Validating Static Warnings via Testing Code Fragments

Ashwin Kallingal Joshy
Iowa State University
Ames, Iowa, USA
ashwinkj@iastate.edu

Xueyuan Chen
Iowa State University
Ames, Iowa, USA
xueyuan@iastate.edu

Benjamin Steenhoek
Iowa State University
Ames, Iowa, USA
benjis@iastate.edu

Wei Le
Iowa State University
Ames, Iowa, USA
weile@iastate.edu

ABSTRACT

Static analysis is an important approach for finding bugs and vulnerabilities in software. However, inspecting and confirming static warnings are challenging and time-consuming. In this paper, we present a novel solution that automatically generates test cases based on static warnings to validate true and false positives. We designed a syntactic patching algorithm that can generate syntactically valid, semantic preserving executable code fragments from static warnings. We developed a build and testing system to automatically test code fragments using fuzzers, KLEE and Valgrind. We evaluated our techniques using 12 real-world C projects and 1955 warnings from two commercial static analysis tools. We successfully built 68.5% code fragments and generated 1003 test cases. Through automatic testing, we identified 48 true positives and 27 false positives, and 205 likely false positives. We matched 4 CVE and real-world bugs using *Helium*, and they are only triggered by our tool but not other baseline tools. We found that testing code fragments is scalable and useful; it can trigger bugs that testing entire programs or testing procedures failed to trigger.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; *Software verification and validation*; *Software defect analysis.*

KEYWORDS

Code Fragments, Syntactic Patching, Testing Static Warnings

ACM Reference Format:
Ashwin Kallingal Joshy, Xueyuan Chen, Benjamin Steenhoek, and Wei Le. 2021. Validating Static Warnings via Testing Code Fragments. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3460319.3464832

1 INTRODUCTION

Static analysis is an important approach to find bugs and vulnerabilities. Path-sensitive static analysis tools [3, 21, 26, 31, 55, 57, 63, 64] are especially useful because it is more precise and reports path information to help diagnose the bugs. Microsoft uses ESP [21], Prefix [15] and ESPx [31] to detect resource leaks and buffer overflows. The companies like Grammatech, Synopsys and MathWorks sell their path-sensitive static analysis tools to more than thousands of customers yearly and find bugs in real-world software daily. Although useful, static analysis tools are expensive to use. There can be a large number of warnings generated, and inspecting these messages to confirm true positives and false positives is not a trivial task.

Due to the importance, there have been persistent efforts to help process static warnings. Flynn et al. developed a classification model that integrated multiple static analysis tools to identify likely true positives [23]. Zhang et al. developed an interactive approach to learn from the users' feedback to prioritize static warnings [76]. There is also a joint effort from Google and the academia that developed a *logistic regression* analysis to predict accurate and actionable static warnings generated from Google software [59].

In this paper, we provide a complementary approach by converting the paths reported from static analysis tools to executable test cases. We further leverage advanced testing techniques such as fuzzers and symbolic executors to automatically test the warnings. By demonstrating the symptoms through testing, static warnings can be validated. Different from previous research [16, 19] that combines static and dynamic approaches for bugs, we propose *testing code fragments*; that is, we aim to generate test cases that are as small as possible but can still encapsulate the reported buggy paths. We believe that when a test case is small and contains a targeted bug, automatic testing tools can be more scalable and effective to trigger the bug. We can also avoid the potentially complicated setups needed for testing integrated software. Previous research [16] has mentioned that testing smaller programs can reduce the time of testing and correction of error by developers.

We identified a set of challenges to achieve this goal. First, the code fragments (warning paths, e.g., see Figure 1a) reported by static warnings typically are not compilable. We need to patch the code fragments to fix the syntax errors. When adding the patch to generate a test case, we should not alter the paths reported in the warnings. Second, we need to isolate the dependencies from the project required to build the code fragments. Third, to enable automation, we need to provide proper test harnesses and inputs that can trigger the symptoms aligned with the warning types.

ISSTA '21, July 11–17, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8459-9/21/07.
https://doi.org/10.1145/3460319.3464832

Additionally, we need to construct test oracles that can match the symptoms reported from testing and static warnings.

We developed the *Helium* framework to address these challenges. It consists of a *Lowest Common Ancestor (LCA)-based syntactic patching* algorithm and a system for building and testing code fragments. We formally define what *syntactic patches* we should generate and prove that following this definition, the warning paths in the original program will be preserved in the test case we generated. Different from the existing techniques of auto-fixing compiler errors [5, 28–30, 33], our approach analyzes the parse trees to preserve the semantics of code fragments and also make sure the patch is as small as possible.

To resolve dependencies, we analyzed the code fragment and its surrounding environment in the original project to determine the header files, the definitions of functions, types and variables, compiler flags, and the libraries needed to compile and link the code fragments.

For testing, we used randomly generated test inputs as well as the automatically generated test inputs through fuzzers like *Radamsa* [1] and the symbolic executors like *KLEE* [2]. Based on testing results, we identify *valid* tests to confirm true positives and false positives. The valid tests match the failures specified in our test oracles. We constructed test oracles using static warnings, including failure locations, e.g., the file names and line numbers, as well as failure symptoms linked to the warning types, e.g., buffer overflows, null-pointer dereferences, and memory leaks. We used *Valgrind* [3] and also added *assert* statements to capture such failures during testing.

We implemented *Helium* and evaluated our techniques using 12 C projects such as `cvs`, `httpd`, `findutils`, `grep`, `make` and `coreutils`. We collected 1955 static warnings from two popular commercial static analysis tools, *PolySpace* from MathWorks and the *Commercial Tool* (anonymized based on our license). We used Radamsa, KLEE and Valgrind as our automatic testing tools. *Helium* compiled 1340 code fragments and generated 1003 test cases. We identified 48 true positives and 27 false positives, and categorized 205 as likely false positives. Our techniques significantly outperformed the baselines, including unit testing, *RLAssist* [29], *MACER* [18], *BovInpsector* [25] and the integrated testing using existing test suites. We matched 4 true positives to CVE and real-world bugs that the other tools did not find.

In this paper, we make the following contributions:

(1) defining *syntactic patching* for code fragments (§3),
(2) designing an algorithm that can generate syntactically valid, semantic-preserving, as small as possible code fragments (§4),
(3) developing a system that automatically builds and tests code fragments (§5); and
(4) building a tool, *Helium*, using which, we demonstrated that our techniques can effectively validate static warnings from real-world code and static analysis tools (§6).

Our paper website is https://sites.google.com/view/helium-2021.

[1] https://gitlab.com/akihe/radamsa
[2] https://klee.github.io/
[3] https://valgrind.org/

2 OVERVIEW

We motivate our work using a real-world example. We then explain at a high level how *Helium* works to help developers.

2.1 A Motivating Example

In Figure 1a, we present a static warning reported by the *Commercial Tool* for `squid-2.3`. The warning includes a list of statements that lead to a buffer overflow at line 1025 in `ftp.c`. These statements are located in three different functions and files. To confirm the warning, the developer need to inspect the path and understand what happens along the path. The task is difficult and time consuming, considering there are hundreds of warnings reported for `squid-2.3` by the *Commercial Tool*.

Using *Helium* , we generate an executable test case shown in Figure 1c. Specifically, *Helium* adds the red lines (lines 11, 14, 19, 21, 23 and 36) to make the code syntactically valid. Note that the code may be parsable by adding just "`}`"s; however, without adding line 11 (this loophead exists in the original program but the *Commercial Tool* did not include it in the static warning), the test case cannot preserve the semantics of the original program, as `break` at line 14 would be paired with the loop at line 10 instead of the one at line 11.

In addition, *Helium* adds the yellow lines to resolve the dependencies needed to build the code fragment, including variables (line 26), type (lines 5–8) and function (line 4) dependencies as well as the header files (lines 1–3). *Helium* also generates the test harness code at lines 37-41 (shown in green). Running the test input shown in Figure 1b, *Helium* triggered the buffer overflow (similar to CVE 2002–0068). We thus can confirm that this warning is a true positive.

2.2 How Helium Works to Help the Developers

Figure 2 (in page 4) presents an overview of our approach. *Helium* takes static warnings as input. It first applies our *LCA-based syntactic patching* algorithm to generate a parsable, semantic-preserving code fragment from the warning. It then resolves the dependencies and builds the code fragment to generate an executable test case. Finally, *Helium* adds assertions and tests code fragments with fuzzers, KLEE and Valgrind. Comparing the test results with our test oracles, it validates whether a warning is a true positive or a false positive. If many tests exercise the paths but do not trigger any failures, we report the warnings as likely false positives. Based on this information, developers then can prioritize which warnings to fix first, and use our test cases to help diagnose the warnings.

When manually inspecting a static warning, the developer aims to understand whether there indeed exists a bug along the paths reported and to predict what consequences this bug may cause. *Helium* automates this manual process by running tests through the paths. When reporting a buggy path, static analysis tools had concluded that the path sufficiently implied the error condition, and some (or all) user inputs can trigger the bug. Our testing finds such user inputs, confirms that the error condition is indeed correctly computed by static analysis, and demonstrates its failure symptoms.

