A scalable, effective and simple Vulnerability Tracking approach for heterogeneous SAST setups based on *Scope+Offset*

James Johnson*, Julian Thome‡†, Lucas Charles§† Hua Yan¶†, Jason Leasure‖†,
†GitLab Inc.
*jamxjohn@gmail.com, ‡jthome@gitlab.com, §lcharles@gitlab.com, ¶hyan@gitlab.com, ‖jleasure@gitlab.com

***Abstract*—Managing software projects using Source Control Management (SCM) systems like Git, combined with automated security testing in Continuous Integration and Continuous Delivery (CI/CD) processes, is a best practice in today's software industry. These processes continuously monitor code changes to detect security vulnerabilities as early as possible.**

Security testing often involves multiple Static Application Security Testing (SAST) tools, each specialized in detecting specific vulnerabilities, such as hardcoded passwords or insecure data flows. A heterogeneous SAST setup, using multiple tools, helps minimize the software's attack surface.

The security findings from these tools undergo Vulnerability Management, a semi-manual process of understanding, categorizing, storing, and acting on them. Code volatility, i.e., the constant change of the project's source code, as well as double reporting, i.e., the overlap of findings reported by multiple tools, are potential sources of duplication imposing futile auditing effort on the analyst.

Vulnerability Tracking is an automated process that helps deduplicating and tracking vulnerabilities throughout the lifetime of a software project. We propose a scalable Vulnerability Tracking approach called *Scope+Offset* for heterogeneous SAST setups that reduces the noise introduced by code volatility as well as code duplication. Our proposed, fully automated method proved to be highly effective in an industrial setting, reducing the negative effect of duplication by approximately 30% which directly translates to a reduction in *futile auditing time* while inducing a negligible performance overhead.

Since its product integration into GitLab in 2022, *Scope+Offset* provided vulnerability tracking to the thousands of security scans running on the GitLab DevSecOps platform every day where the GitLab DevSecOps platform can be considered as a heterogeneous SAST setup as it includes a variety of different SAST tools.

***Index Terms*—Static Application Security Testing, Vulnerability Tracking, Debugging**

I. Motivation

Managing software projects by means of Source Control Management (SCM) systems (such as Git) in combination with automated security testing processes that continuously monitor the constantly changing code to detect security vulnerabilities as early as possible in the software development process is a best practice in today's software industry [1], [2]. The automated security testing process consists of static application security testing (SAST) tools that can automatically detect vulnerabilities in the source code of a software project. SAST tools are often language and problem specific: some SAST tools can detect secrets (e.g., hardcoded passwords) in the source code, while other tools are focused on detecting potential insecure flows or path conditions leading up to a security vulnerability in the program.

In today's software industry, it is common practice to:

1) rely on a heterogeneous SAST setup where multiple SAST tools are executed in combination to minimize the overall attack surface of the software [3], [4].
2) use SCMs in combination with Continuous Integration and Continuous Delivery (CI/CD) processes leveraging Development, Security and Operations (DevSecOps) platforms such as GitLab or GitHub. SCMs enable developers to track code changes, maintain a history of these changes and rollback or revert changes, and CI/CD provides automation and tooling around the integration and deployment of code changes [5].

A heterogeneous SAST setup is composed of various SAST tools that run their analyses on every code commit/push to the SCM repository. The produced vulnerability reports are stored in a common vulnerability database which can then be used to interact with the vulnerability by performing security audits/reviews, prioritizing, dismissing, confirming, delegating or fixing vulnerabilities; we refer to this process as Vulnerability Management (VM). VM is a semi-manual process and for this reason it is important to minimize the amount of noise (e.g., redundant vulnerabilities) presented to the security analyst in order to reduce manual auditing effort.

However, when applying VM in combination with a heterogeneous SAST setup and SCMs, there are two potential sources of noise that can both lead to vulnerability duplication, i.e., reporting the same vulnerability multiple times:

Double Reporting Different SAST tools might report the same vulnerabilities in different ways. The vulnerability location is usually mapped to a source code location. We assume the tuple $\langle filename, linenumber \rangle$ to be the most basic and common form of reporting/locating a vulnerability in the source code.

Code Volatility The location of vulnerabilities is constantly changing due to an evolving code base so that the baseline to which a vulnerability report refers, is constantly changing: source code may be added, removed, edited, shifted, auto-formatted, etc. as the project evolves. Deciding whether a vulnerability in one version of a software project is the same

as a previously detected vulnerability requires a clear notion of what criteria make two vulnerabilities different.

The noise introduced by *Double Reporting* as well as *Code Volatiliy* poses a challenge because it introduces duplication which increases the auditing effort for security analysts by futilely rechecking already acted upon vulnerabilities. We refer to this process as *futile auditing* which can be considered a waste of invested auditing time that does not produce value and distracts auditors from triaging potentially important vulnerabilities. If a heterogeneous SAST setup is based on CI/CD, the negative effect of code duplication can be amplified as new vulnerability data can be produced with every commit/push to the SCM.

Duplication can be the source of vulnerability fatigue [6], where auditors may become desensitized to repeated warnings or issues from SAST tool, ultimately leading to a diminished response or awareness of new, potentially relevant vulnerabilities; the sheer volume of reported findings may overwhelm the auditor to a point where the results are not properly addressed anymore.

Vulnerability Tracking (VT) is an automated process that helps deduplicating and tracking vulnerabilities throughout the lifetime of a software project. The most basic form of vulnerability tracking is line-based where vulnerability locations are defined based on their line number in a file ($\langle$*filename*, *linenumber*$\rangle$). This form of tracking is very limited because it requires the vulnerability to remain at the exact same source code location across security scans to not be considered as new vulnerability which makes it not suitable for environments with a high code volatility.

In this paper, we present the *Scope+Offset* vulnerability tracking method which identifies vulnerabilities based on a *Scope+Offset* fingerprint; this fingerprint identifies vulnerabilities based on the context in which they appear. The main goal of VT in general, and our proposed *Scope+Offset* method in particular, is to reduce *futile auditing time* when working with heterogeneous SAST setups. Hence, *Scope+Offset* balances two competing requirements: (1) alert developers to potential vulnerabilities while (2) deduplicating findings in order to avoid vulnerability fatigue caused by *Double Reporting* and *Code Volatility*.

Our experiments show that the *Scope+Offset* method automatically reduces *futile auditing time* by approximately 30% in comparison to line-based fingerprinting while inducing a negligible performance overhead. Since its product integration into GitLab in 2022, *Scope+Offset* provided vulnerability tracking to the thousands of security scans running on the GitLab DevSecOps platform every day where the GitLab DevSecOps platform can be considered as a heterogeneous SAST setup as it includes a variety of different SAST tools [7].

