# A Study of Static Warning Cascading Tools (Experience Paper)

Xiuyuan Guo
Iowa State University
xiuyuang@iastate.edu

Ashwin Kallingal Joshy
Iowa State University
ashwinkj@iastate.edu

Benjamin Steenhoek
Iowa State University
benjis@iastate.edu

Wei Le
Iowa State University
weile@iastate.edu,

Lori Flynn
Carnegie Mellon University
lflynn@cert.org

## ABSTRACT

Static analysis is widely used for software assurance. However, static analysis tools can report an overwhelming number of warnings, many of which are false positives. Applying static analysis to a new version, a large number of warnings can be only relevant to the old version. Inspecting these warnings is a waste of time and can prevent developers from finding the new bugs in the new version. In this paper, we report the challenges of *cascading warnings* generated from two versions of programs. We investigated program differencing tools and extend them to perform warning cascading automatically. Specifically, we used textual based diff tool, namely *SCALe*, abstract syntax tree (AST) based diff tool, namely *GumTree*, and control flow graph (CFG) based diff tool, namely *Hydrogen*. We reported our experience of applying these tools and hopefully our findings can provide developers understandings of pros and cons of each approach. In our evaluation, we used 96 pairs of benchmark programs for which we know ground-truth bugs and fixes as well as 12 pairs of real-world open-source projects. Our tools and data are available at https://github.com/WarningCas/WarningCascading_Data.

## 1 INTRODUCTION

In an agile software development setting, there is a need to deliver reliable new software releases in a rapid fashion. The big challenge is how we can only analyze and report the software quality issues related to the new version, as the issues of old versions have been addressed previously when shipping the old versions. In particular, static analysis tools, as an important software assurance technique, often generate an overwhelming number of warnings for each version of software [1, 2]. It is confusing which warnings are related to only the old code and have been reviewed in the previous versions, which warnings report new issues in the updated release, and which warnings are about the issues of fixing the old warnings. Consequently, tremendous time and manual efforts can be wasted and not spent on the right problems of the current version.

The goal of *static warning cascading* (also called *matching* or *aligning* static warnings) is to help developers classify warnings into several categories: (1) the cascaded warnings report a same issue in the old and new versions (so we don't need to handle it), (2) the warnings in the old version are fixed in the new version (we can inspect them together to confirm if the fix is indeed correct), (3) the warnings are changed from the old version but the old and new warnings are related (we should inspect them together to understand the problem), and (4) the warnings only report issues in the new version (we should inspect them in the new version of software). For most of static analysis tools, a warning is reported as a line in the source code file. Thus the warning cascading problem can be reduced to map a source code line from the old version to the new version and classify the mapping to be one of the above categories.

There exist a spectrum of program differencing tools [3–5] that can match source code lines. Some representative categories include *textual based diff*, *syntax based diff* and *control flow based diff*. Textual based diff is typically performed on two source code files using the *longest common sequence* algorithm like the one implemented in the *Unix diff* tool. Syntactic based diff tools like GumTree [4] are performed on the *abstract syntax trees (ASTs)*. It compares ASTs from two versions of a source code file and determines if the AST nodes in the two versions should be matched. The control flow based diff uses a representation of *multiple version interprocedural control flow graphs (MVICFG)* [3]. The MVICFG is a union of *Interprocedural Control Flow Graphs (ICFGs)* for a set of program versions. The common nodes and edges in versions are represented only once and each edge is marked with the versions it belongs to.

In this paper, we conducted a study of the three representative program differencing techniques for cascading static warnings. Our goal is to evaluate the pros and cons of each tool and report which tools are the most *useful* and *successful*

for cascading warnings. By useful and successful, we mean when a bug in a program is fixed in the new version, the tool is able to report the warning of the bug does not cascade to the one in the fixed version; when a bug in a program still exists after adding changes, the tool is able to report that the warning in the buggy version is cascaded to the new version. They are the same or related warnings.

Specifically, we used an existing texual based warning cascading tool *SCALe* [1] developed and used at CERT [2]; we also designed and implemented warning cascading routines on top of other two open source tools, GumTree and Hydrogen. We applied the three tools for static analysis warnings generated for two program versions. To compare different tools' behavior when apply warning cascading, we first constructed studies on the 96 pairs of benchmark programs where the ground truth are known. We then collected 12 pairs of real-world open source projects and investigate the use of the three tools in practice.

Our results show that Hydrogen has a slight advantage compared to other two tools for the ground truth benchmarks. When used for real-world programs such as `find`, `grep`, `make` and `coreutils`, Hydrogen is more successful for cascading same "bugs" across versions compared to two other tools, where SCALe shows more advantages in cascading the cases where the warnings for the first version are fixed in the second versions. We sampled a set of our results and reported the analysis on these examples (see §4 for details). Our experience and findings can provide developers knowledge on warning cascading as well as a more general problem of programming differencing.

In summary, this paper made the following contributions:

(1) We reported practical challenges of cascading warnings across program versions and proposed what is considered as a successful warning cascading (§2);
(2) We used and extended three types of programming differencing tools to perform warning cascading (§3);
(3) We designed and performed comprehensive empirical studies to compare the three types of approaches to discover whether, when and why each type of the tools work best for warning alignment (§4); and
(4) We open source our tools and datasets at https://github.com/WarningCas/WarningCascading_Data.

## 2  MOTIVATION AND CHALLENGES

Warning cascading is challenging because when programs are updated in the new versions, the function and variable names may change, and the line numbers of the same statements are also likely changed. Directly performing string matching for the output from static analysis tools cannot work because of the change of context in the newer version. In this section, we provide some examples to explain the challenges of warning cascading, and we also more precisely define what it means by a useful and successful cascading.

### 2.1  Challenging examples

`find.71f10368` has a bug of "crashing in some locales for find -printf '%AX'" and a newer version of `find` added a fix for this bug and also included many additional new changes. If we run static analysis tool, we will get 1600+ warnings for each version. Without a proper warning cascading tool, we cannot easily find which warnings in the previous version are changed in the new version, and determine whether the fix is successful and whether there are more new issues introduced in the fix and other newly added code.

Warning cascading is challenging for several aspects. First, there are many identical warnings between the two versions; however, the same warnings across versions may be reported as different locations of the same files due to the new code added or old code deleted, and there can be changes of variable/function names, e.g., via refactoring, which do not affect the warning semantics. Second, there are often dead code in the project, e.g., `corebench` [3] have `gnulib-tests` folders within the projects which did not affect program behaviors. But static analysis scans all the code to output the warnings. Developers have to filter out those warnings irrelevant to the newly developed code. Such dead code can be project specific and hard to exclude, and thus increase the overhead of warning cascading.

