# FuzzSlice: Pruning False Positives in Static Analysis Warnings through Function-Level Fuzzing

Aniruddhan Murali
University of Waterloo
Waterloo, Canada
a25mural@uwaterloo.ca

Noble Saji Mathews
University of Waterloo
Waterloo, Canada
noblesaji.mathews@uwaterloo.ca

Mahmoud Alfadel
University of Waterloo
Waterloo, Canada
malfadel@uwaterloo.ca

Meiyappan Nagappan
University of Waterloo
Waterloo, Canada
mei.nagappan@uwaterloo.ca

Meng Xu
University of Waterloo
Waterloo, Canada
meng.xu.cs@uwaterloo.ca

## ABSTRACT

Manual confirmation of static analysis reports is a daunting task. This is due to both the large number of warnings and the high density of false positives among them. Fuzzing techniques have been proposed to verify static analysis warnings. However, a major limitation is that fuzzing the whole project to reach all static analysis warnings is not feasible. This can take several days and exponential machine time to increase code coverage linearly.

Therefore, we propose FuzzSlice, a novel framework that automatically prunes possible false positives among static analysis warnings. Unlike prior work that mostly focuses on confirming true positives among static analysis warnings, which inevitably requires end-to-end fuzzing, FuzzSlice focuses on ruling out potential false positives, which are the majority in static analysis reports. The key insight that we base our work on is that a warning that does not yield a crash when fuzzed at the function level in a given time budget is a possible false positive. To achieve this, FuzzSlice first aims to generate compilable code slices at the function level. Then, FuzzSlice fuzzes these code slices instead of the entire binary to prune possible false positives. FuzzSlice is also unlikely to misclassify a true bug as a false positive because the crashing input can be reproduced by a fuzzer at the function level as well. We evaluate FuzzSlice on the Juliet synthetic dataset and real-world complex C projects: openssl, tmux and openssh-portable. Our evaluation shows that the ground truth in the Juliet dataset had 864 false positives which were all detected by FuzzSlice. For the open-source repositories, we were able to get the developers from two of these open-source repositories to independently label these warnings. FuzzSlice automatically identifies 33 out of 53 false positives confirmed by developers in these two repositories. This implies that FuzzSlice can reduce the number of false positives by 62.26% in the open-source repositories and by 100% in the Juliet dataset.

## 1 INTRODUCTION

Static analysis tools report errors in the source code of a program without executing it. These tools enable the discovery of vulnerabilities in the early stages of software development. However, they suffer from major issues. First, an overwhelming number of bugs are suggested by these tools, making it hard for a software developer to verify them, which can also lead a software development team to ignore the static analysis report [1, 2]. Second, a static analysis tool may lack the knowledge of how data flows through the system, the dependencies and software architecture [3, 4]. Therefore, many of the bugs turn out to be false positives [4–7]. Such false positives produced by static analysis tools are a significant barrier to the wide-scale adoption of these tools [8, 9].

Fuzzing is a popular software testing technique that involves supplying arbitrary or randomized input to a computer program with the objective of uncovering unexpected behaviors, including crashes. Prior work in fuzz testing has largely focused on identifying true positives in static analysis reports. For example, Böhme et al. utilized directed grey box fuzzers to direct fuzzing towards a target location [10]. Other techniques use dynamic symbolic execution to verify static analysis reports [11]. However, the drawback of such approaches is the time budget and computational power required to reach all static analysis warnings [12]. In this paper, we propose FuzzSlice, an approach that aims to prune false positives produced by static analysis tools. The novelty of FuzzSlice is twofold:

- **Conceptual Innovation:** While several other techniques (e.g., [10, 13–15]) aim at identifying true positives in static analysis warnings, FuzzSlice is optimized towards pruning possible false positives in a given program. Instead of fuzzing the entire program from the main function, FuzzSlice only fuzzes the function containing the warning to prune possible false positives. FuzzSlice hinges on the novel idea that a flagged code fragment (represented as a warning) executed at least once at the function level and not yielding a crash in a given time budget is a possible false positive.
- **Technical Innovation (close-to-warning fuzzing):** Unlike typical methods that reduce fuzzing cost by independently fuzzing modules and libraries [16–19], FuzzSlice focuses on generating compiled slices that encompass the warning location detected by a static analysis tool.

FuzzSlice generates and fuzzes a separate binary for each warning, which facilitates the coverage of most warnings with reduced computational cost (under 5 minutes of fuzzing). Finally, FuzzSlice is unlikely to misclassify a true bug as a false positive because the crashing input can be reproduced by a fuzzer at the function level.

*ICSE 2024, April 2024, Lisbon, Portugal*
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

We evaluate FuzzSlice on four diverse repositories comprising one synthetic and three open-source datasets that have been reported to contain buffer overflow vulnerabilities. Our evaluation shows that the ground truth in the synthetic Juliet dataset has 864 false positives which were all confirmed by FuzzSlice. For open-source repositories, we found 143 possible false positives among 265 warnings. We reached out to developers from two of these open-source repositories (tmux and openssh-portable) to label these warnings. FuzzSlice automatically identifies 33 possible false positives out of 53 false positives confirmed by developers in these two repositories.

In summary, this paper makes the following contributions:

- We introduce FuzzSlice, a novel design built upon the insight that warnings fuzzed at the function level and not resulting in crashes within a reasonable time budget are possible false positives.
- FuzzSlice efficiently identifies possible false positives in static analysis reports by: (1) automatically generating a minimal compiled code slice for complex real world C code encapsulating *any* arbitrary static analysis warning, and (2) generating a fuzzing wrapper that performs type-based input generation for the function enclosing the warning.
- We develop a prototype tool for FuzzSlice. The tool and datasets along with the docker image are publicly available [20].

## 2 MOTIVATIONAL EXAMPLE

The goal of this study is to prune possible false positives efficiently. In this section, we provide an example to motivate the FuzzSlice approach. An example code of the openssl repository [21] in the C language is shown in Listing 1. The code listing describes a function (i.e., `glue_strings`) that joins an array of strings (the function argument) into a single string (the return value). On line 10, the variable `len` is updated in a loop to hold the sum of the length of all input array strings. On line 12, the variable `ret` is dynamically allocated with a size of `len+1`. On line 16, the string is joined together in pointer variable `ret` by iterating the pointer `p` and copying each input string one by one.

```
1  /* Glue an array of strings together and return it as an
       allocated string.
2  */
3  char *glue_strings(const char *list[])
4  {
5      size_t len = 0;
6      char *p, *ret;
7      int i;
8
9      for (i = 0; list[i] != NULL; i++)
10         len += strlen(list[i]);
11
12     if (!(ret = p = OPENSSL_malloc(len + 1)))
13         return NULL;
14
15     for (i = 0; list[i] != NULL; i++)
16         p += strlen(strcpy(p, list[i])); //False positive
17
18     return ret;
19 }
```

**Listing 1: Code snippet from the openssl project flagged by RATS as buffer overflow [22].**

When a static analysis tool such as RATS [23] is run on this code it flags line 16 as a possible heap buffer overflow. However, this is clearly a false positive because the `strcpy` on line 16 can never exceed the bounds of the allocated pointer `ret`. The reason behind this is that the size of the allocated pointer `ret` will always be one plus the length of all input strings. RATS is not capable of this kind of value flow analysis for variable `len`. Therefore the tool cannot be sure that line 16 will never cause a heap buffer overflow.

Fuzzing has become a popular solution for the verification of static analysis reports. It is possible to compile the whole openssl binary and guide the fuzzing toward line 16 in the code snippet from the `main` method. However, this often takes several days, many CPU cycles and requires the help of appropriate fuzzing dictionaries. In spite of this, there is no guarantee that the given static analysis warning can be covered by fuzzing within a given time budget.

FuzzSlice is an approach that is primarily targeted toward false positives in static analysis warnings. It takes advantage of the fact that the function `glue_strings` in Listing 1 can be fuzzed directly to identify it is a false positive. Given an arbitrary warning location, it automatically constructs a function slice of the program, compiles it including necessary dependencies, generates its own fuzzing wrapper and fuzzes the function slice. When the function slice is fuzzed on its own, the static warning on line 16 can be easily reached. Let us assume that fuzzing this function slice gives no crash on line 16. This implies that fuzzing from `main` will also not result in a crash. This can be derived from the fact that caller functions can only constrain the input to a given function through the function arguments. On the other hand, if a crash is observed on the static analysis warning line then we cannot comment on it being an actual bug or a true positive. This is because a caller function can invalidate the input that causes the observed crash in the slice. FuzzSlice aims to prune all false positives within a static analysis report similar to Listing 1.

