximum number of resolved vulnerabilities DAA can detect. If the SAST tool can not initially detect the vulnerability in P, DAA will not detect if the vulnerability resolves in P'.

4.3. Announcement Level Identification

DAA is responsible for determining if alerts from SAST tools resolve, but it does not determine if devel-

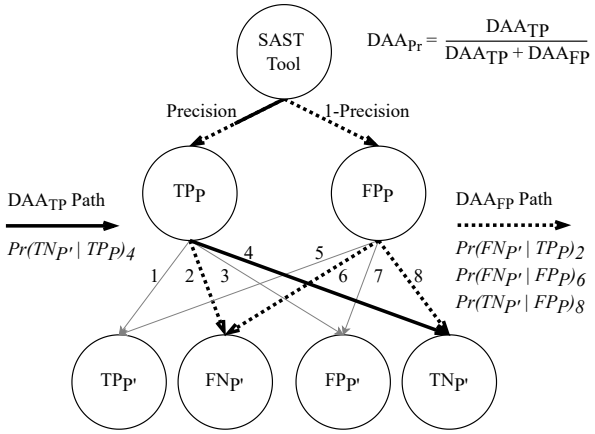

Figure 4: Generating true positives and false positives in DAA in terms of SAST precision. Solid bold lines indicate a path to a TP, while dotted lines show paths to an FP.

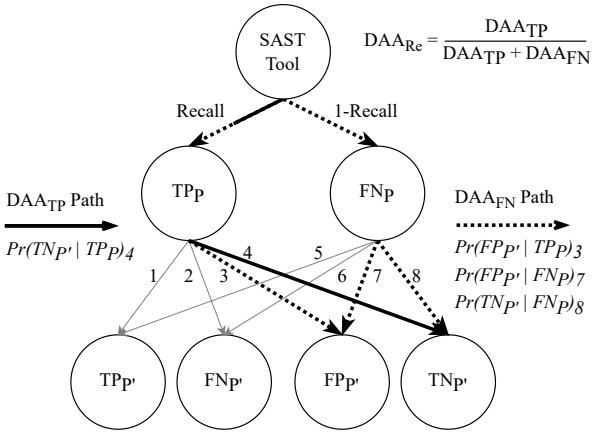

Figure 5: The dotted lines show paths to generate false negatives in DAA in terms of SAST recall. Generating true positives is the same as in Figure 4.

opers announced the fix. To scale the search for silent fixes, we automate a process to determine resolved alert announcement levels. The general concept is that a security announcement should be clearly stated as a security-related fix and be in an accessible location, such as a changelog within a repository. Changelogs come in various formats (e.g., changelog.txt, news.rst, history.md); in some cases, developers use the release tab of GitHub to communicate changes. In addition, CVEs can bring awareness of security fixes. In this subsection, we discuss the methodology of obtaining changelogs from GitHub repositories and how to determine if a security announcement took place, and if there are any CVEs for the fix.

Announcement Level Method: The initial step is to obtain commit metadata. Extracting commit metadata involves acquiring the commit title, commit message, commit date, and the associated commit version tag. We obtain the commit message and tags through the GitHub API [18]. GitHub tags point to a time along the commit history tree and indicate a commit associated with a particular version. For the scope of this paper, we are only concerned with tags and commits on the main branch.

To obtain the changelog, we rely on the project `changelogs` released by PyUp.io [49]. The PyUp project searches a GitHub repository for standard changelog naming conventions (e.g., changelog.txt, news.rst, history.md) to identify the associated changelog. In addition, we obtain the release tab of GitHub as some developers will announce changes in such locations. Built into the `changelogs` project is a parsing feature, which will parse a changelog into separate blocks based on the version within the text.

With the parsed changelog and commit metadata, we then match the block of changelog to the associated commit version tag. Within the associated block, we run git-vuln-finder [10], which finds security-related messages within text based on regexes [4], [67]. We believe it is unlikely that developers would not use security-related terms when disclosing a vulnerability fix.

Any matches in the changelog imply an *announced fix*, while no matches in the changelog text imply a *silent fix*. We discuss the results of this automated announcement analysis within Section 6.3.

Matching CVEs to Resolved Alerts: To match CVE data, we rely on Open Source Insights (OSI) [20]. This Google-based platform gathers known security advisories from the GitHub Security Advisories Database (GHSA) [17] and the Open Source Vulnerability Database (OSVDB) [21] to match advisories to projects. Both GHSA and OSVDB include official CVEs from the national vulnerability database [43]. OSI then handles the process of matching advisories related to the correct project and version. In addition, the advisories contain reference links. These reference links, for example, point to the GitHub commit patch links that fixed the security issue.

We first download the OSI advisory table from the Google BigQuery snapshot [19] to match the resolved alerts to OSI. The dataset contains a title and description of the vulnerability, reference links, project name, affected versions, and the disclosure date. The project name and reference links are key matching points for our pipeline. We first match the resolved alerts on the project name to determine if any matches exist for a given project. If so, we search for the associated commit with a resolved alert in the reference links. Matches on the project name and commit patch reference links are then marked as announced with a known CVE. We discuss the evaluation results of the matching process in Section 6.3.

5. Evaluation

This section describes how we collected a ground truth dataset and selected the necessary underlying tools to perform DAA. We then evaluate the underlying SAST tool against the vulnerable projects in the ground truth dataset. Finally, we evaluate DAA over the dataset in which the SAST tools produce true positives. While DAA will work on any language the underlying tools support, we evaluate DAA for Python and Java due to their popularity.

5.1. Evaluation Setup

We first require a labeled dataset of software with resolved vulnerabilities for both Python and Java. Various databases exist that identify resolved vulnerabilities in Python and Java but lack the exact location of the vulnerability. To identify the location of the vulnerability, we

use information from within the report of the associated CVE, which is cataloged by the National Vulnerability Database (NVD) [43]. In more recent years, the report of a CVE contains a patch link (e.g., a GitHub commit fixing the vulnerability). We obtained changed lines from the GitHub diff of the commit for assistance in identifying the vulnerability location.

Python Ground Truth Dataset: We used the SafetyDB database [48] to identify CVEs associated with vulnerability patch links to PyPI projects for the ground truth dataset. SafetyDB database contains disclosed vulnerabilities in Python projects hosted on PyPI, the Python Package Index used by virtually all Python developers. SafetyDB comes from a team at `pyup.io` that filters CVEs and changelogs looking for vulnerabilities related to PyPI projects. The database continued to receive updates as of our data collection in November 2021. The data in SafetyDB contains the PyPI project name, advisory, CVE-ID (where available), and the vulnerable versions. We only target resolved vulnerabilities with associated CVEs containing patch links to a GitHub repository. We obtained 174 CVEs accounting for 94 unique PyPI projects.

While we believe our approach is reasonable and appropriate, it does have some limitations. We assume the correctness of SafetyDB and that the patch link reported in the CVE is correct. We also must assume the vulnerability is resolved in the version immediately after the last listed vulnerable version in the SafetyDB tag. If this assumption were false, it would imply an error in the database entry. Finally, when a SafetyDB entry did not contain a pointer to a CVE with a link to the relevant patch, we excluded the vulnerability from analysis simply because we could not be confident about where the fix should be.

Java Ground Truth Dataset: We rely on a pre-built database by Ponta et al. [46] that identifies fixes to open-source Java projects containing 624 security fixes. The data was manually built over four years and open-sourced in 2019. The database contains 1,282 commits for security fixes pointing to the project repository, CVE, and commit-patch link for 205 open-source Java projects used within SAP products. As noted in the opening of Section 5.1, we still need to extract `git diff` information of the commit-patch link to obtain changed lines for identifying the potential vulnerability location. Discussion on validating vulnerability location occurs in Section 5.2.