Here, we further show some real-world examples discovered in our study. In the first case, when many new lines are added before the target line, the text diff tools cannot match the warnings. See Figure 1. In the second case, a line added in the new version (green at line 5 in Figure 2) is the same as the target line (blue at line 7 in Figure 2). The diff tools can be confused and mistakenly match newly added line 5 with `i++` in the old version instead of line 7. In the third case, there are non-semantic changes, e.g., changing function name between the two versions or adding a new comment to the target line. As an example, in Figure 3, `fprintf` is changed to `checked_fprintf`, and the text diff tools cannot match them. In Figure 4, a statement at line 7 didn't change at all in the second version but an extra comment is added. These cases can challenge the warning cascading tools.

### 2.2  What is a useful and successful warning cascading?

In the problem of warning cascading, the tool takes static warnings generated from one version and determines if there is a *match* for the warnings in another version. We consider a warning is successfully cascaded for the following two cases. First, the cascading tool reports the two warnings as *same "bugs"* if both versions contain the same "bug" located at the target line (we say the "bugs" in two versions are the same if the sequences of root cause statements along the paths are semantically equivalent). In this case, the warnings have been reviewed in the old version, and developers do not need to further investigate these warnings. Here, "bug" is not confirmed but is the output warning from static analysis

```
static char **                                         1
construct_command_argv_internal                        2
(char *line, char **restp, char *shell,                3
char *shellflags, char *ifs, int flags,                4
char **batch_filename_ptr)                             5
{                                                      6
    ...                                                7
                                                       8
                                                       9
    + if(one_shell)                                   10
    + {                                               11
    + #if defined __MSDOS__ || defined (__EMX__)      12
    + if (unixy_shell)                                13
    + #else                                           14
    + if (is_bourne_compatible_shell(shell))          15
    + #endif                                           16
    ... // more lines are added here                  17
                                                      18
    ...                                               19
    command_ptr = ap;                                 20
}                                                     21
```

**Figure 1: Challenge: many lines (lines 9–17) are added before the target line (line 25)**

```
static bool                                            1
record_exec_dir (struct exec_val *execp)               2
{                                                      3
    ...                                                4
    + i++ // newly added line in version2 could        5
        possibly match two version1
    function1()                                        6
    i++ // shared line between version1 and            7
        version2
    ...                                                8
}                                                      9
```

**Figure 2: Challenge: newly add a line the same as the target line**

```
static reg_errcode_t
internal_function
re_string_reconstruct (re_string_t *pstr, Idx idx, int eflags)
{
    ...
    case KIND_FORMAT:
        switch (segment->format_char[0])
    case 'a':
        - fprintf (fp,segment->text , ctime_format
            (get_stat_atime(stat_buf)));
        + checked_fprintf (fp,segment->text , ctime_format
            (get_stat_atime(stat_buf)));
        break;
    case 'b':
        - fprintf(fp,segment->txt, human_readable ((unintmax_t)
            ST_NBLOCKS (*stat_buf)));
        + checked_fprintf(fp,segment->txt, human_readable
            ((unintmax_t) ST_NBLOCKS (*stat_buf)));
        break;
    ...
}
```

**Figure 3: Challenge: new name is applied during refactoring**

```
static bool                                            1
record_exec_dir (struct exec_val *execp)               2
{                                                      3
                                                       4
    ...                                                5
    i++;                                               6
    + i++; //extra comment                             7
    ...                                                8
}                                                      9
```

**Figure 4: Challenge: add a comment**

tools. "bug" can be false positives, and we can match them if the changes newly added do not affect the semantics of the warnings.

A variant of the first case is that the root cause statements of the "bug" are not exactly the same but have some changes—-for a successful cascading, the tool should report the two warnings as *relevant "bugs"*. So developers can inspect the two warnings together for diagnosis.

In the second case, one version contains the "bug" at the target line and the other version added a fix for the "bug". There is no longer warning reported for this "bug" in the second version. Here, a successful cascading should report *"bug" fix*. This case includes a special situation, where the buggy code is deleted in the second version. The warning cascading in this case is useful especially when the second version aims to fix the bugs in the first version. Warning cascading is able to help determine if the issues in the first version likely are addressed in the new versions, and what are the new issues added in the new version.

If the cascading tools fail to match the same "bugs" (in the first case) or match any "bug" in one version with irrelevant "bugs" in another version (in both first and second cases), we consider such cascading as unsuccessful. In our evaluation, we used benchmarks that are known with ground truth bugs and fixes to evaluate such metrics. For the real-world benchmarks where there is no ground truth, we performed manual inspection to determine if the warning cascading is successful (details see §4).

## 3 THREE TECHNIQUES OF CASCADING WARNING

In this paper, we used three different types of program differencing tools, namely *textual based diff*, *AST based diff* and *CFG based diff*, for warning cascading. Specifically, *SCALe* is a tool developed by CERT and used `Unix diff` for cascading warnings. *GumTree* is a syntactic differencing tool based on abstract syntax trees (ASTs). *Hydrogen* compares programs based on control flow graphs integrated in MVICFG. We extended GumTree and Hydrogen for warning cascading.

We compare the output of these tools to understand the pros and cons of these techniques. We hope our findings can help developers better select warning cascading tools and more efficiently improve their code quality in the continuous integration. In the following, we provide some technical details of the three tools.

## 3.1 Textual based diff tool: SCALe

SCALe [5] takes warnings reported by multiple static analysis tools and first group the warnings of the same issue (reported by different static analysis tools) into one warning. It then applies the textual *diff* tool [4] to maps lines between versions and assess whether the source code line associated with a warning has undergone modification in the updated version.

To apply SCALe, we deployed docker-based virtual machine [5]. For each program analyzed, the static analysis tools were run, and the output was formatted in a way that SCALe could process and understand. These output files were uploaded into SCALe via their web-based interface. The warnings of the second version of the program are added in the same way. To compare the different versions of the program, we leveraged the browser interface's built-in functions and employed the *diff* tool for cascading the differences. This was executed internally by the browser's backend, which allowed for the comparison of the program and performed warning cascading on the program versions. After that, we can find the output of it on the GUI page, and the information contains the *verdict* value of each warning.