## 3 FUZZSLICE APPROACH

We split our approach into two parts. First, we discuss the main design steps of FuzzSlice. Then, we discuss in detail how we achieve each step.

### 3.1 Design

The core functionality of FuzzSlice is to decide *whether a specific static analysis warning is a possible false positive* (and hence, can be de-prioritized for manual triage). More formally, given a warning $w$ in a static analysis report for program $P$, FuzzSlice examines $w$ in three conceptual steps, namely (D1) Minimal Slice Creation; (D2) Fuzzing Input Generation; and (D3) Warning Classification.

**(D1) Minimal Slice Creation.** First, we build an execution environment that fully encloses $w$ at the function level. This is called a slice of the original program denoted by $S$. Ideally, $S$ should have the following properties:

(a) $S \subseteq P$, i.e., the slice should be smaller than the program unless the program is just a single main function.
(b) Consider the function $F$ directly enclosing warning $w$. This function must be part of the slice i.e., $F \subset S$

(c) Consider a function $F_2$ which is called by a function $F_1$ within slice $S$ ($F_1 \subset S$). In that case, $F_2$ is also part of the slice $S$ i.e., $F_2 \subset S$.

By definition of property (c), if a function $F_k$ is not called by *any* function $F_i$ within $S$ then $F_k$ cannot be in $S$. This implies that any execution beginning from $F$ (the function enclosing the warning) can never reach $F_k$. Therefore the defined slice $S$ is a *minimal* slice capturing execution environment related to warning $w$ at the function level.

**(D2) Fuzzing Input Generation.** Next, FuzzSlice generates valid and versatile inputs with the goal of testing the minimal slice $S$ comprehensively. FuzzSlice achieves this with randomly generated arguments used by the function $F$ which encloses the warning. FuzzSlice performs type-based input generation and mutation, i.e., unlike AFL [24] or libfuzzer [25] which blindly mutates raw bytes, the input generator in FuzzSlice recognizes the type of the arguments, including struct pointers and mutate inputs based on typing rules. This helps prevent early-termination by input sanitization logic during the execution of F which improves both the efficiency and effectiveness of fuzzing the slice.

**(D3) Warning Classification.** Finally, we decide whether $w$ is a possible false positive, i.e., based on fuzzing the minimal slice $S$:

(a) If the fuzzer finds a concrete input $I$ that causes a dynamic bug checker (e.g., ASAN) to report an issue on $w$ while executing $S$, it is possible that either this confirms $w$ to be a true positive (hence, higher priority for manual triage) or that $I$ is an infeasible input which can never be generated when executing from the 'main' function.
(b) If the fuzzer cannot find an input that causes a dynamic bug checker to complain about $w$ even after exhausting its computation budget and executing $w$ at least once, it is possible that $w$ is a false positive based on the fact that even with a free-form search, the fuzzer cannot find an offending input to trigger the warning.

Implementing the design steps above presents unique challenges. We discuss how we tackle these challenges in the next section.

## 3.2 Proposed technique

In this section, we discuss the technical aspects of each of the design components in the previous section. An overview of FuzzSlice implementation is shown in Figure 1.

FuzzSlice is used when a software developer would like to validate and prioritize a set of warnings from a static analysis tool. FuzzSlice takes the following steps to rule out potential false alarms in these static analysis warnings: (1) Build project and store build commands; (2) Generate structural information of source code in XML format; (3) Create a minimized compilable slice containing the warning; (4) Generate fuzzing wrapper; and (5) Classify warnings into possible false alarms (remaining are worth further investigation).

Along this process, step (1) and (2) are information collection steps that aim to build a compiler-agnostic representation of the program source code; while steps (3), (4), and (5) outline a concrete implementation to achieve each of the three design goals (D1), (D2), and (D3) respectively.

We first describe each step in detail using the code example in Listing 1. Next, we walk through each step to show how the given code example is finally classified by FuzzSlice to be a possible false positive.

In the example of Listing 1, the static analysis tool flags the `strcpy` on line 16 as a possible heap buffer overflow. FuzzSlice currently uses two static analysis tools RATS [23] and Infer [26] to generate static analysis reports. It is important to note that FuzzSlice can use any static analysis tool in principle for this approach.

**Step 1: Build the project and store build commands.** In the first step of our approach, FuzzSlice performs a build of the repository using its native build system (usually make or cmake for C-based projects). During the build process, FuzzSlice stores important build information (e.g., include paths for header files, compiler options, shared library locations, path to compiled file etc.), which will be used later to compile minimized slice $S$. FuzzSlice utilizes Build EAR - a tool that generates a compilation database for a given build process [27]. Build EAR stores build commands related to each file in a JSON format. In our running example, the code snippet lies in a file called `driver.c`. The build command of `driver.c` is stored in JSON format in our example. FuzzSlice uses the generated JSON files when compiling the code slice in Step 3.

Original source code can have complex structures such as functions which are generated dynamically using macros or preprocessor directives that allow multiple definitions of a function for different operating systems. To alleviate this, we use the first stage of compilation to preprocess the source code files. This is necessary because it makes the code easier for minimization while generating code slices in later steps. The preprocessed files are similar to original source code but stripped of all comments, have inline macros substituted, and preprocessor directives like `#ifdef` removed. We achieve this step by adding some flags to the compilation process (e.g., "`-save-temps`" flag). In the example of our running case in Listing 1, only the comment gets stripped after preprocessing.

**Step 2: Generate structural information of source code in XML format.** In this step, we aim to label AST-like high-level structural information within the entire project repository. This AST-like information will be used to search for code sections that represent relevant functions to add to minimized slice $S$ in the next step. To achieve this, we use srcML, a tool that provides an XML format for structural information of source code [28]. It is lightweight and highly scalable. Also, the output of srcML (XML-format) makes it easier when parsing and searching for C references (functions or variables) while constructing minimized compilable slice. FuzzSlice obtains the srcML of the preprocessed files (obtained in Step 1). The XML produced by srcML labels individual nodes that represent source code components (e.g., functions, declaration statements, for loops etc.). At the end of this step, we have XML equivalent of source code with all high level structural information labelled, which we use in Step 3.

**Step 3: Constructing minimized compilable code slice.** In this step, FuzzSlice obtains a compilable code slice that contains a given

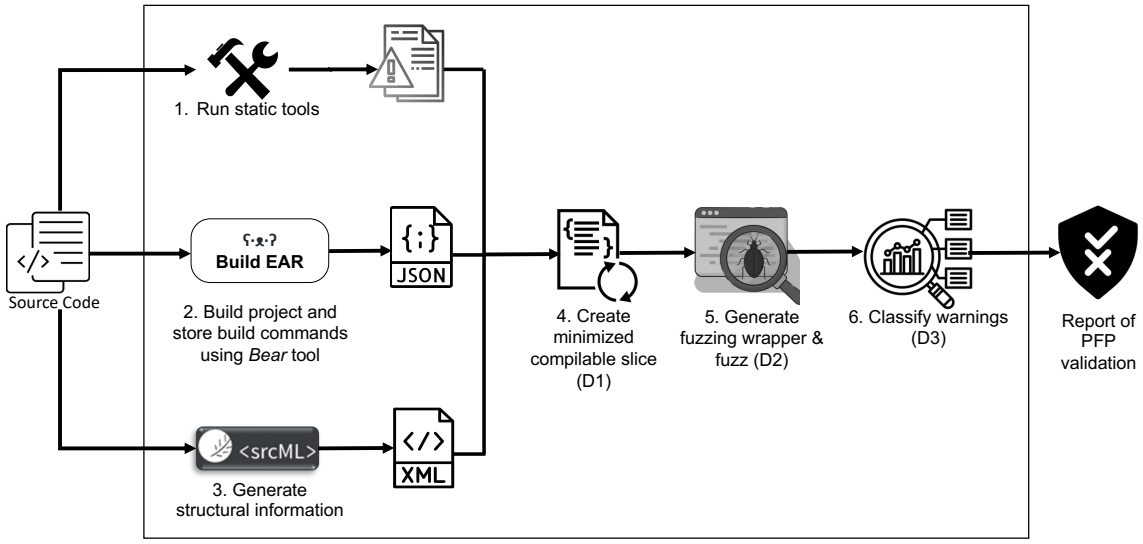


**Figure 1: Overview of the FuzzSlice technique.**

**Algorithm 1** Minimal Slice Creation

```
1:  Input: Function F containing the warning
2:  Output: Slice S
3:  procedure Slice(Dependency F)
4:      file ← EnclosingFile(F)
5:      filesrcML ← GetSrcML(file)
6:
7:      queue ← F                          ▷ File level breadth-first search
8:      for each c ∈ queue do
9:          if c ∈ GetFunctions(filesrcML) then
10:             S ← S ∪ c
11:             queue ← queue ∪ GetCallees(c, filesrcML)
12:             Pop c from queue
13:         end if
14:     end for
15:
16:     extDependencies ← CompileSlice(S)          ▷ Recursion
17:     for each c ∈ extDependencies do
18:         Slice(c)
19:     end for
20: end procedure
```