3 DEFINING SYNTACTIC PATCHING

By *syntactic patching*, we mean patching a code fragment and making it syntactically valid. The goal of our syntactic patching is to generate a small, syntactically valid, and semantic preserving

```
file:lib/rfc1738.c
89: char *rfc1738_do_escape(char *url, int
  encode_reserved) {
102: for (p = url, q = buf; *p != 0; p++) {
103:   do_escape = 1;
107:   if (*p == rfc1738_reserved_chars[i]) {
108:     do_escape = 1;
109:     break;
139:   if (do_escape == 1) {
140:     (void) sprintf(q, "%%%02x", (unsigned char) *p);
141:     q += sizeof(char) * 2;
142:   } else {
143:     *q = *p;
146:  *q = '\0';
147:  return (buf);
```

```
file:lib/rfc1738.c
175: char *rfc1738_escape_part(char *url) {
177:  return rfc1738_do_escape(url, 1);
```

```
file:src/ftp.c
999:void ftpBuildTitleUrl(FtpStateData *ftpState) {
1004:  len = 64 + strlen(ftpState->user)
1005:    + strlen(ftpState->password)
1006:    + strlen(request->host)
1007:    + strLen(request->urlpath);
1008:  t = ftpState->base_href = xcalloc(len, 1);
1020:  strcat(t, "ftp://");
1021:  if (strcmp(ftpState->user, "anonymous")) {
1022:    strcat(t, rfc1738_escape_part(ftpState->user));
1023:    if (strcmp(ftpState->user, "anonymous")) {
1024:      strcat(t, ftpState->user);
1025:      strcat(t, "@");
```

(a) static warning for `squid-2.3`

```
./squid \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \
   \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \
  \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \  password@host.com
```

(b) testing code fragments using the input provided by `squid-2.3`

```
#include <stdlib.h>
#include <squid.h>
...
void *xcalloc(int n, size_t sz);
typedef struct _Ftpdata {
 char user[MAX_URL];
 char password[MAX_URL]; ...
} FtpStateData;
char *rfc1738_do_escape(char *url, int encode_reserved) {
  for (p = url, q = buf; *p != 0; p++) {
    for (i = 0; i < sizeof((rfc1738_reserved_chars)) &&
  encode_reserved > 0; i++) {
      if (*p == rfc1738_reserved_chars[i]) {
        do_escape = 1;
        break; }
      if (do_escape == 1) {
        (void) sprintf(q, "%%%02x", (unsigned char) *p);
        q += sizeof(char) * 2;
      } else {
        *q = *p; }  }
  *q = '\0';
  return (buf); }
char *rfc1738_escape_part(char *url) {
  return rfc1738_do_escape(url, 1); }
void ftpBuildTitleUrl(FtpStateData *ftpState) {
  request_t *request = ftpState->request;
  size_t len; char *t;
  len = 64 + strlen(ftpState->user)
    + strlen(ftpState->password) + strlen(request->host)
    + strLen(request->urlpath);
  t = ftpState->base_href = xcalloc(len, 1);
  strcat(t, "ftp://");
  if (strcmp(ftpState->user, "anonymous")) {
    strcat(t, rfc1738_escape_part(ftpState->user));
    if (strcmp(ftpState->user, "anonymous")) {
      strcat(t, ftpState->user);
      strcat(t, "@"); } } }
int main(int argc, char *argv[]){
  FtpStateData *ftpState=xcalloc(1,sizeof(FtpStateData));
  ftpState->user = argv[1];
  ftpState->password = argv[2]; ...
  ftpBuildTitleUrl(ftpState); }
```

(c) test case generated by ***Helium***

Figure 1 – the buffer overflow example in `squid-2.3` **relevant to CVE 2002–0068**

code fragment. In the following, we provide the details on what this goal means.

3.1 Code Fragments

In this paper, we use *code fragments* and *code segments* interchangeably. We use *tokens* to define code fragments. Precisely, the *token* here means its *lexeme* [6], e.g., the variable "pi" is a lexeme, and the constant "3.1415" is also a lexeme. Each token has a unique identification given by its location in the program.

Definition 1. A *program*, p, is a finite sequence of tokens, $p = t_0 t_1 \dots t_n$. We use the notation $|p|$ to indicate the length of the program in terms of the number of tokens. We say p_1 is *smaller* than p_2 iff $|p_1| < |p_2|$.

Definition 2. Given programs $p = t_0 t_1 \dots t_n$ and $q = t'_0 t'_1 \dots t'_m$, we say q is a *code fragment* of p iff $t'_0 t'_1 \dots t'_m$ is a subsequence [6] of $t_0 t_1 \dots t_n$, denoted by $q \sqsubseteq p$.

3.2 LCA-Based Syntactic Patching

We present three different syntactic patching approaches and show that this problem is challenging: *token-based syntactic patching* may change semantics and *tree-based* can generate a large patched program. Our novel solution is called *LCA-based syntactic patching*.

Definition 3. In *token-based syntactic patching*, given a context-free grammar G, a program p, and a code fragment s, $s \sqsubseteq p$, we say s' is a patched program of s regarding p, iff

(1) $s' \sqsubseteq p$;
(2) $s \sqsubseteq s'$;
(3) s' is recognized by G; and

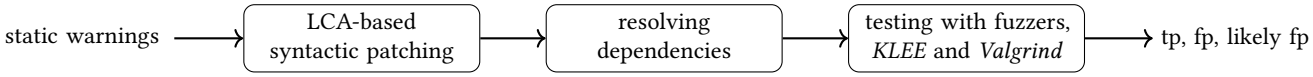


Figure 2 – The Helium Workflow

(4) s' is the smallest program that satisfies the conditions (1)–(3).

This definition is problematic, as demonstrated in Figure 3: when we select `foo` and `b` on the left, or select `while`, `c2` and `s1` on the right, we generate a patched program (see the bottom) that has a different syntax tree from the original program, which can lead to different semantics.

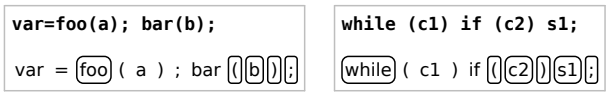


Figure 3 – the top line is the original code, and the boxed nodes at the bottom represent the patched code under Definition 3.

Definition 4. In *tree-based syntactic patching*, given a context-free grammar G, a program p, and a code fragment s, $s \sqsubseteq p$, we say s' is a patched program of s regarding p, iff

(1) $s' \sqsubseteq p$,
(2) $s \sqsubseteq s'$,
(3) s' is recognized by G,
(4) $\exists t$, where t is a common subtree of $parse(s', G)$ and $parse(p, G)$ such that $s \sqsubseteq dft(t)$, and
(5) s' is the smallest program that satisfies the conditions (1)–(4).

Here, *parse(p,G)* returns a parse tree from a given program p and a grammar G. *dft(t)* (depth first traversal) produces a program by sequencing the leaf tokens during traversing the parse tree, t, in a depth-first order [6].

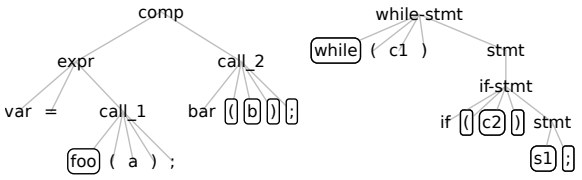


Figure 4 – Addressing the problems in Figure 3

Definition 4 addresses the problem of Definition 3, as shown in Figure 4. On the left, the smallest subtree that contains the selected tokens `foo` and `b` is the entire tree rooted by `comp`. The patched program thus is `var = foo(a); bar(b);` instead of `foo(b)`. Similarly, on the right, when we keep the smallest subtree of the selected nodes `while`, `c2` and `s1`, we include `while(c1)` in the patched program, and `c2` will not be mistakenly used as a condition for the `while` loop.

Definition 4 is still not ideal as it can lead to an unnecessarily large patched program. For example, in Figure 5, given the selected `s1` and `s3`, we would start from the root `comp1` and generate the patched program `s1; s2; s3; s4`. But without `s2` and `s4`, the program `s1; s3` is still compilable and retains the execution order for `s1` and `s3` as in the original program.

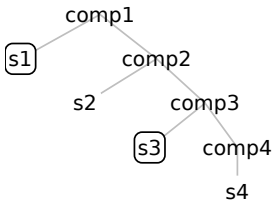


Figure 5 – Generating a large patched program

To address the problems of Definitions 3 and 4, we propose *LCA-based syntactic patching*. This approach aims to keep the syntactic structure, thus semantics, of code fragments during patching while maintaining a smallest parse tree. The idea is to only copy the important non-terminal nodes that can maintain the syntactic relations of any two selected tokens in the parse tree. We identified that such "bridge nodes" are the *Lowest Common Ancestors (LCAs)* [62]. The LCA of x and y is the common ancestor of x and y that is not the ancestor of any other common ancestors of x and y.

Definition 5. An *LCA relation* of two nodes x and y on a parse tree pt, denoted by $R_{lca}(x, y, pt)$, is a 4-tuple (L, r, i_x, i_y), where

(1) L is the LCA of x and y, a non-terminal node on the parse tree pt,
(2) r is the right hand side (RHS) of the production rule, where L is the left hand side (LHS),
(3) i_x is the position of the x's ancestor in r, and
(4) i_y is the position of the y's ancestor in r.