If the warning's verdict value is `true`, that means there is no adjudication on the previous version for the line corresponding to the warning, and there is matched warning in the first version. If such a value is false, that means there is a change in the previous version. The warning in the second version is not matched. We review any warnings present in the first version but not matched as *same "bug"* in the second version and label them as *"bug" fix* cascading. We implemented a script to automate the process of generating warnings from multiple static analysis tools, uploading the results to SCALe, and cascading warnings between two versions of a program.

## 3.2 AST based diff tool: GumTree

GumTree [4] is a syntactic differencing tool that operates based on the Abstract Syntax Tree (AST). Unlike SCALe, which uses the diff tool to compare file versions on a text line level, GumTree parses each file version into an AST representation and directly matches the nodes in the two AST versions. By utilizing the AST representation, GumTree is able to bypass the influence of minor changes that may surround warnings, such as changes in spacing or refactoring of variable names, which are unlikely to affect the warnings. When the warnings in the two versions correspond to the aligned nodes identified by GumTree, we consider the warning is cascaded to the new version.

To perform syntactic warning cascading, we built a custom client interface for the code release of GumTree[6]. GumTree's objective is to compute an *edit script*, a sequence of edit actions made to a source file, which is short and close to the developer's intent. It follows a 2-step process. Step 1 computes mappings between similar nodes in the two ASTs;

the main contribution of GumTree is to maximize the number of mapped nodes. Step 2 deduces the edit script from the AST mapping using the algorithm of Chawathe et al [6]. We only used the AST mapping and did not compute the edit script, since our application only needs to match the AST nodes between two versions of a program.

Given two versions of a source code file $P_1$ and $P_2$, the GumTree parser produces their respective ASTs $T_1$ and $T_2$. Then, GumTree computes the mapping $M_T$ between the similar nodes in $T_1$ and $T_2$. Finally, our client checks the mapping $M_T$ to determine the set of warnings to cascade.

Algorithm 1 defines our cascading algorithm built on top of the GumTree. Since all warnings are placed on concrete lines in the code, we traverse only the leaf nodes of the AST. A warning is cascaded between nodes $t_1 \in T_1, t_2 \in T_2$ if and only if there is at least one warning on the same line as both $t_1$ and $t_2$ (line 3), $t_1$ is mapped to $t_2$ by GumTree (line 4), and the warnings attached to $t_1$ and $t_2$ have the same CWE (Common Weaknesses Enumeration) condition (line 5).

Similar to SCALe's implementation of diff cascading, our implementation uses the results of running GumTree to cascade the warnings without modifying the GumTree AST parser and differencing algorithm. Our implementation preserves the stable and efficient implementation and adds only the little overhead which is necessary for cascading warnings.

---

**Algorithm 1:** Cascade Warnings with GumTree

> **Input** : AST for 2 versions $T_1, T_2$, node mapping $M_T = \{t_1, t_2\}$
>
> **Output:** $M_W = \{w_1, w_2\}$ matched from $T_1$ to $T_2$

1 $M_W \leftarrow \emptyset$
2 **for** $w_1 \in W_1$ **do**
3    **for** $t_1 \in leaves T_1 | t_1.line = w_1.line$ **do**
4      **if** $t_1, t_2 \in M_T$ **then**
5        $M_W = \{w_2 \in W_2 | t_2.line = w_2.line \wedge w_1.condition = w_2.condition\}$
6 **return** $M_W$

---

## 3.3 CFG based diff tool: MVICFG

Hydrogen [3] is a tool that differentiates programs with regards to its control flow. It first generates the ICFG based on IR produced by LLVM [7]. It then combines ICFGs of each version into one via a graph union. The nodes and edges are shared across multiple versions and are marked with versions they belong to. In the end, it builds a program representation for multiple versions of ICFGs, called MVICFG, which shows different control flows and paths between two different program versions.

To perform warning cascading, we developed an extension to Hydrogen's original algorithm. This extended algorithm utilizes various graph traversals to detect the cascading of warnings, shown in Algorithms 2 and 3.

Algorithm 2 takes as input the two program versions ($V_1$ & $V_2$) and their respective collection of static warnings ($SW_1$ & $SW_2$). The algorithm outputs ($W_m$ & $W_u$) as a matched

---

[4]https://www.gnu.org/software/diffutils
[5]https://github.com/cmu-sei/SCALe/tree/scaife-scale
[6]https://github.com/GumTreeDiff/gumtree

warning (cascading bugs) and unmatched warning (cascading fixes) respectively.

We generate MVICFG of the two versions of the program at line 3 of Algorithm 2. Then we embed the warnings from both versions of $SW_1$ and $SW_2$ into the MVICFG at line 4, based on its location in the code, including the file path, file name, function name, and line number. After embedding, each warning contains meta-data that specifies the version from which it originates, the type and message associated with the warning, and its node in MVICFG. At lines 5 and 6, we iterate through all the warnings from $SW_1$. For each warning, we obtain its corresponding MVICFG node based on the metadata provided by the warning, and we then see if node $n$ is a common node shared across two versions [8]. If so, the warnings at these locations have the possibility of being cascaded. If the node is a common node and it also contains the warning from the second version, we add it into $W_m$. Otherwise, if the node is not a common node, we put it into *CheckBetween* (discussed later) function to further identify whether it can be cascaded.

---

**Algorithm 2:** Cascade warnings

**Input**   : Program versions $[V_1, V_2]$,
            Resp. warning sets $[SW_1, SW_2]$
**Output**: Matched warnings $[W_m]$,
            Exclusive warnings $[W_u]$

1  Initialize $W_m, W_u$, MVICFG;
2  **Function** CascadeWarning($V_1, V_2, SW_1, SW_2$)
3      MVICFG $\leftarrow$ GenMVICFG($V_1, V_2$);
4      EmbedInMVICFG($SW_1, SW_2$);
5      **while** $SW_1 \neq \emptyset$ **do**
6          Remove a warning w from $SW_1$;
7          n $\leftarrow$ GetNodeFromWarningData(MVICFG, w);
8          **if** n.*IsSharedNode* **and** n.*HasWarning_2* **then**
9              Add w to $W_m$;

---

Since there is a possibility of the line being marked as modified due to changes surrounding it, Unix diff tool will report the line as changed. MVICFG used Unix diff tool and thus the changed lines will be represented in two different nodes. But we can recover such weakness of Unix diff tools by further checking if the (buggy) paths lead to this line in two versions are actually the same. If so, we can category the warnings as matched. See Algorithm 3 *CheckBetween*.