warning, i.e., the code slice that FuzzSlice generates comprises the entire function that encapsulates the warning and its dependencies in the same file as well as in other files. In the design section, we defined dependencies as function callees. In this section, we generalize the concept of dependency to a reference that includes functions, structs, or global variables required for successful code slice compilation. In our running example, `OPENSSL_malloc` is a dependency of the function `glue_strings`. The function `glue_string` exists in a file called `driver.c` while its dependency `OPENSSL_malloc` exists in another file called `alloc.c` in the openssl project.

Within the design section, we discussed properties (a), (b) and (c) of minimized slices in (D1). Here we enforce these different properties of the minimized slice. To enforce these properties, we utilize the stored build commands (from Step 1) and source code srcML outputs (from Step 2).

First, we enforce property (b) of the minimized slice in the design section (D1) by identifying the function $F$ enclosing the warning and adding it to slice $S$. For our example in Figure 1, the enclosing function is `glue_strings` which can be identified by parsing the srcML. This function is the first function added to the code slice.

Now, we try to enforce properties (a) and (c) in (D1) while creating minimized slice $S$. We recursively identify all other dependencies required by the function enclosing the warning in all files across the repository. We collect all the required dependencies (including callee functions) automatically over several iterations using compiler logs. To achieve that, we perform the following steps: (1) code minimization within the file, (2) Attempt to compile; if unsuccessful - identify other files containing the missing dependencies (3) Recurse until compilation in step (2) succeeds. We describe the pseudocode of our steps in algorithm 1. We explain the pseudocode as we describe each step in detail.

(1) *Code minimization within a file.* In this step, we retain all dependencies needed by a given function (e.g., `glue_strings`) within its file. To do this, we obtain the file containing the required function which initially is the file containing the warning. We automatically recurse the callees of the given function in a breadth-first search manner. We use srcML labelled nodes (from Step 2) to identify function calls within the given file. These dependencies are retained during the minimization process. Finally, we filter unused dependencies (not covered by the breadth-first search) which are not relevant to the code slice that FuzzSlice is trying to create in this file. We shall henceforth call this file the minimized file. Lines 4-14 in algorithm 1 create the minimized file through a breadth-first search before adding to the slice $S$. For the code in listing 1, the file

to be minimized is `driver.c`. Only the function `glue_strings` is retained within `driver.c`.

(2) *Attempt to compile.* We use the compile commands (stored by Build EAR in Step 2) for the given minimized file. Using these compile commands, we attempt to compile and obtain the object files. In the case that the compilation is successful, the compilable code slice is ready for the next step (i.e., generating a fuzzing wrapper). If the build fails, we automatically parse the compiler error logs to identify missing references. These missing references can be external dependencies in other files that we have not minimized yet. This step is shown in line 16 in algorithm 1, where the external dependencies are obtained from the compiler logs. In our running example, `OPENSSL_malloc` is a missing reference thrown as an error by the compiler as shown in Listing 2. Hence, FuzzSlice searches for this reference among other files in the srcML representation of the repository and locates it in another file `alloc.c`.

```
driver.c:12:19: error: implicit declaration of
    function 'OPENSSL_malloc' is invalid
  if (!(ret = p = OPENSSL_malloc(len + 1)))
```

**Listing 2: Compiler error requesting additional references.**

(3) *Recurse until compilation succeeds.* Previously, we attempted to compile the minimized file. Only when we fail, we recurse over new required references. In our running example, `OPENSSL_malloc` is the new required reference and `alloc.c` is the new file that must be minimized. This recursion step is shown in lines 17-19 in algorithm 1. We repeat (1) with `OPENSSL_malloc` as the required function and `alloc.c` as the file to be minimized. After this recursion, our running example will become a compilable code slice.

By the end of this step, we obtain all references within multiple minimized files required for the function enclosing the warning $F$. This is the minimized slice $S$ described in the design section. Additionally, we have also successfully compiled these files. Finally, we have a list of object files which will be linked along with a fuzzing wrapper in the next step.

**Step 4: Generate fuzzing wrapper.** In this step, we generate a fuzzing wrapper tailored to each function that contains a given static analysis warning. This step aims to generate versatile inputs to fuzz the minimized slice $S$ according to (D2). We require a fuzzing wrapper which is a piece of code that will correctly initialize the arguments to this function and all its fields (based on type) so that we can reach the warning through fuzzing.

To create the fuzzing wrapper, we write a Python script that looks at the argument type and initializes it correctly depending on the argument type. For primitive C types (eg. char, int, bool, double etc.), the fuzzing wrapper handles each case in a unique way that appropriately fuzzes them. Within our example, the function `glue_strings` has only one function argument `char** list` (list of strings) that needs to be fuzzed. The generated fuzzing wrapper for the example is shown in Figure 3. It has two inputs `Fuzz_Data` which is the fuzzing bytes and `Fuzz_Size` which is the length of these fuzzing bytes. The fuzzing wrapper uses these fuzzing bytes to randomly initialize the argument `list`. For this purpose on line 7 the first few fuzz bytes are read which is used to split the fuzz bytes into chunks on line 10. The wrapper then iterates in a for loop allocating all the strings on line 16 and copying the fuzz bytes on line 18. This fuzzed function argument `list` is then passed to the function `glue_strings`.

There can also be other cases where user-defined structs or objects are passed as arguments to the function to be fuzzed. These objects can have their own fields which must be correctly initialized. In this case, we use GDB - a debugger for C [29] to resolve the object into the C primitive types automatically. In the fuzzing wrapper, FuzzSlice initializes the fields appropriately within the object and finally fuzzes only the primitive types.

FuzzSlice has now generated a fuzzing wrapper for the code slice that finally calls the function enclosing the warning. Once the fuzzing wrapper is in place, the fuzzing wrapper is compiled. Then all the object files from Step 3 and the fuzzing wrapper are linked together using the link commands from Build EAR stored in Step 1. We then use ASAN [30] as an oracle which crashes the program during stack and heap buffer overflows during the fuzzing process. At the end of this step, we have a compiled binary ready to be fuzzed. We then use libfuzzer [25] for the purpose of fuzzing.

```
int LLVMFuzzerTestOneInput(uint8_t* Fuzz_Data, size_t
     Fuzz_Size)
{
    uint8_t * pos = Fuzz_Data;
    // Use fuzz bytes to find no. of strings
    char  **list;
    size_t num_ptr;
    memcpy(num_ptr, pos, sizeof(size_t));
    num_ptr = 1 + abs(num_ptr) % Fuzz_Size;
    // Find length of each string from fuzz bytes
    size_t str_size = Fuzz_Size/num_ptr;
    // Allocate pointers first
    list = malloc(num_ptr * sizeof(char*));
    for (int i=0; i< num_ptr; i++ )
    {
        // Allocate string
        list[i]= malloc(str_size);
        // Copy fuzzed characters
        memcpy(list[i], pos, str_size);
        pos += str_size;
    }
    // Call target function
    glue_strings(list);
    //Free allocated variables after this
}
```

**Listing 3: Fuzzing wrapper.**

**Step 5: Classify warnings.** This step is aimed at tackling the classification of the warnings after fuzzing the minimal slice $S$ (D3). At this stage of our approach, each static analysis warning has its own binary which is compiled and linked. FuzzSlice fuzzes each binary after which the llvm-coverage [25] is obtained to show the number of times each line is executed during fuzzing. We classify the output of fuzzing the binary in only one of four states as follows:

(1) There is at least one crash/buffer overflow at warning location - **Crash (C)**
(2) There is no crash/buffer overflow at the warning location, but the line is executed - **Possible False Positive (PFP)**
(3) The warning line is not executed - **Not Reachable (NR)**
(4) The slice is not compiled - **Not Compiled (NC)**

When there is no crash or buffer overflow at the warning location but the line is executed according to coverage, then we can predict that the warning has a high chance of being a false positive. This is because a caller of the function enclosing the warning can only constrain the function argument values compared to the fuzzer. However, in the case of a crash, we cannot confirm that the warning is a true positive because the caller function may invalidate the crashing input. Similarly, if a given line is not executed according to coverage or if a code slice fails to compile, we cannot say anything about the warning.