II. Preliminaries

A. Common Weakness Enumeration

Common Weakness Enumeration (CWE) [8] is a category system to classify and communicate hardware and software weaknesses including security vulnerabilities. CWE provides a stable, detailed, and hierarchical ontology as a foundation to communicate vulnerabilities [9]. A CWE identifier is usually prefixed with *CWE* followed by a number that uniquely identifies the weakness. The CWE system includes hundreds of different categories. For example, *CWE-22* [10] represents a path traversal vulnerability which is the parent of *CWE-23* [11] (Relative Path Traversal) and *CWE-36* [12] (Absolute Path Traversal).

B. Scope

A scope can be considered as a source code block that contains statements and expressions including definitions/declarations of entities such as variables/functions/classes/structs/types etc. Scopes limit the visibility of these entities to a part of the program.

A *scope tag* is composed of *scope elements* and uniquely identifies a scope; a *scope element* refers to a concept, such as packages, modules, namespaces, types/classes/function definitions, which are relevant with respect to scoping.

The scope tag a▷b denotes that we have two scope elements `a` and `b` where `a` scopes `b`, or conversely `b` is nested in or scoped by `a`. Note that the filename is treated as a separate scope to ensure that there is always at least a single scope element present to capture a given vulnerability.

```
1 module Widget
2   class CustomWidget
3     def run(user_supplied_arg)
4     end
5   end
6 end
```

Listing 1: The four scopes in `custom_widget.rb` highlighted in different shades of gray: the file itself, `Widget`, `CustomWidget`, and the function `run`.

Table 1 displays the four scopes/scope tags from Listing 1 including their line numbers. Column *Scope Tag* displays the scope tag that represents the scope, whereas the *Start* and *End* columns display the start and end line of the scope which we also refer to as *scope fringe*; the start and end line numbers correspond to the upper and lower boundaries of the gray areas that are visualized in Listing 1.

Table 1: The four scope tags from Listing 1.

Scope Tag	Start	End
`custom_widget.rb`	1	6
`custom_widget.rb▷Widget`	1	6
`custom_widget.rb▷Widget▷CustomWidget`	2	5
`custom_widget.rb▷Widget▷CustomWidget▷run`	3	4

The different scopes (apart from the file scope), are visually highlighted in Listing 1 as boxes that are highlighted in different shades of gray; the deeper the nesting level, the darker the level of gray.

For the remainder of this document we use ▷ as a delimiter that separates scope elements. In addition, for simplicity, we do not always include the file component from the scope tag

so that we write Widget▷CustomWidget instead of custom_widget.rb▷Widget▷CustomWidget for brevity.

III. Vulnerability Equivalence

The *Ship of Theseus* [13] is a philosophical thought experiment about a ship that gets all of its components replaced with new parts over time. One of the central questions is whether the ship, after having all initial parts replaced, can be still considered the same/original ship or whether it should be considered a different ship. This thought experiment raises questions about two key points that directly relate to VT: *Identity*, i.e., what makes a vulnerability identifiable, and *Change*, i.e., how a vulnerability can maintain its identity over time despite code volatility.

To gain a better, more intuitive, understanding of the genesis of a vulnerability through a changing code base, we walk through a sequence of changes on an example Ruby program custom_widget.rb in Listing 2. Each of the steps can be considered as a snapshot that enables us to probe whether or not the identity of a vulnerability changed between two snapshots (the one we are currently looking at vs. the previous one).

The purpose of this example is to develop a notion of *Vulnerability Equivalence* that provides the conditions under which we consider two vulnerabilities as equivalent (or not) and that is the foundation of our *Scope+Offset* method.

Step 1: A vulnerable code snippet

Listing 2 illustrates a vulnerable Ruby code snippet custom_widget.rb consisting of a module Widget, a class CustomWidget, and a function run that takes a parameter user_supplied_arg. The exec call in the body of the run function in line 5 is vulnerable to OS Command Injection (*CWE-78*) because the user input from line 5 (user_supplied_arg) is directly passed as an argument to the string which in its turn is passed onto the exec call in line 5; this enables an attacker to inject any code through the user_supplied_arg variable. The vulnerability below is enclosed in the scopes that are highlighted by gray rectangles; the scope of the vulnerability can be denoted as custom_widget.rb▷Widgets▷CustomWidget▷run:

```
1 module Widget
2   class CustomWidget
3     def run(user_supplied_arg)
4       puts "running"
5       exec("echo #{user_supplied_arg}")
6     end
7   end
8 end
```

Listing 2: custom_widget.rb with OS Command Injection through the exec call in the function run.

Step 2: Adding a function

Listing 3 below differs from Listing 2 by providing an additional function implementation foo enclosed in the Widget▷CustomWidget scope without changing the run function. Although the code is different from the one in Listing 2, the vulnerability is identical to the one illustrated in step 1. However, the vulnerability moved to a different source code line from line 5 in Listing 2 to line 7 in Listing 3.

```
1  module Widget
2    class CustomWidget
3      def foo(arg) end
4
5      def run(user_supplied_arg)
6        puts "running"
7        exec("echo #{user_supplied_arg}")
8      end
9    end
10 end
```

Listing 3: Modified custom_widget.rb that still includes the same vulnerability with an OS Command Injection in the function run from Listing 2.

Step 3: Renaming a function

Listing 4 differs from Listing 3 by changing the name of the function in which the vulnerability is enclosed from run to bar. If we assume that the name of a function tells us something about its purpose, a name change can indicate a functional/semantic change which we use as a signal that the vulnerability may be worth re-checking.

Hence, with *Scope+Offset*, the vulnerability in Listing 4 would not be considered identical to the vulnerability from Listing 3.

```
1  module Widget
2    class CustomWidget
3      def foo(arg) end
4
5      def bar(user_supplied_arg)
6        puts "running"
7        exec("echo #{user_supplied_arg}")
8      end
9    end
10 end
```

Listing 4: Modified custom_widget.rb where the function run from Listing 3 was renamed to bar.

Step 4: Changing the vulnerability type

Listing 5 differs from Listing 4 by calling to Dir.mkdir instead of exec. The function Dir.mkdir takes the user input and creates a directory so that the example below contains a path traversal vulnerability (*CWE-22*) instead of the original command injection vulnerability (*CWE-78*). Both vulnerabilities are not identical because they differ in their vulnerability category/type. The vulnerability category/type is provided by the SAST tools where the majority of SAST tools use CWE as a baseline.

```ruby
module Widget
  class CustomWidget
    def foo(arg) end

    def bar(user_supplied_arg)
      puts "running"
      Dir.mkdir(user_supplied_arg)
    end
  end
end
```

Listing 5: Modified `custom_widget.rb` where the call to `exec` from Listing 4 was changed to `Dir.mkdir` (create directory).

Step 5: Adding code before the vulnerable call

Listing 6 is different from Listing 5 because of the additional code added before the call to `Dir.mkdir` which modifies the value of `user_supplied_arg` before it is used as an argument. This may change the nature of vulnerability so that the vulnerability below is not considered identical to the one from Listing 5.

```ruby
module Widget
  class CustomWidget
    def foo(arg) end

    def bar(user_supplied_arg)
      puts "running"
      // ...
      modified_arg = ...
      Dir.mkdir(modified_arg)
    end
  end
end
```

Listing 6: Modified `custom_widget.rb` whith additional code included that impacts the argument of `Dir.mkdir`.