In Algorithm 3, at lines 2 and 3, we traverse MVICFG to get the *divergent/convergent* nodes nearest to $n$. A *divergent* node of $n$ on the MVICFG is defined as a nearest *matched* node (matched across two versions) found by traversal of the predecessor edges from $n$. A *convergent* node of $n$ is a nearest matched node found by traversal of a successor edge of $n$. We provide an example to further clarify the two definitions. Figure 9 showed a snippet of MVICFG. Node $n_1$ is a matched node shared between two versions. From this node, there are two edges which are called *version branches* in the MVICFG. Nodes $n$ and $n_2$ on the left version-branch belong to version

---

**Algorithm 3:** Categorize matched warnings

**Input**   : Warning w at MVICFG node n
**Output**: Updated $W_m, W_u$

1  **Function** CheckBetween(w,n)
2      DivN $\leftarrow$ FirstDivNodeInMVICFG(n);
3      ConvN $\leftarrow$ FirstConvNodeInMVICFG(n);
4      Stmt $\leftarrow$ Statement at n;
5      MN $\leftarrow$ StmtInMVICFG(Stmt,DivN,ConvN,$V_2$);
6      **if** MN $\neq \emptyset$ **and** MN.*HasWarning_2* **then**
7          Add w to $W_m$;
8      **else**
9          Add w to $W_u$

---

1. Node $n$ on the right version-branch belongs version 2. Here, node 1 is the divergent node for $n$ and $n2$, and $n_3$ is the convergent node for $n$ and $n2$. The two nodes $n$ along the two version branches have the same statements. We will use algorithm 3 to mark those two as matched by leveraging the use of divergent and convergent nodes.
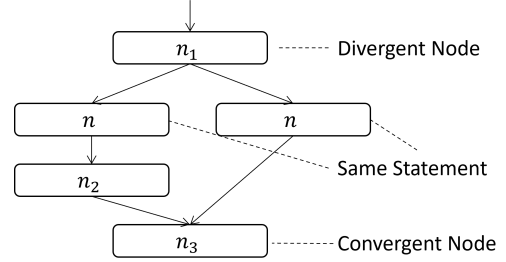


**Figure 5: Divergent and Convergent Nodes on the MVICFG**

Specifically, at line 2 in Algorithm 3, `FirstDivNodeInMVICFG` performs a breadth-first search backwards from n and returns the divergent node with the shortest path to n. Similarly, at line 3, `FirstConvNodeInMVICFG` performs a breadth-first search forwards from n and returns the convergent node with the shortest path from n. Because n is a modified node, there will be at least one divergent node in its ancestors and at least one convergent node in its successors. At line 10, we extract the statement as a string and trim whitespace characters. Then, at line 11, `StmtInMVICFG` searches all nodes between $DivN$ and $ConvN$ for a node in $V_2$ whose text exactly matches $Stmt$ after trimming whitespace. If such a node exists and it contains a warning for the second version, we consider it as matched with n and categorize the warning as cascaded $w_m$; otherwise, we add it into $W_u$

## 4  EVALUATION

In the evaluation, we plan to investigate:

(1) Which approaches perform the best for static warning cascading?
(2) When and why each approach does not perform well?

## 4.1 Experimental setup

*Experiments.* We designed two experiments, namely a *ground-truth setting* and a *real-world setting*. In the ground-truth setting, we collected a set of buggy programs, where we know the location of the bugs in each program. Each buggy program has two variants: a *buggy-buggy variant* consisting of the original buggy program and a version in which a refactoring irrelevant to the bug is introduced, and a *buggy-fix* variant, consisting of the original buggy program and a version in which the bug is fixed. We selected the buggy programs such that at least one of our static analysis tools can correctly report the warnings for the known bugs. That way, we can count and analyze how these warnings are cascaded in its variants, and compare the cascading results with the ground truth.

In the real-world setting, we collected a set of programs consisting of real-world bugs and their fixed versions from open source projects. We then observed the cascading of static warnings from the first version to the second version. This setting helps us understand the usefulness and challenges of cascading approaches in the real-world application settings.

*Software subject selection.* To fulfill the two experiment settings, we used C programs from two benchmarks: SARD [9] and CoREBench [10]. From the SARD dataset, we used ABM and Toyota as the ground truth setting. The two benchmarks consist of 96 pairs of synthetic programs, where static analysis tools are able to report warnings for the buggy version.

CoREBench consists of a total 12 pairs of real-world projects, including `make, find, grep,` and `coreutils`. These are open-source programs with a long contribution history of over 33k commits. Each program is documented with real-world bug reports and their corresponding fix introducing commit. The program represents a wide variety of project sizes, ranging in size from 9.4k LOC (grep) to 83.1k LOC (coreutils).

*Static analysis tools.* To generate the static warnings for cacading, we used five different tools: GCC [11], Clang [12], Cppcheck [13], Rosecheckers [14] and CodeSonar[15]. These tools are currently supported by SCALe and frequently used in CERT [16] for scanning vulnerabilities. We used SCALe first to aggregate the warnings generated from different static analysis tools into one warning. We then cascade the warning using the three tools.

*Metrics and confirmation of the results.* : For the benchmarks where we know the ground-truth bugs, we used the warnings reported at the buggy lines in the first version as subjects and determined the successfulness of warning cascading based on our criteria given in Section 2.2.

For the real-world programs, we sampled 12% of total warnings from one version and manually inspected if the warnings are cascaded successfully. For each pair of programs, we report (1) the warnings of the two versions are matched as "bugs", and (2) the warnings in the first version are removed in the second versions. We then compared the results from three tools and also performed manual inspection to evaluate

whether such two types of match are performed correctly by the three tools. For example, in case (1), a mistake is reported if the two warnings are not the same but paired incorrectly by a tool, or the two warnings are supposed to be matched, but one tool fails to do so; in case (2), the warning is supposed to be removed in the second version; however, it is matched with some random warning incorrectly.

All the manual inspection is done by two code reviewers. The code reviewers first inspect the cascaded warnings by themselves and then compared and discussed the results with another code reviewer so that we report confident results.

*Running experiments.* We ran all of our experiments on RedHat 20.4 Linux distribution on a virtual machine with 32 GB of memory and 32 cores available. We implemented our tools using LLVM-8.0, Python 3 and Bash scripts.

