Vulnerability Tracking using Normalized *Scope+Offset*

Preprint (author's version)

The definitive Version of Record is available at `https://doi.org/10.1145/3832783.3834513`.

Vulnerability Tracking using Normalized *Scope+Offset*

Julian Thome
GitLab Inc.
Luxembourg
jthome@gitlab.com

Hua Yan
GitLab Inc.
Australia
hyan@gitlab.com

Lucas Charles
GitLab Inc.
USA
lcharles@gitlab.com

Craig Smith
GitLab Inc.
Australia
csmith@gitlab.com

Jason Leasure
Independent Researcher
USA
jpleasu@gmail.com

Abstract

In heterogeneous Static Application Security Testing setups, multiple tools continuously monitor code changes to detect security vulnerabilities. Vulnerability Tracking deduplicates and tracks these vulnerabilities throughout a project's lifetime, reducing *futile auditing* caused by code volatility and double reporting. In prior work, we introduced the *Scope+Offset* method which identifies vulnerabilities based on a scope and offset fingerprint, reducing futile auditing by approximately 30%. However, the offset component is sensitive to non-functional code changes such as added comments or whitespace, which can cause spurious duplication. In this paper, we present *Scope+Offset*$_N$, an improved method that normalizes the offset computation by excluding non-functional source code lines. We detail the normalization algorithm and demonstrate its correctness on worked examples; on a targeted public benchmark it reduces unique fingerprints by 43%. *Scope+Offset*$_N$ has been deployed in GitLab since May 2025, providing tracking to thousands of daily security scans.

CCS Concepts

• **Software and its engineering** → **Software testing and debugging**; • **Security and privacy** → **Software and application security**.

Keywords

Static Application Security Testing, Vulnerability Tracking, Fingerprinting

1 Introduction

Continuously monitoring the constantly changing code of a software project with automated security testing to detect vulnerabilities as early as possible is a best practice in today's software industry [16, 22]. In heterogeneous SAST setups, multiple SAST tools are executed in combination to minimize the overall attack surface [3, 19]; such setups are commonly integrated with CI/CD processes on DevSecOps platforms such as GitLab or GitHub [24].

Vulnerability Tracking (VT) is an automated process that helps deduplicating and tracking vulnerabilities throughout the lifetime of a software project. In prior work [12], we introduced the *Scope+Offset* method which identifies vulnerabilities based on a fingerprint that combines the narrowest *scope* (the enclosing code context such as modules, classes, and functions) with an *offset* (the line distance from the scope boundary to the vulnerability). *Scope+Offset* proved effective, reducing *futile auditing* by approximately 30% compared to line-based fingerprinting. Since its integration into GitLab in 2022, *Scope+Offset* has provided tracking to thousands of daily security scans.

However, as noted in the limitations of the original work [12], the offset component is sensitive to *all* source code lines between the scope boundary and the vulnerability, including non-functional code such as comments and whitespace. Adding or removing comments within a scope changes the offset and can lead to the reintroduction of previously tracked vulnerabilities, causing unnecessary duplication. Addressing this sensitivity *coarsens* the underlying equivalence relation with respect to such changes: *Scope+Offset*$_N$ merges findings that differ only in non-functional code, eliminating the spurious duplicates *Scope+Offset* produces. The revised relation was adopted only after field reports involving C/C++ and JavaScript projects.

In this paper, we present *Scope+Offset*$_N$, an improved method that normalizes the offset computation by excluding non-functional source code lines. *Scope+Offset*$_N$ has been deployed in GitLab since May 2025, closing the reported gap in production. The key contributions are:

(1) A production-deployed improvement of *Scope+Offset* that eliminates spurious duplicates caused by non-functional edits; it reduces unique fingerprints by 43% on a targeted public benchmark and measurably lowers finding growth in production.
(2) A normalized notion of Vulnerability Equivalence that excludes non-functional code from the offset computation, realized by the `countNFL` algorithm which counts non-functional lines between a scope boundary and a vulnerability location by traversing the syntax tree.

Although realized within *Scope+Offset*, the normalization is not specific to it. Any line- or offset-based fingerprinting scheme, as used by other SAST tools or for secret detection, suffers from the same sensitivity to non-functional edits, and `countNFL` transfers directly: it only requires a parse tree that distinguishes functional from non-functional nodes.