**Novelty of the approach:** The novelty of the FuzzSlice approach lies in its ability to generate compiled slices for fuzzing. This means that for a static analysis warning anywhere in the repository, the FuzzSlice framework automatically identifies the required dependencies of the enclosing function, compiles, and links the slice with the correct compiler options. We generate a unique binary aimed at covering each warning. We believe this is a novel idea within the FuzzSlice framework especially when combined with fuzzing wrapper generation to prune possible false positives.

In the next section, we evaluate the FuzzSlice approach and discuss resulting classes, with a focus on minimizing false positives. FuzzSlice aims to assist developers in efficiently de-prioritizing false positives without extensive manual effort.

# 4 EVALUATION

In this section, we evaluate our proposed approach. In particular, we aim to answer the following research questions in our evaluation:

- **RQ1.** How many PFPs can FuzzSlice confirm on a synthetic dataset?
- **RQ2.** How many PFPs can FuzzSlice confirm on a real-world project dataset?
- **RQ3.** How does FuzzSlice perform in terms of coverage, warning executions and compilation for PFP warnings?

In order to answer the above research questions, we first introduce our evaluation setup in Section 4.1. Then, we present our evaluation results of each research question in Section 4.2, Section 4.3, and Section 4.4.

## 4.1 Evaluation Setup

In this section, we describe the static analysis tools and the benchmarks used to evaluate FuzzSlice.

**Static analysis tools.** We use two static analysis tools, RATS [23] and Infer [26]. RATS is an open-source tool that utilizes a vulnerability database to flag similar code as a warning. RATS detects buffer overflows and race conditions. The second tool Infer is developed and used internally by Meta. Infer performs abstract interpretation that reasons about mutations to computer memory to detect buffer overflows and null dereferences. Both of these tools output warnings at different severity levels. We use the "High" and "Medium" severity of warnings for RATS and L1 and L2 severity of warnings for Infer because they are the most faithful warnings for these tools. Both static analysis tools provide the warning at the line level within a given file. For our evaluation benchmark, we obtain two sets of warnings, one from each tool.

**Datasets.** We use two datasets to evaluate FuzzSlice. We are interested in pruning false positive buffer overflow warnings in both datasets. First, we use a synthetic benchmark called *Juliet test suite* [31]. We run FuzzSlice on Juliet test suite v1.2 for C/C++, which is a benchmark created by the US National Security Agency (NSA) specifically for assessing the capabilities of static analysis tools. The benchmark labels each possible warning location with comments to indicate that the warnings are true or false positives.

Second, we evaluate FuzzSlice on three real world open-source repositories, namely, *openssl, tmux, and openssh-portable* [21, 32, 33]. The selected packages for evaluation represent various domains. Openssl is a robust, commercial-grade toolkit for the Transport Layer Security (TLS) protocol. Tmux is an open-source multiplexer for Unix-like operating systems. Using tmux, multiple terminal sessions can be accessed in a single window. Openssh is the primary connectivity tool for remote connectivity through ssh protocol eliminating eavesdropping and hijacking. Table 1 presents descriptive statistics on the selected packages. The data shows that these repositories are actively maintained and have a large number of lines of code. The following are the git versions of each repository used for the analysis: openssl (894f2166ef), tmux (70ff8cfe), and openssh-portable (5f93c483).

In selecting datasets for our evaluation, we opted to consider both synthetic and open-source repositories. Our reasoning for considering both synthetic and open-source repositories stems from the observation that synthetic benchmarks, such as Juliet, tend to yield a higher ratio of true positives to false positives, making them an effective means of assessing the performance of FuzzSlice on a large number of true positives. Additionally, a synthetic dataset provides ground truth, which can be used to *objectively* evaluate the effectiveness of FuzzSlice. In contrast, static analysis warnings obtained from open-source repositories are likely skewed toward false positives. Hence, we believe that by considering both synthetic benchmarks and open-source repositories, we can obtain a more comprehensive and reliable evaluation of FuzzSlice performance.

**FuzzSlice Configuration.** To evaluate the warnings produced by the static analysis tools, we subject each warning to a fuzzing process lasting five minutes. Our fuzzing procedure was conducted on a Headless Server equipped with a powerful hardware configuration consisting of 64 cores of Intel Xeon Gold 6226R processors, operating at a speed of 3.900GHz, and 128GB of RAM on Ubuntu 20.04 LTS. It is worth noting that certain warning locations may only be compilable on specific operating systems as defined by preprocessor directives. As a result, FuzzSlice may not be able to generate a compilable code slice for these warning locations due to the lack of build information. Therefore, we excluded such warnings from our analysis.

**Table 1: Statistics on the three project repositories in our benchmark.**

| Repository | Lines of code | Latest commit |
|---|---|---|
| openssl | 450,982 | 29/03/2023 |
| tmux | 106,528 | 15/03/2023 |
| openssh-portable | 60,387 | 29/03/2023 |

**Table 2: FuzzSlice performance on Juliet test suite.**

| Ground Truth | Total | PFP | C | NR | NC |
|---|---|---|---|---|---|
| True positive | 1,059 | 20 | 1,039 | 0 | 0 |
| False positive | 864 | 864 | 0 | 0 | 0 |
| Total | 1,923 | 884 | 1,039 | 0 | 0 |

## 4.2 RQ1: Juliet Test Suite

The goal of evaluating FuzzSlice on a synthetic data set (which provides ground truth) is to validate the following foundational insights behind FuzzSlice- a false alarm by a static analysis tool should not be flagged by the dynamic checker in FuzzSlice while a true bug can and is likely to be caught by the dynamic checker in FuzzSlice as well.

Our evaluation involved the use of version 1.2 of the Juliet test suite, which provides ground truth for all static analysis warnings, thereby facilitating their classification as either true or false positives. We run RATS and Infer over the Juliet dataset, resulting in a total of 1,923 unique static warnings which comprises of 864 false positives and 1,059 true positives. Subsequently, these warnings were subjected to fuzzing using FuzzSlice.

Table 2 shows that FuzzSlice identified all of the 864 false positives within the static analysis warnings. It is worth noting that FuzzSlice was able to execute all of the warning lines classified as false positives without observing any crashes. This suggests that FuzzSlice has the potential to effectively prune false positives in a static analysis report.

In addition, FuzzSlice was able to compile and execute all of the static analysis warnings, including 1,059 true positives that suggest possible buffer overflow vulnerabilities. Of the total 1,059 warnings labelled as true positives, indicating a possible buffer overflow at the warning line number, FuzzSlice was able to crash the warning line in 1,039 cases. However, there were only 20 instances in which FuzzSlice was unable to crash the warning line due to the involvement of global variables. In these 20 instances, FuzzSlice wrongly classified them as false positives. Since FuzzSlice does not currently fuzz these global variables, default values with which they are initialized were used instead. We discuss more about this in Section 6.

**Summary of RQ1:** We find that FuzzSlice is able to compile minimized slices for all warnings in the Juliet dataset. Out of 864 false positive warnings in the Juliet dataset, FuzzSlice confirms all of them by executing the warning without observing any crashes.

