A novel method to study the emergence of software vulnerabilities in code

Malo MONIN
Intern
ENS Rennes, Univ. Rennes
Bruz, France

Djamel E. Khelladi
DiverSE
CNRS, Univ. Rennes, IRISA, Inria
Rennes, France

Arnaud Blouin
DiverSE
INSA, Univ. Rennes, IRISA, Inria
Rennes, France

***Abstract*—Large codebases with many contributors inevitably fall victim to the emergence of bugs and vulnerabilities. The precise way they are progressively introduced is not yet fully understood. Prior works have focused mainly on small amounts of Java codebases because they rely on slow tools and require human intervention. We propose and evaluate a novel process for extracting from a Git commit history the changes that induce a fixing commit and mining patterns from those changes. For tracking down fix-inducing changes, we use HyperAST, a tool that far surpasses its predecessors both in terms of memory usage and in terms of execution time.**

***Index Terms*—Software vulnerabilities, detecting fix-inducing commits, polyglot code analysis, empirical software engineering**

I. Introduction

When working on codebases where singular contributors cannot have a complete understanding of the intricate interactions between different parts, it is inevitable that some modifications will inadvertently contribute to the emergence of bugs, vulnerabilities, or security breaches. Previous works [1], [2], [3], [4] have focused on identifying changes within a codebase history that contribute to the emergence of a bug or vulnerability ("fix-inducing changes"). This process can performed at the commit level, or more finely at the level of particular changes within those commits. One of the ways of determining the fix-inducing changes for a specific bug or vulnerability is, given a fixing commit, to track down the changes that previously affected the same pieces of source code. However, no satisfying systematic method exists for applying this procedure. Indeed, previous works use slow methods involving tools such as Spoon [5], which can take up to multiple days to analyze an entire project [6], and often only consider a single programming language (typically, Java [5], [7], [8]). We present and implement a novel method for tracking down changes affecting the same pieces of code as a fixing commit using HyperAST [6]. We apply this method and use pattern mining algorithms [9] to attempt to reproduce some of the results from the literature [1], [2], [3], [4].

More specifically, we propose a generic approach that, starting from a given commit (typically, a fixing commit) and its repository, traces back the related changes in the project's history and compute an abstract summary. An *abstract summary* is a succession of sets of edits. Each set of edits describes a commit, and an edit is either an insertion, a deletion, a move, or an update, associated with semantic information. For example, adding a function could result in the edit "**Insert** FunctionDefinition." An abstract summary describes changes to apply to a starting point (in our case, the project's initial commit) to get to an end point (in our case, the parent of the fixing commit). Then, we mine frequent closed sequential patterns from the SPMF pattern mining java library [10] to mine patterns from those abstract summaries.

We formulate the hypothesis that results analogous to the state of the art's can be reproduced more efficiently using HyperAST. To test this hypothesis, we run an evaluation on Defects4J [11], a dataset of real-world defects. Out of 854 defects, we were able to generate abstract summaries for 280 of them, and the pattern mining algorithms detected 191,594,297 patterns. This exceedingly large number of patterns, which indicates that the pattern mining algorithm we used is not appropriate. We considered other algorithms, but did not have the time to apply them.

In the Section II, we clarify our goal by laying out the state of the art, briefly presenting HyperAST, and formulating a hypothesis and research questions. In Section III, we unfold the entire process, from the collection of vulnerability-contributing changes, to the implementation of the tracker, to the choice pattern mining algorithms. In Section IV, we compare our results to the state of the art and answer the research questions. Section V provides an overview of related works and contextualizes the work presented here.

II. Goal

In this section, we present existing studies on the emergence of bugs, vulnerabilities, and complex functions, in code. We detail some of the methods used in those works to retrace the life of pieces of code corresponding to bugs or vulnerabilities. Next, we give a background on HyperAST, the tool we use for this purpose. Finally, we formulate the hypothesis that results analogous to the state of the art's can be reproduced more efficiently using HyperAST, and introduce our research questions.

A. State of the art

Previous studies [1], [2] have been able to find valuable information regarding the emergence of vulnerabilities in source code. This information ranges from change patterns (i.e., number of lines added and removed, [1], [2], number of files touched [1], complexity of the changes [2]) to meta-information (i.e., time since the project creation, time since

the file creation, experience of the contributor, workload of the contributor [1]).