In summary, we define two vulnerabilities v_1 and v_2 to be equivalent if all of the following conditions hold:

1) v_1 and v_2 are located in the same scope (Steps 1-3).
2) v_1 and v_2 have the same vulnerability type (Step 4).
3) v_1 and v_2 have the same offset (line number distance) to the line number where their parent scope begins (Step 5).

This is the notion of Vulnerability Equivalence used by the *Scope+Offset* approach.

IV. Approach

Figure 1 illustrates how our VT approach works on a high level and how it fits into the software development and VM workflows. In Figure 1 processes are depicted as round-boxes. Input/output relationships are depicted as labeled arrows where the label describes what information is passed between processes. Input/output items are represented as squared boxes. The source code is managed through an SCM; one or multiple SAST scanners check the source code for vulnerabilities and generate vulnerability reports which contain (meta-)information about the detected vulnerabilities, including their

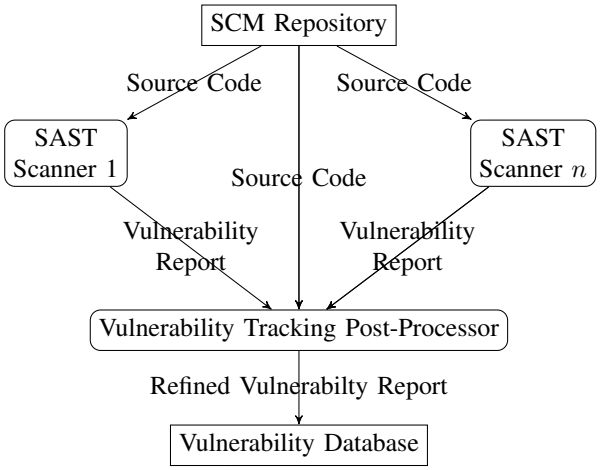

Figure 1: Vulnerability Tracking Approach.

locations. We conservatively assume that a SAST scanner provides the most basic form of a vulnerability location which is its source code line number ($\langle$*filename*, *linenumber*$\rangle$).

These reports are digested by the *Vulnerability Tracking Post-Processor* component which computes the scopes from the source code; the scopes are used to generate *Scope+Offset* fingerprints, i.e., hashes that uniquely identify vulnerabilities based on their context/scope. The fingerprints are required for both deduplication and vulnerability tracking. The *Vulnerability Tracking Post-Processor* generates a *Refined Vulnerability Report* that includes a set of deduplicated vulnerabilities with their fingerprinting information. The data is then stored in a *Vulnerability Database* that collects and stores all the data used for the purpose of VM.

Figure 2 zooms more closely into the *Vulnerability Tracking Post-Processor* component from Figure 1. Macro-blocks are illustrated as dashed boxes; input/output components are depicted as squared boxes and processes are depicted as round-boxes. Input/output relationships are depicted as labeled arrows where the label describes what information is passed between processes. As input, the post-processor takes one or more Vulnerability Reports that provide Vulnerability information from the SAST scanners that are part of the heterogeneous SAST setup. In addition, the *Vulnerability Tracking Post-Processor* also takes the source code that originates from the SCM repository; the source code is required for computing the scopes that are then used to determine the optimal (most narrow) scope for the vulnerability findings originating from the SAST tools. As an output, the *Vulnerability Tracking Post-Processor* produces a *Refined Vulnerability Report* that includes the set of deduplicated vulnerabilities.

The *Vulnerability Tracking Post-Processor* includes the following five sub-steps each of which is explained in sequence and in more detail in the following sub-sections.

1) *Generic Parsing*: In this phase we parse the source code to obtain a parse tree, i.e., a structural/hierarchical representation of the source code.

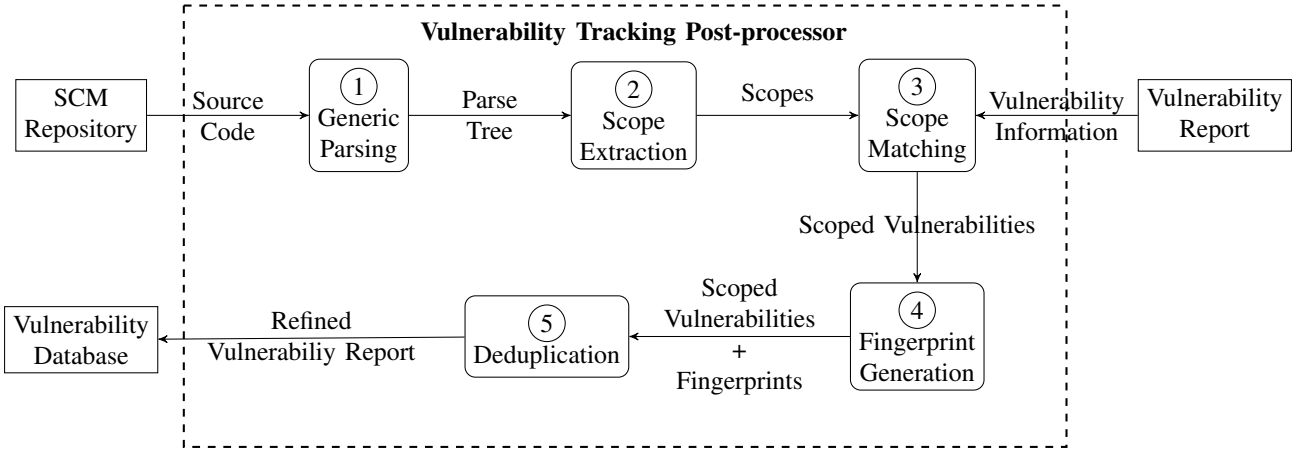

Figure 2: Vulnerability Tracking Architecture

2) *Scope Extraction*: This phase uses the parse tree provided by *Generic Parsing* to extract all scope tags.
3) *Scope Matching*: In this phase, the vulnerabilities are matched against the scope information. The goal of this step is to find the optimal scope for a vulnerability, i.e., the most narrow scope in which the vulnerability appears.
4) *Fingerprint Generation*: Once the vulnerabilities have been successfully scoped, the actual *Scope+Offset* fingerprints are generated from the scoping information as well as the vulnerability category/type.
5) *Deduplication*: The last step identifies which vulnerabilities are identical based on their fingerprint and deduplicates them. This step yields a set of refined vulnerabilities which can finally be stored in the *Vulnerability Database*.

A. Generic Parsing

Parsing can be described as the process of applying a (formal) language definition, usually provided in the form of a Context Free Grammar (CFG) for a given programming language. This process results in the construction of a parse tree or syntax tree, which represents the syntactic structure of the input based on the rules defined by the CFG [14].

As our proposed VT method is language-agnostic, we do not make any assumption about the parser that is being used as long as it can parse source code and produce a corresponding parse tree. For the remainder of this section, we assume that the generic parsing step is performed by a *Generic Parser* component.