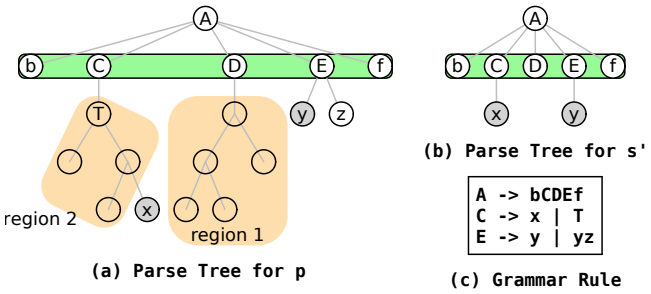


Figure 6 – An example of the LCA relation

In Figure 6 (a), A is the LCA of the two leaf nodes x and y; r is $bCDEf$, the RHS of A; the ancestors of x and y in the sequence $bCDEf$ are C and E respectively, and thus $i_x = 2$ and $i_y = 4$.

Therefore, in this example, $R_{lca}(x, y, pt) = (A, bCDEf, 2, 4)$. This relation captures the production rule used from the LCA node A as well as the nonterminals C and E that will eventually derive to the selected tokens x and y.

Definition 6. In *LCA-based syntactic patching*, given a context-free grammar G, a program p, and a code fragment s, $s \sqsubseteq p$, we say s' is a patched program of s regarding p, iff

(1) $s' \sqsubseteq p$,
(2) $s \sqsubseteq s'$,
(3) s' is recognized by G,
(4) $\forall x, y \in s$,
$R_{lca}(x, y, parse(p, G)) = R_{lca}(x, y, parse(s', G))$, and
(5) s' is the smallest program that satisfies the conditions (1)–(4).

Condition (4) says that the LCA relation of any two tokens should remain unchanged when generating s' from p. Unlike Definition 4, which keeps the entire subtree rooted from the common ancestor of tokens, here we only copy the "first level" of the subtree, including the LCA and its production rule, for generating a small s'. In Figure 6 (b), the parse tree of the patched program s' keeps A, the LCA of x and y, and its production rule $A \rightarrow bCDEf$. When generating patches, instead of using the original product rules for C and E, we use $C \rightarrow x$ and $E \rightarrow y$ to keep the patch small.

3.3 Preserving Semantics

When performing syntactic patching, we need to *preserve semantics*, which means that the paths in static warnings should be retained even with the patch, and hence the bug in the warning can be reproduced. We developed two theorems and used the concept of *partial order* [8] to demonstrate that our LCA syntactic patching retains the partial order of any two selected tokens; as a result, the execution order of statements along the warning paths are the same in the patched program and in the original program. The proof of Theorem 1 considers *sequential*, *loop* and *branch*, three types of control flow and their nested cases. The proof of Theorem 2 used the Theorem 1 as well as the definitions of partial order and subsequence. Due to the space, we moved the proof sketches here [4].

Definition 7. *partial order* is defined on two nodes on the control flow graph. We say node x_1 is ordered before x_2 ($x_1 < x_2$) iff along some acyclic path, x_1 is a predecessor of x_2. If x_1 and x_2 do not appear together on any acyclic path, they do not have an order.

Theorem 1. If s' is a patched program of the code fragment s regarding the original program p via LCA-based syntactic patching, two statements n_1 and n_2 in s' have the same partial order in s and p.

Theorem 2. If two statements n_1 and n_2 in s' have the same partial order in p, any path in s' is a subsequence of some path in p, i.e., $\forall$ path t in s', $\exists$ path ω in p, such that $t \sqsubseteq \omega$.

4 COMPUTING SYNTACTIC PATCHES

We developed an LCA-based syntactic patching algorithm, consisting of two steps: *preserving LCA relations* (Section 4.1) and *generating minimal patches* (Section 4.2).

[4] https://sites.google.com/view/helium-2021

4.1 Preserving LCA Relations

In Algorithm 1 at line 4, N stores the parse tree nodes whose LCAs need to be computed. Initially, it is set to s. Δs stores the patch in progress. Lines 5–10 present the key step of identifying LCA nodes for the selected tokens. At lines 5– 6, we use a worklist to traverse the nonterminal nodes from the bottom of the parse tree pt. For any such node l, we check if its descendants overlap with the nodes in N. If we found more than two of such nodes, denoted by the set C_{lca} at line 8, we add l to N (l is an LCA) and remove C_{lca} from N at line 10. As the algorithm progresses through the loop, the tokens in N are gradually replaced by their LCAs until the most top level LCA is reached, and the worklist at line 6 becomes empty.

At lines 11-15, we generate the patch based on the LCA relation discovered above. At line 12, when the children of the LCA node are terminals, we directly add them to the patch based on the parse tree, denoted by $\Delta s \leftarrow \Delta s \cup c$. If c is a nonterminal, we use GenMinPatch at line 15 to find a minimum patch that derives from c to $target$. Using the same GenMinPatch, at line 16 when $worklist$ is empty, we find a desired derivation from the start symbol, the parse tree root, to the top LCA (the last element processed by $worklist$) to generate the patched program recognizable by the grammar.

Algorithm 1 LCA-based Syntactic Patching

```
1:  INPUT: p (program), s (code fragment), G (grammar)
2:  OUTPUT: s' (patched program)

3:  pt ← Parse(p)
4:  N ← s, Δs ← ∅
5:  worklist ← SortByLevel(pt.nonterminals)
6:  while worklist ≠ ∅ do
7:      remove l from worklist
8:      C_lca = l.descendants ∩ N
9:      if |C_lca| ≥ 2 then
10:         N ← (N \ C_lca) ∪ l
11:         for all c ∈ l.children do
12:             if c is terminal then Δs ← Δs ∪ c
13:             else
14:                 target ← c.descendants ∩ C_lca
15:                 s' ← s' ∪ GenMinPatch(c, target)
16: s' = s' ∪ GenMinPatch(pt.root, N)
```

Example: In Figure 6 (a), when traversing the parse tree in a bottom up fashion, we determine that A is the LCA for the selected tokens x and y (at lines 8–9 in Algorithm 1, A is l, and x and y are in N). We add b and f to the patch since they are terminals (line 12). For the nonterminals C, D and E, we use GenMinPatch to find $C \rightarrow x$ and $E \rightarrow y$ that can generate a minimal patch (lines 14 and 15). If A is not the root, we also use GenMinPatch to generate a patch that derives the root to A (line 16).

4.2 Generating Minimal Patches

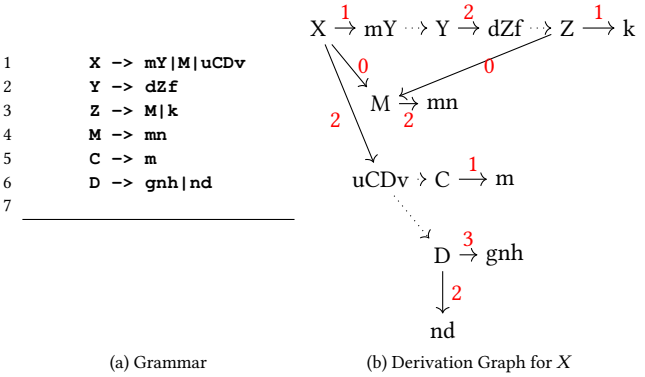
GenMinPatch in Algorithm 1 aims to find a *shortest derivation* from a nonterminal to a target, defined as follows:

Definition 8. Given a nonterminal X, a target y (it can be either terminal or nonterminal), and a string α which derives from X

and consists of y and terminals, a *shortest derivation* from X to y regarding α generates the string β, such that (1) $\beta \sqsubseteq \alpha$, (2) $y \in \beta$, (3) β can also be derived from X, and (4) β is the shortest string that satisfies the above conditions.

To compute the shortest derivation, we define *derivation graph* and use it to reduce this problem to finding the shortest paths on the derivation graph.

Definition 9. A *derivation graph* regarding a context-free grammar and its nonterminal X is a directed graph $G = (V, E)$, where the node represents either a *nonterminal* or the RHS of a production rule. Correspondingly, an edge is either (1) a *Type I* edge from a nonterminal to its RHS, and the edge is weighted with the number of terminals used in RHS; or (2) a *Type II* edge from RHS to a nonterminal used in the RHS, and the edge does not have a weight.



(a) Grammar (b) Derivation Graph for X

Figure 7 – An example of derivation graph

Example: In Figure 7, the solid edges from X are the Type I edges. The weights on the edges starting from X indicate that the RHSs of X used 1, 0 and 2 terminals respectively. Some example Type II edges (dashed) are from $uCDv$ to C, and from $uCDv$ to D.

In Algorithm 2, we show how to find a shortest derivation from X to the target y. The output $\mathrm{D}_{min}(X,y)$ records the *cost* of this derivation (the length of the string generated from the derivation). Once the derivation is found, we test if the generated string β is the subsequence of a given α (see Definition 8). If yes, we report the solution; if not, we continue finding the next shortest derivations until β is a subsequence of α.

At line 4, we compare two cases: in D_{min}-*I* at line 5, the target y is reached through only Type I edges and in D_{min}-*II* at line 14, y is reached through Type I and Type II edges. In the first case, at lines 7–8, we first find all the nodes that contain y. For each of such node u, we run a shortest path algorithm on the paths consisting of only Type I edges to find the minimal cost from X to u (line 9). For any Type II edges from u (line 10), we compute the minimal cost from v to the *final* node that only contains terminals (line 11). At line 12, we select the minimum for all such u.