SAST Tool Selection: As discussed in Section 4, DAA relies on an underlying SAST tool. We have selected two tools for evaluation purposes for Python: Bandit [47] and CodeQL [16]. We evaluate our DAA approach for Java using CodeQL and SpotBugs [54]. Bandit, CodeQL, and Spotbugs are oriented toward detecting security-related flaws in software and are actively maintained. CodeQL, used in Section 6 for our ecosystem analysis, is adopted by GitHub as the primary tool for scanning project code [15].

Bandit is explicitly designed to find security flaws in Python, is lightweight, and is actively maintained. Bandit analyzes the AST for each file in a project and produces alerts at a line-level granularity. CodeQL is a versatile tool that can run checks for several languages. CodeQL uses data-flow analysis to find flaws in code. CodeQL creates a queryable database extracted from code and uses an object-oriented query language (.QL) to allow end-users to search for known bug patterns. CodeQL relies on the build process for Java to extract syntactic and semantic data. For Python, CodeQL extracts information directly from the source code. SpotBugs is a static analysis tool designed specifically for Java applications. SpotBugs checks for over 400 bug patterns and has some specifically related to security. Like CodeQL, SpotBugs needs to compile Java-based projects to scan the bytecode. More detailed information regarding each tool is in their respective documentation.

Adapting DAA to SAST tools is straightforward. Generally, SAST tools export alerts with line level and file name granularity. DAA only needs to extract the function name (Algorithm 2). DAA can then be applied with alerts at the project, file, function, and line levels. We excluded commercial tools in our study due to not having licenses for them. In the following section, we discuss the evaluation process of these tools on ground truth data.

5.2. SAST Evaluation

SAST Tool Evaluation Method: The goal of evaluating the underlying SAST tool is to determine the initial precision and recall values to validate the DAA theory previously described in Section 4. To do so, we must identify the true and false positives that each tool produces. We also use the true positives as the primary evaluation set for DAA, as discussed in Section 5.3.

We began by running the selected SAST tools across all parent commits of the commit-patch link in the ground truth dataset. The parent commit, in theory, should be the source code before the patch and still contain the vulnerability concerning the CVE. Bandit and SpotBugs are run using default settings. For CodeQL, we used two query packs (*python-security-and-quality* and *python-security-extended*) to search for security related flaws in Python. For Java, we used the *java-security-and-quality* and *java-security-extended* query packs. When running CodeQL, we confirmed current scans overwrote any existing databases from a previous scan of the target projects.

For easy build management of the Java ground truth database, we constructed a Jenkins pipeline integrating CodeQL and SpotBugs to scan the associated projects. For simplicity, we only targeted Maven-based projects and used the Maven 3.5.4 JDK-8 docker images in Jenkins for the build environments. We have a four-stage build process for the Jenkins CI/CD. The first stage is the build stage which invokes CodeQL to create a database using Maven. We skipped test cases and used a clean install for each target project when compiling with Maven. The second stage analyzes the compiled CodeQL database with the desired query packs. In the third stage, we run SpotBugs using Jenkins's built-in plugins. The final stage was copying alert records (i.e., CodeQL and SpotBugs reports) outside the Docker environment for future analysis. Due to the simplicity of scanning Python projects, we opted for a custom-built script to run Bandit and CodeQL.

Once alerts return for the associated target project, we distinguish false positives from true positives. We automatically identified the set of potential true positive alerts by comparing the alert location to the locations of the changed code from the commit patch. To establish ground truth, we manually validate each tagged alert and determine if the alert was directly related to the vulnerability. If the alert matched the root cause of the vulnerability from the

TABLE 1: Evaluation of SAST tools on ground truth datasets. Precision is calculated with alert TP, and Recall is calculated with CVE TP.

Tool	GT CVE	TP CVE	TP Alert	FP Alert	Precision Alert	Recall CVE
Python-Bandit	174	25	64	35,871	0.18%	14.37%
Python-CodeQL	174	14	28	18,028	0.16%	8.05%
Java-SpotBugs	130	13	41	181,739	0.01%	10.00%
Java-CodeQL	130	15	25	10,494	0.24%	11.54%

commit patch link, we could be confident the alert is a true positive. Otherwise, if an alert is not within the location of the changed code from the commit patch link, we know it is unrelated to the vulnerability. We conservatively assume all of those alerts to be false positives; if they are indeed vulnerable, it would merely improve the reported precision of the tool. The count of false negatives is the difference between the number of known vulnerabilities within the project and the number of true positives identified. We retain these metrics at the vulnerability type level to help us gain insight into the relationship between SAST metrics and DAA in Section 5.3.

5.2.1. SAST Tool Results. SAST tools are well-known to produce false positives [25], [52], [8]. Understanding the type of false positives produced by the underlying SAST tools is outside the scope of this paper. We report results of each SAST tool across the entire project rather than the commit's diff. The deciding factor was to show that despite the number of false positives produced by a tool, DAA can still perform with high precision.

Python SAST Tool Evaluation Results: The two Python SAST tools correctly identified 39 CVEs compared to Python's 174 CVE ground-truth dataset. For Python, we scan all 94 projects for the 174 CVEs. Between Bandit and CodeQL, an overlap of 6 CVEs occurred. Table 1 provides a breakdown of the SAST evaluation. Precision was calculated based on alerts and recall at the CVE level. Recall at the CVE level is appropriate since we are only aware of the vulnerabilities described in the projects.

Bandit had a precision of 0.18% and recall of 14.37% against the vulnerable projects in the ground truth dataset. Bandit produced 35,935 alerts across the vulnerable parent commit of projects in the ground truth dataset. We concluded 64 of those alerts to be true positives, resulting in 35,871 false positives and 149 false negatives concerning the CVE count. The most common detection true positive type for Bandit is related to improper neutralization of special elements in output used by downstream components, also known as injections. Precisely, cross-site scripting (CWE-79) vulnerabilities with five CVEs were the most common and followed by operating system command injections (CWE-78) with three associated CVEs. The three most common missed vulnerabilities were type cross-site scripting (CWE-79), open redirects (CWE-601), and improper input validation (CWE-20).

CodeQL for Python had a precision of 0.16% and recall of 8.05% against the vulnerable projects in the ground truth dataset. We concluded 28 of those alerts to be true positives, resulting in 18,028 false positives and 160 false negatives to the CVE count. The most common detection true positive type for CodeQL is related to open redirects (CWE-601) associated with four CVEs. The next most common vulnerability type was of type cross-site scripting (CWE-79), assigned to three CVEs. Similar to Bandit, the most missed vulnerability was for cross-site scripting (CWE-79), followed by improper input validation (CWE-20), and then open redirects (CWE-601).

Java SAST Tool Evaluation Results: For Java, issues arose when compiling the projects. We could only compile 46.94% of the projects accounting for 130 CVEs, which align with previous work when compiling Java applications [60], [22]. The Ponta et al. database was released in 2019 and contained CVEs in 2007. The older projects lead to a natural removal of dependencies from their respective repository, making compilation impossible.

SpotBugs for Java had a precision of 0.01% and recall of 10.00% against the vulnerable projects in the ground truth dataset. We concluded 41 of those alerts to be true positives, resulting in 181,739 false positives and 117 false negatives with the CVE count. The greater number of alerts in SpotBugs can be explained by having more security checks integrated within the tool. The most common detection true positive type for SpotBugs is improper input validation (CWE-20) associated with three CVEs and permissions, privileges, and access control vulnerabilities (CWE-264) assigned to two CVEs. The three most missed vulnerability types for SpotBugs were improper input validation (CWE-20), deserialization of untrusted data (CWE-502), and cross-site request forgeries (CWE-352).