Although existing works use various different methods to identify the fix-inducing changes or commits (SZZ algorithm [1], [4], backporting regression tests [3], etc.), they all heavily rely on human intervention. The state of the art primarily focuses on Java code, even though it has been shown that the notion of code complexity, which plays an important role in the emergence of vulnerabilities [12], varies across languages [13]. Additionally, the reliance on slow tools, as well as the requirement for manual classification of commits and changes, make the state of the art methods unsuitable for large-scale polyglot analyses. In the following section, we introduce a data structure that solves those issues.

B. HyperAST

HyperAST [6] is a new data structure that represents an abstract syntax tree that evolves across time. It is based on Git and uses Tree-sitter [14] to parse source files. As such, it is compatible with any language for which a Tree-sitter grammar can be provided. From a Git repository, HyperAST builds a concrete syntax tree (CST) with a temporal dimension. It associates Git object identifiers (OIDs) with CST nodes, thus making it possible to share the representation of identical objects in the tree. The overall approach is illustrated in Fig. 1 HyperAST has been implemented in Rust and shows up to a 99.9% reduction in memory footprint, and up to a 99.99% gain in construction time compared to Spoon [5], the more traditional approach. The input files are parsed without error in 99.98% of the cases. One of the use cases of HyperAST is tracing the evolution of code elements. [6, Section 4.4, use case #3] This can be applied to the pieces of code affected by a fixing commit to trace them back in time and determine the commits that affected them.

Additionally, we make use of HyperDiff [15], an extension of HyperAST that enables diffing of commits and the generation of edit scripts that contain the semantic information required to build abstract summaries.

C. Research questions

We formulate the hypothesis that results analogous to the state of the art's can be reproduced more efficiently using HyperAST. To test this hypothesis, we propose the following research questions.

RQ1 Can we track down changes affecting a piece of code and generate abstract summaries efficiently with HyperAST?

RQ2 Can we mine patterns analogous to the state of the art's from the generated abstract summaries?

III. Methodology

The method presented here is separated into three independent steps that each use the result of the previous step. It is illustrated in Fig. 2. The first step (Section III.A) is the collection of Git repositories and fixing commits. This information can be obtain for different sources, independently of the other steps. For example, one might source data from

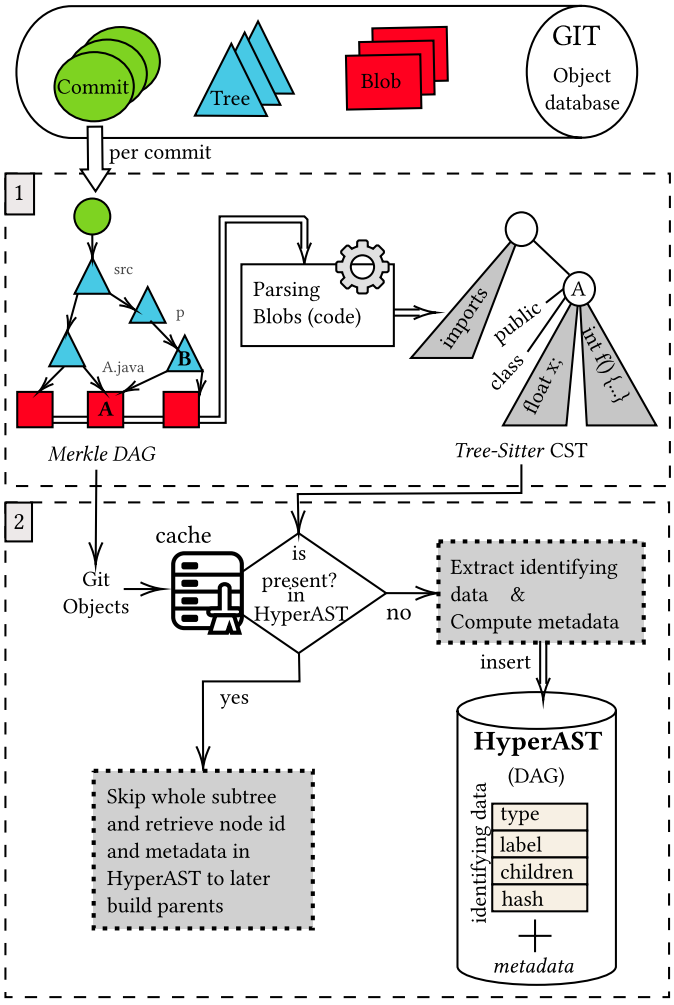

Fig. 1: Overall approach of HyperAST [6, Figure 3].