In the second case, at line 16, we first find the minimal cost from X to u, where X and u are reachable through only Type I edges, u has outgoing Type II edge(s), and $y \notin u$. For any Type II edge (u, v),

Algorithm 2 Generating Minimal Patches

```
1:  Input: X (nonterminal), g (derivation graph of X), y (target)
2:  Output: D_min(X,y)

3:  procedure D_min(X,y)
4:      return Min(D_min-I(X,y), D_min-II(X,y))

5:  procedure D_min-I(X,y)
6:      C_Xy ← ∞
7:      for all u ∈ g.V do                          ▷ all nodes in g
8:          if y ∈ u then
9:              C_Xy ← ShortestPath(X,u)
10:             for all v where (u,v) ∈ g.E_2 do
11:                 C_Xy ← C_Xy + D_min(v, final)
12:     C_Xy_min ← select the minimal C_Xy for all u
13:     return C_Xy_min

14: procedure D_min-II(X,y)
15:     for all u reachable from X via Type I edges and has outgoing
        Type II edge(s), y ∉ u do
16:         C_Xu ← ShortestPath(X,u); C_vy ← ∞
17:         for all v where (u,v) ∈ g.E_2 do
18:             C_vy ← D_min(v,y); C_tf ← 0
19:             for all t where (u,t) ∈ g.E_2 ∧ t ≠ v do
20:                 C_tf ← C_tf + D_min(t, final)
21:             C_vy ← C_vy + C_tf
22:         C_vy_min ← select the minimal C_vy for all v
23:         C_Xy ← C_Xu + C_vy_min
24:     C_Xy_min ← select the minimal C_Xy for all u
25:     return C_Xy_min
```

we compute the minimum cost from v to the target y at line 18 and add the cost for the rest of the nodes connected from u at lines 19–21. At lines 22 and 24, we select the best v and u respectively.

Example: In Figure 7, suppose we want to compute a shortest derivation from X to m containing Type II edges. Along $X \rightarrow uCDv$, C_{Xu} at line 16 in Algorithm 2 equals 2. At line 17, the two choices of v are the nodes C and D. Along $C \rightarrow m$, C_{vy} at line 18 equals 1 and C_{tf} at line 20 equals 2 (along $D \rightarrow nd$). On the other hand, when v at line 17 is D, along $D \rightarrow gnh$ and $D \rightarrow nd$, C_{vy} at line 18 equals ∞, as D cannot reach m. Therefore, $C_{vy_{min}}$ at line 22 equals 3, and C_{Xy} at line 23 equals 5. The loop at line 15 only iterates once, and we conclude $C_{Xy_{min}} = 5$ at line 24.

5 BUILDING AND TESTING CODE FRAGMENTS

In this section, we present our approaches of resolving dependencies and testing code fragments.

5.1 Resolving Dependencies for Code Fragments

We developed two approaches to identify the dependencies needed to build code fragments. In the first approach, we process the project and store all its definitions into a database. When handling each

segment from the project, we retrieve the definitions on demand and patch them based on their orders in the original project. This approach does not require to build the project; as long as its syntax errors do not affect the segment, we still can compile the segment.

In the second approach, *Helium* finds the header files that contain the symbol definitions needed by the segment based on where the segment is located in the original project. *Helium* then includes these header files in the code fragment and link the segment with the object files of the original project.

To generate an executable, we also developed two approaches to prepare a build script for the code fragment. In the first approach, *Helium* uses the header files included in the code fragments to figure out which object files in the original project and which libraries should be linked to. In the second approach, we recorded all the compiler and linker flags used to build the original project via tools like *bear* [49] and the CMake's *export* command. We then create a build script based on these flags.

5.2 Testing with KLEE, Fuzzers and Valgrind

Creating test harnesses: *Helium* adds a `main` function, which serves as the top level function for the code fragment. It also inserts the code that can supply generated test data to the input variables. We used the approach in [37] and performed *def-use* analyses on the code fragments to identify the input variables: Variable v is an *input variable* if there is no definition of v found before a use of v in the code fragment.

Generating test inputs: If we can determine the value of a variable before the code fragment, we use this value to initialize the variable. Otherwise, we initialize input variables using automatically generated test inputs. We currently implemented random input generation that supports integers, floats, chars, arrays, pointers and the data structures composed with these types. We also applied Radamsa and KLEE to further generate test inputs. We selected Radamsa because it can use a random input as a seed to generate useful inputs, and it has successfully found many vulnerabilities in real-world software such as *chrome*, *firefox*, *php*, *tor*, *libxslt* and *vlc*.

Constructing test oracles: We identified two types of useful tests, *valid* and *pass* tests. A valid test case triggers the failure specified in our test oracles. A test oracle consists of the failure location and failure symptoms, extracted from the corresponding static analysis warning. During testing, we compare if the actual failure location is matched to the one specified in the oracle, similar to the approach in [38]. For failure symptoms, we developed a mapping from warning types to the failure types that Valgrind and KLEE can report. Some warning types cannot be directly linked to the symptoms reported by dynamic tools. We thus also add assertions to help with the match. For example, we add `assert(false)` at the dead code; when the assertion is triggered, we confirm it as a false positive. Valgrind and KLEE can both report assertion failures.

We analyzed the static warning types and classify them into two categories. Ideally, path-sensitive tools can report warning paths that contain all the statements of relevance to the bugs. In such cases, the *positive* warnings require only one valid test case to demonstrate the bug, e.g., buffer overflows and divide-by-zero. We confirm them as true positives. For such warnings, when testing never triggers the failures defined in the oracles, we report them as likely false positives. Developers can prioritize them accordingly. The *negative* warnings specify the bugs with "the code cannot", e.g., unreachable conditions or dead code. For these warnings, we confirm them as false positives by showing one counter example through testing that such conditions can happen.

6 THE SCOPE AND LIMITATIONS OF OUR APPROACH

Helium provides a syntactic patch for a static warning path by analyzing the parse trees of the program. If the original warning path misses a data dependency, our syntactic patching algorithm may add it when fixing syntax errors, but it does not guarantee that all the data dependencies missed by static analyses are patched. When the necessary failure-inducing statements are missed in the warning paths, or when the paths are actually not reachable from the beginning of the program, i.e., dead code, *Helium* can report inaccurate results, manifesting as false positives and false negatives; however, from our experience with commercial static analysis tools, we have found that these tools work well with our assumptions. They are conservative and often provide more statements than necessary to help developers to diagnose the warnings.

One drawback of static analysis tools is that they can generate many false positive paths as a result of over-approximation. *Helium*, as a testing tool, is useful for distinguishing buggy paths from false positive paths. Importantly, using *Helium*, developers now can obtain the actual, failure-inducing input and executable, in addition to static paths, to diagnose the issue. Of course, there can be cases where *Helium* fails to find a failure-inducing input and thus fails to prioritize the true positive warnings. Also, there are types of bugs, such as concurrency bugs, that can be difficult for tests to manifest even when the warning is valid.

7 EVALUATION

Our evaluation aims to answer the following questions:

- **RQ1.** How effectively can our LCA-based patching and dependency resolving techniques compile code fragments?
- **RQ2.** How effectively can we generate harnesses, inputs and oracles for testing code fragments? Can we preserve semantics of warnings to trigger their bugs in testing?
- **RQ3.** How effectively can we validate real-world static warnings?

7.1 Experimental Setup

To answer the research questions, we implemented *Helium* for C programs using Clang [42, 43], pycparser [12] and srcML [2]. The LCA patching algorithm is implemented in *Racket* [1] and the testing framework is implemented in C++. For building the code fragments, we tried the two approaches specified in Section 5.1. We used the second approaches for both compiling and generating build scripts, as they were shown to better handle real-world code. We manually (one-time effort) created a mapping between static and dynamic warnings similar to [38]. The mapping is implemented in a script to enable automatic matching for individual segments.

Static analysis tools: We surveyed all the static analysis tools (for C/C++ code) listed on the NIST website [5]. Among the 24 tools, all of the 8 commercial static analyzers [3, 26, 50, 55, 57, 63, 64, 67] report paths in the static warnings while for the 16 open-source tools, only *clang-analyzer* and *Oink-stack* provided paths. *Oink-stack* cannot be compiled, and *clang-analyzer* is not *CWE* [6] complete and reported very limited warnings for our benchmarks. To demonstrate that *Helium* can generally handle many categories of warnings, we used the two popular, CWE complete, commercial static analyzers, *PolySpace* and the *Commercial Tool* (anonymous for our license agreement) in our studies.

Benchmarks: We used C projects in CoreBench [14] (we used the first listed buggy version of all 4 projects) and BugBench [48] (it released 11 programs and 8 are C programs which we used). The 12 projects consist of 2.9 million lines of code (sloc) [7], shown in the first two columns of Table 1. From the *Commercial Tool* and *PolySpace*, we processed a total of 1955 warnings of 41 categories. We did not include code smell warnings that manifest no dynamic symptoms, and we also did not include the warnings that require dynamic-checks not yet supported by KLEE, Valgrind or our assertions.