## 4.2 Results for RQ1

**Table 1: Successful cascading for buggy-buggy versions**

| Benchmark | Total | Hydrogen | SCALe | GumTree |
|-----------|-------|----------|-------|---------|
| ABM | 20 | 20 | 19 | 20 |
| Toyota | 37 | 36 | 35 | 36 |
| Both | 57 | 56 | 54 | 56 |

**Table 2: Successful cascading for buggy-fixed versions**

| Benchmark | Total | Hydrogen | SCALe | GumTree |
|-----------|-------|----------|-------|---------|
| ABM | 13 | 13 | 3 | 10 |
| Toyota | 26 | 26 | 5 | 23 |
| Both | 39 | 39 | 8 | 33 |

*4.2.1 Results for the ground-truth setting.* Tables 1 and 2 show the results of each cascading tool for the benchmarks of ABM and Toyota. We leverage the ground truth bugs and patches in the benchmarks to confirm the successfulness of our warning cascading. Table 1 lists a total of 57 (20 from ABM and 37 from Toyota) pairs of programs in the two benchmarks. Each pair of programs contains two buggy versions, where each version contains one bug, and the second buggy version is a refactored version of the first buggy version. The two versions contain the same bug. In Table 2, under *Total*, we show that there are 39 (13 from ABM and 26 from Toyota) pairs of programs. Each pair of programs contains a buggy version and a fixed version. The buggy version contains one bug. The fixed version is the patched version for this bug. For both the cases, we focused on the static warnings generated for the buggy lines for this study (the fault locations are provided by the benchmarks) and determine if this warning is successfully cascaded.

In Tables 1 and 2, each cell in columns *Hydrogen*, *SCALe*, and *GumTree* reported the number of targeted warnings that were successfully cascaded by the tool. Specifically, following in the criteria in Section 2.2, for Table 1, we say a tool made

a successful cascading if each warning of the buggy version in the first version is aligned with the warning of the same bug in the second version. For Table 2, we say a tool made a successful cascading if each warning of the buggy line (first versions) did not find any match on a fixed version (second version). Here, the bug is fixed, and there is no warning for the same bug in the second version. Thus the warning in the first version should not match any other random warnings in the second version.

As shown in Table 1, out of the 57 buggy-buggy program pairs, `Hydrogen` and `GumTree` cascaded 56 paired programs successfully, followed by `SCALe` at 54. For buggy-fixed pairs, as shown in Table 2, out of the total 39 pairs of programs, `Hydrogen` was able to correctly cascade all 39 of them, followed by `GumTree` at 33 and `SCALe` at 8. The results based on the known ground truths show that `Hydrogen` outperformed the other baselines by successfully cascading 95 out of the total 96 pairs, followed by `GumTree` of 89 successful pairs and `SCALe` with 62 successful pairs. SCALe performed very poorly for cascading warnings for buggy-fixed pairs.

*4.2.2　Results for the real-world setting.* In this section, we compared the output generated from the three tools by cascading the warnings from real-world benchmarks and displayed the results using the Venn diagrams. We ran static analysis tools to generate the warnings for 12 pairs of programs. After obtaining the warnings, we did a preprocessing step before providing the warnings to the cascading tools, which included: 1) removing all the irrelevant warnings that have no effect on the execution of the programs, e.g., files from testing folders, obsolete library code, 2) aggregating warnings from different static analysis tools and removing all the duplicate warnings that are reported by different static analysis tools. After the preprocessing step, the warnings (of the first version) were reduced from 19305 to 2113. These are the warnings we used for cascading.

In Figures 6 and 7, the blue, red, and green circles represent GumTree, Hydrogen, and SCALe results respectively. Figure 6 is a Venn diagram to show how many warnings are cascaded to the same "bugs" between the two versions. Figure 7 shows how many of warnings are cascaded as the *"bug" fix* between the two versions.

The numbers located in the intersections of circles represent the shared cascading results across multiple tools. For example, 1101 on the center of Figure 6 represents warnings that have the same cascading across the three tools. 1254 at the intersection of the blue and red circles represents warnings that have the same cascading between Hydrogen and GumTree, including the ones shared by the three tools. The number located in the circle alone (not in the intersection area) represents the number of warnings cascaded by that tool, including the ones shared with other tools. For example, 1301 in the center of the red circle in Figure 6 means the total number of warnings cascaded by Hydrogen. Venn diagrams show that the three tools share a large part of warning cascading results, which brings in confidence that these cascading results are correct
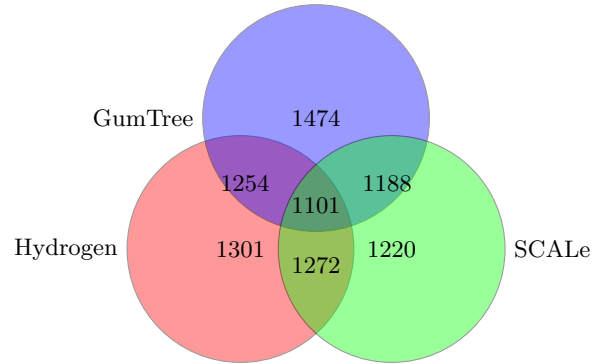


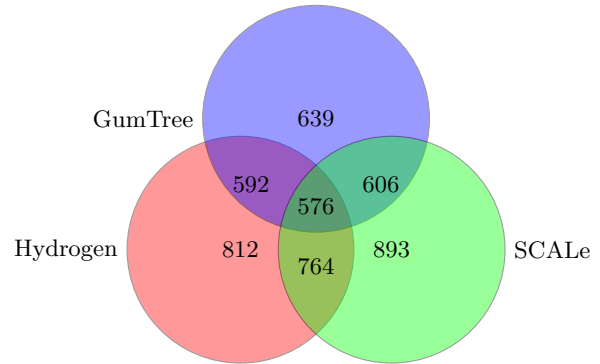**Figure 6: Number of warnings cascaded as *same "bugs"* (defined in Section 2.2)**



**Figure 7: Number of warnings cascaded as *"bug" fixes* (defined in Section 2.2)**

We randomly selected 12% of total warnings from the results, covering each category of overlapped sets. The results of manual inspection are shown in Tables 3 and 4. The numbers presented in Tables 3 and 4 represent warnings of the first version that has been successfully cascaded. For example, in Table 3 Row `find` and Column *Total*, we show that we inspected a total of 33 warnings from the first version that are cascaded as same "bugs". Hydrogen successfully cascaded all the warnings, SCALe successfully cascaded 31 and GumTree matched 32. As shown in Table 3, out of the total 132 warnings that we inspected, `Hydrogen` was able to successfully cascade all 132 of them, followed by `GumTree` and `SCALe` with the same result at 99.