Listing 7 shows an example program `example.rb` consisting of three modules/classes with four functions and four calls. The parser is able to understand the syntactic elements of the source code and constructs a hierarchical representation of the code which is called syntax tree or parse tree.

Figure 3 illustrates a simplistic view of a parse tree as it may be produced by the *Generic Parser*. The parse tree structure captures the scoping through parent/child node relationships: parent nodes scope their children; child nodes are scoped by their parent node. For example, the child scope `OuterClass▷InnerClassB▷function_A` cannot exist without the parent scope `OuterClass▷InnerClassB`. Figure 3 also depicts the node types for the nodes that are provided by the *Generic Parser* in *italic*; for example the `OuterClass` is a node of type *class*, i.e., a node that refers to a class or module declaration as it appears in the source code; the *funcdef* node refers to a function definition whereas *body* refers to a function body.

```
1  module OuterClass
2    class InnerClassA
3      def function_A
4        puts "function A"
5      end
6      def function_A
7        puts "function A"
8      end
9    end
10   class InnerClassB
11     def function_A
12       puts "function A"
13     end
14     def function_B
15       puts "function B"
16     end
17   end
18 end
```

Listing 7: A Ruby file `example.rb` that includes the classes `OuterClass`, `InnerClassA` and `InnerClassB` with four function calls.

Generic Parsing ensures that the node types are normalized to be identical across language boundaries so that Ruby, Java, Scala classes etc., are all mapped to the node type *class*. This normalization significantly simplifies the *Scope Extraction* step.

B. Scope Extraction

During the *Scope Extraction* phase, the scopes are extracted from the parse tree produced by means of the *Generic Parsing*.

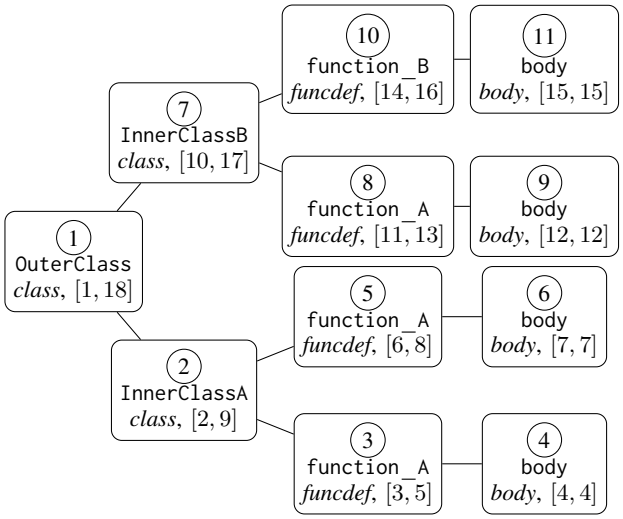

Figure 3: Parse Tree from the code snippet in Listing 7. The numbers denote the node visiting sequence using depth-first pre-order traversal. The numeric ranges refer to the scope fringes.

For the remainder of this section, we assume that the *Scope Extraction* step is performed by a *Scope Extractor* component.

For Listing 7, the *Scope Extractor* would extract the 7 scopes that are listed below:

1) `OuterClass`
2) `OuterClass`▷`InnerClassA`
3) `OuterClass`▷`InnerClassA`▷`function_A` (2×)
4) `OuterClass`▷`InnerClassB`
5) `OuterClass`▷`InnerClassB`▷`function_A`
6) `OuterClass`▷`InnerClassB`▷`function_B`

The scope computation is done in a depth-first (DF) pre-order traversal. Pre-order traversal ensures that we compute the parent scope before computing the child scopes which is necessary due to their interdependence: The scoping information for `OuterClass`▷`InnerClassB` has to be computed before `OuterClass`▷`InnerClassB`▷`function_A`.

Figure 3 depicts the parse tree, the order in which nodes are visited is indicated by the numbers (enclosed in parentheses) in the node labels.

The *Scope Extraction* relies on a stack of scope tags (that are computed based on the identified scope elements), as well as a set of tags that captures all the tags that have been generated. During the tree traversal, every time the *Scope Extractor* encounters a node that corresponds to a scope, a new scope tag is generated. For this purpose, the *Scope Extractor* relies on a predefined set of scope elements that are deemed relevant with regards to scoping; these elements include files, class declarations, function definitions, package/module identifiers, namespace declarations. In our example we only focus on the *class* and *funcdef* scope elements that represent class/module declarations and function definitions, respectively.

In the example (Figure 3) the node visiting sequence is represented by means of circled numbers. At traversal step 1, the *Scope Extractor* encounters a node of type *class* which is relevant with regards to scoping as it represents a class/module declaration. Hence, it generates a corresponding scope tag for `OuterClass`, adds it to the set of scope tags, and pushes it onto the tag stack so that it is accessible for the child nodes that the *Scope Extractor* is going to visit next. At traversal step 3, the *Scope Extractor* encounters a *funcdef* (i.e., a function definition node); the scope extractor generates a tag by peeking the tag item that was generated last from the stack (i.e., `OuterClass`▷`InnerClassA`); this information is then used to generate a new tag item (`OuterClass`▷`InnerClassA`▷`function_A`). Note that once we are done processing a node, we pop its corresponding tag item from the stack. For example, once the scope extractor is done with traversal steps 3 and 4, it pops `OuterClass`▷`InnerClassA`▷`function_A` from the stack so that at traversal step 5, `OuterClass`▷`InnerClassA` is our last tag item on the stack.

Per default, the traversal algorithm walks down (descends) the tree irrespective of the node type without creating a tag. For node types that are relevant with respect to scoping, the *Scope Extractor* creates a tag and descends. Some node types that are irrelevant for scoping are skipped.

Some languages (including Ruby) allow methods/classes to be (re-)defined multiple times. In Listing 7 `function_A` is defined two times within `InnerClassA` which both translate to `OuterClass`▷`InnerClassA`▷`function_A` so that a concatenation of scope elements does not guarantee tag uniqueness. The scope extractor applies a numbering scheme to the scope elements that ensures the generation of unique tags by maintaining an internal counter that tracks how often a scope element has been encountered within its parent scope. For our code example, the scope extractor computes the scope tags below. The scope element counters are indicated by the numbers enclosed in brackets. Thanks to the scope numbering scheme, instead of using the same scope tag for `function_A` that appears twice in `InnerClassA`, we now have dedicated scope tags for both occurrences with `OuterClass[0]`▷`InnerClassA[0]`▷`function_A[0]` and `OuterClass[0]`▷`InnerClassA[0]`▷`function_A[1]`.

1) `OuterClass[0]`
2) `OuterClass[0]`▷`InnerClassA[0]`
3) `OuterClass[0]`▷`InnerClassA[0]`▷`function_A[0]`
4) `OuterClass[0]`▷`InnerClassA[0]`▷`function_A[1]`
5) `OuterClass[0]`▷`InnerClassB[0]`
6) `OuterClass[0]`▷`InnerClassB[0]`▷`function_A[0]`
7) `OuterClass[0]`▷`InnerClassB[0]`▷`function_B[0]`

As depicted in Figure 3, all the scope tags produced by the *Scope Extractor* include source coordinates that capture the *scope fringes* or boundaries, i.e., the start and end source code lines of a given scope.