Experimental design, metrics and baselines: To answer RQ1, we used three metrics: 1) the number of code fragments successfully parsed under *Syntactic Patching* in Table 1 (for evaluating LCA algorithm), 2) the number of code fragments successfully compiled under *Dependencies* (for evaluating our dependency solver), and 3) the average size of code fragments patched under *Code Fragment Size* (for demonstrating LCA patch size). Here, we compared Helium with three baselines: Column *np* shows the results from the code fragments directly taken from static warnings without patching; Columns *R* and *M* report the results from RLAssist [29] and MACER [18], the two state-of-the-art compiler error fixing tools, respectively. To avoid gathering results in favor of our tool, when computing compilation rate, we ran our dependency solver for each segment and then send it to RLAssist and MACER for patching. Hence, the data under *Syntactic Patching* and *Dependencies* are observed under the same input for all the four settings.

To answer RQ2, we report the number of executable test cases generated and also the number of test cases triggered by random testing, Radamsa and KLEE under Columns *Executable Tests*, *Random*, *Radamsa*, and *KLEE* in Table 2. We provide the total valid/pass test cases generated by the three approaches under Column *Summary of testing*. In addition to RLAssist and MACER (Columns *R* and *M*), we used *Unit Testing* as another baseline (Columns *U* in Table 2). Unit testing is a common industry practice, and combining static checking with unit testing has been shown useful for Java code [19]. It will be interesting to see how effectively unit testing can validate static warnings generated from the integrated C software. To construct a unit test, we took the functions where intraprocedural warnings are reported. We used Helium to resolve their dependencies and create test harnesses.

In random testing, we ran 20 randomly generated test inputs for each code fragment. Initially, we ran 100 random inputs but found that the error is mostly triggered within the first 20 inputs. For Radamsa, we configured the fuzzing time as 15 minutes and used 100 random inputs as seeds. Similar to random inputs, we observed empirically that increasing the time further didn't improve the results. To ensure reproducibility, we ran the fuzzing twice and only reported the results that were consistent across the runs.

For RQ3, we use the approach in Section 5.2 to determine how many test cases confirm true positives and false positives (Columns *tp* and *fp* in Table 3), and how many help increase our confidence that the warnings are likely false positives (Column *likely fp*). We assigned three authors to validate a random selection of 50% of the results in Table 3, following the literature [34]. The three authors first independently validate the results for 5 code fragments and then met to discuss. After learning and agreeing on the inspection details, the authors independently inspected the other segments and compared the results. If a disagreement occurs for a particular output, the authors discuss it until either the agreement is reached, or it would be listed as unknown.

We added two baselines to compare testing segments with testing entire programs. First, we ran *BovInspector* [25], the only tool we found that applied symbolic execution over entire software to trigger bugs, guided by the paths in static warnings. We also ran existing test suites shipped with the benchmarks using Valgrind and determine if any of the static warnings are triggered in testing.

Running the experiments: All of our experiments, except the training for RLAssist and MACER, were run on a VM with an 8 core Intel Haswell processor, 16GB memory and CentOS 8. We trained RLAssist and MACER using the hyperparameters recommended in [18, 29]. Training RLAssist and Macer took 2.5 days and 15 minutes, respectively, on an Intel Xeon Gold 6152 CPU @ 2.10GHz, with 128GB of RAM and 2x NVIDIA Tesla V100(32GB) GPUs.

7.2 Results of RQ1

In Table 1 in Columns *np* under *Syntactic Patching* and *Dependencies*, we show that only 56 out of 1955 (2.8%) warnings can be directly parsed, and that only 91 (4.6%) warnings are compiled with the dependencies Helium provided. The results confirmed our assumption that most of the warnings generated by static analysis tools cannot be compiled directly.

Applying LCA syntactic patching, we successfully parsed 618 (31.6%) code fragments, shown under *Syntactic patching/LCA*, and after further resolving dependencies, we compiled a total of 1340 (68.5%) code fragments, shown under *Dependency/Helium*. We tried three tools to collect parsing rates, *pycparser*, *Clang* and *GCC* with `-fsyntax-only` flags. We found that all of the tools require some type and macro information to report successful parsing [11, 24, 69]. As a result, after we provide dependency information, the compilation rate is significantly improved. We used *pycparser*, as it requires the least dependencies and more accurately reports the syntax errors than the other two parsing tools.

Our results show that *Helium* works consistently for all the warning types and projects we have experimented with. We inspected 20% of the compilation failures and found that about 60% of the failures are caused by incorrect compiler flags. We used one generic makefile per project to compile all the code fragments; however, for some projects, certain segments require additional compiler flags to handle macros and variable declarations. About 40% of failures are caused by bugs in srcML and also by the fact that our prototype

[5] https://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html
[6] https://nvd.nist.gov/vuln/categories
[7] the lines are calculated by *tokei*, https://github.com/XAMPPRocky/tokei

Table 1 – Results of RQ1: Compiling code fragments

Projects	Size(sloc)	Static Warnings		Syntactic Patching				Dependencies				Code Fragment Size	
		C	P	np	LCA	R	M	np	Helium	R	M	np	LCA
polymorph-0.4.0	3.6k	6	7	0	13	0	0	0	12	0	0	21.9	41.4
ncompress-4.2.4	6.7k	15	2	0	17	0	0	0	13	0	0	157.1	183.9
man-1.5h1	26.9k	37	18	3	22	4	3	1	36	5	2	65.5	99.9
gzip-1.2.4	35.3k	22	19	1	20	0	1	2	38	1	2	35.3	52.7
bc-1.06	49.9k	6	45	1	4	0	1	1	41	2	1	72.8	75.3
squid-2.3.STABLE5	329.9k	118	85	2	95	1	7	5	187	1	4	43.9	61.9
cvs-1.11.4	462.3k	447	138	13	150	3	13	14	355	5	10	123.3	132.3
httpd-2.0.48	724.4k	94	N/A	1	1	1	1	7	61	6	0	61.1	84.6
findutils-e6680237	92.5k	92	116	6	28	7	9	11	106	6	11	65.2	89.2
make-0afbbf85	130.4k	91	26	6	0	3	6	5	77	1	3	78.1	86.7
grep-c32c0421	407.0k	79	91	1	91	3	3	3	124	3	4	53.9	76.7
coreutils-0928c241	676.7k	293	108	22	117	7	39	42	290	22	45	73.4	92.6
summary	2,942.6 k	1300	655	56	618	29	79	91	1340	52	82	70.9	89.8

Table 2 – Results of RQ2: Testing code fragments

Project	Executable Tests				Random (valid/pass)				Radamsa (valid/pass)				KLEE (valid/pass)				Summary of testing (valid/pass)			
	H	U	R	M	H	U	R	M	H	U	R	M	H	U	R	M	H	U	R	M
polymorph-0.4.0	9	0	0	0	0/4	0	0	0	0/4	0	0	0	0/5	0	0	0	0/13	0	0	0
ncompress-4.2.4	14	10	0	0	2/4	0/2	0	0	1/3	0	0	0	2/3	0	0	0	5/10	0/2	0	0
man-1.5h1	33	12	5	1	5/18	0/7	0/4	0/1	7/9	0	0/1	0	5/17	0/7	0/5	0/1	17/44	0/14	0/10	0/2
gzip-1.2.4	24	6	1	2	0/16	0/4	0/1	0/2	0/13	0/1	0	0	0/14	0/2	0	0/2	0/43	0/7	0/1	0/4
bc-1.06	37	1	0	0	0/23	0/1	0	0	1/22	0	0	0	0/22	0/1	0	0	1/67	0/2	0	0
cvs-1.11.4	312	61	5	7	9/158	3/37	0/4	0/6	10/51	3/3	0	0/1	7/126	3/15	0/3	0/5	26/335	9/55	0/7	0/12
squid-2.3.STABLE5	155	31	0	3	14/54	2/11	0	0/1	20/36	2/1	0	0	19/70	1/14	0	0	53/160	5/26	0	0/1
httpd-2.0.48	25	18	1	0	0/16	0/5	0/1	0	0/3	0/2	0	0	0/5	0/2	0/1	0	0/24	0/9	0/2	0
findutils-e6680237	89	37	4	8	5/28	3/22	1/3	1/3	4/15	4/6	1/3	0/3	4/34	2/22	1/3	1/3	13/77	9/50	3/9	2/9
make-0afbbf85	64	14	0	2	3/31	0/10	0	0/2	2/11	0/2	0	0	1/24	0/5	0	0/2	6/66	0/17	0	0/4
grep-c32c0421	109	13	1	0	13/66	1/9	0	0	17/14	1/4	0	1/0	7/63	1/5	0/1	0	37/143	3/18	0/1	1/0
coreutils-0928c241	132	68	14	20	5/100	1/50	0/12	0/16	7/39	1/17	0/2	0/4	5/74	1/31	0/5	0/13	17/213	3/98	0/19	0/33
Total	1003	271	31	43	56/518	10/158	1/25	1/31	69/220	11/36	1/6	1/8	50/457	8/104	1/18	1/26	175/1195	29/298	3/49	3/65
	1348				68/732				82/270				60/605				210/1607			

does not yet support all of the C language's features e.g. nested macros.