CodeQL for Java had a precision of 0.24% and recall of 11.54% against the ground truth dataset. We concluded 25 of those alerts to be true positives, resulting in 10,494 false positives and 115 false negatives with the CVE count. The most common detection true positive type for CodeQL is path traversal vulnerabilities (CWE-22) associated with six CVEs. The next most common vulnerability type was of type improper input validation (CWE-20), assigned to five CVEs. The most missed vulnerability was for improper input validation (CWE-20), deserialization of untrusted data (CWE-502), and then cross-site scripting (CWE-79). With the underlying performance of the SAST tools, we can now evaluate DAA.

5.3. DAA Evaluation

We evaluate DAA against a ground truth dataset to demonstrate the following research questions:

RQ1: *Does DAA work better than simple line-level approaches for identifying resolved vulnerabilities (e.g., LGTM)?*

RQ2: *How does DAA performance relate to the underlying SAST tool performance?*

5.3.1. DAA Evaluation Method. To evaluate DAA, we run the ground truth vulnerable parent commits and patched commits through our DAA technique described in Section 4.1. The entire process was automated using a Jenkins pipeline to run the SAST tools and a Python implementation of Algorithm 1 and 2 for DAA.

The output of DAA will emit true positives or false positives. True positives in terms of DAA are the resolved vulnerabilities. False positives are alerts emitted by DAA not corresponding to true resolved vulnerabilities. False

TABLE 2: Evaluation results of industry standard (line-level) for identifying resolved alerts vs. DAA for identifying resolved alerts on ground truth data based on 67 CVEs. Precision and recall values are based on the labeled alert counts from the underlying SAST evaluation in Section 5.2.

Language	Tool	Ground Truth		Industry Standard (Line-Level)					DAA				
		CVE Total	Alert Total	TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall
Python	Bandit	25	64	57	152	7	27.27%	89.06%	43	0	21	100.0%	67.19%
Python	CodeQL	14	28	19	147	9	11.45%	67.86%	14	0	14	100.0%	50.00%
Java	SpotBugs	13	41	39	99	2	28.26%	95.12%	35	2	6	94.59%	85.37%
Java	CodeQL	15	25	17	1	8	94.44%	68.00%	10	0	15	100.0%	40.00%
Totals		67	158	132	399	26	24.86%	83.54%	102	2	56	98.08%	64.56%

negatives are when DAA misses resolved vulnerabilities. The number of resolved vulnerabilities DAA detects is driven by the number of true positives the underlying SAST tool initially detects. Due to this fact, we only run DAA on the projects for which the underlying SAST tool produced a true positive in Section 5.2. When considering these projects, we still account for all possible checks from the underlying SAST tool emitted during the scan. Doing so results in thousands of alerts from the SAST tool that DAA has to analyze. The DAA evaluation ground truth dataset consists of 67 CVEs from real Python and Java-based open-source projects.

We also compare the *industry standard* (line-level) technique for identifying resolved vulnerabilities [34]. The line-based analysis is a naive differential alert analysis that only considers the type of vulnerability and line-level of alerts. The line-based approach is prone to false positives triggered by code refactors. For example, minor refactors, such as moving functions within the file, can trigger fixes using the line-based approach.

5.3.2. DAA Evaluation Results (RQ1). DAA resulted in an average precision of 98.08% and an average recall of 64.56% across the four separate evaluations based on alerts in Table 2. When comparing DAA to the industry standard line-level approach, our approach increases precision by 294.53% while only having a 22.72% decrease in recall. Note we only ran DAA and the industry-level approach evaluation against entire projects where the SAST tool could detect a TP. Therefore, the related recall is a calculation of the TP alert counts from Section 5.2. As predicted, DAA performed with high precision despite our test set's dismal (though typical for SAST) precision of Bandit, CodeQL, and SpotBugs. As Section 4 mentions, DAA only emits alerts when a positive alert was in P and transitions to a negative alert in P'. The critical property that allowed DAA to perform with high precision is that the probability of an FP in P resolving to an alert of type negative in P' was nearly negligible as only two FPs transitioned in our ground truth dataset. The reason false positives do not resolve from P to P' is apparent: they are not vulnerabilities, so developers do not fix them.

DAA detected true positives for various vulnerability types. For Python with Bandit, DAA identified 43 TP alerts. The most common type of TP was for the improper control of a resource through its lifetime, which had five associated CVEs. For Python with CodeQL, DAA found 14 TP alerts. The most common alert types were a combination of improper control of a resource through its lifetime (three CVEs) and improper neutralization (three CVEs). DAA also detected two other CVEs related to improper access controls. For Java with CodeQL, DAA discovered 10 TP alerts. DAA identified three CVEs related to improper control of a resource through its lifetime and three of improper neutralization. Regarding SpotBugs, DAA detected 35 TP alerts and observed the widest variety of CWE true positives. These TPs included alerts related to improper neutralization, access control, credential management errors, cryptographic issues, and improper control of a resource throughout its lifetime.

For recall, DAA will only detect a fix if the underlying SAST tool detects the initial vulnerability, the primary limitation of DAA. The overarching theme among missed vulnerabilities for DAA is the CWE class of improper neutralization of special elements in output used by downstream components (CWE-74). For example, vulnerabilities missed by DAA added sanitization of input before the downstream function or component call. The primary limitation of such instances is the SAST tool's inability to verify a sanitized data flow and issue a true negative for the fixed version. The higher recall for the line-level approach can be misleading due to the underlying weaknesses of the SAST tool. Even though the alerts missed by DAA remain in the P', they only shifted a few lines during the patch, triggering a positive alert for the line-based approach. We found that DAA outperforms the industry standard line-level approach. In the next section, we dive deeper into the underlying relationship between SAST metrics and DAA.

Takeaway: DAA finds resolved vulnerabilities in a ground-truth dataset, outperforming the industry standard line-level method with an increase of 294.53% in precision and only a 22.72% reduction in recall.

5.3.3. DAA Underlying SAST Relationship (RQ2). To gain insight into the relationship between the performance of DAA and SAST tools, we evaluated the results at a granular level of vulnerability type. As detailed in Section 5.2, we established the performance of each SAST tool at the level of individual vulnerability types, specifically their checks. We calculated the underlying SAST precision/recall for each tool's check and vice-versa for DAA. The analysis provides valuable insights into how DAA operates with SAST performance, enabling us to understand the dynamics between the two better.

Specifically, we conducted a multilinear regression to analyze the relationship between SAST precision and recall (independent variables) and DAA precision and recall (dependent variables). We perform separate regression analyses for DAA precision and recall. We consider all

TABLE 3: Multilinear regression analysis between the underlying SAST metrics and DAA metrics. SAST recall is a significant predictor for DAA recall, while the other SAST metrics have little impact on DAA precision.

Independent Variables	Dependent Variable	Coefficient	$p > \|t\|$
SAST Recall	DAA Recall	0.59	0.004
SAST Precision		-0.16	0.68
SAST Recall	DAA Precision	-0.09	0.11
SAST Precision		0.03	0.75

false and true positive alert types from DAA for the regression analysis for DAA precision. If DAA triggered an alert, we would consider the other alerts of the same type from the SAST tool. As shown in Table 2, our ground truth analysis consisted of 158 true positive alerts from the underlying SAST tools. Considering the same checks that generated true positives, we have 36,508 false positives alerts from the underlying SAST tools. We use these positive alerts to perform the regression analysis regarding DAA precision. For DAA recall, we consider all of the positive alerts along with the false negatives that were missing by DAA. Table 3 presents the regression results for the dependent variable of DAA precision and recall to the underlying SAST metrics. Next, we discuss the SAST relationship with DAA recall and precision.