2 Motivation

We briefly recap the *Scope+Offset* method from our prior work [12]. *Scope+Offset* computes a fingerprint for each vulnerability that

```
1  module Widget
2    class CustomWidget
3      def run(arg)
4        # create directory
5        if arg != ""
6          Dir.mkdir(arg)
7        end
8      end
9    end
10 end
```

Listing 1: `example.rb` with a path traversal vulnerability on line 6.

```
1  module Widget
2    class CustomWidget
3      def run(arg)
4        # create directory
5        # TODO: add validation
6
7        if arg != ""
8          Dir.mkdir(arg)
9        end
10     end
11   end
12 end
```

Listing 2: `example.rb` with additional comments; the same vulnerability is now on line 8.

encodes its context:

$$\textit{scope tag} \triangleright \textit{offset} \triangleright \textit{vulnerability classifier}$$

The *scope tag* identifies the enclosing code context (e.g., `Widget[0]▷CustomWidget[0]▷run[0]`), the *offset* is the line distance $\textit{offset} = l - \textit{start}(s)$ where l is the vulnerability line and s is the matching scope, and the *vulnerability classifier* categorizes the vulnerability (e.g., via CWE identifiers).

Two vulnerabilities are considered equivalent under *Scope+Offset* if they share the same scope, vulnerability type, and offset. This fingerprint is resilient to code shifts (e.g., adding functions before the enclosing scope) but remains sensitive to changes *within* the scope that alter line distances, including the addition or removal of non-functional code. We consider source code as either *functional* (defining program behavior) or *non-functional* (comments and whitespace added for formatting and readability). We abbreviate a **n**on-**f**unctional source code **l**ocation as NFL and denote a sequence of one or more NFLs as a *gap*.

Consider the Ruby file `example.rb` in Listing 1 which includes a path traversal vulnerability on line 6 where user input is directly passed to `Dir.mkdir`; the example is inspired by real failure cases reported in practice. With *Scope+Offset*, the fingerprint for this vulnerability includes the offset $\textit{offset} = 6 - 3 = 3$.

Listing 2 shows a modified version with additional comments and whitespace inserted before the call to `Dir.mkdir`. The vulnerability moved to line 8 and *Scope+Offset* computes $\textit{offset} = 8 - 3 = 5$, which differs from the offset in Listing 1. Since the functional code has not changed, these vulnerabilities are identical and should be deduplicated. Instead, the finding resurfaces as a new vulnerability whose triage state, such as a prior dismissal, does not carry over: an auditor has to re-examine it, and every non-functional edit near a tracked finding repeats this cycle. In this paper, we present *Scope+Offset*$_N$, which excludes non-functional lines from the offset computation, yielding identical fingerprints for Listing 1 and Listing 2.

3 Normalized Offset

We assume that every node n in the syntax tree can be mapped to source code line numbers by means of $s(n)$ and $e(n)$ representing the start and end lines, respectively, with $s(n) \leq e(n)$.

Vulnerability Equivalence. We revise the notion of Vulnerability Equivalence from [12]. Two vulnerabilities v_1 and v_2 are equivalent if:

(1) v_1 and v_2 are located in the same scope, **and**
(2) v_1 and v_2 have the same vulnerability type, **and**
(3) v_1 and v_2 have the same *normalized* offset to their parent scope boundary, *excluding non-functional code lines.*

Normalized Offset Computation. *Scope+Offset*$_N$ replaces the original offset with:

$$\textit{offset}_N = \textit{offset} - \texttt{countNFL}(\textit{scope}, l)$$

where $\texttt{countNFL}(\textit{scope}, l)$ counts non-functional lines between the start of the scope and the vulnerability line l.

Algorithm 1 details the `countNFL` algorithm. The algorithm operates on the syntax tree produced by the *Generic Parser* [12], a language-agnostic component that parses source files and identifies scope boundaries. `countNFL` takes two parameters: *node*, a node in this syntax tree, and *line*, the source code line of the vulnerability finding. It is initially invoked with the node that corresponds to the best-fitting scope for the vulnerability. The algorithm uses a counter variable *nfl* to accumulate non-functional line counts.

The algorithm collects the child nodes of *node* into an ordered sequence C and iterates over each child $c_i \in C$ using a depth-first left-to-right traversal. We assume that starting lines of visited nodes on the same level are non-decreasing. Only nodes on or before the vulnerability *line* need to be processed; once $s(c_i) > \textit{line}$, the loop terminates.