Helium outperformed both RLAssist and MACER significantly. Under Columns *R* and *M*, we show that the parsing rates for RLAssist and MACER are 29 (1.4%) and 79 (4%) respectively, and the compilation rates are 52 (2.6%) and 82 (4.1%) respectively. We observed that some warnings miss only ";" or "}" where the two tools can help. However, the two tools are not effective for the majority of real-world static warnings, as we found that they mutated and deleted lines to fix compiler errors. We analyzed the patches produced by *Helium*, and found that similar to the types of constructs added in Figure 1, the patches included the relevant control-dependent nodes and sometimes data-dependent nodes missed by static analysis tools.

Under *Code Fragment Size*, we show that the LCA patch size is small, 18.8 sloc on average, and the average patched code fragment is 89.8 sloc. The patched programs generated by RLAssist and MACER mostly have the same sizes as unpatched versions. We report an average of 0.7 s per segment for LCA patching and 16.8 s for generating a compilable unit from a warning. Our approach is scalable in that the time of processing a warning is independent of the size of the software project. Therefore, we can handle warnings from all of the real-world programs in our benchmarks, the biggest of which is 676.7 k sloc.

7.3 Results for RQ2

Helium successfully generated 1003 executable test cases, compared to 271, 31 and 43 for unit testing, RLassist and MACER respectively shown under *Executable Tests* in Table 2. We were not able to generate executables for all the complied code fragments because of the link errors and the errors of initializing input variables in test harnesses, specifically, (1) certain macros and function definitions can take different arguments and return types depending on the compiler flags, which we did not always set correctly; (2) some functions in the project are declared as *static* or *inline* and only can be linked by the functions in the same files but not by our code fragments; and (3) our prototype is not yet able to recognize constant variables and tried to initialize them with test inputs.

Among the tools, *Helium* triggered the most bugs and assertions and reported 175 valid test inputs, shown under *Summary of testing*

Column *H*. Unit testing, RLAssist and MACER reported 29, 3, and 3 valid tests respectively under *Summary of testing* Columns *U*, *R* and *M*. *Helium* can handle interprocedural warnings (which are 59% of the total warnings) that unit testing cannot handle. Even comparing only intraprocedural warnings, *Helium* triggered 33 of the 802 warnings while unit testing only triggered 11. This indicates that testing code fragments more effectively triggered the bugs and assertions than testing entire procedures.

Specifically, among the total 802 intraprocedural warnings, *Helium* and the unit testing approach produced 235 common test cases, 6 of which are triggered by both approaches. This provides the evidences, besides our reasoning in Section 3.3, that *Helium* preserved the semantics of static warnings. Importantly, we have seen the advantages of testing code fragments over unit testing even for handling intraprocedural warnings. First, there are four cases, where unit testing failed to reach the bug due to the larger size of the function, but *Helium* triggered the bug. Second, there are 23 cases where *Helium* succeeded and unit testing failed, and 5 cases where unit testing succeeded but *Helium* failed, mostly due to the build errors. We can see that *Helium* doesn't yet support all the C features, which impacted both *Helium* and unit testing but unit testing is affected more and produced more unbuildable cases because of its additional code and complexity.

For RLAssist and MACER, we found that all the warnings triggered by the two tools are also trigged by *Helium*. Our inspection showed that RLAssist and MACER changed control flow of the segments and function signatures, which lead to low success rates. This further confirms the importance of preserving semantics during syntactic patching.

Comparing the three testing approaches in the row *Total* in Table 2, we found that Radamsa (fuzzing) performed the best for triggering bugs in static warnings, and reported 82 valid test cases, followed by random testing (generated 68 valid test cases), and then KLEE (60 valid test cases). Radamsa reported the least number of pass test cases, as we observed that the fuzzer tried to crash our test harnesses. Radamsa and Random both triggered some unique warnings, but KLEE did not.

7.4 Results for RQ3

In Table 3, we show that Helium significantly outperformed other baselines regarding the confirmed true positives (under Column *tp*), confirmed false positives (under *fp*), and likely false positives (under *likely fp*). We see that *Helium* confirmed 48 true positives, 27 false positives and 205 likely false positives. On the other hand, unit testing confirmed only 11 false positives and 35 likely false positives, and RLAssist and MACER both confirmed 1 false positive and 6 and 10 likely false positives respectively. Note that multiple test cases in Column *Summary of testing* in Table 2 can be generated from a same warning by different tools. The numbers under *tp* and *fp* in Table 3 counted validated warnings, and thus are smaller.

BovInspector only finished running with `make` and did not trigger any buffer overflows. The rest of benchmarks are either not terminated after 60 minutes or cannot be handled by KLEE. This indicates that testing code fragments can provide the scalability and practicality that cannot be achieved by performing symbolic execution on the entire software.

We ran a total of 7748 existing tests found in our benchmarks, excluding `polymorph`, `gzip`, `ncompress`, `httpd`, `man` and `squid` in BugBench that do not have test suites. We matched 2 memory leak warnings for `coreutils`.

Table 3 – RQ3: Validating static warnings: Columns *tp*, *fp* and *likely fp* list the tools' output.

Tool	tp	fp	likely fp
Helium	48	27	205
Unit testing	0	11	35
RLAssist	0	1	6
MACER	0	1	10
BovInspector	0	0	0
Existing test suite	2	0	0

One of the *Helium* true positives listed in Table 3 is matched to CVE-2001-1413, and the other one is matched to a real-world bug [8]. We also triggered two more real-world bugs documented in BugBench, after fixing some bugs in the test harnesses of two segments. All of the four matched real-world bugs are interprocedural bugs and are only triggered by *Helium* but not other tools.

We manually validated 50% of true positives and false positives reported in Table 3 (process given in Section 7.1). We found that 75% of the reviewed results are indeed true and false positives as stated by *Helium*, among which, the correct confirmation rate for true positives is 83%. Our further investigation shows that the imprecision is due to the incompleteness of the *Helium* implementation, e.g., we did not include a bounds-checking related to a nested macro (not yet supported by *Helium*) and triggered buffer overflow; our test harness failed to correctly initialize a structure variable used in an *Unreachable Call* warning.

7.5 Threats to Validity

To address the external threats to validity, we selected two popular commercial static analysis tools and processed close to 2000 warnings of 40+ types from 12 C projects and 2.9 million lines of code. To address the internal threat to validity, we inspected 20-30% code fragments at each step of our implementation, and 50% of final results to assure they are correct (following a protocol documented in [34]). After validating our results, we believe that our success rates can be further improved after we handle more C features. We used the models trained with the dataset shipped with RLAssist and MACER. The dataset may not represent the distributions of our code fragments.

8 RELATED WORK

Automatically processing static warnings: Our work is related to verifying and testing software based on static warnings [46, 47, 53, 54, 56, 75, 76]. Muske et al. added assertions for divide-by-zero and array out-of-bounds warnings and applied model checking to verify the assertions [53]. Parvez et al. and Zhang et al. used symbolic execution to trigger the warnings in binary applications [56, 75]. Li et al. validated memory leaks using concolic testing along the

[8] https://bugzilla.redhat.com/show_bug.cgi?id=40400

paths reported in the warnings [46, 47]. Our work generated a small possible test case for each warning instead of testing entire software.

There are also approaches that identify patterns from warnings, source code and software repositories for predicting false positives [7, 9, 17, 40, 45, 59, 71, 74], and that use machine learning techniques to learn what are likely true and false positives [7, 23, 39, 45, 59, 70**?**]. For example, Zhang et al. automatically learned and integrated the users' feedback to rank the warnings [76]. Finally, there are tools that use multiple static tools to cross-validate and rank the warnings to increase their reliability [23, 73].

Auto-fixing compiler errors: Existing techniques addressed subproblems of compilation issues, such as resolving identifier names [68, 77], inferring types [20] and fixing parsing errors [5, 28, 33, 65]. Recently, deep learning and reinforcement learning have been applied to fix compiler errors based on the supervised dataset and feedback of compiler warnings [4, 13, 29, 30, 51, 60, 66, 72]. Automatic program repair tools also reported some successes to fix compiler errors [10, 41]. These tools synthesized the patches based on bug reports or past fixes. Our approach analyzed parse trees for syntactic patching. It relies on neither supervised dataset nor sufficient tests and oracles for correctness.

Parse tree analysis: There has been work on transforming parse trees to reduce the size of programs for debugging compilers. *Hierarchical delta debugging (HDD)* performed delta debugging on parse trees to generate smaller programs [52]. *Generalized tree reduction* [32] improves HDD and performs transformations on a sub-tree of the parse tree. These techniques used a search-based technique to transform parse trees, and are different from our syntactic patching algorithm.

Testing code fragments for other applications: There have been research interests in analyzing and executing code fragments [22, 27, 35–37, 44, 58, 61, 78]. Godefroid proposed *micro execution* [27], a VM technique for executing code fragments at the binary level. EqMiner ran contiguous lines of code to detect semantic clones [37]. Segmented symbolic analysis applied dynamic analysis on loops and library calls to help static analysis [44]. All the above work handled continuous lines of code. None have focused on systematically building and testing any code fragments.