The results of "bug" fixes type of cascading are shown in Table 4. Out of the 120 total warnings, `SCALe` cascaded 112 warnings correctly, followed by `GumTree` at 91 and `Hydrogen` at 85. It indicated that compared with other tools, Hydrogen has higher accuracy when detecting cascading bugs, and SCALe has more accuracy when determining if the "bug" is fixed. GumTree performed as a middle ground where have a better result than Hydrogen on cascading related to fixes and a better results than SCALe on cascading bugs.

Xiuyuan Guo, Ashwin Kallingal Joshy, Benjamin Steenhoek, Wei Le, and Lori Flynn

**Table 3: Successful cascading of same "bugs" in real-world program pairs**

| Benchmark | Total | Hydrogen | SCALe | GumTree |
|---|---|---|---|---|
| find | 33 | 33 | 31 | 32 |
| grep | 22 | 22 | 2 | 20 |
| make | 25 | 25 | 25 | 20 |
| coreutil | 52 | 52 | 41 | 27 |
| Total | 132 | 132 | 99 | 99 |

**Table 4: Successful cascading of "bug" fixes in real-world program pairs**

| Benchmark | Total | Hydrogen | SCALe | GumTree |
|---|---|---|---|---|
| find | 21 | 20 | 24 | 1 |
| grep | 24 | 24 | 21 | 20 |
| make | 10 | 10 | 10 | 10 |
| coreutil | 65 | 31 | 57 | 60 |
| Total | 120 | 85 | 112 | 91 |

## 4.3 Results for RQ2

To answer RQ2, we further analyzed and grouped the warning cascading results from the three tools into the following categories:

(1) GumTree failed to cascade
(2) SCALe failed to cascade
(3) SCALe and GumTree failed to cascade
(4) Hydrogen failed to cascade

*4.3.1 GumTree failed to cascade.* GumTree can fail for two reasons. The first reason is that GumTree cannot process macros in the programs. In the presence of macros, the ASTs sometimes are parsed incorrectly and do not match the source code. This prevents GumTree from matching the warnings correctly. The second reason is that GumTree used a heuristic algorithm to map the AST nodes based on their syntax, regardless of their semantic meaning. This approach can lead to incorrect mapping of the AST nodes across versions, causing warning cascading to fail.

Figure 8 shows an example that only GumTree made the wrong cascading among the three tools. In this example, a section of code is surrounded by a conditional compilation using the macro `_AMIGA`. The presence of such a macro, caused the AST to parse the information in the incorrect way. In the AST diff, this region of code is considered as deleted in the second version, which fails to match the rest of the AST nodes in the two versions. SCALe and Hydrogen both can handle such a case and made a correct cascading.

GumTree makes mistakes also because of its syntactic diff algorithm. In Figure **??**, we showed diffs of two functions: `wrong_001` (abbreviated) and `wrong_014`. GumTree AST diff algorithm incorrectly matched the first version of `wrong_001` to the second version of `wrong_014` (an irrelevant function), instead of the second version of `wrong_001`. The blue arrow shows the nodes which GumTree considers to be matched

```
static char **
construct_command_argv_internal
(char *line, char **restp, char *shell,
char *shellflags, char *ifs, int flags,
char **batch_filename_ptr)
{
    ...
    #ifdef _AMIGA
+   if(one_shell)
+   {
+   #if defined __MSDOS__ || defined (__EMX__)
+   if (unixy_shell)
+   #else
+   if (is_bourne_compatible_shell(shell))
+   #endif
    ...

    ap = new_line;
    memcpy (ap, shell, shell_len);
    ap += shell_len;
    *(ap++) = '␣';
    memcpy (ap, shellflags, sflags_len);
    ap += sflags_len;
    *(ap++) = '␣';
    command_ptr = ap;
    ...
    #endif _AMIGA
}
```

**Figure 8: GumTree failed due to macro**

between the two versions. The nodes `a`, `fptr`, and `arr` in function `wrong_001` were mapped to identify nodes in the irrelevant function `wrong_014`, but they should have been mapped to the version 2 of `wrong_001`. This incorrect mapping caused the warning on line 8 to be incorrectly cascaded to line 26 (version 2 of `wrong_014`) instead of line 10 (version 2 of `wrong_001`).

*4.3.2 SCALe failed to cascade.* SCALe can generate two types of errors when performing warning cascading. First, two snippets of code may match textually but a change in referenced elements, e.g., a change in the called function, can cause different execution behaviors. SCALe can incorrectly match the warnings. Second, some textual differences have no impact on program behaviors related to the warnings, but SCALe would falsely report the warnings cannot match. In the following, we further provide two examples to demonstrate the weakness of SCALE.

Figure 10 shows the diff between versions `401d8194` (shown in red) and `54d55bba` (green) of `kwset.c` in `Grep`. The static analysis tool reports a warning at line 22 (shown in blue) for both the versions as *-1 is coerced from int to unsigned long*. SCALe, however, reports the lines 7-22 as being modified instead of just the lines 7-18. Due to the line containing the warning (line 22) being misclassified as modified, SCALe fails to cascade this warning. However, this warning should have been matched because (1) line 22 was not modified between the versions and (2) adding the keywords "register" at lines 13–19 should not change the semantics related to this type of bug.

```
int wrong_arguments_func_pointer_001()
{
    int arr[5] = {1,2,3,4,5} ;
    - int (*fptr)(int *);
    + int (*fptr)(int);
    int a;
    - fptr = (int (*)(int
        *))wrong_arguments_func_pointer_001_func_001;
    - a =fptr(arr);
    + fptr = wrong_arguments_func_pointer_001_func_001;
    + a =fptr(arr[0]);
}

int wrong_arguments_func_pointer_014()
{
    - float f;
    - f=0.7
    + long arr[5];
    if(wrong_arguments_func_pointer_014_func_001(0) == 0)
    {
        - long (*fptr)(float *);
        + long (*fptr)(long [],int);
        long a;
        - fptr = (long (*)(float *))wron...
        - a =fptr(&f);
        + fptr = wrong_arguments_func_pointer_014_func_002;
        + a = fptr(arr,5)
    }
}
```