security breach databases, or from bug databases. The second step (Section III.B) accepts the list of Git repositories and fixing commits collected in the first step in a format agnostic of the initial database and, for each fixing commit, tracks down the fix-inducing changes and summarizes them into an abstract summary, using HyperAST. Finally, the third step (Section III.C) mines patterns from the abstract summaries generated in the second step. We detail the inner workings of each step in the following sections.

A. Gathering data from databases

The first step consists of collecting a list of Git repositories and fixing commits that can be provided to the tracker. This step is independent of both the forge where the Git repository is stored, and the contents of the repository, including the programming languages used. However, it should be adapted to each different data source. Indeed, the goal of this step is to convert data from a given source to a unified format that can be parsed by the tracker.

The information that the tracker needs is a list of fixing commits and corresponding Git repositories. For simplicity, and as further discussed in Section III.B, only repositories

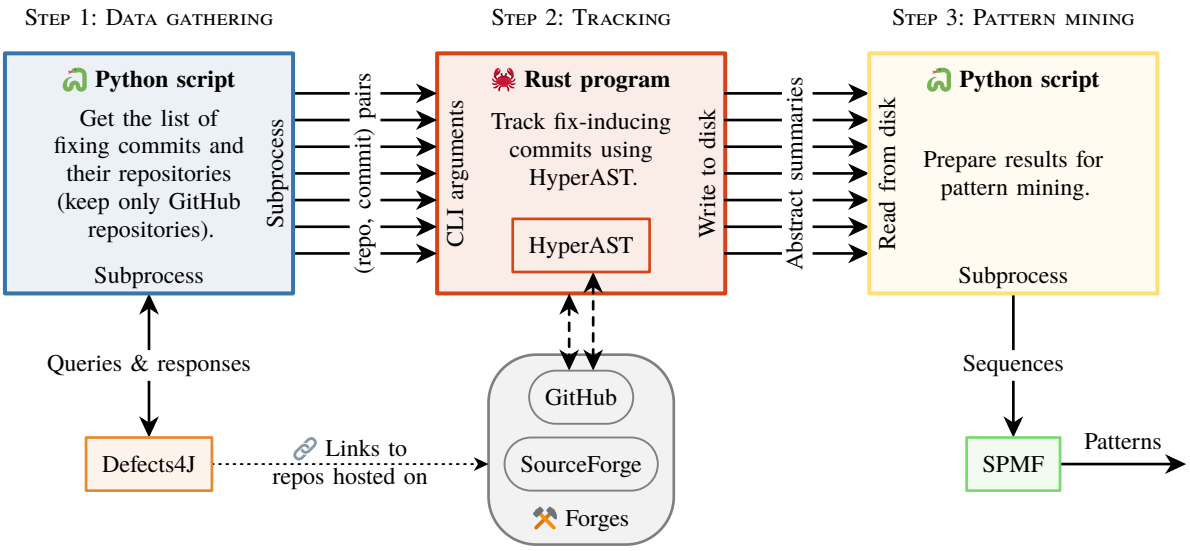

Fig. 2: Pipeline architecture of the proposed method. Each column corresponds to a step of the process. The top row contains the programs that we wrote, while the bottom row consists of the existing resources we make use of. Notice how the first and last steps are completely oblivious of each other. Dashed arrows indicate communication with distant forges using Git.

hosted on GitHub[1] are supported for now. This is a temporary restriction that should be lifted in the future. The information is provided to the tracker as command line arguments. This makes it easy to plug any bug or vulnerability database into the tracker using a simple script.