Non-functional lines arise from two sources: *explicit* NFLs that correspond to concrete syntax tree nodes (e.g., comments), and *implicit* gaps that are not represented by any node and can only be inferred from line number differences between parent/child or sibling nodes.

Explicit NFLs. For each child c_i, `countNFLSpan` (Algorithm 2, detailed below) checks whether c_i represents non-functional code and returns a count of occupied NFLs together with a boolean *skip* indicating whether the implicit gap computation can be skipped for this child.

To properly count non-functional lines inside sub-scopes, the algorithm recurses into relevant children selected by `shouldDescend`(c_i), which returns true if $e(c_i) - s(c_i) > 0 \wedge |\text{children}(c_i)| > 0$,

Algorithm 1 `countNFL`: Count non-functional lines between a scope node and a vulnerability line.

```
1:  function countNFL(node, line)
2:      nfl ← 0
3:      C ← children(node)
4:      for i = 1 to |C| do
5:          c_i ← C[i]
6:          if s(c_i) > line then break
7:          count, skip ← countNFLSpan(C, i)
8:          nfl ← nfl + count
9:          if skip then continue
10:         if shouldDescend(c_i) then
11:             nfl ← nfl + countNFL(c_i, line)
12:         end if
13:         if i = 1 then                    ▷ Gap: parent to first child
14:             nfl += max(s(c_i) − (s(node) + 1), 0)
15:         else if i > 1 then               ▷ Gap: between siblings
16:             nfl += max(s(c_i) − (e(c_{i−1}) + 1), 0)
17:         end if
18:     end for
19:     last ← C[|C|]
20:     if s(last) ≤ line then               ▷ Trailing gap
21:         nfl += max(e(node) − (e(last) + 1), 0)
22:     end if
23:     return nfl
24: end function
```

i.e., nodes that span more than a single line and have children of their own.

Implicit gaps. After processing explicit NFLs and recursing, the algorithm computes implicit gaps from three cases:

- $i = 1$: gap between the start of the parent and its first child (blank lines after an opening delimiter), computed as $s(c_i) - (s(node) + 1)$.
- $i > 1$: gap between consecutive siblings, computed as $s(c_i) - (e(c_{i-1}) + 1)$.
- Trailing gap: after the loop, if $s(last) \leq line$, the gap between the last child and the scope boundary (blank lines before a closing delimiter), computed as $e(node) - (e(last) + 1)$.

Counting explicit NFLs. Algorithm 2 details `countNFLSpan` which takes the ordered set of children C and the index i of the current child from `countNFL`. For explicitly represented newlines, the NFL count is $e(c_i) - s(c_i) + 1$ and *skip* is **true**. For comments, non-functional and functional code can share the same line; the *shared* variable accounts for this overlap by looking backward ($i > 1$) and forward ($i < |C|$) at adjacent non-comment nodes, using Ω to measure shared lines, and the NFL count is $e(c_i) - s(c_i) + 1 - shared$, clamped to zero.

Overlap helper. The function $\Omega(first, second)$ computes the overlap in source lines between two consecutive nodes:

$$\Omega(a, b) = \begin{cases} \min(e(a), e(b)) - \max(s(a), s(b)) + 1, & \text{if } e(a) \geq s(b) \\ 0, & \text{otherwise} \end{cases}$$

For example, a comment a spanning lines $[3, 5]$ and a neighboring node b starting on line 5 share $\Omega(a, b) = 1$ line.

Algorithm 2 `countNFLSpan`: Count explicit non-functional lines for a node.

```
1:  function countNFLSpan(C, i)
2:      c_i ← C[i]
3:      if isComment(c_i) then
4:          shared ← 0
5:          if i > 1 ∧ ¬isComment(c_{i−1}) then
6:              shared ← shared + Ω(c_{i−1}, c_i)
7:          end if
8:          if i < |C| ∧ ¬isComment(c_{i+1}) then
9:              shared ← shared + Ω(c_i, c_{i+1})
10:         end if
11:         return max(e(c_i) − s(c_i) + 1 − shared, 0), false
12:     else if isNewline(c_i) then
13:         return e(c_i) − s(c_i) + 1, true
14:     end if
15:     return 0, false
16: end function
```