The regression analysis shows a strong correlation between SAST recall and DAA recall. The coefficient for SAST recall is 0.59, indicating that a one-unit increase in SAST precision corresponds to a 0.59 unit increase in DAA recall. The p-value of 0.004 indicates a statistically significant relationship. However, the coefficient for SAST precision is -0.16, indicating an inverse relationship with DAA recall. The p-value of 0.68 for SAST precision shows no statistically significant relationship to DAA recall. For example, Bandit's recall for the check permissions, privileges, and access control vulnerabilities (CWE-264) was 80%, while the check for cross-site scripting (CWE-79) was 32%. Additionally, both checks had a precision under 1%. DAA achieved 100% recall for CWE-264 and 40% for CWE-79, demonstrating a higher SAST recall results in a higher DAA recall. Overall, SAST recall significantly impacts DAA recall, while SAST precision has little effect.

Furthermore, the regression analysis revealed that the underlying metrics of the SAST tool have minimal impact on DAA precision. The lower coefficient values of SAST recall (-0.09) and SAST precision (0.03) suggest any increase or decrease in SAST metrics has a negligible effect on DAA precision. For example, CodeQL's precision for the check related to a path traversal vulnerability (CWE-22) was 13.6% and a recall of 66.7%. The check XML external entity references (CWE-611) had a precision of 0.47 and a recall of 15%. These underlying checks for DAA produced a precision of 100%, demonstrating that neither the precision nor recall impacted DAA precision. As previously mentioned, the high precision of DAA is driven primarily by developers not fixing false positives simply because they are not actual vulnerabilities. Therefore, one should focus on SAST recall to obtain optimal DAA results.

Takeaway: Despite low-precision SAST tools, DAA produces high-precision results. Furthermore, increasing SAST tool recall will increase DAA recall.

6. Ecosystem Study

We now seek to apply DAA at an ecosystem scale. Because DAA relies on SAST alerts, we can leverage an existing alert database made available by LGTM, a code review platform [2]. The LGTM platform scans an entire project's GitHub commit history with CodeQL and provides alerts for each commit. We focus on projects from NPM, PyPI, Maven, and Go ecosystems, as these are the intersection of scanned languages on LGTM and the available data from Open Source Insights [20]. The Open Source Insights (OSI) platform is a project hosted by Google to catalog dependency information (e.g., full dependency graphs, advisories) from various ecosystems. We break our dataset into two parts: a breadth and depth collection. The breadth study consists of running DAA on the latest ten commits for all available projects scanned on LGTM from the NPM, PyPI, Maven, and Go ecosystems. The depth study targets the latest 1,000 commits on the top 1,000 most depended upon projects from NPM, PyPI, Maven, and Go that have been scanned by LGTM. We start our discussion by describing the data collection and analysis methods, then provide results and case studies from the DAA analysis across projects in LGTM. We seek to demonstrate that DAA finds vulnerability fixes *regardless of their disclosure to the public.*

6.1. LGTM Methods

Data Collection: The initial step is to obtain projects from NPM, PyPI, Maven, and Go. We used a Google BigQuery Data snapshot released by OSI to obtain the available projects quickly [19]. The data contains project version information, dependency graphs for each project, and associated security advisories. The data is available to query and download.

From a March 3, 2022 scrape of LGTM using their APIs, the platform contained 104,121 projects from NPM, PyPI, Maven, and Go. Most of the projects came from NPM, with 62,551 scanned projects on LGTM, followed by Go (18,649), PyPI (16,143), and then Maven (6,778). We use these projects as the basis for a breadth and depth analysis. We scraped at a rate of 5,000 API calls per hour to avoid significant stress on the LGTM servers.

Breadth Data Collection: For the breadth analysis, we restrict the data collection process to the latest ten commits on the intersection of LGTM scanned projects and OSI data. We used GitHub APIs to obtain commit information on the latest ten commits for each project before March 5, 2022. While not all projects had ten commits, we confirmed at least two commits exist for the project to allow for DAA. We automatically confirmed the commits held a parent-child relationship when extracting commit information. In total, we obtain alerts for 535,000 commits across 104,121 projects for the breadth analysis.

2. Since publication, LGTM was deprecated and moved scanning capabilities to GitHub [34].

Depth Data Collection: For the depth study, we targeted the latest 1,000 commits on the 1,000 most depended-upon projects from NPM, PyPI, Maven, and Go ecosystems. We defined most depended-upon projects as a ranking of projects with the highest number of direct and transitive dependents. The Open Source Insights platform compiled each project's dependency graph and released the data in a Google Big Query snapshot to allow for a simple query.

In total, 1,980 of the most depended upon projects have been scanned by LGTM from our initial scrape on March 3, 2022. The breakdown of projects for each ecosystem is as follows: NPM (720 projects), Go (541 projects), PyPi (545 projects), and Maven (174 projects). The initial target was to pull the latest 1000 commits for each project. When scraping each project's GitHub repository for commits, we found the average number of commits to be 387 across the 1,980 projects, representing the entire lifecycle for most projects. In total, we successfully obtained alerts for 406,222 commits across 1,980 projects.

LGTM Challenges: The LGTM alert data posed a challenge for DAA. The returned data from LGTM was unreliable when CodeQL produced alerts, reducing the number of commits in our ecosystem analysis. For example, the project *Celery With Flask* is scanned on LGTM. The project runs a Flask application in debug mode, which triggers a CodeQL alert for allowing an attacker to run arbitrary code through the debugger. Figure 6 in the appendix shows the parent commit, `fc45071`, returns an alert from the LGTM API. When querying the API for the child commit, `c3bbeeb` (Figure 7), which only updates the `requirements.txt` file, the alert is no longer returned by the API. The subsequent commit to the child, `76f785e` (Figure 8), only updating the `README.md` file, reports the Flask alert for the project from the API. We only considered resolved alerts changing the associated file during the commit using git diff commit information to avoid such instances. We discuss the results and totals in Section 6.2.

Classifying Resolved Alerts: With DAA alerts obtained, we are next interested in two key questions: *Are these alerts correct?* and *Was this fix publicly disclosed?*. The first question will validate the precision of DAA in this setting, while the second question will give us a window into how well DAA works in leading us to undisclosed vulnerability fixes. Our claims on this second point will be conservative: silent fixes are indeed a phenomenon that occurs with some regularity. By using LGTM and CodeQL, and the nature of vulnerability discovery in general, we cannot claim to definitively identify all or even a majority of fixes, silent or otherwise.

Because this section aims to explore fixes without even partial ground truth, we rely on our automated announcement pipeline to help identify announced alerts, thus reducing effort on our end. To validate our findings, we conduct a principled manual review to confirm our automated announcement pipeline results. The manual review determines whether a vulnerability is fixed and validates whether project owners disclose the fix. The manual review process is in the Appendix A.

As mentioned in Section 2 the fixed announcement location can vary in prominence and specificity; therefore, we have created a hierarchy to classify the announcement. An **announced fix** is one where the *specific vulnerability fix* is mentioned in a place where it would be reasonable for a developer to use the project to check for updates. Our other classification for fixes is **silent fixes**. These silent fixes will contain no mention in the changelog or a CVE. For our purposes, a "silent fix" may have a note in a commit log or a code comment but not in one of the locations mentioned earlier. Our justification for this decision is that it is unreasonable to expect a developer to review every commit message and code comment for every project they include in their software.

6.2. Fixes in LGTM

In this subsection, we discuss our results from applying DAA to the LGTM data, and then we validate these findings and classify whether the fixes were silent or announced. We demonstrate that DAA is a precise tool for identifying fixes and that silent fixes are sufficiently common to merit concern. We discuss both the breadth and depth results providing high-level takeaways and examples of silent fixes, announced fixes, and false positives.