**Table 3: FuzzSlice performance on the three studied open source repositories.**

| Repository | Tool | Total | PFP | C | NR | NC |
|---|---|---|---|---|---|---|
| openssl | RATS | 30 | 21 | 1 | 8 | 0 |
| | Infer | 163 | 88 | 18 | 45 | 12 |
| tmux | RATS | 5 | 4 | 1 | 0 | 0 |
| | Infer | 18 | 6 | 2 | 10 | 0 |
| openssh-portable | RATS | 20 | 6 | 2 | 10 | 2 |
| | Infer | 29 | 18 | 1 | 3 | 7 |
| Total | | 265 | 143 | 25 | 76 | 21 |

## 4.3 RQ2: Real-world Dataset Verified by Developers

We evaluate FuzzSlice on real-world open source projects with the goal of assessing practicality of FuzzSlice in handling large codebases. We also would like to see if developers agree with the labelling provided by FuzzSlice.

We evaluate FuzzSlice on three popular open-source repositories: openssl, openssh-portable and tmux [21, 32, 33]. Our results are presented in Table 3. As shown in the table, FuzzSlice was able to prune 143 instances of possible false positives (PFP) (54%) out of 265 warnings. This means that FuzzSlice executed 143 static analysis warnings without any observed crashes at the warning location. Additionally, FuzzSlice detected 25 crashes (9.4%) at the warning location out of the 265 warnings. It should be noted that these crashes may be false positives if the callers of the enclosing function invalidate the crashing inputs. However, these warnings can still be useful for developers to prioritize for further manual triage. Lastly, we point out that FuzzSlice detected 76 warnings that were not reachable, and encountered 21 warnings (8%) for which it could not generate a compiled slice. The relatively low number of not compiled cases (i.e., 21) is a testament to FuzzSlice ability to minimize complex code with features such as macros, function pointers, and structs. This demonstrates that FuzzSlice is capable of minimizing and compiling a wide range of real-world, complex C code. We delve into the reasons behind the not reachable and not compiled cases in Section 6.

**Verification by developers.** Since there is no readily available ground truth for the warnings in the open-source repositories we studied, we reach out to core developers across all three repositories to obtain a ground truth on these warnings. We are also interested to see if developers agree with the labelling provided by FuzzSlice.

To achieve this, we first parse the git logs on the files containing the warnings we obtained from the three repositories. Specifically, we identify the developers who modified these warning lines the most along with their email IDs. We then send emails to three developers (one from each repository) and received responses from all three developers. The developers of tmux and openssh-portable agreed to collaborate with us. However the developer of openssl let us know that they would be unable to collaborate with us as it would take time away from their core development work.

We provide each developer with the static analysis warning reports we obtained earlier from both RATS and Infer, and then

**Table 4: Developer labels of warnings from the three open source repositories.**

| Repository | FuzzSlice label | Total | Developer label | | |
|---|---|---|---|---|---|
| | | | FP | TP | Ambiguous |
| tmux | PFP | 10 | 9 | 0 | 1 |
| | C | 3 | 1 | 0 | 2 |
| | NR | 10 | 3 | 1 | 6 |
| openssh-portable | PFP | 24 | 24 | 0 | 0 |
| | C | 3 | 3 | 0 | 0 |
| | NR | 13 | 13 | 0 | 0 |

ask them to review each static analysis warning and classify it as a True Positive (actual warning), False Positive (not a vulnerability), or ambiguous (if they are uncertain about it). We also give the developers the option to provide a reason for their decision.

It is worth noting that we opted not to send the warnings that were not compiled by our approach to the developers, as we recognized that such warnings would not contribute towards the analysis and could potentially waste the valuable time of open-source developers. Furthermore, we refrain from sharing the results obtained from FuzzSlice, as this could potentially bias the opinion of developers when classifying the warnings, and thereby introduce an unwanted element of subjectivity into the analysis. Finally, it is important to note that for the openssh-portable project, the developer identified by our git log approach had invited another developer, as both were heavily involved in implementing the code flagged by these static analysis tools.

We present the results of our developer label analysis in Table 4. The table shows the labels assigned to the warnings by FuzzSlice (on the left side) and the labels assigned by the developers (on the top side). As shown in Table 4, the openssh-portable developers were able to confirm that all the possible false positives detected by FuzzSlice across both static analysis tools were indeed false positives. This outcome highlights the accuracy and reliability of our approach. Table 4 also reveals that the developer from tmux was able to confirm that 9 out of the 10 possible false positives detected by FuzzSlice were indeed false positives. However, for the remaining warnings, the developer labeled them as ambiguous because they reported that such cases require a precise call stack to analyze the warning further, as multiple callers can call the function. Overall, our analysis shows that the developers' labels largely matched with the FuzzSlice classification of warnings as PFPs.

Table 4 also reveals that the developers within the tmux repository labelled 9 warnings as ambiguous because they lacked sufficient context within large encryption-related functions. Interestingly, of these 9 ambiguous warnings, FuzzSlice labelled 6 warnings as unreachable. This is due to the inability of FuzzSlice to trigger the warnings within these large functions after 5 minutes of fuzzing, partially indicating agreement between ambiguous and not reachable warnings in the tmux repository. Furthermore, the developer from tmux reported one warning with undefined behavior, which could potentially result in a buffer overflow. However, FuzzSlice was unable to execute that warning within tmux and classified it as not reachable.

**Table 5: Evolution of possible false positives over time.**

| Repository | # PFP (3 years ago) | # persistant PFP |
|---|---|---|
| openssl | 55 | 55 |
| tmux | 11 | 8 |
| openssh-portable | 20 | 20 |

For most warnings that were labeled as false positives in tmux and openssh-portable, the developer supported their label with a rationale. For example, the developers reported that in most false positives, the arguments passed to libc functions are within the correct bounds. Also, they mentioned that the index variable could not exceed buffer size within several loops. In other cases, bounds checking happens close to the warning, which prevents buffer overflow from occurring. Finally, developers in openssh-portable also mentioned that they use third-party library calls from OpenBSD that are known to be completely safe.

**Summary of RQ2:** We find that FuzzSlice is able to compile minimized slices for 244 out of 265 warnings in the 3 open source repositories. In the tmux and openssh-portable repositories, FuzzSlice was able to identify 33 out of 53 false positives confirmed by the developers.

## 4.4 RQ3: Coverage, Warning Executions and Compilation of FuzzSlice

Sections 4.2 and 4.3 presents the overall effectiveness of FuzzSlice with both synthetic and real-world case studies. In this section, we seek to provide a better understanding of FuzzSlice from more fine-grained aspects. Specifically, we examine FuzzSlice from three perspectives: coverage, number of executions on warning, and compile time.

**Coverage.** Code coverage is important to analyze because it indicates how much of the code within a given minimized code slice has been exercised during the fuzzing process. A higher code coverage means that more parts of the code have been exercised, which in turn increases the chances of confirming both true bugs and false alarms in static analysis reports.

We first evaluate the code coverage of the PFP code slices. Note that we obtain the coverage of the minimized slice in Step 5 of Section 3.2. The ratio of the number of lines that are executed to the total number of lines is the code coverage percentage. Figure 2(a) shows the code slice coverage across the examined benchmarks. Note that this coverage is the coverage within the constructed code slice, not the coverage over the entire repository. Among the four datasets we examined, openssh and juliet showed the highest code coverage, with a median of 100%. tmux was not far behind, with a median of 84% code coverage. However, in the case of openssl, the median code coverage is close to 30%, which is lower than other repositories. In fact, this is because some static analysis warnings in openssl are within functions that exercise whole modules within openssl, creating larger slices, which require

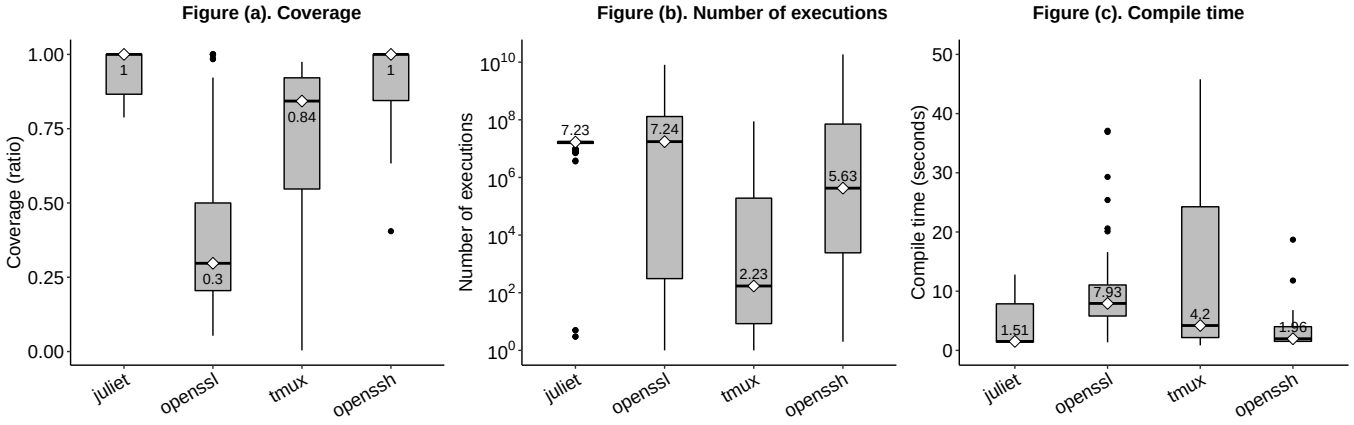


**Figure 2: Coverage, number of executions on warning and compile time of minimized code slices.**

extra time to achieve better coverage (recall that we limit the fuzzing time to five minutes).