As a proof of concept, we rely on Defects4J [11] as our bug database. Defects4J is a database of Java bugs that provides the necessary information. Defects4J contains a total of 854 defects from 17 different repositories. They use Git as their version control system, with the exception of a single one (responsible for 26 defects) which uses Apache Subversion, and had to be excluded for this reason. Defects4J does not provide a direct link to the project's repositories. Instead, we derive it from the report URL. Out of the 828 defects from the repositories using Git, 280 (33.8 %) have report URLs under the domain github.com. From those, we infer the URL of the corresponding GitHub repositories. The corresponding 280 fixing commits and associated GitHub repositories are the ones we use as inputs for the tracker. In total, we thus use 32.8 % of the defects from Defects4J. In a Python script, we query Defects4J through its command line interface, and call the tracker using `subprocess`.[2] An example of a fixing commit is provided in Listing 1.

Other databases were considered, such as OSV,[3] a vulnerability database maintained by Google. However, lack of time prevented our applying the method to these databases.

B. *Tracking pieces of code*

The second step consists of tracking down changes affecting the same pieces of code as a given commit in prior commits (i.e., ancestors in the git tree) within a given repository, and generating a corresponding abstract summary. An *concrete summary* is an ordered sequence of sets of edits, where each

```
--- a/Main.java
+++ b/Main.java
@@ -51,12 +51,14 @@
 String code;
 switch (color) {
     case RED:
         code = "#FF0000";
         break;
     case GREEN:
         code = "#00FF00";
         break;
     case BLUE:
         code = "#0000FF";
+        break;
     case YELLOW:
         code = "#FFFF00";
 }
```

Listing 1: Simplified diff for a hypothetical fixing commit.

set of edits corresponds to a single commit, and an edit affects a given text within a file, and is either an insertion, a deletion, a modification, or a move. An *abstract summary* is similar a concrete summary where text is abstracted away and replaced with semantic information (e.g., an edit consisting of adding a new function might become "**Insert** FunctionDefinition"). Listing 2 presents an example of a fix-inducing commit, and Fig. 3 the corresponding abstract summary.

This is achieved using a Rust program that relies on HyperAST as a library. The tracker accepts a list of Git repositories and commits as command line arguments and generates an abstract summary for each of them in parallel. For now, the repositories have to be hosted on GitHub.[1] This restriction is temporary and allowed for faster prototyping of the tracker.

Each command line argument received by the tracker corresponds to a single abstract summary generation task and is independent of the other tasks. As such, the tasks are embarrassingly parallelizable, which results in considerable speed

[1]https://github.com/
[2]https://docs.python.org/3/library/subprocess.html
[3]https://osv.dev/

```
--- a/Color.java
+++ b/Color.java
@@ -2,4 +2,5 @@ enum Color {
      RED,
      GREEN,
      BLUE,
+     YELLOW,
  }
--- a/Main.java
+++ b/Main.java
@@ -51,11 +51,13 @@
  String code;
  switch (color) {
      case RED:
          code = "#FF0000";
          break;
      case GREEN:
          code = "#00FF00";
          break;
      case BLUE:
          code = "#0000FF";
+     case YELLOW:
+         code = "#FFFF00";
  }
```

Listing 2: Simplified diff for a hypothetical fix-inducing commit leading to the fixing commit from Listing 1.

up on machines with many cores. The computed abstract summaries are written to disk at the end of each task.

The tracker uses HyperAST in multiple places to interact with Git repositories. Notably, HyperAST is used to track down pieces of code, and to generate abstract summaries. HyperAST proved to be a powerful tool, although the interface is imperfect and still evolving as HyperAST is developed. For this reason, we resort to using the Git command line interface to detect the pieces of code affected by a fixing commit instead of relying of HyperAST (left dashed arrow in Fig. 2). This allowed for faster development of the tracker, but results in possibly slightly less exact results and worse code quality. Additionally, some minor modifications had to be applied to the HyperAST library. Notably, we had to expose 21 struct fields, a module, a type, and a function, as well as remove a panic case, raise a constant, and derive `Debug` for a type (Rust's way of making values of a type printable for debug purposes).

As a developing tool, HyperAST sometimes proves unable to track pieces of code from a specific commit to its parent. This happens on 93.9 % of the inputs, although some repositories cause more problems than others, and may take the form of a panic, an error, or an empty result. In case of a panic or error, a possible solution is to simply stop the algorithm as soon as a problem is encountered, effectively acting as if

- Set 1 (corresponding to the only commit)
  - **Insert** EnumValue
  - **Insert** SwitchCase

Fig. 3: A simplified abstract summary[4] corresponding to the fix-inducing commit from Listing 2 (we assume no other commit affects this piece of code in the project history).