In the breadth analysis of the ten most recent commits across the four ecosystems scanned on LGTM, DAA produced 7,241 alerts representing potential security patches. For the depth analysis, DAA initially produced 3,749 potentially resolved security alerts. As described in Section 6.1, we noticed inconsistent responses from the LGTM platform on returned CodeQL alerts. Therefore, we filtered alerts based on if files changed during the commit were also the files with an associated alert. After automated filtering, DAA tagged 172 projects accounting for 304 security-related resolved alerts for the breadth analysis and 112 projects accounting for 209 alerts for the depth analysis. We note that these are purely security-related alerts as defined by CodeQL. We also confirmed that DAA did not duplicate alerts between the breadth and depth analysis; any initial duplicated alerts were counted within the breadth analysis. Next, we determine the correctness and advisory classification of these resolved alerts.

6.2.1. Fix Validation Advisory Classification. As previously discussed, we are interested in the correctness of DAA and the correctness of our automated announcement levels process of the fixes. Table 4 displays the resolved alerts by project and alert count aggregated by CWE pillar type. The results presented in the table have been manually validated for correctness, along with the announcement level. We discuss the performance of our automated announcement pipeline in Section 6.3. The following section discusses more alert fix classifications with samples from both silent and announced fixes, as well as false positives.

Silent Fixes: From our DAA alerts, we confirmed 111 projects with silent fixes, accounting for 237 alerts across the breadth and depth study. We noted a surprising amount of silent fixes related to improper neutralization, such as reflective cross-site scripting and log injections for the breadth study. In the depth analysis, we noted many fixes related to improper control of resources, such as regex denial of service (ReDoS) fixes. Additionally, we saw fixes related to integer overflows, which ranked 13th in the most dangerous software weaknesses of 2022 [40]. Despite these fixes, project owners did not provide any form of a security advisory for the patched vulnerability.

TABLE 4: LGTM DAA classification by CWE pillar type across the breadth (172 projects) and depth (112 projects) analysis.

CWE Pillar Type	Breadth Analysis						Depth Analysis					
	Silent		Announced		False Positive		Silent		Announced		False Positive	
	Proj.	Alert	Proj.	Alert	Proj.	Alert	Proj.	Alert	Proj.	Alert	Proj.	Alert
CWE-284: Improper Access Control	22	79	2	2	2	5	2	3	2	2	0	0
CWE-710: Improper Coding Standards	7	7	2	2	0	0	5	7	5	7	0	0
CWE-664: Improper Control of a Resource	21	26	19	29	27	44	10	19	26	38	13	24
CWE-707: Improper Neutralization	18	34	17	27	15	22	6	8	9	13	6	12
CWE-682: Incorrect Calculation	8	9	4	4	1	6	8	39	7	14	6	14
CWE-693: Protection Mechanism Failure	2	2	3	4	2	2	2	4	3	3	2	2
Totals	78	157	47	68	47	79	33	80	52	77	27	52
Percentage of Whole	45.35%	51.64%	27.33%	22.37%	27.33%	25.99%	29.46%	38.28%	46.43%	36.84%	24.11%	24.88%

An example silent fix is within three high-profile NPM projects fixing ReDoS vulnerabilities. Those three projects are node-tar (18.7M weekly downloads), clean-CSS (12.7M weekly downloads), and ua-parser-js (7.8M weekly downloads). Specifically, in node-tar, on August 3, 2021, a developer pushed a commit to fix a potential regular ReDoS vulnerability. While the developer notes in the commit log that the vulnerability is unlikely to be exploited, it would be theoretically possible to exploit it if a user passes untrusted input into a particular function. Various releases appeared for the project, but the repository's changelog has not been updated since May 24, 2020. We consider the fix to be a silent fix in a highly used project.

For another silent fix example, we reference the Python project streamlit, which allows for easy transformation of data scripts into web applications [55]. On April 21, 2022, a developer opened a commit to remove a path from a 404 response, which falls under a stack trace exposure vulnerability. Such vulnerabilities could lead to an attacker learning sensitive information. The developer simply removed the path from the response code to fix the vulnerability. Deeper in the pull request, the developer acknowledged they should not risk exposing such information through the browser. In the changelog, the project failed to mention any vulnerability type surrounding information exposure or stack trace exposure.

We also note that some of the alerts resolved in Java fall under dependency issues for using JFog Bintray, which houses JCenter for hosting Java artifacts. Access to the deprecated artifact repository JFrog could lead to eventual supply chain attacks from attackers targeting deprecated repositories. A Yahoo-owned project, EGADS, on December 22, 2021, removed the Bintray dependency link and the Log4j dependency from the projects `pom.xml` file, following the Log4J vulnerability (CVE-2021-44228). Based on Defintion 4, we consider this a silent fix as no changelog exists for the project, and maintainers removed the dependencies from the project.

Announced Fixes: From our sampled data, we confirmed 47 projects from the breadth analysis and 52 from the depth analysis with announced fixes. The most commonly announced fix among the data was for ReDoS attacks. Generally announced fixes were mentioned in the changelog and often added the initial pull request or a commit SHA that points back to the security fix.

An example announced fix from the breadth study is from a WebSocket emulation Python-based server, SockJS-tornado [53]. The project has 857 stars on GitHub, with 2,089 other projects depending on SockJS-tornado. On October 19, 2018, a user opened an issue to fix an URL path XSS vulnerability. The issue includes a proof of concept to replicate the vulnerability on the users' local host. The same day, a project owner merged a fix into the main branch to escape the user's callback input. The project owner also bumped the project to version 1.0.6 on PyPI within the same day, accompanied by a changelog stating: *XSS security fix for the HTMLFILE transport*.

TABLE 5: Evaluation of the automated announcement pipeline on LGTM results across the 382 announced and silent fixes from Table 4.

	TP	FP	FN	TN	Precision	Recall	F1
Breadth	157	39	0	29	80.10%	100.00%	88.95%
Depth	74	51	6	26	59.20%	92.50%	72.20%
Totals	231	90	6	55	71.96%	97.47%	82.80%

False Positives: While expected, the precision dropped from the evaluation to the LGTM ecosystem analysis due to more deletions and advanced code refactors. The false positives were generally related to refactors or changed context undetectable by CodeQL after the change, thus not triggering an alert. For example, one project with 21 associated resolved alerts entirely moved code to a new repository and then used the new repository as an import.

Another example of a false positive is from the Apache Airflow project [57]. The original CodeQL alert regarded an untrusted URL redirection in four locations. A developer pushed a commit for some code cleanup to remove duplicating code and create a single function with the same functionality that contained the untrusted URL redirection. DAA detected that the overall count of alerts of the same type was reduced, producing resolved alerts. We consider this a weakness of CodeQL to not correctly understand the code's data flow to realize the function is still called four times throughout the program.

6.3. Automated Announcement Evaluation

During the manual evaluation process of the LGTM DAA alerts, we also assessed the correctness of the automated announcement pipeline. We evaluate the breadth and depth analysis approach across 382 positive alerts from Section 6.2.1. As mentioned in the previous section, when evaluating the DAA results, we confirmed the project uses a changelog and the announcement level (i.e., announced in the changelog or a CVE) for a fix. We would then read the changelog and confirm that the developers either announced or did not announce the security-related fix from DAA. For the CVE match, we initially matched against

all of the project names and then confirmed whether the DAA fix was related to the CVE. True positives for the automated announcement pipeline are correctly labeled silent fixes. False positives are labeled as silent, but the project announced the fix. False negatives are labeled as announced, but the project silently fixes the security-related issue. True negatives are correctly labeled announced fixes.

Table 5 displays the results from the analysis across the 382 DAA alerts. Overall, the automated pipeline produces an acceptable F1 score of 82.80%. In rare cases, the pipeline would produce false negatives, for example, labeling fixes as announced, but the fix was silently patched. These instances occurred when the announcement was related to a different fix and not the fix reported by DAA. False positives were generated in cases when the pipeline labeled a fix as silent but was announced. An example is when a project does not maintain a changelog within GitHub but on its product's website. Other false positive cases were when the text did not contain standard security terms. For example, the text *Fix unsafe shell command constructed from library input* is in a release. In such instances, git-vuln-finder cannot detect the context of the message. Future research should explore more advanced approaches to determine the context of these messages as security-related fixes.