The *Scope Extractor* generates a scope table that includes the scope tag as well as the *scope fringes* denoted by start and end lines of the scope in the source code.

Listing 8 depicts another, smaller Ruby code snippet which serves as a running example for the remainder of this section. Table 2 details the scope tags as well as the start and end lines for all the scopes in Listing 8.

```
1  module Widget
2    class CustomWidget
3      def foo(arg) // ...

10     end

20     def run(user_supplied_arg) // ...

30     end
31   end
32 end
```

Listing 8: Scope Extraction Example `custom_widget.rb`

For example, in Table 2, the scope of the function run with scope name `custom_widget.rb▷Widget[0]▷CustomWidget[0]▷run[0]` starts on line 20 and ends on line 30.

C. Scope Matching

In the *Scope Matching* phase, the vulnerabilities that are included in vulnerability reports from possibly multiple SAST scanners are scoped by means of the set of scope tags S that were computed by the *Scope Extractor*. We assume that a vulnerability is reported with enough information to locate it in the source code; at a bare minimum, we assume the presence of file name and source code line number $\langle \textit{filename}, \textit{linenumber} \rangle$. For the remainder of this section, we assume that the scope matching step is performed by a *Scope Matcher* component.

The *Scope Matcher* uses the scope fringes in conjunction with the line number of the vulnerability to position a vulnerability. For every scope tag $s \in S$ that was extracted by the *Scope Extractor*, $\textit{start}(s)$ and $\textit{end}(s)$ denote the start and end lines of scope s (*scope fringe*) in the source code file. The *Scope Matcher* computes the optimal (most narrow) scope that scopes the reported vulnerability whose position is identified by its source code line $l \in L$ where L represents all the vulnerable lines reported by the SAST tools. A scope s is considered optimal for a vulnerability l if (1) l lies inside the scope fringe and (2) the *scope fringe* is most narrow.

For all vulnerability lines $l \in L$ and scopes $s \in S$ with $\textit{start}(s) \leq l \leq \textit{end}(s)$, we compute a *scope match* value that measures the placement of the vulnerability by means of $\textit{smatch}(s, l)$:

$$\textit{smatch}(s,l) = \begin{cases} \textit{end}(s) - \textit{start}(s), & \text{if start}(s) \leq l \leq \text{end}(s) \\ \text{-}, & \text{otherwise} \end{cases}$$

The smaller the scope match value, the narrower the scope that nests the vulnerability that is located on line l. By minimizing the $\textit{smatch}(s, l)$ which we write as $min(\textit{smatch}(s, l))$, we make sure that we find the optimal scope for a vulnerability.

Using our running example `custom_widgets.rb` from Listing 8, if we assume that a SAST tool reports a vulnerability on line $l = 25$ in the source code, we can compute the matching scope based on the scope table below that is provided by the *Scope Extractor*. In Table 2, *smatch* is minimal (row 5) for the scope `custom_widget.rb▷Widget[0]▷CustomWidget[0]▷run[0]` with $min(\textit{smatch}) = 10$; the *Scope Matcher* identified this scope as the matching scope for the reported vulnerability; row 4 is omitted because it does not satisfy the constraint $\textit{start}(s) \leq l \leq \textit{end}(s)$.

D. Scope+Offset Fingerprint Generation

In the fingerprint generation phase, we compute hashes (fingerprints) from the scoped vulnerabilities.

After *Scope Matching* has determined the optimal scope for a given line, *Fingerprint Generation* computes a hash that includes the information below:

scope tag ▷ *offset* ▷ *vulnerability classifier*

The *scope tag* component is the optimal scope tag s computed in the *Scope Matching* step, *offset* is the offset of the vulnerability to the boundary start line of the matching scope. $\textit{offset} = l - \textit{start}(s)$; *vulnerability classifier* is the vulnerability classifier returned by the SAST tool. To make the hash comparable across different analyzers, CWEs, CVEs and or OWASP categories are the most commonly used classification/ranking systems. However, to ensure generic and stable identifiers, CWE is the ideal classification system due to its comprehensiveness. Using our example `custom_widgets.rb`, if we assume that a SAST tool reports a vulnerability with *CWE-22* on line 25, we can compute the hash below:

```
custom_widget.rb▷Widget[0]▷CustomWidget[0]▷
run[0]▷5▷CWE-22
```

E. Deduplication

The deduplication of vulnerabilities is performed based on the generated hashes. Any pair of vulnerabilities that can be translated to the same fingerprint is considered identical. During the *Deduplication* step, all unique vulnerability fingerprint hashes are assembled in a refined vulnerability report that is stored in the vulnerability database.

Using our running example `custom_widgets.rb` from Listing 8, if we assume that a developer shifts the `run` function by $+100$ source code lines, the SAST tool would report the *CWE-22* vulnerability with a $+100$ line difference on line 125. Thanks to the fingerprint hash, we can deduplicate the vulnerabilities from lines 25 and 125. For both vulnerabilities, the fingerprint hash is identical:

```
custom_widget.rb[0]▷Widget[0]▷CustomWidget[0]▷
run[0]▷5▷CWE-22
```

V. Evaluation

For our evaluation, we leveraged a Record & Replay framework for Git histories called *SourceWarp* [5] which enables us to automatically reproduce the set of commits that are included in a slice of the Git commit history. *SourceWarp* simulates user input (code changes) by generating a patch sequence from the

Table 2: Scope Extractor results generated for the program in Listing 8 with a vulnerability in line 25. The table shows the scope tags, their start and end line numbers as well as the *scope match* values where smaller value indicates a better match for the given vulnerability (line).

	Scope Tag	Start Line	End Line	*smatch*
1	`custom_widget.rb`	1	33	32
2	`custom_widget.rb▷Widget[0]`	1	33	32
3	`custom_widget.rb▷Widget[0]▷CustomWidget[0]`	2	32	30
4	`custom_widget.rb▷Widget[0]▷CustomWidget[0]▷foo[0]`	3	10	-
5	`custom_widget.rb▷Widget[0]▷CustomWidget[0]▷run[0]`	20	30	10

Git history slice in a record phase; the same patch sequence can then be replayed against different target system in the replay phase. *SourceWarp* collects statistics from the target system while doing so.

In our experiment, a target system consists of an SCM which serves as the replay target for *SourceWarp* and the SAST tool that runs its scans on the SCM every time a patch is applied to the SCM. To assess the performance of *Scope+Offset*, we did a differential analysis between two target systems, one equipped with our *Scope+Offset* methodology, and one without *Scope+Offset* using line-based fingerprinting.

We configured *SourceWarp* to collect the produced vulnerability reports as well as statistics about the execution time. As output *SourceWarp* generated two reports for both systems under test whose content is presented in Table 3.