Invariance. Provided the parser classifies every non-functional node as such and node spans do not include leading non-functional lines, the case analysis of `countNFL` counts each non-functional source line between $start(s)$ and l exactly once and never counts a line that contains functional code; blank lines trailing the last child are included as well, which does not affect the argument below. Explicit nodes are handled by `countNFLSpan`, where *skip* prevents double counting of explicitly represented newlines and Ω subtracts lines shared with functional code. Unrepresented lines fall into exactly one of the three gap cases, and `shouldDescend` extends the argument to nested sub-scopes. As a consequence, an edit that only inserts or removes non-functional lines before the finding changes *offset* and `countNFL`(s, l) by the same amount, leaving $offset_N$ and hence the $Scope{+}Offset_N$ fingerprint unchanged.

4 Example Walkthrough

Example 1. Figure 1 sketches a possible syntax tree for Listing 1. The scoping function node n_3 (highlighted) encloses the vulnerable call n_{11} on line 6. We invoke `countNFL`$(n_3, 6)$.

Table 1 traces the recursive evaluation. For Example 1, the comment node n_7 on line 4 contributes 1 NFL via `countNFLSpan`, yielding $offset_N = (6 - 3) - 1 = 2$.

Example 2. For Listing 2, the tree (Figure 2) gains a comment node n_8 (line 5) followed by an implicit gap (line 6); the vulnerable call moves to n_{12} on line 8. Both comments and the gap contribute $nfl = 3$, yielding $offset_N = (8 - 3) - 3 = 2$.

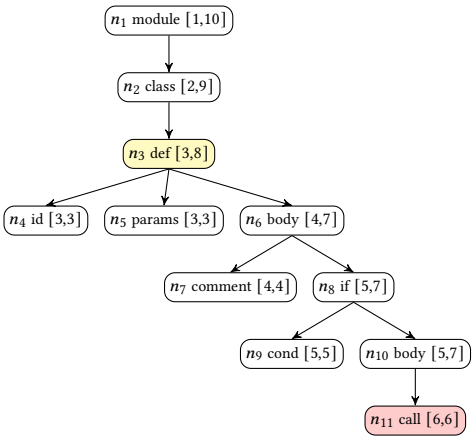

Figure 1: Syntax tree for Listing 1. The scope node n_3 is highlighted in yellow; the vulnerable call n_{11} in red.

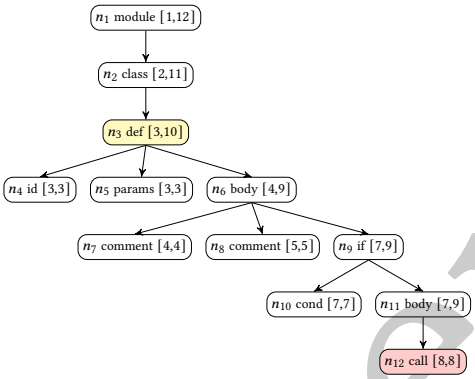

Figure 2: Syntax tree for Listing 2. The scope node n_3 is highlighted in yellow; the vulnerable call n_{12} in red.

Table 1: Evaluation traces for `countNFL` on Listing 1 (left) and Listing 2 (right).

`countNFL`$(n_3, 6)$		`countNFL`$(n_3, 8)$	
Invocation	*nfl*	Invocation	*nfl*
`countNFL`$(n_3, 6)$	0	`countNFL`$(n_3, 8)$	0
`countNFL`$(n_6, 6)$	0	`countNFL`$(n_6, 8)$	0
n_7: comment	$+1 \rightarrow 1$	n_7: comment	$+1 \rightarrow 1$
`countNFL`$(n_8, 6)$	1	n_8: comment	$+1 \rightarrow 2$
`countNFL`$(n_{10}, 6)$	1	`countNFL`$(n_9, 8)$	2
n_{11}: call	$+0 \rightarrow 1$	`countNFL`$(n_{11}, 8)$	2
		gap: $7 - (5+1)$	$+1 \rightarrow 3$
Result	**1**	**Result**	**3**

The normalized offset $offset_N = 2$ is identical for both Listing 1 and Listing 2, so *Scope+Offset*$_N$ deduplicates the vulnerability, whereas *Scope+Offset* produces different offsets (3 vs. 5) and treats them as distinct findings.

5 Evaluation

We evaluate *Scope+Offset*$_N$ along two dimensions: fingerprint stability under controlled perturbations (RQ1) and real-world impact on production deduplication (RQ2).