In terms of matching CVEs, we were able to correctly detect eight advisories from the OSI database for true positive DAA alerts. Of particular interest is the project *path-parse* with nearly 26M downloads per week and used by 13.8M. On May 13, 2021, the project pushed a patch to avoid a ReDoS attack. CVE-2021-23343 was published on May 4, 2021, disclosing the possible attack. The project does not have a changelog and fails to mention updates in the releases tab of GitHub, meaning if users do not use other sources (e.g., NVD, OSI, DAA), they, too, would miss such a critical patch. The automated pipeline missed two advisories from OSI, which did not have the commit patch link directly in the reference links. We did find the commits for a missing reference through an external proof of concept link, and the other commit was referenced through a pull request. The method of matching purely on the project name and commit patch links proved effective.

6.4. Threats to Validity

Like all research, the analysis of LGTM-enabled projects has several validity threats. The principal concern is that Section 5 showed that each SAST tool has a low recall for known vulnerabilities. Because we have no ground truth for the LGTM projects, we can assume that we only see a fraction of the total vulnerabilities. Second, our study is not longitudinal, as it is meant only to demonstrate the scalability and precision of DAA. Third, while we relied on independent review and a thorough process for validating each alert, we may have misclassified an alert just as in any other human endeavor.

7. Discussion

Throughout the paper, we worked with a commit level analysis. The decision to pursue the commit level represents what DAA would encounter in the real world (e.g., through CI/CD implementations or how LGTM operates). Sometimes, a commit level analysis may not be appropriate, as an alternative semantic version level analysis for DAA is possible. That is when P is the semantic version (SemVer[3]) that comes immediately after P'. In an earlier approach to DAA, we used semantic versions to evaluate and perform an empirical study of the PyPI ecosystem.

Similar to our current approach, our previous approach enhanced the set of alerts produced by the SAST tool with additional program context but then used existing code-clone detection tools to identify the existence of refactoring. Using code-clone detection tools restricted DAA resulting in a lower recall than our current approach of a hierarchy analysis. Despite this, we applied the technique to a May 2021 snapshot of PyPI on the latest two semantic versions using only Bandit as the underlying SAST tool. Similarly, we found 58.33% of resolved alerts were silent fixes.

Responsible Disclosure: Prior to disclosure, we emailed developers informing them of our plans to release the fixes publicly. We gave developers an appropriate timeline (30 days) to respond before the disclosure; two developers requested non-disclosure, which we did not include in the public release. After the timeline, we submitted our current silent fixes findings to the Global Security Database [9] to disclose the results from Section 6.2.1. The Global Security Database is an open-source community project supported by the Cloud Security Alliance that aims to improve the quality and usability of vulnerability databases by involving the community in a collaborative effort. During the previous iteration, we disclosed the findings to PyUp, the parent company of SafetyDB [48]. Their engineers validated the vulnerabilities and added 55 projects to their vulnerability database. The two databases provide security updates to thousands of companies.

8. Related Work

Our work relates to multiple areas: static analysis for security, large-scale vulnerability detection, identifying vulnerability fixes, and differential static analysis.

Static Analysis in Security: Static analysis is now standard practice for finding security flaws [7], [2], [5] in projects of virtually every major language ecosystem [13], [65], [23], [38], [27], [62], [36]. While static analysis can be effective, analysts must overcome poor initial configuration, faulty warning messages, and frequent false positives [25], [52], [8], [41]. Analysts must also accept that static analysis cannot detect all vulnerabilities [1], [64], the use of multiple tools can improve recall [11].

Static analysis is particularly well-suited for large-scale analysis ranging from projects to entire ecosystems [30]. In fact, static analysis has demonstrated how security vulnerabilities can propagate through software supply chains [3], [45]. Zimmermann et al. [68] recently analyzed the npm ecosystem and found that up to 40% of projects have a dependency with a known vulnerability. They also discovered that, on average, an NPM project would depend on 79 third-party projects, severely increasing the attack surface. Duan et al. [12] found 339 malicious projects across registries (PyPI, Npm, and Ruby Gems) by applying metadata, static, and dataflow analysis techniques. Kula

3. major.minor.patch (https://semver.org/)

et al. [31] found across 4,600 GitHub projects that nearly 81.5% of them have outdated dependencies.

Identifying Vulnerability Fixes: Prior work explored discovering vulnerability fixes with machine learning. These approaches all suffer from similar weaknesses that DAA surpasses: intensive ground truth datasets continuously requiring updates for training and restricted to specific languages, binary classification of commits without vulnerability context, unknown resolved vulnerability locations, and manual effort to determine announcement levels. Some approaches focus on identifying security-relevant fix commits by building classifiers on the commit log message and patch information of a commit [59], [51] and more recently by appending commit-issue links [42]. Wang et al. [61] use 61 different features that may appear in a patch (e.g., the number of changes for specific operators) to detect vulnerability fixes. VulFixMiner [66] uses deep learning at a commit level to detect vulnerability fixes on 52 projects. Additionally, these prior approaches scale to only tens of projects, whereas we scale to thousands.

Differential Static Analysis: Differential static analysis has started to attract researchers [33] but has yet to use different granularity levels to identify resolved vulnerabilities. Brumley et al. [6] leveraged static and dynamic analysis to determine various check differences between two programs to build exploits for unpatched software versions. Partush and Yahav [44] focused purely on C to compute the abstract semantic difference between two programs, one being the unpatched version and the other being the patched version. Differential assertion checking (DAC) verifies the correctness of assertions over subsequent program versions and pairs with SymDiff to verify bug fixes [32]. Unlike our approach, Verification Modulo Versions (VMV) [39] focuses on identifying new alerts and reducing static analysis alarms using semantic information from previous program versions. SPAIN [63] considers two binary versions of a project and discovers security patches by comparing control flow graphs and semantic function count to determine if a patch is security-related. However, SPAIN is less scalable than DAA, requiring more heavyweight analysis and knowing when developers might have applied security patches.

LGTM [34] is a platform running CodeQL against OSS projects, and then a user interface displays the alert types and location. While not a forefront feature, they also display new and fixed alerts throughout the commit history. LGTM uses three strategies to understand new and fixed alerts. The first is location-based matching, which considers the starting line position of the alert. Second, snippet-based matching considers the start and end positions of the alerts. Finally, the hash-based alert matching considers the surrounding code to handle refactored code. While LGTM attempts to handle refactors for fixes, their approach is too fine-grained and often imprecise, leading to false positives.

9. Conclusion

Software vendors increasingly rely on security advisories for open-source software dependencies to ensure the security of their products. Unfortunately, not all vulnerability fixes are announced and pose harm to the unaware end consumer. This paper introduced Differential Alert Analysis (DAA), an algorithm that uses lightweight and imprecise static analysis security testing (SAST) tools to discover the existence of vulnerabilities in software projects. We described our design and proof-of-concept implementation of DAA for various tools and languages. Using a ground-truth dataset of vulnerabilities for both Java and Python, we showed that DAA provides very high precision even with a noisy and imprecise SAST tool. We further used our DAA tool to evaluate a March 2022 snapshot of the LGTM projects in a breadth and depth study, including the most depended projects, in four separate ecosystems, finding a variety of silent fixes. We demonstrate that DAA can provide a valuable primitive for transparency to aid software vendors who rely on open-source software.

Acknowledgements

This work is supported in part by NSF grants CNS-1946273 and CNS-2207008. Any findings and opinions expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] H. H. AlBreiki and Q. H. Mahmoud, "Evaluation of static analysis tools for software security," in *2014 10th International Conference on Innovations in Information Technology (IIT)*, Nov. 2014, pp. 93–98.