We assessed the following research questions:

- *RQ1: By which margin can Scope+Offset mitigate the effect of futile auditing?*: Our hypothesis being that the new Vulnerability Tracking method reduces *futile auditing* time, we want to know an estimate of the margin by which the new method works better in comparison to the line-based method.
- *RQ2: What is the performance overhead of Scope+Offset?*: The overhead (in terms of execution time) when applying the new Vulnerability Tracking method.

Evaluation Setup

As input *SourceWarp* used the GitLab source code Git repository [15] because it is a real-world repository hosting a large code-base. This evaluation is based on the GitLab release v13.6.6-ee which included 34K files, with 3.7 million lines of code out of which 1.2 million lines of code were Ruby on Rails (RoR) code with a history of 200K commits. As a Git slice, we arbitrarily picked the time window between 2020-10-31 and 2020-12-31.

Leveraging the data from the GitLab SCM repository replayed the same slice of the history on two target systems under test both of which were provided as Docker [16] images:

1) As baseline we used the dockerized SAST tool for RoR GitLab brakeman v2.12.2 [17] which wraps the free and open source RoR SAST tool brakeman [18]. We refer to this system as *brakeman*. For this analyser, we employ line-based VT.
2) We used a dockerized version of brakeman v2.12.2 enhanced with our new *Scope+Offset* Vulnerability Tracking approach as a post-analyzer. We refer to this system as *brakeman+VT*. *brakeman+VT* is basically *brakeman* equipped with the *Vulnerability Tracking Post Processor* as illustrated in Figure 1. The *Vulnerability Tracking Post Processor* is implemented by means of the `tracking-calculator` tool which we made publicly available [19] in binary form and as a Docker image; the tool is based on the (incremental) parser generator library `tree-sitter` [20]. The tool comprises approximately 2KLOC of Go code excluding library code. The integration of `tracking-calculator` required no changes on brakeman, as both tools implemented the same exchange format, i.e., the GitLab SAST report format [7].

This setup directly translates to a heterogeneous SAST setup due to the reasons laid out below.

1) The experiment is conducted with a single SAST tool. Hence, the results provide us with the lower bound on the improvement in regard to deduplication.
2) The normalization provided by *Generic Parsing* implemented in `tracking-calculator` based on `tree-sitter` abstracts away language-specific scope elements. Thanks to this abstraction, `tracking-calculator` supports a wide variety of programming languages: C#, C/C++, Go, Java, Javascript, Python, Ruby and PHP.
3) The GitLab security report exchange format provides a layer of abstraction that is used to pass vulnerability reports from any SAST tool to `tracking-calculator`. Thanks to this abstraction `tracking-calculator` can be used in conjunction with any combination of SAST tools that support the common SAST exchange format.

The experiment was conducted on Linux Laptop (using version 5.10.12) with a Core i7 hexa-core (2.2GHz per core) with an Intel SSD Pro 7600p using Docker 20.10.3.

RQ1: By which margin can Scope+Offset mitigate the effect of futile auditing?

Table 3 shows the data included in the result/metrics report produced by *SourceWarp* for both *brakeman* and *brakeman+VT*. Column *Record Time* displays the time *SourceWarp* took during the record phase; column *Replay Time* shows the time *SourceWarp* required to replay all the patches in the patch sequence; column *Average Replay Time Per Patch* shows how much time *SourceWarp* spent on average to replay a single patch; column ↯ displays the number of crashes *SourceWarp* observed and column *#Unique fingerprints for* $patch_i$ shows the number of unique vulnerability findings observed after replaying $patch_i$ where the patch sequence numbers are provided in the row below. Note that for *#Unique*

Table 3: Evaluation Results when running *SourceWarp* on two SUTs: *brakeman* and *brakeman+VT* equipped with *Scope+Offset*

	Recording Time	Replay Time	Average Replay Time Per Patch	Overall Time	↯	#Unique Fingerprints for $patch_i$												
						1	2	3	4	5	6	7	8	9	10	11	12	13
brakeman	54 min 30 s	18 min 19 s	1 min 24 s	1 h 12 min 49 s	0	94	94	97	97	102	102	118	118	125	125	128	128	132
brakeman+VT	54 min 30 s	17 min 50 s	1 min 22 s	1 h 12 min 20 s	0	83	83	84	84	84	84	91	91	91	91	91	91	92
Δ_{abs}	0 s	29 s	2 s	29 s	0	11	11	13	13	18	18	27	27	34	34	37	37	40
$\Delta_{\text{rel}}(\%)$	0	2.6	2.6	0.6	0	11	11	13	13	17	17	22	22	27	27	28	28	30

Columns *Record Time*, *Replay Time* display the time *SourceWarp* took for the record and replay phases, respectively; column *Average Replay Time Per Patch* shows how much time *SourceWarp* spent on average to replay a single patch; column ↯ displays the number of crashes; column *#Unique fingerprints for* $patch_i$ shows the number of unique vulnerability findings observed after replaying $patch_i$ where the patch sequence numbers are provided in the row below. Row *brakeman* shows all the results that were related to the brakeman SAST tool we used at the time with line-based fingerprinting; row *brakeman+VT* shows the results for the tool that used *Scope+Offset*. The rows Δ_{abs} and Δ_{rel} display absolute and relative difference expressed in percentage (improvement).

fingerprints for $patch_i$ smaller numbers denote an improvement: the less unique fingerprints are generated, the better *Scope+Offset* performed with regards to deduplication. The *brakeman* row shows all the results that were related to the unchanged/original brakeman SAST tool we used at the time with line-based fingerprinting; the *brakeman+VT* row shows the results for the tool that used the enhanced Vulnerability Tracking algorithm *Scope+Offset*. The rows Δ_{abs} and Δ_{rel} display absolute and relative difference expressed in percentage (improvement) between *brakeman+VT* and the baseline *brakeman*. It is important to collect multiple data-points during the Replay phase in order to properly reflect and observe the progression in terms of the number of vulnerabilities that accumulate in the backend (vulnerability) database over time to which ultimately a user of the system would be exposed.

We can see that the *brakeman+VT* performed significantly better than *brakeman* in terms of deduplicating findings. After the application of the first patch $patch_1$, *brakeman+VT* deduplicated 11 vulnerabilities; for every subsequently applied patch the number Δ_{abs} of successfully eliminated, deduplicated, redundant findings when using *brakeman+VT* constantly increases reaching 40 after the application of the $patch_{13}$ which is a relative improvement of 30% in comparison to the traditional, line-based tracking method.

We achieved cost savings of 30%, i.e., reduction of *futile auditing* without any negative impact in terms of robustness (no crashes observed).

RQ2: What is the performance overhead of Scope+Offset?

When looking at *brakeman* and *brakeman+VT*, the replay time was almost the same in both cases; in fact *brakeman+VT* was 29 s (2.6%) faster which was due to the overhead incurred when creating the Docker containers. From the fact that the Vulnerability Tracking overhead has been shadowed by Docker container creation time, we can conclude that the overhead of the new Vulnerability Tracking algorithm is negligible in practice.