- *RQ1: Does normalizing the offset computation reduce spurious fingerprint duplicates caused by non-functional code changes?*
- *RQ2: Does the improvement translate to a measurable reduction in production?*

To answer RQ1, we leveraged the *SourceWarp* Record & Replay framework [24]: it records a patch sequence from a slice of a Git commit history and replays it against a target system while collecting statistics.

5.1 Setup

As input for *SourceWarp* we used `sast-rules` [6], a multilingual benchmark repository containing known vulnerability samples across C/C++, C#, Go, Java, JavaScript, Python, and Ruby (439 source files). Vulnerability locations in the benchmark are marked, which allows us to systematically locate and perturb the code adjacent to each known vulnerability.

To simulate non-functional code changes, we created 2247 commits each of which inserts exactly one perturbation—a comment line or a blank line—directly before a marked vulnerability location. The perturbations are applied bottom-up within each file (processing the last vulnerability location first) to preserve line numbers for subsequent insertions. This approach ensures that each commit perturbs exactly one vulnerability location.

The target system under test consists of a dockerized version of the GitLab Semgrep analyzer v6 [5] equipped with the `tracking-calculator` [8] post-analyzer. For each scan, the analyzer produces a SAST report in the GitLab security report format [4] containing all detected findings with both *Scope+Offset* and *Scope+Offset*$_N$ fingerprints.

We configured *SourceWarp* with a sampling interval of 50 patches, yielding 45 probing points. At each probing point, the analyzer scans the repository and produces a report. For each fingerprinting algorithm, we track the cumulative set of unique fingerprints across all probing points where each fingerprint combines the algorithm-specific value with the vulnerability's primary identifier to distinguish different vulnerability types. Fewer unique fingerprints indicate better deduplication.

5.2 Results

Table 2 shows the cumulative unique fingerprint counts for *Scope+Offset* and *Scope+Offset*$_N$ at sampled probing points. Both algorithms start at 1768 unique fingerprints which constitutes the baseline before any perturbations are applied.

RQ1. *Scope+Offset*$_N$ performed better than *Scope+Offset* in terms of deduplication. *Scope+Offset* accumulates 1361 duplicate fingerprints (77% growth over the baseline, reaching 3129), whereas *Scope+Offset*$_N$ produces zero duplicates and remains at the baseline of 1768 throughout the entire experiment—a 43% reduction in overall unique fingerprints. This confirms that the normalized offset computation successfully absorbs all perturbations that consist solely

Table 2: Evaluation results: cumulative unique fingerprints after applying $50 \times i$ perturbation patches, for the original (*Scope+Offset*) and normalized (*Scope+Offset$_N$*) methods; Δ_{abs} and Δ_{rel} show the absolute and relative reduction achieved by *Scope+Offset$_N$*. Smaller counts indicate better deduplication.

	#Unique Fingerprints at probe$_i$																					
	1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39	41	43	45
Scope+Offset	1768	1852	1937	2053	2170	2251	2352	2427	2465	2474	2480	2481	2556	2649	2743	2832	2922	3008	3098	3105	3114	3114	3129
Scope+Offset$_N$	1768	1768	1768	1768	1768	1768	1768	1768	1768	1768	1768	1768	1768	1768	1768	1768	1768	1768	1768	1768	1768	1768	1768
Δ_{abs}	0	84	169	285	402	483	584	659	697	706	712	713	788	881	975	1064	1154	1240	1330	1337	1346	1346	1361
Δ_{rel} (%)	0	5	9	14	19	21	25	27	28	29	29	29	31	33	36	38	39	41	43	43	43	43	43

of non-functional code changes. This experiment is a targeted validation: the perturbations exercise exactly the weakness class *Scope+Offset$_N$* is designed to eliminate, so the reduction is an upper bound on the achievable improvement.

RQ2. Production data shows that after the deployment of *Scope+Offset$_N$*, the average monthly growth rate of SAST findings decreased by 4.86 percentage points. The measurement compares month-over-month growth rates of newly created SAST findings over a two-year window split at the May 2025 deployment, excluding outlier months in each period. The externally reported re-auditing cases caused by non-functional changes, including those that inspired the running example, were resolved after deployment. The results are complementary: RQ1 gives the best case under purely non-functional edits; RQ2 reflects more realistic conditions where such edits interleave with functional changes.