[2] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using Static Analysis to Find Bugs," *IEEE Software*, vol. 25, no. 5, pp. 22–29, Sep. 2008.

[3] D. Balzarotti, M. Monga, and S. Sicari, "Assessing the risk of using vulnerable components," in *Quality of Protection*, ser. Advances in Information Security, D. Gollmann, F. Massacci, and A. Yautsiukhin, Eds. Boston, MA: Springer US, 2006, pp. 65–77.

[4] G. Barish, M. Michelson, and S. Minton, "Mining commit log messages to identify risky code," in *Proceedings on the International Conference on Artificial Intelligence (ICAI)*, 2017, pp. 345–349.

[5] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, Feb. 2010. [Online]. Available: https://doi.org/10.1145/1646353.1646374

[6] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*, May 2008, pp. 143–157, iSSN: 2375-1207.

[7] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security Privacy*, vol. 2, no. 6, pp. 76–79, Nov. 2004.

[8] M. Christakis and C. Bird, "What developers want and need from program analysis: an empirical study," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. Singapore, Singapore: Association for Computing Machinery, Aug. 2016, pp. 332–343. [Online]. Available: https://doi.org/10.1145/2970276.2970347

[9] Cloud Security Alliance, "Global security database." [Online]. Available: https://gsd.id/

[10] CVE-Search, "Finding potential software vulnerabilities from git commit messages." 2022. [Online]. Available: https://github.com/cve-search/git-vuln-finder

[11] A. Delaitre, B. Stivalet, P. E. Black, V. Okun, A. Ribeiro, and T. S. Cohen, "SATE V report: ten years of static analysis tool expositions," National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep. NIST SP 500-326, Oct. 2018. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.500-326.pdf

[12] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages," *arXiv:2002.01139 [cs]*, Dec. 2020, arXiv: 2002.01139. [Online]. Available: http://arxiv.org/abs/2002.01139

[13] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE Software*, vol. 19, no. 1, pp. 42–51, Jan. 2002.

[14] FireEye, "Highly evasive attacker leverages solarwinds supply chain to compromise multiple global victims with sunburst backdoor," https://www.fireeye.com/blog/threat-research/2020/12/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor.html, Dec. 2020.

[15] GitHub, "About code scanning - GitHub Docs," 2022. [Online]. Available: https://docs.github.com/en/code-security/code-scanning/automatically-scanning-your-code-for-vulnerabilities-and-errors/about-code-scanning

[16] GitHub, "Codeql: the libraries and queries that power security researchers around the world." 2022. [Online]. Available: https://codeql.github.com/

[17] GitHub, "GitHub Advisory Database," 2022. [Online]. Available: https://github.com/github/advisory-database

[18] GitHub, "GitHub REST API - GitHub Docs," 2022. [Online]. Available: https://docs.github.com/en/rest

[19] Google, "Dependencies, vulnerabilities and licenses of open source software," 2022. [Online]. Available: test

[20] Google, "Open Source Insights," 2022. [Online]. Available: https://deps.dev/

[21] Google, "OSV - Open Source Vulnerabilities," 2022. [Online]. Available: https://github.com/google/osv.dev

[22] F. Hassan, S. Mostafa, E. S. L. Lam, and X. Wang, "Automatic building of Java projects in software repositories: a study on feasibility and challenges," in *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '17. Markham, Ontario, Canada: IEEE Press, Nov. 2017, pp. 38–47. [Online]. Available: https://doi.org/10.1109/ESEM.2017.11

[23] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM SIGPLAN Notices*, vol. 39, no. 12, pp. 92–106, Dec. 2004. [Online]. Available: https://doi.org/10.1145/1052883.1052895

[24] hunter-helper, "Security Fix for Arbitrary Code Execution - huntr.dev by huntr-helper · Pull Request #9669 · tensorflow/models," 2021. [Online]. Available: https://github.com/tensorflow/models/pull/9669

[25] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 672–681, iSSN: 1558-1225.

[26] P. Johnson. Top 10 open source vulnerabilities in 2020. [Online]. Available: https://resources.whitesourcesoftware.com/blog-whitesource/top-security-open-source-vulnerabilities-2020

[27] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: a static analysis tool for detecting Web application vulnerabilities," in *2006 IEEE Symposium on Security and Privacy (S P'06)*, May 2006, pp. 6 pp.–263, iSSN: 2375-1207.

[28] A. Kalam and A. Aboobacker, "Code Injection in tensorflow/models," 2020. [Online]. Available: https://huntr.dev/bounties/1-other-models

[29] J. Kaplan-Moss, "Piston and Tastypie security releases issued | Weblog | Django," 2011. [Online]. Available: https://www.djangoproject.com/weblog/2011/nov/01/piston-and-tastypie-security-releases/

[30] S. Kim, S. Woo, H. Lee, and H. Oh, "VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery," in *2017 IEEE Symposium on Security and Privacy (SP)*, May 2017, pp. 595–614, iSSN: 2375-1207.

[31] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, Feb. 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9521-5

[32] S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel, "Differential assertion checking," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. Saint Petersburg, Russia: Association for Computing Machinery, Aug. 2013, pp. 345–355. [Online]. Available: https://doi.org/10.1145/2491411.2491452

[33] S. K. Lahiri, K. Vaswani, and C. A. R. Hoare, "Differential static analysis: opportunities, applications, and challenges," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, ser. FoSER '10. Santa Fe, New Mexico, USA: Association for Computing Machinery, Nov. 2010, pp. 201–204. [Online]. Available: https://doi.org/10.1145/1882362.1882405

[34] LGTM, "LGTM - Code Analysis Platform to Find and Prevent Vulnerabilities," 2022. [Online]. Available: https://lgtm.com/

[35] F. Li and V. Paxson, "A Large-Scale Empirical Study of Security Patches," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, Oct. 2017, pp. 2201–2215. [Online]. Available: https://doi.org/10.1145/3133956.3134072

[36] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, "Static analysis of android apps: A systematic literature review," *Information and Software Technology*, vol. 88, pp. 67–95, Aug. 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584917302987

[37] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "VulPecker: an automated vulnerability detection system based on code similarity analysis," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*. Los Angeles California USA: ACM, Dec. 2016, pp. 201–213. [Online]. Available: https://dl.acm.org/doi/10.1145/2991079.2991102

[38] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," in *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, ser. SSYM'05. Baltimore, MD: USENIX Association, Jul. 2005, p. 18.

[39] F. Logozzo, S. K. Lahiri, M. Fähndrich, and S. Blackshear, "Verification modulo versions: towards usable verification," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 294–304, Jun. 2014. [Online]. Available: https://doi.org/10.1145/2666356.2594326

[40] MITRE, "CWE - CWE-1387: Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses (4.8)," 2022. [Online]. Available: https://cwe.mitre.org/data/definitions/1387.html

[41] T. Muske and A. Serebrenik, "Survey of Approaches for Handling Static Analysis Alarms," in *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Oct. 2016, pp. 157–166, iSSN: 2470-6892.

[42] G. Nguyen-Truong, H. Jin Kang, D. Lo, A. Sharma, A. Santosa, A. Sharma, and A. Ming Yi, "Hermes: and Using Commit-Issue Linking to Detect Vulnerability-Fixing Commits," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Mar. 2022.

[43] NIST, "National Vulnerability Database." 2022. [Online]. Available: https://nvd.nist.gov/

[44] N. Partush and E. Yahav, "Abstract Semantic Differencing for Numerical Programs," in *Static Analysis*, ser. Lecture Notes in Computer Science, F. Logozzo and M. Fähndrich, Eds. Berlin, Heidelberg: Springer, 2013, pp. 238–258.