Our VT method *Scope+Offset* was integrated into the product in the year 2022 and proved to be an effective and efficient solution to the Vulnerability Tracking problem in a heterogeneous SAST setups like GitLab: it is easy to understand and maintain, does not induce a large performance penalty and is versatile as it can be easily integrated as a post-analyzer in combination with various SAST tools across several languages.

VI. Limitations

Given that both the SAST and VT tools are necessarily approximate, it is possible that new vulnerabilities track to previously dismissed false positives and the location is therefore not re-reported. Since developers are alerted to a security concern at the location at least once, *Scope+Offset* accepts erring on the side of avoiding re-reports in favor of reducing futile auditing time and vulnerability fatigue [6]. In our concrete application context at GitLab, it is worth noting that *dismissed* findings are readily available with a toggle for more exhaustive security reviews.

Another limitation of the *Scope+Offset* method lies in the fixed *Offset* component. Since our method does not have any semantic understanding of the source code, we conservatively assume that any source code that may be added in front of a given line that corresponds to a vulnerability finding within the same scope could potentially impact the finding and, hence, needs to be rechecked (section III, step 5).

Scope+Offset does not take into consideration newlines and comments when computing the offset component which can lead to the reintroduction of vulnerabilities if comments are added or removed. Our simple design enabled us to start out with a low performance overhead which was important because our tracking approach operates at a large scale. However, even without considering newlines and comments *Scope+Offset* reduces the negative effect of duplication by approximately 30%.

To identify scopes accurately, our approach relies on traversal of named nodes in the syntax tree. This imposes a limitation in that anonymous functions are not considered as part of the generated signatures. In addition, method overloading based on argument types is not supported, yet. While the numbering-scheme presented in subsection IV-B prevents the generation of multiple identical tags that refer to different scopes, it imposes a positional constraint with respect to the scope position.

Our current implementation scopes vulnerabilities based on the line numbers. Assuming a SAST tool to provide most basic form of reporting a vulnerability by means of a line number is conservative as it makes our approach versatile and compatible

with most SAST tools which is important for heterogeneous SAST setups.

VII. Related Work

A. Library Classification System

The *Scope+Offset* methodology is inspired by existing library classification systems that are used to organize books based on their subjects and allow the addition of new books to an appropriate location based on their subject. Some classification systems include the Dewey Decimal Classification [21] or the Library of Congress Classification [22]. These classification systems provide a mechanism to organize and locate books based on meta-information about them. Using library classification systems as an analogy, with the *Scope+Offset* method, the vulnerability would be analogous to the book, the *Scope* component could be considered analogous to the shelf whereas the *Offset* component is analogous to the book's position in the shelf.

B. Code Clone Detection

This work is related to Clone Detection [23] that aims to detect code duplication which tends to inflate maintenance costs especially if the code duplicates digress over time. The approaches can be classified based on the code representation they leverage ranging from text-based [24], token-based [25], [26], syntax-tree-based [27], [28] to more complex models such as program-dependence graphs [29].

While *Scope+Offset* does not aim to reduce maintenance time, it aims to reduce *futile auditing time*, there is some similarity with regards to the used techniques. Some code clone detection methods use fingerprinting/hashing to identify source code patterns based on syntax tree [28] which is conceptually similar to our *Scope+Offset* fingerprint.

C. Code Fingerprinting in Cybersecurity

In cybersecurity, vulnerabilities and malware programs are often tracked via various types of fingerprinting.

Fingerprints can be computed based on the line [30], function [31] or whole-file [32] levels to support different granularities. Sophisticated program representations like program dependence graph [29] can be employed to achieve partial semantics-awareness [33].

Fingerprinting is widely used in static malware program analysis to track known malware where cryptographic hashing techniques such as MD5, SHA-1, and SHA-256 are commonplace to index known malware samples. To counter subtle evasion techniques, fuzzy hashing, which tolerates minor variations, is applied in detecting metamorphic [34], [35], polymorphic [36], [37], and even obfuscated malware [38], [39]. Fingerprinting can also be applied to higher-level program representations, such as abstract syntax trees [40] and program dependence graphs [41], to perform more in-depth analysis.

In comparison, our fingerprinting method is based on the structure of the syntax tree which allows to account for program modifications more flexibly.

D. Bug Report Deduplication

Deduplication of bug reports has been a traditional research topic of software engineering [42]. In addition to code, a real-world bug report typical contains summary texts, product/component names, version information and detailed descriptions, which is suitable for natural language processing and information retrieval techniques to be applied [43], [44].

In contrast to these approaches, *Scope+Offset* is based on the source code and applied before the actual VM takes place.

E. Static Application Security Testing

The field of SAST has seen significant research activity over the course of the last decades, with many studies addressing various aspects of the problem. SAST tools can be largely divided into three categories: linting-based approaches, intraprocedural [45] and interprocedural [46], [47], [48] approaches.

A substantial body of work exists on exploring various program representation/models and approaches which we will not be able to adequately capture here. However, VT including our *Scope+Offset* method complements SAST by making the findings produced by SAST traceable which is important for Vulnerability Management especially when SAST is applied in conjunction with DevSecOps platforms, CI/CD environments and heterogeneous SAST setups.

F. Vulnerability Management

Vulnerability Management [49], [50] is the task of interacting with and/or acting on the vulnerabilities reported by the SAST tool. Our work relates to VM by supporting and simplifying the auditing of vulnerabilities by reducing the amount of duplication and *futile auditing time*.

VIII. Conclusion & Future Work

In this paper, we present a scalable, effective and simple method to track vulnerabilities in heterogeneous SAST setups. With a negligible overhead, we reduce the negative effect of *futile auditing*, i.e., auditing of vulnerability duplicates, by 30% without any negative impact in terms of robustness (no crashes observed).

Our VT method *Scope+Offset* runs in the GitLab product since the year 2022 and proved to be an effective and efficient solution to the Vulnerability Tracking problem in a heterogeneous SAST setups like GitLab: it is easy to understand and maintain, does not induce a large performance penalty and is versatile as it can be easily integrated as a post-analyser in combination with various SAST tools across several languages

In the future, we would like to continue working on more refined VT methods by better supporting SAST tools that return data-flows or code paths.

REFERENCES

[1] KPMG, "Agile transformation," https://assets.kpmg/content/dam/kpmg/be/pdf/2019/11/agile-transformation.pdf, 2019.

[2] StackOverflow, "Stackoverflow developer survey 2022," https://survey.stackoverflow.co/2022, 2022.

[3] N. I. of Standards and Technology, "Security and privacy controls for information systems and organizations," NIST, Tech. Rep., 2020. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-53/rev-5/final

[4] Gartner, "Magic quadrant for application security testing," https://www.gartner.com/en/documents/4013799, 2023.

[5] J. Thome, J. Johnson, I. Dawson, D. Bolkensteyn, M. Henriksen, and M. Art, "Sourcewarp: A scalable, scm-driven testing and benchmarking approach to support data-driven and agile decision making for ci/cd tools and devops platforms," in *Proceedings of AST'23*, 2023.