[4]Not generated by the tracker.

the commit history stopped at this point. This solution is not satisfactory because vulnerabilities are often introduced over large periods of time: vulnerabilities take an average of 4.2 years, and up to 20 years, to be fully introduced [1, Section 3.1], and are not fixed until 2.6 years after their full introduction on average [1, Section 3.3]. Instead, we simply skip the parent and compare the child commit to its grandparent. Seeing as our end goal is to construct an abstract summaries, skipping some commits is not an issue as long as we do not skip too many successive commits.

C. Mining patterns

The third and final step consists of mining patterns from the abstract summaries generated in the previous step. Although options exist for pattern mining specifically in Git commit histories [16], [17], we used more a generic solution for simplicity: an abstract summary is an ordered collection of unordered sets of edits. This structure matches that of the inputs of *sequential pattern mining algorithms*. Sequential pattern mining algorithms mine patterns from ordered sequences of sets of items [9]. More precisely, the mined patterns are sequences that occur in more than some given amount of input sequences. A sequence $X_1, ..., X_k$ is said to *occur in another sequence* $Y_1, ..., Y_n$ if and only if, there exists integers $1 \leq i_1 < ... < i_k \leq m$ such that $X_1 \subseteq Y_{i_1}, ..., X_k \subseteq Y_{i_k}$. The *support* of a sequence is the number of input sequences in which it appears. A mined sequence is said to be *closed* when it does not occur in a different mined sequence having the same support.

We relied on SPMF [10], a Java pattern mining library. It provides implementations for a large collection of pattern mining algorithms [9]. Specifically, we used the frequent sequential pattern mining algorithm PrefixSpan.[5]

IV. Results

In this section, we present the results of our works. In particular, we focus on the benefits of HyperAST and the quality of the mined patterns. We answer our research questions.

A. Can we track down changes affecting a piece of code and generate abstract summaries efficiently with HyperAST? (RQ1)

On a ProLiant DL365 Gen10 Plus (2 × AMD EPYC 7543 32-Core CPU @ 2.8GHz; 32/32 cores, 64 threads / CPU) with a RAM of 756GB, we were able to handle 280 inputs in about four hours. Since each task is parallelizable, the running time is determined by the longest-running tasks. Most run in under five minutes, but some may take multiple hours. Tasks that result in an empty abstract summary are not always among the fastest to execute, but tasks that result in an error or panic tend to halt in a matter of minutes.

With only 6.1 % of the tasks resulting in a non-empty result, it is not possible to answer RQ1 affirmatively without first solving this issue. This is further discussed in Section VI.

[5]https://www.philippe-fournier-viger.com/spmf/PrefixSpan.php

TABLE I: The first four out of 191,594,297 patterns mined using the PrefixSpan algorithm. The support of a pattern is the number of sequences in the input that contain the pattern. The last column indicates whether a pattern is closed among the patterns in the table.

Pattern	Support	Closed?
• In some commit ‣ **Insert** OpeningBrace	17	No
• In some commit ‣ **Insert** Openingbrace ‣ **Insert** WildcardBound	12	No
• In some commit ‣ **Insert** Openingbrace ‣ **Insert** ClosingBrace	17	Yes
• In some commit ‣ **Insert** Openingbrace ‣ **Insert** ClosingBrace ‣ **Insert** WildcardBound	12	Yes

B. Can we mine patterns analogous to the state of the art's from the generated abstract summaries? (RQ2)

The only pattern mining algorithm we had the time to test was PrefixSpan [9], and it produced 191,594,297 patterns from 17 abstract summaries. This amount is too high to allow for manual interpretation of the generated patterns. In particular, we were not able to find patterns analogous to the state of the art's.

PrefixSpan mines frequent sequential patterns from sequences. It is not restricted to closed patterns. In other words, many of the mined patterns are related for the sequence inclusion relation. For example, the first four mined patterns are shown in Table I. Frequent closed sequential pattern mining algorithms should be considered in the future for a better chance of meaningful results. This is further discussed in Section VI.

C. Comparison to the state of the art

We were not able to reproduce results analogous to the states of the art's. Whether this is due to limitations of HyperAST, HyperDiff, the pattern mining algorithms used, or the general method, is to be determined as part of future works. For now, it is not possible to validate the hypothesis that results analogous to the state of the art's can be reproduced with HyperAST.