6 Deployment

Scope+Offset$_N$ is implemented in `tracking-calculator` [8], a Go-based post-analyzer that uses the incremental parser generator `tree-sitter` [25]. The implementation extends the existing *Scope+Offset* fingerprint generation step with the `countNFL` algorithm and supports C#, C/C++, Go, Java, JavaScript, Python, Ruby, and PHP. *Scope+Offset$_N$* was integrated into the GitLab product in May 2025 where it is available as the `scope_offset_compressed` algorithm, complementing the original *Scope+Offset* method that has been operational since 2022 [12]. The normalization step reuses the same parse tree that is already constructed during *Generic Parsing*; we observed no regressions in scan times. The GitLab SAST report exchange format [4] remains unchanged, ensuring seamless integration with any combination of SAST tools. Spurious duplicates still occur when functional code is inserted or removed between the scope boundary and the finding, and, by design of the conservative fallback, when a parser cannot classify a node as non-functional; such cases delimit what the normalization can absorb. GitLab itself benefits from the reduced re-auditing effort: it applies its own SAST and Vulnerability Tracking to its internal, predominantly Ruby codebase, which also motivated the choice of Ruby for the running example.

7 Related Work

A comprehensive discussion of related work is provided in [12]; we briefly summarize the most relevant areas here. In cybersecurity, vulnerabilities and malware are tracked via fingerprints computed at the line [10], function [15] or whole-file [11] level, via program dependence graphs for partial semantics-awareness [17, 27], or via fuzzy hashing [18, 20]. In comparison, our fingerprinting method is based on the structure of the syntax tree; the normalization introduced by *Scope+Offset$_N$* makes the fingerprint insensitive to non-functional code changes. This work is also related to Clone Detection [13] that leverages text-based [21], token-based [14], and syntax-tree-based [1, 2] representations; while clone detection aims to reduce maintenance time, *Scope+Offset* and *Scope+Offset$_N$* aim to reduce *futile auditing time*. Bug report deduplication [23, 26, 28] relies on natural language processing, whereas *Scope+Offset* and *Scope+Offset$_N$* operate on source code before Vulnerability Management takes place.

8 Limitations

The limitations of the original *Scope+Offset* method [12], other than the sensitivity to non-functional code changes, still apply to *Scope+Offset$_N$*. Since both methods identify findings by position rather than content, the coarser equivalence of *Scope+Offset$_N$* raises the risk of conflating genuinely distinct findings whose offsets coincide. The accuracy of the normalization depends on the parser's ability to distinguish functional from non-functional nodes. Edge cases arise from deviating parser conventions such as grammar-specific comment node types or node spans that begin before their first token; when in doubt, a node is treated as functional, so such cases degrade conservatively into *Scope+Offset*-like duplicates rather than incorrect merges.

The practical benefit also depends on programming style: where non-functional-only changes are rare, the normalization has fewer duplicates to avoid. As noted in the evaluation, the benchmark is favorable to *Scope+Offset$_N$* by construction, so the RQ1 gap may be smaller in practice. The production improvement observed in RQ2 is subject to confounding factors such as changes in the user base, detection rules, and project activity, which prevent a strict causal attribution solely to *Scope+Offset$_N$*.

9 Conclusion

We present *Scope+Offset$_N$*, an improvement to *Scope+Offset* [12] that normalizes the offset by excluding non-functional lines, producing fingerprints insensitive to comment and whitespace changes. *Scope+Offset$_N$* has been deployed in GitLab since May 2025, serving thousands of daily security scans [9].

Data Availability. SourceWarp [24], the evaluation patches [7], and `sast-rules` [6] (version `d580dedc`) are publicly available. The `tracking-calculator` [8] is an internal GitLab project.

References

[1] Brenda S. Baker. 1995. On finding duplication and near-duplication in large software systems. In *Proceedings of WCRE'95*. IEEE. doi:10.1109/WCRE.1995.514697

[2] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings of ICSM'98*. IEEE. doi:10.1109/ICSM.1998.738528

[3] Gartner. 2022. Magic Quadrant for Application Security Testing. https://web.archive.org/web/20230201221830/https://www.gartner.com/en/documents/4013799.

[4] GitLab. 2026. GitLab SAST. https://docs.gitlab.com/ee/user/application_security/sast/.