[6] R. He, H. He, Y. Zhang, and M. Zhou, "Automating dependency updates in practice: An exploratory study on github dependabot," 2023.

[7] GitLab, "Gitlab sast," https://docs.gitlab.com/ee/user/application_security/sast/, 2024.

[8] MITRE, "Common weakness enumeratino (cwe)," https://cwe.mitre.org/, 2024.

[9] S. Pan, L. Bao, X. Xia, D. Lo, and S. Li, "Fine-grained commit-level vulnerability type prediction by cwe tree structure," in *Proceedings of ICSE'23*, 2023.

[10] MITRE, "Cwe-22," https://cwe.mitre.org/data/definitions/22.html, 2024.

[11] ——, "Cwe-23," https://cwe.mitre.org/data/definitions/23.html, 2024.

[12] ——, "Cwe-36," https://cwe.mitre.org/data/definitions/36.html, 2024.

[13] B. Simon, *The Oxford Dictionary of Philosophy*. Oxford University Press, 2016.

[14] A. Aho, J. Ullman, M. Lam, and R. Sethi, *Compilers Principles, Techniques, and Tools*. Pearson Deutschland, 2013.

[15] GitLab, "Gitlab source code repository," https://gitlab.com/gitlab-org/gitlab, 2024.

[16] D. Inc., "Docker," https://www.docker.com/, 2024.

[17] GitLab, "brakeman analyzer," https://gitlab.com/gitlab-org/security-products/analyzers/brakeman, 2024.

[18] Brakeman, "brakeman scanner," https://brakemanscanner.org/, 2024.

[19] GitLab, "tracking-calculator post analyzer," https://gitlab.com/gitlab-org/security-products/post-analyzers/tracking-calculator, 2024.

[20] TreeSitter, "The treesitter parser generator," https://tree-sitter.github.io/tree-sitter/, 2024.

[21] M. Dewey, "A classification and subject index for cataloguing and arranging the books and pamphlets of a library [dewey decimal classification]," https://www.gutenberg.org/files/12513/12513-h/12513-h.htm, 1876.

[22] L. S. . I. Technology, "Library of congress classification," https://www.librarianshipstudies.com/2017/11/library-of-congress-classification.html, 2020.

[23] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Proceedings of ICSE'09*. IEEE, 2009.

[24] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using," in *Proceedings of ICPC'08*. IEEE, 2008.

[25] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, "On detection of gapped code clones using gap locations," in *Proceedings of APSEC'02*. IEEE, 2002.

[26] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *TSE*, 2002.

[27] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings of ICSM'98*. IEEE, 1998.

[28] B. Baker, "On finding duplication and near-duplication in large software systems," in *Reverse Engineering - Working Conference Proceedings'95*. IEEE, 1995.

[29] J. Krinke, "Identifying similar code with program dependence graphs," in *Proceedings of ICSE'01*. IEEE, 2001.

[30] J. Jang, A. Agrawal, and D. Brumley, "Redebug: finding unpatched code clones in entire os distributions," in *S&P'12*. IEEE, 2012.

[31] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *S&P'17*. IEEE, 2017.

[32] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of ICSE'07*. IEEE, 2007.

[33] Y. Xiao, B. Chen, C. Yu, Z. Xu, Z. Yuan, F. Li, B. Liu, Y. Liu, W. Huo, W. Zou *et al.*, "Mvp: Detecting vulnerabilities using p̈atch-enhancedv̈ulnerability signatures," in *Proceedings of USENIX Security'20)*. USENIX, 2020.

[34] K. Monnappa, "Automating linux malware analysis using limon sandbox," *Black Hat Europe*, vol. 2015.

[35] A. P. Namanya, "A heuristic featured based quantification framework for efficient malware detection. measuring the malicious intent of a file using anomaly probabilistic scoring and evidence combinational theory with fuzzy hashing for malware detection in portable executable files," Ph.D. dissertation, University of Bradford, 2016.

[36] E. G. Lazo, "Combing through the fuzz: Using fuzzy hashing and deep learning to counter malware detection evasion techniques," https://www.microsoft.com/en-us/security/blog/2021/07/27/combing-through-the-fuzz-using-fuzzy-hashing-and-deep-learning-to-\counter-malware-detection-evasion-techniques/, 2021.

[37] F. Pagani, M. Dell'Amico, and D. Balzarotti, "Beyond precision and recall: understanding uses (and misuses) of similarity hashes in binary analysis," in *Proceedings of CODASPY'18*. ACM, 2018.

[38] L. Glanz, S. Amann, M. Eichberg, M. Reif, B. Hermann, J. Lerch, and M. Mezini, "Codematch: obfuscation won't conceal your repackaged app," in *Proceedings of FSE'17*. ACM, 2017.

[39] Y. Li, S. C. Sundaramurthy, A. G. Bardas, X. Ou, D. Caragea, X. Hu, and J. Jang, "Experimental study of fuzzy hashing in malware clustering analysis," in *Proceedings of CSET'15*. USENIX, 2015.

[40] R. Potharaju, A. Newell, C. Nita-Rotaru, and X. Zhang, "Plagiarizing smartphone applications: attack strategies and defense techniques," in *Proceedings of ESSoS'12*. Springer, 2012.

[41] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of p̈iggybackedm̈obile applications," in *Proceedings of CODASPY'13*. ACM, 2013.

[42] T. Zhang, D. Han, V. Vinayakarao, I. C. Irsan, B. Xu, F. Thung, D. Lo, and L. Jiang, "Duplicate bug report detection: How far are we?" *Transactions on Software Engineering and Methodology*, 2023.

[43] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *Proceedigns of ASE'11*. IEEE, 2011.

[44] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proceedings of ICSE'08*. ACM, 2008.

[45] R. Gu, Z. Zuo, X. Jiang, H. Yin, Z. Wang, L. Wang, X. Li, and Y. Huang, "Towards efficient large-scale interprocedural program static analysis on distributed data-parallel computation," *Transactions on Parallel and Distributed Systems*, 2021.

[46] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," in *Proceedings of SSYM'05*. USENIX, 2005.

[47] N. Ayewah, W. Pugh, J. Morgenthaler, J. Penix, and Y. Zhou, "Using findbugs on production software," in *Proceedings of OOPSLA'07*. ACM, 2007.

[48] H. Yan, Y. Sui, S. Chen, and J. Xue, "Spatio-temporal context reduction: a pointer-analysis-based static approach for detecting use-after-free vulnerabilities," in *Proceedings of ICSE '18*. ACM, 2018.

[49] B. Wu and A. Wang, "A multi-layer tree model for enterprise vulnerability management," in *Proceedings of SIGITE'11*. ACM, 2011.

[50] D. Manatova, I. Kouper, and S. Samtani, "Designing a vulnerability management dashboard to enhance security analysts' decision making processes," in *Proceedings of PEARC'22*. ACM, 2022.