**Figure 9: GumTree failed to cascade `Toyota` warnings because AST diff algorithm cannot align ASTs of the two versions correctly**

```
static size_t                                          1
cwexec (kwset_t kws, char const *text,                 2
size_t len, struct kwsmatch *kwsmatch)                 3
{                                                      4
    struct kwset const *kwset;                          5
        ...                                             6
    - unsigned char c;                                  7
    - unsigned char const *delta;                       8
    - int d;                                            9
    - char const *end, *qlim;                          10
    - struct tree const *tree;                         11
    - char const *trans;                               12
    + register unsigned char c;                        13
    + register unsigned char const *delta;             14
    + register int d;                                  15
    + register char const *end, *qlim;                 16
    + register struct tree const *tree;                17
    + register char const *trans;                      18
    ...                                                19
    kwset = (struct kwset *) kws;                       20
    if (len < kwset->mind)                             21
        return -1;                                     22
    ...                                                23
}                                                      24
```

**Figure 10: SCALe failed to cascade `grep` warnings**

Figure 11 shows an example that only SCALe made the wrong match cascading among the three tools. Similar to the example shown in Figure 10, this program has a newly added

```
static void
bytes_split (uintmax_t n_bytes, char *buf,
size_t bufsize)
{
    size_t n_read;
    bool new_file_flag = true;
    size_t to_read;
    uintmax_t to_write = n_bytes;
    char *bp_out;
    + uintmax_t opened = 0;
    ...
}
```

**Figure 11: SCALe fails to casade**

```
+ # MACRO_BEGIN
char
human_readable (...
- uintmax_t from_block_size, uintmax_t to_block_size)
    + uintmax_t from_block_size, uintmax_t to_block_size)
{
    ...
    unsigned int base = human_base_1024 ? 1024 : 100;
    ...
}
+ # MACRO_END
```

**Figure 12: GumTree and SCALe fail to cascade**

line below the target line that we aim to cascade. Due to this change, the Unix diff reported that the target line has changed in the new version, which caused SCALe to fail to match warnings where AST and CFG-based diff tools can cascade successfully.

*4.3.3 SCALe and GumTree failed to cascade.* Figure 12 illustrates a scenario in which both GumTree and SCALe made incorrect warning cascading. In the case of GumTree, the failure is due to a large macro that surrounds the warning location on the second version. This macro makes it difficult for GumTree to parse the code into an AST, which results in a failure to match the warnings. In the case of SCALe, the failure is caused by a difference in the text located directly above the warning statement. This difference affects the results of the Unix diff tool but does not change the semantics of code. On the other hand, Hydrogen will not be affected by such changes because 1) it successfully parses the code within macro and built it into the MVICFG; 2) it uses control flow graphs to perform diff. It can confirm that this change does not affect program control flow and semantics, and thus will mark the blue statement as the matched warnings in MVICFG (See algorithm 2).

The main reason Hydrogen fails in some cases is that it used LLVM to compile the program, and some code that cannot be handled by LLVM is excluded from MVICFG. For example, `if` statements that do not have brackets, a statement expanding across multiple lines (it covers 46.8% undetected cascading for Hydrogen), the internal function

used for library and some conditional compilation code is not covered by this build.

Hydrogen used textual diff tool to build MVICFG and sometimes has the disadvantage similar to SCALe. In Section 2.1, Figure 3 shows a snippet of code where function `fprintf` is refactored to `checked_fprintf`. In this example, the text has changed between two versions of a related statement. However, the functionality still remains the same. Thus the warnings should be cascaded as the same "bug". Hydrogen fails to cascade this warning correctly because the target line has been marked as modified (unmatched node in MVICFG). If the textual statement wasn't changed, Hydrogen could possibly make a correct detection by using algorithm 3. However, since the function names in the statements have been changed, Hydrogen is not able to handle it. GumTree is able to make a correct cascading by leveraging the AST structures.

## 5 THREATS TO VALIDITY

To address the external threats to validity, we applied 12 pairs of C real-world projects as well as 96 pairs of benchmark programs where we know ground truth. We applied a set of static analysis tools often used by CERT to make sure we generated all types of practical static warnings. The benchmarks also had varying commits between versions to ensure heterogeneous diffs.

To address the internal threat to validity, we first inspected the output of three tools to make sure the implementations of the warning cascading algorithms are correct. We inspected 100% of cascaded warnings from the ground truth and 12% for real-word programs across by two code reviewers to confirm our findings.

## 6 RELATED WORK

There have been many works that focus on matching and prioritizing warnings or faults between multiple versions of a program [17–25] . [17–19, 24] uses GNU diff, AST, and Verification Modulo Versions to provide matching, while [21, 22] use source control revisions to prioritize static warnings. To the best of our knowledge, there has been no study on tools of warning cascading.

Spacco et al. [20] developed two methods to match warnings in the static analysis tool FindBugs at the line granularity level. The first approach, 'pairing', matches warnings based on their source code location. First, it identifies exact matches of package, class, and method name. Then, for those warnings that do not match exactly, the approach uses progressively 'fuzzier' criteria. The second approach is called 'warning signatures'. This approach transforms each warning into a string format that includes information about the warning, and then matches string-formatted warnings with the same MD5 code. Both the 'pairing' and 'warning signature' methods perform best when using a single static analyzer tool and use textual diff to identity places to do the cascading.

Logozzo et al. [18] present a solution to the common problem of verifying software with multiple versions. To tackle this challenge, the authors introduce a novel verification framework called Verification Modulo Versions (VMV), which is specifically designed to enhance the efficiency and effectiveness of software verification for multi-version systems. While this work does verification and relies on their own framework, our work tackles cascading warnings generated by off-the static analyzers and compares the usefulness and efficiency of different cascading methods independently of the specific analyzer.

Palix et al. [19] build an AST based on code changes to improve tracking changes similar to GumTree[4]. However, their work focuses on tracking changes between multiple versions, while our work focuses on studying how different change-tracking methodologies affect warning cascading.

Finally, there are many methods to track changes [4, 26–30] which can be roughly separated into textual, syntactic, and semantic methods. None of them directly deals with the problem of matching warnings between versions, but some of them are used in other works about matching warnings. Here, we discuss the state of the art for each category.