9 CONCLUSIONS AND FUTURE WORK

This paper presents the techniques and a tool to validate static warnings. The key idea is to take the warning and patch it to generate semantically equivalent executable code fragments. We can then use existing testing tools to trigger the failures in the warnings. We formally defined what we mean by patching a code fragment and developed an algorithm to automatically generate such patches. We built a system that addressed the challenges of building and testing code segments. We achieved 68.5% build rate for the complicated warnings reported by commercial tools. Our tool scales to all the real-world large C projects used in our evaluation. We confirmed the true positives that match to CVE and real-world bugs, in which other baseline methods did not succeed. In the future, we plan to further improve the robustness of our tool and explore more applications of testing code fragments.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. We thank Sebastian Elbaum for kindly shepherding our paper. We thank Ekene Shiafiwi Okeke for collecting static warnings from the *PolySpace* static analysis tool. This research is supported by the US National Science Foundation (NSF) under Award 1816352.

REFERENCES

[1] [n.d.]. Racket, the Programming Language. Available at https://racket-lang.org (2020/8/26).
[2] [n.d.]. srcML: an XML format for source code. Available at https://www.srcml.org (2020/8/26).
[3] ABSINT. [n.d.]. Astree Static Analyzer. https://www.absint.com/astree/index.htm.
[4] Toufique Ahmed, Vincent Hellendoorn, and Premkumar Devanbu. 2019. Learning Lenient Parsing & Typing via Indirect Supervision. *arXiv preprint arXiv:1910.05879* (2019).
[5] Alfred V. Aho and Thomas G. Peterson. 1972. A Minimum Distance Error-Correcting Parser for Context-Free Languages. *SIAM J. Comput.* 1, 4 (1972), 305–312. https://doi.org/10.1137/0201022 arXiv:https://doi.org/10.1137/0201022
[6] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. Compilers, principles, techniques. *Addison wesley* 7, 8 (1986), 9.
[7] Enas A. Alikhashashneh, Rajeev R. Raje, and James H. Hill. 2018. Using Machine Learning Techniques to Classify and Predict Static Code Analysis Tool Warnings. In *2018 IEEE/ACS 15th International Conference on Computer Systems and Applications (AICCSA)*. 1–8. https://doi.org/10.1109/AICCSA.2018.8612819
[8] Frances E. Allen. 1970. Control Flow Analysis. In *Proceedings of a Symposium on Compiler Optimization* (Urbana-Champaign, Illinois). Association for Computing Machinery, New York, NY, USA, 1–19. https://doi.org/10.1145/800028.808479
[9] Hirohisa Aman, Sousuke Amasaki, Tomoyuki Yokogawa, and Minoru Kawahara. 2019. A Survival Analysis-Based Prioritization of Code Checker Warning: A Case Study Using PMD. In *Studies in Computational Intelligence*. Springer International Publishing, 69–83. https://doi.org/10.1007/978-3-030-24405-7_5
[10] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to Fix Bugs Automatically. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 159 (Oct. 2019), 27 pages. https://doi.org/10.1145/3360585
[11] Eli Bendersky. [n.d.]. On parsing C, type declarations and fake headers. Available at https://eli.thegreenplace.net/2015/on-parsing-c-type-declarations-and-fake-headers (2020/8/26).
[12] Eli Bendersky. [n.d.]. pycparser: Complete C99 parser in pure Python. Available at https://github.com/eliben/pycparser (2020/8/26).
[13] Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. 2018. Neuro-Symbolic Program Corrector for Introductory Programming Assignments. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 60–70. https://doi.org/10.1145/3180155.3180219
[14] Marcel Böhme and Abhik Roychoudhury. 2014. CoREBench: Studying Complexity of Regression Errors. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 105–115. https://doi.org/10.1145/2610384.2628058
[15] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. 2000. A Static Analyzer for Finding Dynamic Programming Errors. *Softw. Pract. Exper.* 30, 7 (June 2000), 775–802.
[16] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. 2012. Program Slicing Enhances a Verification Technique Combining Static and Dynamic Analysis. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing* (Trento, Italy) *(SAC '12)*. Association for Computing Machinery, New York, NY, USA, 1284–1291. https://doi.org/10.1145/2245276.2231980
[17] Foteini Cheirdari and George Karabatis. 2018. Analyzing False Positive Source Code Vulnerabilities Using Static Analysis Tools. In *2018 IEEE International Conference on Big Data (Big Data)*. 4782–4788. https://doi.org/10.1109/BigData.2018.8622456
[18] Darshak Chhatbar, Umair Z. Ahmed, and Purushottam Kar. 2020. MACER: A Modular Framework for Accelerated Compilation Error Repair. *arXiv preprint arXiv:2005.14015* (2020).
[19] Christoph Csallner and Yannis Smaragdakis. 2005. Check 'n' Crash: Combining Static Checking and Testing. In *Proceedings of the 27th International Conference on Software Engineering* (St. Louis, MO, USA) *(ICSE '05)*. Association for Computing Machinery, New York, NY, USA, 422–431. https://doi.org/10.1145/1062455.1062533
[20] Barthélémy Dagenais and Laurie Hendren. 2008. Enabling Static Analysis for Partial Java Programs. *SIGPLAN Not.* 43, 10 (Oct. 2008), 313–328. https://doi.org/10.1145/1449955.1449790

[21] Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: Path-sensitive Program Verification in Polynomial Time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. ACM, New York, NY, USA, 57–68. https://doi.org/10.1145/512529.512538

[22] Florian Deissenboeck, Lars Heinemann, Benjamin Hummel, and Stefan Wagner. 2012. Challenges of the Dynamic Detection of Functionally Similar Code Fragments. In *2012 16th European Conference on Software Maintenance and Reengineering*. 299–308. https://doi.org/10.1109/CSMR.2012.38

[23] Lori Flynn, William Snavely, David Svoboda, Nathan VanHoudnos, Richard Qin, Jennifer Burns, David Zubrow, Robert Stoddard, and Guillermo Marce-Santurio. 2018. Prioritizing Alerts from Multiple Static Analysis Tools, Using Classification Models. In *Proceedings of the 1st International Workshop on Software Qualities and Their Dependencies* (Gothenburg, Sweden) *(SQUADE '18)*. ACM, New York, NY, USA, 13–20. https://doi.org/10.1145/3194095.3194100

[24] Free Software Foundation. [n.d.]. 3.8 Options to Request or Suppress Warnings. Available at https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html (2020/8/26).

[25] Fengjuan Gao, Linzhang Wang, and Xuandong Li. 2016. BovInspector: Automatic Inspection and Repair of Buffer Overflow Vulnerabilities. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. Association for Computing Machinery, New York, NY, USA, 786–791. https://doi.org/10.1145/2970276.2970282

[26] GHS. [n.d.]. Double Check Static Analyzer. https://www.ghs.com/products/doublecheck.html.

[27] Patrice Godefroid. 2014. Micro Execution. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. ACM, New York, NY, USA, 539–549. https://doi.org/10.1145/2568225.2568273

[28] Susan L. Graham and Steven P. Rhodes. 1973. Practical Syntactic Error Recovery in Compilers. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Boston, Massachusetts) *(POPL '73)*. Association for Computing Machinery, New York, NY, USA, 52–58. https://doi.org/10.1145/512927.512932

[29] Rahul Gupta, Aditya Kanade, and Shirish Shevade. 2019. Deep Reinforcement Learning for Syntactic Error Repair in Student Programs. *Proceedings of the AAAI Conference on Artificial Intelligence* 33, 01 (Jul. 2019), 930–937. https://doi.org/10.1609/aaai.v33i01.3301930

[30] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence* (San Francisco, California, USA) *(AAAI'17)*. AAAI Press, 1345–1351.

[31] Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. 2006. Modular Checking for Buffer Overflows in the Large. In *Proceedings of the 28th International Conference on Software Engineering* (Shanghai, China) *(ICSE '06)*. ACM, New York, NY, USA, 232–241. https://doi.org/10.1145/1134285.1134319

[32] Satia Herfert, Jibesh Patra, and Michael Pradel. 2017. Automatically Reducing Tree-Structured Test Inputs. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, 861–871.

[33] E. T. Irons. 1963. An Error-Correcting Parse Algorithm. *Commun. ACM* 6, 11 (Nov. 1963), 669–673. https://doi.org/10.1145/368310.368385

[34] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 510–520. https://doi.org/10.1145/3338906.3338955

[35] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 96–105.

[36] Lingxiao Jiang and Zhendong Su. 2008. Profile-Guided Program Simplification for Effective Testing and Analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Atlanta, Georgia) *(SIGSOFT '08/FSE-16)*. Association for Computing Machinery, New York, NY, USA, 48–58. https://doi.org/10.1145/1453101.1453110

[37] Lingxiao Jiang and Zhendong Su. 2009. Automatic Mining of Functionally Equivalent Code Fragments via Random Testing. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* (Chicago, IL, USA) *(ISSTA '09)*. Association for Computing Machinery, New York, NY, USA, 81–92. https://doi.org/10.1145/1572272.1572283

[38] Vineeth Kashyap, Jason Ruchti, Lucja Kot, Emma Turetsky, Rebecca Swords, Shih An Pan, Julien Henry, David Melski, and Eric Schulte. 2019. Automated Customized Bug-Benchmark Generation. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 103–114. https://doi.org/10.1109/SCAM.2019.00020

[39] Ugur Koc, Parsa Saadatpanah, Jeffrey S. Foster, and Adam A. Porter. 2017. Learning a Classifier for False Positive Error Reports Emitted by Static Code Analysis Tools. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Barcelona, Spain) *(MAPL 2017)*. Association for Computing Machinery, New York, NY, USA, 35–42. https://doi.org/10.1145/3088525.3088675

[40] Ugur Koc, Shiyi Wei, Jeffrey S. Foster, Marine Carpuat, and Adam A. Porter. 2019. An Empirical Assessment of Machine Learning Approaches for Triaging Reports of a Java Static Analysis Tool. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 288–299. https://doi.org/10.1109/ICST.2019.00036

[41] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 2019. IFixR: Bug Report Driven Program Repair. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 314–325. https://doi.org/10.1145/3338906.3338935

[42] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD Conference*. 1–2.