[45] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vulnerable open source dependencies: counting those that matter," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '18. Oulu, Finland: Association for Computing Machinery, Oct. 2018, pp. 1–10. [Online]. Available: https://doi.org/10.1145/3239235.3268920

[46] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi, and C. Dangremont, "A manually-curated dataset of fixes to vulnerabilities of open-source software," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 383–387.

[47] PyCQA, "bandit: Security oriented static analyser for python code." [Online]. Available: https://bandit.readthedocs.io/en/latest/

[48] PyUp, "Safety DB: A curated database of insecure python packages." [Online]. Available: https://github.com/pyupio/safety-db

[49] PyUp.io, "changelogs - A changelog finder and parser for packages available on pypi, npm and rubygems." 2022. [Online]. Available: https://github.com/pyupio/changelogs

[50] E. Remaley, "Software Bill of Materials Elements and Considerations (NTIA–2021–0001)," Jun. 2021. [Online]. Available: https://www.ntia.doc.gov/files/ntia/publications/uscc_-_2021.06.17.pdf

[51] A. Sabetta and M. Bezzi, "A Practical Approach to the Automatic Classification of Security-Relevant Commits," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2018, pp. 579–582, iSSN: 2576-3148.

[52] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford, "Questions developers ask while diagnosing potential security vulnerabilities with static analysis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, Aug. 2015, pp. 248–259. [Online]. Available: https://doi.org/10.1145/2786805.2786812

[53] SockJS-tornado, "WebSocket emulation - Python server," 2022. [Online]. Available: https://github.com/mrjoes/sockjs-tornado

[54] SpotBugs, "Find bugs in Java Programs." 2022. [Online]. Available: https://spotbugs.github.io/

[55] Streamlit, "Streamlit - the fastest way to build data apps in python." 2022. [Online]. Available: https://github.com/streamlit/streamlit

[56] Synopsys. [Analyst Report] 2022 Open Source Security and Analysis Report | Synopsys. [Online]. Available: https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html

[57] The Apache Software Foundation, "Apache Airflow - A platform to programmatically author, schedule, and monitor workflows," 2022. [Online]. Available: https://github.com/apache/airflow

[58] G. Thomas. Additional Fixes in Microsoft Security Bulletins – Microsoft Security Response Center. [Online]. Available: https://msrc-blog.microsoft.com/2011/02/14/additional-fixes-in-microsoft-security-bulletins/

[59] Y. Tian, J. Lawall, and D. Lo, "Identifying Linux bug fixing patches," in *2012 34th International Conference on Software Engineering (ICSE)*, Jun. 2012, pp. 386–396, iSSN: 1558-1225.

[60] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "There and back again: Can you compile that snapshot?: There and Back Again: Can you Compile that Snapshot?" *Journal of Software: Evolution and Process*, vol. 29, no. 4, p. e1838, Apr. 2017. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/smr.1838

[61] X. Wang, K. Sun, A. Batcheller, and S. Jajodia, "Detecting "0-Day" Vulnerability: An Empirical Study of Secret Security Patch in OSS," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Jun. 2019, pp. 485–492, iSSN: 1530-0889.

[62] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages," in *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, ser. USENIX-SS'06. Vancouver, B.C., Canada: USENIX Association, Jul. 2006.

[63] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, "SPAIN: Security Patch Analysis for Binaries towards Understanding the Pain and Pills," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, May 2017, pp. 462–472, iSSN: 1558-1225.

[64] A. Xypolytos, H. Xu, B. Vieira, and A. M. T. Ali-Eldin, "A Framework for Combining and Ranking Static Analysis Tool Findings Based on Tool Performance Statistics," in *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, Jul. 2017, pp. 595–596.

[65] J. Yang, T. Kremenek, Y. Xie, and D. Engler, "MECA: an extensible, expressive system and language for statically checking security properties," in *Proceedings of the 10th ACM conference on Computer and communications security*, ser. CCS '03. Washington D.C., USA: Association for Computing Machinery, Oct. 2003, pp. 321–334. [Online]. Available: https://doi.org/10.1145/948109.948153

[66] J. Zhou, M. Pacheco, Z. Wan, X. Xia, D. Lo, Y. Wang, and A. E. Hassan, "Finding A Needle in a Haystack: Automated Mining of Silent Vulnerability Fixes," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2021, pp. 705–716, iSSN: 2643-1572.

[67] Y. Zhou and A. Sharma, "Automated identification of security issues from commit messages and bug reports," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, Aug. 2017, pp. 914–919. [Online]. Available: https://doi.org/10.1145/3106237.3117771

[68] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small-world with high risks: a study of security threats in the npm ecosystem," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC'19. Santa Clara, CA, USA: USENIX Association, Aug. 2019, pp. 995–1010.

1. Manual Inspection Methodology

1) An automated set of files containing the following information is generated to familiarize the reviewer with the resolved alert they are reviewing.
   - Project Name
   - LGTM Link
   - GitHub Commit Link
   - Alert Info
   - Alert Type
   - Alert Location (File, Function, Line)
2) The next process is to confirm the vulnerability fix through a commit in the associated repository for the project.
   - Follow the GitHub commit link within the file.
   - Go to the line location with the fixed vulnerability in the repository.
   - Confirm the change in code that fixed the alert.
     - Validation is simply noting developer changed the code that initially tagged the alert to a safer form.
     - E.g., `yaml.load`→`yaml.safe_load`
3) Find the changelog associated with the project, generally within the repository/releases/tags tab. In some cases, projects maintain a changelog on their own site.
   - Typical files for a changelog (HISTORY.rst, CHANGES.md, CHANGELOG.MD, etc...).
   - "Releases" tab in GitHub
4) Review the changelog for the associated patched version of the project and determine the level of the announcement.
   - Announced: Explicit/Implicit mention of vulnerability fix
   - Silent: No mention
   - See Definition 2
5) Finally, search for a project (i.e., OSVDB, GHSA, OSI) to determine if a CVE exists to determine if classification changes.

Figure 6: Parent commit (fc45071) for project Miguelgrinberg/flask-celery-example


```
{
  "version": "2.0.0",
  "runs": [
    {
    "tool": {
      "name": "Semmle",
      "version": "1.31.0-SNAPSHOT",
    },
    "files": {
      "app.py": {
      "fileLocation": {
        "uri": "app.py"
      }
      }
    },
    "results": [
      {
      "ruleId": "python-queries:py/flask-debug",
      "partialFingerprints": {
        "primaryLocationLineHash": "592eb5113a7053ce:1"
      }
      }
    ],
    "resources": {
      "rules": {
      "python-queries:py/flask-debug": {...}
      }
      }
    }
  ]
}
```


Figure 7: Child commit (c3bbeeb) for project Miguelgrinberg/flask-celery-example with missing results.


```
{
  "version": "2.0.0",
  "runs": [
    {
    "tool": {
      "name": "Semmle",
      "version": "1.31.0-SNAPSHOT",
      "language": "en-US"
    },
    "resources": {
      "rules": {}
    }
    }
  ]
}
```


Figure 8: Subsequent commit (76f785e) for project Miguelgrinberg/flask-celery-example with matching results from the initial parent commit.


```
{
  "version": "2.0.0",
  "runs": [
    {
    "tool": {
      "name": "Semmle",
      "version": "1.31.0-SNAPSHOT",
    },
    "files": {
      "app.py": {
      "fileLocation": {
        "uri": "app.py"
      }
      }
    },
    "results": [
      {
      "ruleId": "python-queries:py/flask-debug",
      "partialFingerprints": {
        "primaryLocationLineHash": "592eb5113a7053ce:1"
      }
      }
    ],
    "resources": {
      "rules": {
      "python-queries:py/flask-debug": {...}
      }
      }
    }
  ]
}
```