Syntactic methods such as GumTree [4] work at the granularity of ASTs, which reflects the source code structure and hence can be more precise than textual diff. AST helps in avoiding common pitfalls of textual based diff like missing refactoring based changes and spacing issues. Yang et al. [26] developed a syntactic-based comparing method for dynamic programming languages like scheme. Similar to GumTree, Fluri et al. [28] also proposed an AST based approach using a tree-differencing algorithm to detect source code changes. We choose GumTree because it is recent, open-source, and has been widely used by other works. Huang et al. [31] present an approach called ClDiff to linked code differences with the aim of simplifing code review. While these works aim to improve tracking changes between versions, our work focuses on studying the impact of using different tracking methods for cascading warnings.

## 7 CONCLUSIONS AND FUTURE WORK

Cascading static warnings is a practical but challenging problem. This paper applied three tools to explore their pros and cons of addressing this problem. We found that SCALe, the textual diff based tool, fails when there are textual changes but not semantics changes related to the bugs. It also fails when the referred calls or global variables have changed outside the current functions. GumTree has the weakness of not being able to handle macros, and the AST tree matching algorithm faces some failures due to its heuristic nature. Hydrogen relied on LLVM and cannot process all the code in the repositories due to the requirement of building the project. It used textual diff tool to build MVICFG and sometimes has the disadvantage similar to SCALe. In the future, we plan to integrate more static analysis tools like `CodeSonar`. Such tools produce paths as static warnings, and we envision that CFG based diffs can have greater advantages.

# REFERENCES

[1] N. Imtiaz, A. Rahman, E. Farhana, and L. Williams, "Challenges with Responding to Static Analysis Tool Alerts," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, May 2019, pp. 245–249, iSSN: 2574-3864.

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

[3] W. Le and S. D. Pattison, "Patch verification via multiversion interprocedural control flow graphs," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 1047–1058. [Online]. Available: https://doi.org/10.1145/2568225.2568304

[4] J.-r. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and Accurate Source Code Differencing To cite this version : Fine-grained and Accurate Source Code Differencing," *Ase*, 2014.

[5] R. Seacord, W. Dormann, J. McCurley, P. Miller, R. W. Stoddard, D. Svoboda, and J. Welch, "Source Code Analysis Laboratory (SCALe)," 4 2012. [Online]. Available: https://kilthub.cmu.edu/articles/report/Source_Code_Analysis_Laboratory_SCALe_/6584273

[6] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," *SIGMOD Rec.*, vol. 25, no. 2, p. 493–504, jun 1996. [Online]. Available: https://doi.org/10.1145/235968.233366

[7] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California, Mar 2004, 2004. [Online]. Available: https://llvm.org/pubs/2004-01-30-CGO-LLVM.html

[8] W. Le and S. D. Pattison, "Patch verification via multiversion interprocedural control flow graphs," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, p. 1047–1058. [Online]. Available: http://dl.acm.org/citation.cfm?id=2568304

[9] "Welcome to the nist software assurance reference dataset project." [Online]. Available: https://samate.nist.gov/SARD/index.php

[10] M. Böhme and A. Roychoudhury, "Corebench: Studying complexity of regression errors," in *Proceedings of the 23rd ACM/SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA, 2014, pp. 105–115.

[11] Free Software Foundation, "GNU compiler collection (GCC)." [Online]. Available: https://gcc.gnu.org/

[12] LLVM Project, "Clang: a c language family frontend for llvm." [Online]. Available: https://clang.llvm.org/

[13] C. Team, "Cppcheck - a tool for static c/c++ code analysis." [Online]. Available: http://cppcheck.sourceforge.net/

[14] R. C. Team, "Rose: Compiler infrastructure to build custom source code analyzers." [Online]. Available: https://rosecompiler.llnl.gov/

[15] I. GrammaTech, "Codesonar: Advanced static analysis for c/c++ and java." [Online]. Available: https://www.grammatech.com/products/codesonar

[16] C. M. U. Software Engineering Institute, "Cert division." [Online]. Available: https://www.sei.cmu.edu/about/divisions/cert/index.cfm

[17] "Populating a Release History Database from Version Control and Bug Tracking Systems," *IEEE International Conference on Software Maintenance, ICSM*, pp. 23–32, 2003.

[18] F. Logozzo, S. K. Lahiri, M. Fähndrich, and S. Blackshear, "Verification Modulo Versions: Towards usable verification," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 294–304, 2014.

[19] N. Palix, J. R. Falleri, and J. Lawall, "Improving pattern tracking with a language-aware tree differencing algorithm," *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings*, pp. 43–52, 2015.

[20] J. Spacco, D. Hovemeyer, and W. Pugh, "Tracking defect warnings across versions," *Proceedings - International Conference on Software Engineering*, pp. 133–136, 2006.

[21] S. Kim and M. D. Ernst, "Which warnings should i fix first?" *6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2007*, pp. 45–54, 2007.

[22] "Prioritizing warning categories by analyzing software history," *Proceedings - ICSE 2007 Workshops: Fourth International Workshop on Mining Software Repositories, MSR 2007*, pp. 0–3, 2007.

[23] C. Boogerd and L. Moonen, "Evaluating the relation between coding standard violations and faults within and across software versions," *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories, MSR 2009*, pp. 41–50, 2009.

[24] B. Chimdyalwar and S. Kumar, "Effective false positive filtering for evolving software," *Proceedings of the 4th India Software Engineering Conference 2011, ISEC'11*, pp. 103–106, 2011.

[25] P. Avgustinov, A. I. Baars, A. S. Henriksen, G. Lavender, G. Menzel, O. De Moor, M. Schäfer, and J. Tibble, "Tracking static analysis violations over time to capture developer characteristics," *Proceedings - International Conference on Software Engineering*, vol. 1, pp. 437–447, 2015.

[26] W. Yang, "Identifying Syntactic Differences between Two Programs," *Softw. Pract. Exper.*, vol. 21, no. 7, pp. 739–755, jun 1991. [Online]. Available: https://doi.org/10.1002/spe.4380210706

[27] M. Pawlik and N. Augsten, "RTED: A robust algorithm for the tree edit distance," *Proceedings of the VLDB Endowment*, vol. 5, no. 4, pp. 334–345, 2011.

[28] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[29] Y. Higo, A. Ohtani, and S. Kusumoto, "Generating simpler ast edit scripts by considering copy-and-paste," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 532–542.

[30] V. Frick, "Understanding software changes: Extracting, classifying, and presenting fine-grained source code changes," *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 226–229, 2020.

[31] K. Huang, B. Chen, X. Peng, D. Zhou, Y. Wang, Y. Liu, and W. Zhao, "Cldiff: Generating concise linked code differences," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018, pp. 679–690.