D. Limitations & threats to validity

Mainly due to lack of time, this work has limitations and threats to validity.

Notably, it prevented our making the abstract summary generation phase work on all inputs, as well as testing more pattern mining algorithms. This is detrimental to the quality of the final result, which may unfairly represent the true potential of the proposed method. Currently, the tracker fails to generate abstract summaries in two ways: by erroring or panicking when calling HyperAST or HyperDiff, and by not detecting any change, therefore generating empty summaries. Out of the 280 defects that we used as input to the tracker, only 17 abstract summaries were generated, which amounts to a 93.9 % of unsuccessful cases.

Additionally, the necessity to modify the source code HyperAST deprived us of the ability to depend on HyperAST as a crates.io[6] dependency, as is usual with Cargo-built Rust programs, instead requiring us to include altered sources of HyperAST as part of the tracker. In particular, this means future updates to HyperAST will be non-trivial to apply as the tracker essentially behaves as a fork of HyperAST. However, the modifications in question are minor and could become part of HyperAST itself in the future, allowing us to depend on it as a remote Git dependency, or even as a crates.io dependency.

V. Related works

Previous works have already tackled similar topics. While most focus on Java [5], [7], [8], some propose polyglot methods, such as CodeShovel [18], a tool that reconstructs the history of a method in a codebase. This is similar to our second step, where we track changes related to individual pieces of code across a repository's commit history, but is more limited in scope, CodeShovel focuses on methods, whereas we may consider any kind of code element.

This builds upon Le Dilavrec's work on HyperAST [6] and HyperDiff [15] and presents a use case for those tools: finding changes affecting the same code elements as a given commit throughout a Git repository's history. Some existing work do not use specialized tools for similar goals, and instead rely on Git alone to track pieces of code to prevent having to parse source files [7], [8]. This has the advantage of much faster execution times (up to the order of magnitude of a minute) compared to HyperAST and other tools that do not rely on Git mechanisms.

VI. Conclusion and future works

We proposed and implemented a novel method to study the mergence of security breaches in code. We showed how HyperAST has the potential to outperform existing tools in terms of compute time to generate abstract summaries, which we can then mine for patterns, although we lacked time to make the tracker work in a majority of inputs. Using HyperAST and HyperDiff, we were able to successfully track the changes in a project's history that affected the same pieces of code as a given fixing commit, and generate a corresponding abstract summaries for 2 % of the defects from Defects4J, a dataset of Java defects.

Future works on this matter should expand on what was done here and address the limitations mentioned in Section IV.D. In particular, improving HyperAST, HyperDiff, and our implementation of the tracking phase to make it work on more inputs. Indeed, we currently generate non-empty abstract summaries for 6.1 % of the inputs. It appears HyperAST errors or panics on some inputs, but this could be due to improper usage in the tracker.

Additionally, the only pattern mining algorithm we had the time to consider was PrefixSpan [9], which is a frequent sequential pattern mining algorithm. This generated a very high amount of patterns (191,594,297), which makes it harder

[6]https://crates.io/

to interpret them and extract useful information. Instead, frequent *closed* sequential pattern mining algorithms should be considered. For example, SPMF [10] implements CloFast,[7] CM-ClaSP,[8] as well as a post-processing phase for PrefixSpan to keep only closed patterns.[9]

Finally, we only generated and mined patterns on abstract summaries. Intuitively, abstract summaries get rid of unnecessary information such as variable names or even whitespace, allowing for more meaningful patterns to be mined. However, it might be that concrete summaries actually result in better-quality patterns. This hypothesis should be properly tested.

References

[1] E. Iannone, R. Guadagni, F. Ferrucci, A. De Lucia, and F. Palomba, "The Secret Life of Software Vulnerabilities: A Large-Scale Empirical Study," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 44–63, Jan. 2023, doi: 10.1109/TSE.2022.3140868.

[2] M. Jiang, J. Jiang, T. Wu, Z. Ma, X. Luo, and X. Zhou, "Understanding Vulnerability Inducing Commits of the Linux Kernel," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 7, Art. no. 170, Sep. 2024, doi: 10.1145/3672452.