We also compute the slice size in the form of the total number of lines of code in the minimized slice (LOC). A higher slice size usually implies more dependencies needed to compile the function enclosing the warning. Overall, we find that the size of slices varies depending on the examined project dataset and dependencies of the function enclosing the warning. For example, a minimized slice in the tmux project has a median of 7,691 LOC. However, the number is much lower for juliet, openssh, and openssl. The slice sizes (on median) in these projects are 58, 495, and 431, respectively.

**Frequency of warning hit.** We analyze the frequency with which the warning line was executed for PFP. By examining the number of executions for a warning line, we gain valuable insights into the likelihood that the warning may be a false positive. If a warning line is executed frequently without triggering a crash, this strongly suggests that it may not be indicative of an actual vulnerability. This is because the number of executions is closely tied to the diversity of input values evaluated by the fuzzer. This information can be obtained through the coverage information which is collected for fuzzing anyway. This allows us to track the number of times a given line of code is executed during testing.

Figure 2(b) presents the median number of executions for PFP warnings across each benchmark on a logarithmic scale. Our analysis reveals that the median PFP warning was executed approximately 72.3 million times for juliet benchmark, 72.4 million times for openssl benchmark, 223 times for tmux benchmark, and 563 thousand times for openssh benchmark, respectively. The executions on the warning vary depending on the warning location in the code, e.g., if they are surrounded by guard conditions, within for loops, etc. Our examination of code coverage and the number of executions on warning lines without crashes indicates that many of these warnings may indeed be false positives. The high levels of code coverage and the number of executions strongly indicate that these warnings do not correspond to actual vulnerabilities.

**Performance.** To further assess the runtime performance of FuzzSlice, we conduct an evaluation of the time taken to generate the minimized code slices. Specifically, we measured the time required by FuzzSlice to create and compile each complete minimized slice for every warning. Figure 2(c) presents boxplots of the compile time in FuzzSlice for each benchmark. Our analysis reveals that code slices can be compiled within a range of 1.51-7.93 seconds. Notably, we did observe some outliers in the case of openssl, where functions belonging to multiple modules were searched and compiled, resulting in compile times of up to 80 seconds. However, overall, our results indicate that FuzzSlice is fast at constructing minimized code slices across all datasets.

Overall, these findings show that FuzzSlice is an effective tool for pruning possible false positives in static analysis warnings. By quickly generating minimized code slices, FuzzSlice can help developers and security professionals prune and mitigate PFP more efficiently. The process of constructing these slices takes on average less than 8 seconds, making FuzzSlice a valuable automatic approach for optimizing manual triage efforts.

> **Summary of RQ3:** The median minimized slice coverage of FuzzSlice across all 3 repositories for PFPs is 92.26%. The median execution on PFP warnings across all 3 repositories is 69.87 million times. FuzzSlice is also able to compile minimal slices for most warnings under 8 seconds.

## 5 RETROSPECTIVE ANALYSIS OF PFP

Although we were unable to obtain a ground truth for openssl from its developers, we can still evaluate the accuracy of FuzzSlice by analyzing the evolution of PFP detected over time. This technique was inspired by the work of Di Penta et al. [34] and Aloraini et al. [35], who observed that warnings that persist in the same code for long periods without being removed are possible false positives. The basic idea of the technique is that if a warning persists in the same code segment across multiple versions of the software and over a long time, then it is less likely to be a genuine vulnerability as it was not considered worth removing under any circumstances.

To provide a comprehensive evaluation of our approach, we conduct an analysis not only on the openssl repository, but also

on the two other open-source repositories included in our evaluation. Specifically, we consider a version of each repository that was 3 years old (i.e., the latest commit was made before January 1st, 2020), with the following git versions: openssl (5f95fbf399), tmux (566ab9aa), and openssh-portable (c4b2664b). We run RATS and Infer on these older versions of the repositories to flag buffer overflow warnings, and then use the FuzzSlice technique to prune all possible false positives (PFP) among these warnings. Next, we attempt to match these warnings with warnings in a more recent version of the repository. Specifically, we focus on the following two criteria: **(i)** the static analysis warning line in the older version (before 3 years) is identified as a possible false positive through FuzzSlice technique, and **(ii)** the warning line is still flagged by the respective static analysis tool in a recent version of the repository.

Table 5 presents the results of the evolution of the examined warnings. The results indicate that for the openssl repository, all of the possible false positives are still flagged by the static analysis tool after 3 years. This suggests that there is a high likelihood that all of these warnings are indeed false positives. Similarly, for the tmux repository, 8 out of 11 PFP are still flagged after 3 years. In the case of openssh-portable, all 20 warnings matched both criteria. These findings support the efficacy of our approach in pruning false positives in static analysis warnings, and highlight the importance of considering the longevity and persistence of warnings in assessing their validity. These results also indicate that FuzzSlice has the potential to help developers to deprioritize several such warnings for manual triage.

We found that in three cases, warnings that were identified as false positives in the 3-year-old version of the repository could not be matched to any warnings in the recent version. This occurred because, in two out of three cases, the code containing the warning had been deleted from the repository. The commits involved in deleting such code were not related to buffer overflow bugs. In the remaining case, the warning line had been modified, resulting in the static analysis warning being removed. This was due to a new feature in tmux that replaced internal representation of strings from UTF-8 to wide characters which modified the library call involved in this warning.

## 6 LIMITATIONS AND FUTURE WORK

In Section 4, we report that we obtained 21 outlier cases that were not compiled. This was mainly due to the fact that FuzzSlice relies on srcML to parse C code. SrcML may misparse code and produce incorrect XML output. For example, srcML cannot correctly label code that contains inline assembly language within openssl. This is because srcML uses a grammar to parse the code and inline assembly language is not integrated into this grammar. As a result, errors in the srcML output can lead to required dependencies not being resolved and the slice not being created. Despite these limitations of srcML, it has been adopted by several previous works [36, 37].

Also, we reported that 76 warnings were not reachable, meaning that the line could not be executed. This can be due to certain constraints on input, the requirement of external files, etc. Several of these cases, especially in openssl and openssh-portable, contain function pointers as arguments within the code slice. However, FuzzSlice currently does not fuzz function pointers. A possible way to address this limitation in future work is by first identifying all functions with matching signatures and return values that can be assigned to the function pointer. Future work should also consider using a more efficient approach to refine indirect call targets [38].

Another limitation of FuzzSlice is related to fuzzing global variables. Currently, FuzzSlice only provides a default initialization for global variables and does not mutate them, which led to the misclassification of 20 cases in the Juliet test suite as possible false positives. However, despite not mutating function pointers and global variables, FuzzSlice is still capable of minimizing code that requires these components in their minimized slice.