[5] GitLab. 2026. GitLab Semgrep Analyzer. https://gitlab.com/gitlab-org/security-products/analyzers/semgrep.

[6] GitLab. 2026. SAST Rules Benchmark. https://gitlab.com/gitlab-org/security-products/sast-rules.

[7] GitLab. 2026. Tracking Benchmark. https://gitlab.com/gitlab-org/secure/static-analysis/experiments/tracking-benchmark.

[8] GitLab. 2026. tracking-calculator Post analyzer. https://gitlab.com/gitlab-org/security-products/post-analyzers/tracking-calculator.

[9] GitLab. 2026. Vulnerability Tracking Development Documentation. https://docs.gitlab.com/development/sec/vulnerability_tracking/.

[10] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: Finding unpatched code clones in entire OS distributions. In *Proceedings of S&P'12*. IEEE. doi:10.1109/SP.2012.13

[11] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stéphane Glondu. 2007. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proceedings of ICSE'07*. IEEE. doi:10.1109/ICSE.2007.30

[12] James Johnson, Julian Thome, Lucas Charles, Hua Yan, and Jason Leasure. 2025. A scalable, effective and simple Vulnerability Tracking approach for heterogeneous SAST setups based on Scope+Offset. In *Proceedings of ICSE-SEIP'25*. IEEE. doi:10.1109/ICSE-SEIP66354.2025.00053

[13] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. 2009. Do code clones matter?. In *Proceedings of ICSE'09*. IEEE. doi:10.1109/ICSE.2009.5070547

[14] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *TSE* (2002). doi:10.1109/TSE.2002.1019480

[15] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A scalable approach for vulnerable code clone discovery. In *Proceedings of S&P'17*. IEEE. doi:10.1109/SP.2017.62

[16] KPMG. 2019. Agile Transformation. https://web.archive.org/web/20201117085708/https://assets.kpmg/content/dam/kpmg/be/pdf/2019/11/agile-transformation.pdf.

[17] Jens Krinke. 2001. Identifying similar code with program dependence graphs. In *Proceedings of WCRE'01*. IEEE. doi:10.1109/WCRE.2001.957835

[18] K A Monnappa. 2015. Automating Linux malware analysis using Limon sandbox. Black Hat Europe. https://www.blackhat.com/docs/eu-15/materials/eu-15-KA-Automating-Linux-Malware-Analysis-Using-Limon-Sandbox-wp.pdf

[19] National Institute of Standards and Technology. 2020. *Security and Privacy Controls for Information Systems and Organizations*. Technical Report. NIST. doi:10.6028/NIST.SP.800-53r5

[20] Fabio Pagani, Matteo Dell'Amico, and Davide Balzarotti. 2018. Beyond precision and recall: understanding uses (and misuses) of similarity hashes in binary analysis. In *Proceedings of CODASPY'18*. ACM. doi:10.1145/3176258.3176306

[21] Chanchal K. Roy and James R. Cordy. 2008. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of ICPC'08*. IEEE. doi:10.1109/ICPC.2008.41

[22] Stack Overflow. 2025. Stack Overflow Developer Survey 2025. https://survey.stackoverflow.co/2025.

[23] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. 2011. Towards more accurate retrieval of duplicate bug reports. In *Proceedings of ASE'11*. IEEE. doi:10.1109/ASE.2011.6100061

[24] Julian Thome, James Johnson, Isaac Dawson, Dinesh Bolkensteyn, Michael Henriksen, and Mark Art. 2023. 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*. IEEE. doi:10.1109/AST58925.2023.00011

[25] Tree-sitter. 2026. The Tree-sitter parser generator. https://tree-sitter.github.io/tree-sitter/.

[26] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. 2008. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of ICSE'08*. ACM. doi:10.1145/1368088.1368151

[27] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, and Wenchang Shi. 2020. MVP: Detecting vulnerabilities using patch-enhanced vulnerability signatures. In *Proceedings of USENIX Security'20*. USENIX. https://www.usenix.org/conference/usenixsecurity20/presentation/xiao

[28] Ting Zhang, DongGyun Han, Venkatesh Vinayakarao, Ivana Clairine Irsan, Bowen Xu, Ferdian Thung, David Lo, and Lingxiao Jiang. 2023. Duplicate bug report detection: How far are we? *TOSEM* (2023). doi:10.1145/3576042