[3] M. Maes-Bermejo, A. Serebrenik, M. Gallego, F. Gortázar, G. Robles, and J. M. González Barahona, "Hunting bugs: Towards an automated approach to identifying which change caused a bug through regression testing," *Empirical Software Engineering*, vol. 29, Art. no. 66, May 2024, doi: 10.1007/s10664-024-10479-z.

[4] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, May 2005, doi: 10.1145/1082983.1083147.

[5] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code," *Software: Practice and Experience*, vol. 46, pp. 1155–1179, Sep. 2025, doi: 10.1002/spe.2346.

[6] Q. Le Dilavrec, D. E. Khelladi, A. Blouin, and J.-M. Jézéquel, "Hyper-AST: Enabling Efficient Analysis of Software Histories at Scale," in *ASE 2022: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, Association for Computing Machinery, Oct. 2022, pp. 1–12. doi: 10.1145/3551349.3560423.

[7] H. Hata, O. Mizuno, and T. Kikuno, "Historage: fine-grained version control system for Java," in *IWPSE-EVOL '11: Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, Association for Computing Machinery, Sep. 2011, pp. 96–100. doi: 10.1145/2024445.2024463.

[8] Y. Higo, S. Hayashi, and S. Kusumoto, "On tracking Java methods with Git mechanisms," *Journal of Systems and Software*, vol. 165, Jul. 2020, doi: 10.1016/j.jss.2020.110571.

[9] P. Fournier-Viger, J. C.-W. Lin, R. U. Kiran, and Y. S. Koh, "A survey of sequential pattern mining," *Data Science and Pattern Recognition*, vol. 1, no. 1, pp. 54–77, 2017, [Online]. Available: https://www.philippe-fournier-viger.com/dspr/dspr-paper5.pdf

[10] P. Fournier-Viger *et al.*, "The SPMF Open-Source Data Mining Library Version 2," in *Proc. 19th European Conference on Principles of Data Mining and Knowledge Discovery (PKDD 2016) Part III*, Springer, Sep. 2016, pp. 36–40. doi: 10.1007/978-3-319-46131-1_8.

[11] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: a database of existing faults to enable controlled testing studies for Java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, Association for Computing Machinery, Jul. 2014, pp. 437–440. doi: 10.1145/2610384.2628055.

[12] A. G. Koru and J. Tian, "An empirical comparison and characterization of high defect and high complexity modules," *Journal of Systems and Software*, vol. 67, no. 3, pp. 153–163, Sep. 2003, doi: 10.1016/S0164-1212(02)00126-7.

[13] M. Lopes and A. Hora, "How and why we end up with complex methods: a multi-language study," *Empirical Software Engineering*, vol. 27, Art. no. 115, May 2022, doi: 10.1007/s10664-022-10144-3.

[14] M. Brunsfeld *et al.*, *tree-sitter/tree-sitter: An incremental parsing system for programming tools*. doi: 10.5281/zenodo.4619183.

[15] Q. Le Dilavrec, D. E. Khelladi, A. Blouin, and J.-M. Jézéquel, "HyperDiff: Computing Source Code Diffs at Scale," in *ESEC/FSE 2023: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Association for Computing Machinery, Nov. 2023, pp. 288–299. doi: 10.1145/3611643.361631.

[16] M. Janke and P. Mäder, "FS^{3}_{change}: A Scalable Method for Change Pattern Mining", *IEEE Transactions on Software Engineering*, vol. 49, no. 6, pp. 3616–3629, Apr. 2023, doi: 10.1109/TSE.2023.3269500.

[17] M. Janke and P. Mäder, "Graph Based Mining of Code Change Patterns From Version Control Commits," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 848–863, Mar. 2022, doi: 10.1109/TSE.2020.3004892.

[18] F. Grund, S. A. Chowdhury, N. C. Bradley, B. Hall, and R. Holmes, "CodeShovel: Constructing Method-Level Source Code Histories," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, IEEE, May 2021, pp. 1510–1522. doi: 10.1109/ICSE43902.2021.00135.

[7] https://www.philippe-fournier-viger.com/spmf/CLOFAST.php

[8] https://www.philippe-fournier-viger.com/spmf/CM-ClaSP.php

[9] https://www.philippe-fournier-viger.com/spmf/SPAM_PrefixSpan_Closed.php