**Future work: fuzzing global variables.** In this work, we limit our techniques to provide a default initialization for global variables. One potential extension for future work is to identify all global variables involved in the minimized slice and mutate them within the fuzzing wrapper in a similar way as function argument mutation.
**Future work: finer-grained slicing.** An interesting direction to explore would be to further reduce the size of the program slice for fuzzing. In FuzzSlice, the entire function enclosing the vulnerability is considered for fuzzing. As future work, we plan to construct intra-function slices that minimally enclose the static analysis warning (both in terms of control-flow and data-flow). This can further ease the cost of fuzzing in pruning possible false positives.
**Future work: supporting a diverse set of static analysis tools.** Similar to related work [10, 39], we illustrate FuzzSlice on a specific bug pattern: buffer overflow vulnerabilities. However, the concept of FuzzSlice can be extended to support static analysis tools that target different types of bugs, including but not limited to integer overflow, null-pointer dereference, use-after-free, dead code elimination, and even semantic and logic bugs. To facilitate false alarm filtering on these types of bugs, we only need to replace the oracle that detects the violation at runtime (e.g., UBSAN is an oracle for integer overflows [40]) within the FuzzSlice framework.

## 7 RELATED WORK

A common approach to fuzzing has been to fuzz independent submodules, drivers or libraries separately. There is a rich literature focused on fuzzing independent libraries such as Transport Layer Security (TLS), deep learning, C/C++ libraries [16–18]. For example, Corina et al. [41] proposed fuzzing for kernel drivers effectively finding bugs within them. FuzzSlice differs from such approaches since it involves fuzzing at the function level for any arbitrary warning location, aiming to prune possible false positives.

There exists a rich literature on directing fuzzing towards a given location [10, 14, 42–45]. The core idea behind such methods is to mutate inputs that are closer to reaching the target location. The main difference between FuzzSlice and directed fuzzing is that FuzzSlice does not use `main` method as the entrypoint. Instead, FuzzSlice creates the minimal slice enclosing the warning first and then we confine the state space exploration within the slice. In fact, their techniques to direct input mutation towards a certain location are orthogonal to our approach and can be used as a complementary technique within FuzzSlice.

Fuzzing has been applied to binary-level code slices as well. For instance, Chen et al. [19] implemented fuzzing on independent code

snippets extracted from real-time operating systems (RTOS) binaries. In contrast, FuzzSlice takes a different approach by creating code slices at the source code level instead of the binary level. By generating fully compiled slices, FuzzSlice identifies possible false positives in static analysis reports, eliminating the need to deal with unstable fuzzing caused by incomplete context.

The work closest to FuzzSlice utilizes symbolic execution. For example, Engler et al. [46] proposed the idea of under-constrained symbolic execution (UC-KLEE). UC-KLEE takes an arbitrary function and symbolically executes it without initializing any of its data structures and without doing any environment setup, with the goal of finding quality bugs in drivers within the Linux operating system. While program slicing helps to reduce the search space of symbolic execution, UC-KLEE still suffers from other sources of path explosion and imprecision such as unbounded loops, pointer arithmetic, memory modeling, and invocation of library functions. FuzzSlice, on the other hand, is a dynamic analysis tool at its core and does not suffer from the above-mentioned limitations. Instead, FuzzSlice is subject to a different set of limitations such as code coverage and effectiveness of mutation strategies.

Kallingal et al. [47] generate code slices aimed at identifying true bugs through the Helium framework. Helium can work only with certain static analysis tools - those that can provide a list of several statements leading to a given warning. In the Helium approach, the slice considered is a least common ancestor subtree of the parse tree over the full path leading to the warning. FuzzSlice on the other hand does not make any assumptions about the static analysis tool and does not rely on the accuracy of the static analysis tool in reporting paths leading to a warning. Additionally, since FuzzSlice aims to only prune false positives, the framework can disregard most of the program and directly fuzz only the function enclosing the warning. Furthermore, Kallingal et al. mention they are able to compile 68.5% of their code fragments. Since FuzzSlice compiles a smaller slice containing the enclosing function and its dependencies it compiled 244 out of 265 slices (92%) in open-source repositories and all the slices (100%) in the Juliet test suite. Such statistics from both works highlight the difficulty of generating compilable slices relevant to a warning and prove that the task is not trivial.

A plethora of work proposed approaches that utilized machine learning to reduce false positives in static analysis tools [48–51]. Hanam et al. [50] create a feature vector based on code characteristics at the site of each warning. The technique leverages machine learning techniques to build an actionable alert prediction model. Yedida et al. [51] proposed locally adjusting decision boundaries of models for actionable warnings to improve overall performance. FuzzSlice differs from such machine learning-based works since it actually dynamically executes the program to classify the warning.

Recent techniques such as [52–55] focused on improvements in automatic fuzzing driver generation, which is orthogonal to the FuzzSlice approach. FuzzSlice can benefit from the recent state of the art in this area. Tip et al. [56] surveyed algorithmic aspects of program slicing techniques. However, in FuzzSlice, the slice must also be compiled and linked into an executable with the correct compiler options, which requires storing relevant information of the build system. This increases the complexity of the slicing component within the FuzzSlice framework.

# 8 CONCLUSION

This paper introduces FuzzSlice, a framework that automates the pruning of false positives from static analysis tool warnings. We achieve this by fuzzing warnings at the function level, as it identifies non-crashing fuzzed warnings as potential false positives. The framework employs two steps: (1) creating a minimal compiled code slice containing any warning and (2) generating a fuzzing wrapper that performs type-based input generation for the enclosing function. Evaluation on synthetic and real-world C codebases demonstrates FuzzSlice's effectiveness. In the synthetic Juliet test suite, FuzzSlice identifies all 864 false positives (100%). In open-source repositories (tmux and openssh-portable), where developers independently labeled warnings from the static analysis tools, FuzzSlice identifies 33 potential false positives out of the 53 confirmed by developers (62.2%). Thus, FuzzSlice substantially reduces the effort required for developers to examine warnings. Additionally, in the Juliet test suite, 20 of the 884 possible false positives detected by FuzzSlice were actually true positives (2.2%). We were able to confirm that the incorrect classification in the Juliet test suite was due to our inability to fuzz global variables. In tmux and openssh-portable, of the 34 warnings we determined as possible false positives, 33 were confirmed as false positives by the developers, and 1 case was deemed as ambiguous (2.9%). These results validate the key insight of the FuzzSlice framework that a warning that does not yield a crash when fuzzed at the function level in a given time budget is a possible false positive.

# REFERENCES


[1] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, pp. 18–26.

[2] M. Alfadel, D. E. Costa, E. Shihab, and B. Adams, "On the discoverability of npm vulnerabilities in node. js projects," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, pp. 1–27, 2023.

[3] F. Cheirdari and G. Karabatis, "Analyzing False Positive Source Code Vulnerabilities Using Static Analysis Tools," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, pp. 10–13.

[4] M. Nadeem, B. J. Williams, and E. B. Allen, "High false positive detection of security vulnerabilities: a case study," in *ACM-SE '12: Proceedings of the 50th Annual Southeast Regional Conference*. New York, NY, USA: Association for Computing Machinery, Mar. 2012, pp. 359–360.

[5] H. J. Kang, K. L. Aw, and D. Lo, "Detecting false alarms from automatic static analysis tools: how far are we?" in *ICSE '22: Proceedings of the 44th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, May 2022, pp. 698–709.

[6] J. Park, I. Lim, and S. Ryu, "Battles with False Positives in Static Analysis of JavaScript Web Applications in the Wild," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, May 2016, pp. 61–70. [Online]. Available: https://ieeexplore.ieee.org/document/7883289

[7] B. Aloraini and M. Nagappan, "Evaluating State-of-the-Art Free and Open Source Static Analysis Tools Against Buffer Errors in Android Apps," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Sep. 2017, pp. 295–306.

[8] M. Christakis and C. Bird, "What developers want and need from program analysis: an empirical study," in *ASE '16: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: Association for Computing Machinery, Aug. 2016, pp. 332–343.

[9] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, May 2013, pp. 672–681.

[10] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed Greybox Fuzzing," in *CCS '17: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 2329–2344.