[43] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[44] Wei Le. 2013. Segmented symbolic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. 212–221. https://doi.org/10.1109/ICSE.2013.6606567

[45] Seongmin Lee, Shin Hong, Jungbae Yi, Taeksu Kim, Chul-Joo Kim, and Shin Yoo. 2019. Classifying False Positive Static Checker Alarms in Continuous Integration Using Convolutional Neural Networks. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 391–401. https://doi.org/10.1109/ICST.2019.00048

[46] Mengchen Li, Yuanjun Chen, Linzhang Wang, and Guoqing Xu. 2013. Dynamically Validating Static Memory Leak Warnings. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis* (Lugano, Switzerland) *(ISSTA 2013)*. Association for Computing Machinery, New York, NY, USA, 112–122. https://doi.org/10.1145/2483760.2483778

[47] X Li, Y Zhou, MC Li, YJ Chen, GQ Xu, LZ Wang, and XD Li. 2017. Automatically validating static memory leak warnings for C/C++ programs. *Ruan Jian Xue Bao/Journal of Software* 28, 4 (2017), 827–844.

[48] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. [n.d.]. Bugbench: Benchmarks for evaluating bug detection tools.

[49] Many. 2017. Build EAR: a tool that generates a compilation database. https://github.com/rizsotto/Bear. Accessed: 2017-05-07.

[50] Mathworks. [n.d.]. Polyspace. https://www.mathworks.com/products/polyspace.html.

[51] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. DeepDelta: Learning to Repair Compilation Errors. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 925–936. https://doi.org/10.1145/3338906.3340455

[52] Ghassan Misherghi and Zhendong Su. 2006. HDD: Hierarchical Delta Debugging. In *Proceedings of the 28th International Conference on Software Engineering* (Shanghai, China) *(ICSE '06)*. Association for Computing Machinery, New York, NY, USA, 142–151. https://doi.org/10.1145/1134285.1134307

[53] Tukaram Muske and Uday P. Khedker. 2015. Efficient elimination of false positives using static analysis. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. 270–280. https://doi.org/10.1109/ISSRE.2015.7381820

[54] Thu Trang Nguyen, Pattaravut Maleehuan, Toshiaki Aoki, Takashi Tomita, and Iori Yamada. 2019. Reducing False Positives of Static Analysis for SEI CERT C Coding Standard. In *2019 IEEE/ACM Joint 7th International Workshop on Conducting Empirical Studies in Industry (CESI) and 6th International Workshop on Software Engineering Research and Industrial Practice (SER IP)*. 41–48. https://doi.org/10.1109/CESSER-IP.2019.00015

[55] ParaSoft. [n.d.]. Ctest Static Analyzer. https://www.parasoft.com/ctest/static-analysis.

[56] Riyad Parvez, Paul A. S. Ward, and Vijay Ganesh. 2016. Combining Static Analysis and Targeted Symbolic Execution for Scalable Bug-Finding in Application Binaries. In *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering* (Toronto, Ontario, Canada) *(CASCON '16)*. IBM Corp., USA, 116–127.

[57] Perforce. [n.d.]. Klockwork Static Analyzer. https://www.perforce.com/products/klocwork.

[58] Marcelino Rodriguez-Cancio, Benoit Combemale, and Benoit Baudry. 2016. Automatic Microbenchmark Generation to Prevent Dead Code Elimination and Constant Folding. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. Association for Computing Machinery, New York, NY, USA, 132–143. https://doi.org/10.1145/2970276.2970346

[59] Joseph R. Ruthruff, John Penix, J. David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. 2008. Predicting Accurate and Actionable Static Analysis Warnings: An Experimental Approach. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE '08)*. Association for Computing Machinery, New York, NY, USA, 341–350. https://doi.org/10.1145/1368088.1368135

[60] Eddie Antonio Santos, Joshua Charles Campbell, Dhvani Patel, Abram Hindle, and José Nelson Amaral. 2018. Syntax and sensibility: Using language models to detect and correct syntax errors. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 311–322. https://doi.org/10.1109/SANER.2018.8330219

[61] Abdus Satter and Kazi Sakib. 2017. A Similarity-Based Method Retrieval Technique to Improve Effectiveness in Code Search. In *Companion to the First International Conference on the Art, Science and Engineering of Programming* (Brussels, Belgium) *(Programming '17)*. Association for Computing Machinery, New York, NY, USA, Article 39, 3 pages. https://doi.org/10.1145/3079368.3079372

[62] Baruch Schieber and Uzi Vishkin. 1988. On Finding Lowest Common Ancestors: Simplification and Parallelization. *SIAM J. Comput.* 17, 6 (1988), 1253–1262. https://doi.org/10.1137/0217079 arXiv:https://doi.org/10.1137/0217079

[63] SonarSource. [n.d.]. SonarQube Static Analyzer. https://www.sonarqube.org/.

[64] Synopsis. [n.d.]. Coverity. https://scan.coverity.com/.

[65] Rohit Takhar and Varun Aggarwal. 2019. Grading Uncompilable Programs. *Proceedings of the AAAI Conference on Artificial Intelligence* 33 (July 2019), 9389–9396. https://doi.org/10.1609/aaai.v33i01.33019389

[66] Daniel Tarlow, Subhodeep Moitra, Andrew Rice, Zimin Chen, Pierre-Antoine Manzagol, Charles Sutton, and Edward Aftandilian. 2019. Learning to Fix Build Errors with Graph2Diff Neural Networks. *arXiv preprint arXiv:1911.01205* (2019).

[67] Grammar Tech. [n.d.]. Code Sonar. https://www.grammatech.com/products/codesonar.

[68] Valerio Terragni, Yepang Liu, and Shing-Chi Cheung. 2016. CSNIPPEX: Automated Synthesis of Compilable Code Snippets from Q&A Sites. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 118–129. https://doi.org/10.1145/2931037.2931058

[69] The Clang project. [n.d.]. clang - the Clang C, C++, and Objective-C compiler. Available at https://clang.llvm.org/docs/CommandGuide/clang.html (2020/8/26).

[70] Omer Tripp, Salvatore Guarnieri, Marco Pistoia, and Aleksandr Aravkin. 2014. ALETHEIA: Improving the Usability of Static Security Analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. Association for Computing Machinery, New York, NY, USA, 762–774. https://doi.org/10.1145/2660267.2660339

[71] Han Wang, Min Zhou, Xi Cheng, Guang Chen, and Ming Gu. 2018. Which Defect Should Be Fixed First? Semantic Prioritization of Static Analysis Report. In *Software Analysis, Testing, and Evolution*. Springer International Publishing, 3–19. https://doi.org/10.1007/978-3-030-04272-1_1

[72] Jiangtao Xue, Xinjun Mao, Yao Lu, Yue Yu, and Shangwen Wang. 2019. History-Driven Fix for Code Quality Issues. *IEEE Access* 7 (2019), 111637–111648. https://doi.org/10.1109/ACCESS.2019.2934975

[73] Achilleas Xypolytos, Haiyun Xu, Barbara Vieira, and Amr M.T. Ali-Eldin. 2017. A Framework for Combining and Ranking Static Analysis Tool Findings Based on Tool Performance Statistics. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 595–596. https://doi.org/10.1109/QRS-C.2017.110

[74] Xueqi Yang, Jianfeng Chen, Rahul Yedida, Zhe Yu, and Tim Menzies. 2020. How to Recognize Actionable Static Code Warnings (Using Linear SVMs). arXiv:2006.00444v1 [cs.SE]

[75] Sai Zhang, David Saff, Yingyi Bu, and Michael D. Ernst. 2011. Combined Static and Dynamic Automated Test Generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (Toronto, Ontario, Canada) *(ISSTA '11)*. Association for Computing Machinery, New York, NY, USA, 353–363. https://doi.org/10.1145/2001420.2001463

[76] Xin Zhang, Radu Grigore, Xujie Si, and Mayur Naik. 2017. Effective Interactive Resolution of Static Analysis Alarms. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 57 (Oct. 2017), 30 pages. https://doi.org/10.1145/3133881

[77] Hao Zhong and Xiaoyin Wang. 2017. Boosting Complete-Code Tool for Partial Program. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, 671–681.

[78] Meital Zilberstein and Eran Yahav. 2016. Leveraging a Corpus of Natural Language Descriptions for Program Similarity. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Amsterdam, Netherlands) *(Onward! 2016)*. Association for Computing Machinery, New York, NY, USA, 197–211. https://doi.org/10.1145/2986012.2986013