[11] M. Christakis, P. Müller, and V. Wüstholz, "Guiding dynamic symbolic execution toward unverified program executions," in *ICSE '16: Proceedings of the 38th International Conference on Software Engineering*. New York, NY, USA: Association

for Computing Machinery, May 2016, pp. 144–155.
[12] M. Böhme and B. Falk, "Fuzzing: on the exponential cost of vulnerability discovery," in *ESEC/FSE 2020: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, Nov. 2020, pp. 713–724.
[13] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection," in *2010 IEEE Symposium on Security and Privacy*. IEEE, pp. 16–19.
[14] G. Lee, W. Shim, and B. Lee, "Constraint-guided directed greybox fuzzing," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 3559–3576. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/lee-gwangmu
[15] X. Zhu, S. Liu, X. Li, S. Wen, J. Zhang, C. Seyit, and Y. Xiang, "DeFuzz: Deep Learning Guided Directed Fuzzing," *arXiv*, Oct. 2020.
[16] A. Wei, Y. Deng, C. Yang, and L. Zhang, "Free lunch for testing: fuzzing deep-learning libraries from open source," in *ICSE '22: Proceedings of the 44th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, May 2022, pp. 995–1007.
[17] J. Jang and H. K. Kim, "FuzzBuilder: automated building greybox fuzzing environment for C/C++ library," in *ACSAC '19: Proceedings of the 35th Annual Computer Security Applications Conference*. New York, NY, USA: Association for Computing Machinery, Dec. 2019, pp. 627–637.
[18] J. Somorovsky, "Systematic Fuzzing and Testing of TLS Libraries," in *CCS '16: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 1492–1504.
[19] L. Chen, Q. Cai, Z. Ma, Y. Wang, H. Hu, M. Shen, Y. Liu, S. Guo, H. Duan, K. Jiang, and Z. Xue, "SFuzz: Slice-based Fuzzing for Real-Time Operating Systems," in *CCS '22: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 485–498.
[20] "FuzzSliceICSE," Sep. 2023, [Online; accessed 11. Sep. 2023]. [Online]. Available: https://github.com/NobleMathews/FuzzSliceICSE
[21] openssl, "openssl," Jan. 2023, [Online; accessed 29. Jan. 2023]. [Online]. Available: https://github.com/openssl/openssl
[22] ——, "openssl," Jan. 2023, [Online; accessed 29. Jan. 2023]. [Online]. Available: https://github.com/openssl/openssl/blob/master/test/testutil/driver.c
[23] andrew d., "rough-auditing-tool-for-security," Jan. 2023, [Online; accessed 29. Jan. 2023]. [Online]. Available: https://github.com/andrew-d/rough-auditing-tool-for-security
[24] "american fuzzy lop," Mar. 2023, [Online; accessed 10. Mar. 2023]. [Online]. Available: https://lcamtuf.coredump.cx/afl
[25] "libFuzzer – a library for coverage-guided fuzz testing. — LLVM 17.0.0git documentation," Jan. 2023, [Online; accessed 29. Jan. 2023]. [Online]. Available: https://llvm.org/docs/LibFuzzer.html
[26] C. Calcagno and D. Distefano, "Infer: An Automatic Program Verifier for Memory Safety of C Programs," in *NASA Formal Methods*. Berlin, Germany: Springer, 2011, pp. 459–465.
[27] rizsotto, "Bear," Jan. 2023, [Online; accessed 30. Jan. 2023]. [Online]. Available: https://github.com/rizsotto/Bear
[28] M. L. Collard, M. J. Decker, and J. I. Maletic, "srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration," in *2013 IEEE International Conference on Software Maintenance*. IEEE, pp. 22–28.
[29] R. Stallman, R. Pesch, S. Shebs *et al.*, "Debugging with gdb," *Free Software Foundation*, vol. 675, 1988.
[30] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," 2012.
[31] T. Boland and P. E. Black, "Juliet 1.1 C/C++ and Java Test Suite," *Computer*, vol. 45, no. 10, pp. 88–90, Oct. 2012.
[32] openssh, "openssh-portable," Jan. 2023, [Online; accessed 30. Jan. 2023]. [Online]. Available: https://github.com/openssh/openssh-portable
[33] tmux, "tmux," Jan. 2023, [Online; accessed 30. Jan. 2023]. [Online]. Available: https://github.com/tmux/tmux
[34] M. D. Penta, L. Cerulo, and L. Aversano, "The life and death of statically detected vulnerabilities: An empirical study," *Information and Software Technology*, vol. 51, no. 10, pp. 1469–1484, Oct. 2009.
[35] B. Aloraini, M. Nagappan, D. M. German, S. Hayashi, and Y. Higo, "An empirical study of security warnings from static application security testing tools," *Journal of Systems and Software*, vol. 158, p. 110427, Dec. 2019.
[36] N. D. Q. Bui, Y. Yu, and L. Jiang, "InferCode: Self-Supervised Learning of Code Representations by Predicting Subtrees," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, May 2021, pp. 1186–1197.
[37] ——, "Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations," in *SIGIR '21: Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. New York, NY, USA: Association for Computing Machinery, Jul. 2021, pp. 511–521.
[38] K. Lu and H. Hu, "Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis," in *CCS '19: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 1867–1881.
[39] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *CCS '13: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. New York, NY, USA: Association for Computing Machinery, Nov. 2013, pp. 511–522.
[40] "UndefinedBehaviorSanitizer — Clang 17.0.0git documentation," Jun. 2023, [Online; accessed 27. Jun. 2023]. [Online]. Available: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html
[41] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "DIFUZE: Interface Aware Fuzzing for Kernel Drivers," in *CCS '17: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 2123–2138.
[42] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, "BEACON: Directed Grey-Box Fuzzing with Provable Path Pruning," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2022, pp. 36–50.
[43] V. Wüstholz and M. Christakis, "Targeted Greybox Fuzzing with Static Lookahead Analysis," *arXiv*, May 2019.
[44] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a Desired Directed Grey-box Fuzzer," in *CCS '18: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 2095–2108.
[45] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding Kernel Race Bugs through Fuzzing," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2019, pp. 754–768.
[46] D. Engler and D. Dunbar, "Under-constrained execution: making automatic code destruction easy and scalable," in *Proceedings of the 2007 international symposium on Software testing and analysis*, 2007, pp. 1–4.
[47] A. Kallingal Joshy, X. Chen, B. Steenhoek, and W. Le, "Validating static warnings via testing code fragments," in *ISSTA 2021: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, Jul. 2021, pp. 540–552.
[48] J. Yoon, M. Jin, and Y. Jung, "Reducing False Alarms from an Industrial-Strength Static Analyzer by SVM," in *APSEC '14: Proceedings of the 2014 21st Asia-Pacific Software Engineering Conference - Volume 02*. USA: IEEE Computer Society, Dec. 2014, pp. 3–6.
[49] U. Koc, P. Saadatpanah, J. S. Foster, and A. A. Porter, "Learning a classifier for false positive error reports emitted by static code analysis tools," in *MAPL 2017: Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. New York, NY, USA: Association for Computing Machinery, Jun. 2017, pp. 35–42.
[50] Q. Hanam, L. Tan, R. Holmes, and P. Lam, "Finding patterns in static analysis alerts: improving actionable alert ranking," in *MSR 2014: Proceedings of the 11th Working Conference on Mining Software Repositories*. New York, NY, USA: Association for Computing Machinery, May 2014, pp. 152–161.
[51] R. Yedida, H. J. Kang, H. Tu, X. Yang, D. Lo, and T. Menzies, "How to Find Actionable Static Analysis Warnings: A Case Study With FindBugs," *IEEE Trans. Software Eng.*, vol. 49, no. 4, pp. 2856–2872, Jan. 2023.
[52] D. Babic, S. Bucur, Y. Chen, F. Ivancic, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, "FUDGE: Fuzz Driver Generation at Scale," *Google Research*, 2019. [Online]. Available: https://research.google/pubs/pub48314
[53] M. Zhang, J. Liu, F. Ma, H. Zhang, and Y. Jiang, "IntelliGen: automatic driver synthesis for fuzz testing," in *ICSE-SEIP '21: Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice*. IEEE Press, May 2021, pp. 318–327.
[54] K. K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "FuzzGen: automatic fuzzer generation," in *SEC'20: Proceedings of the 29th USENIX Conference on Security Symposium*. USA: USENIX Association, Aug. 2020, pp. 2271–2287.
[55] G. Fraser and A. Arcuri, "EvoSuite: automatic test suite generation for object-oriented software," in *ESEC/FSE '11: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. New York, NY, USA: Association for Computing Machinery, Sep. 2011, pp. 416–419.
[56] F. Tip, *A Survey of Program Slicing Techniques*. CWI (Centre for Mathematics and Computer Science), Jul. 1994.