The Journal of Systems & Software 193 (2022) 111450

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

VulSlicer: Vulnerability detection through code slicing[☆]

Solmaz Salimi, Mehdi Kharrazi [*]

S4Lab, Department of Computer Engineering, Sharif University of Technology, Tehran, Iran

ARTICLE INFO

Article history:
Received 8 September 2020
Received in revised form 11 July 2022
Accepted 15 July 2022
Available online 20 July 2022

Keywords:
Code slicing
Static analysis
Vulnerability detection

ABSTRACT

There has been a multitude of techniques proposed for identifying vulnerabilities in software. Forcing a program into a vulnerable state has become increasingly unscalable, given the size of the programs and the number of possible execution states. At the same time, techniques that are looking for vulnerability signatures are marred with weak and incomplete signatures. This is not to say that such techniques have failed to identify previously unknown vulnerabilities in the code. However, they have inherent weaknesses, which result in identifying vulnerabilities that are limited in type and complexity.

We propose a novel technique to extract succinct vulnerability-relevant statements representing the self-contained nature of vulnerabilities and reproduce the vulnerable behavior independently of the rest of the program. We also introduce an innovative technique to slice target programs and search for similar vulnerability-relevant statements in them. We developed VulSlicer, a prototype system capable of extracting vulnerability-relevant statements from vulnerable programs and searching for them on target programs at scale. Furthermore, we have examined four candidate open-source projects and have been able to identify 118 potential vulnerabilities, out of which 94 were found to be silently patched, and from the remaining reported cases, three were confirmed by obtaining a CVE designation.

1. Introduction

Software vulnerabilities are the root cause of most security breaches while detecting them in a complicated program is challenging. The dramatic increase in the number of detected vulnerabilities (CVE, 2019) suggests that while newer vulnerability types are being discovered, e.g., speculative buffer overflows (Kiriansky and Waldspurger, 0000), known vulnerability types, e.g., simple buffer overflows like `strcpy` misuse, remain prevalent in common software. A good example of a recurring yet straightforward vulnerability is CVE-2019-1663, a simple `strcpy` buffer overflow discovered in the web-based management interface of multiple Cisco devices, including popular SOHO routers, which resulted in unauthorized access to at least 12 000 devices online (Rapid7 report, 2022).

Even though many automated techniques have been proposed (Cha et al., 2012; Jang et al., 2012; Newsome and Song, 2005; Perl et al., 2015; Stephens et al., 2016; Yamaguchi et al., 2014, 2011, 2015) to help with identifying security vulnerabilities, with today's growing software complexity, there are still many remaining challenges. While some formal approaches focus on proving the *absence* of certain types of vulnerabilities, vulnerability detection tools primarily focus on triggering a particular undefined behavior as a sign for the *existence* of a vulnerability.

Given the size of current-day programs and the huge number of possible execution paths within them, one of the most critical challenges for vulnerability detection tools is exploring a program and pushing it into a vulnerable state while avoiding the path explosion problem. Alternatively, given the large size of the programs, a set of techniques have focused on searching for the reoccurrence of vulnerability signatures (i.e., a set of program statements) in the code. Nevertheless, using a contiguous code segment, i.e., a function, as the vulnerability signature is inadequate since it may include unrelated statements to the vulnerability.

Therefore, instead of using a vulnerable program as a signature, by extracting the chain of relevant statements with which the program enters a vulnerable state, denoted as **VRS** (i.e., vulnerability-relevant statements), it is possible to change the exhaustive search over the whole program, to an undoubtedly more limited search only on a subset of program statements.

Therefore two specific challenges need to be addressed: **(i)** automatically extracting the VRS from **known** and patched vulnerabilities, and **(ii)** search within target programs, given a set of VRSs, to identify parts of a target code that contain new vulnerabilities similar to the known vulnerabilities.

The problem of extracting VRSs has been studied previously (Jang et al., 2012; Li et al., 2016; Kim et al., 2017). Either the

☆ Editor: Alexander Serebrenik.
* Corresponding author.

E-mail addresses: s.salimi@sharif.edu (S. Salimi), kharrazi@sharif.edu (M. Kharrazi).

https://doi.org/10.1016/j.jss.2022.111450
0164-1212/© 2022 Elsevier Inc. All rights reserved.

VRS is extracted from parts of the contiguous code, or the entire vulnerable program is considered a basic signature to find similar vulnerabilities. However, in most cases, there is a need for a systematic approach for eliminating irrelevant statements to the vulnerability, particularly in the case of complicated vulnerabilities where the fragments of code related to vulnerabilities are noncontiguous, similar to type-3 clone pairs (Alomari and Stephan, 2020; Xue et al., 2018).

Motivated by the above-noted observation, we address the problem of extracting succinct, self-contained, and noncontiguous VRSs from known vulnerable programs that can be found and searched for in real-world applications. We leveraged the static program slicing technique (Weiser and slicing, 1981) as a tool to extract VRSs. By using this technique, we find type-3 clone pairs with VRSs, resulting in new yet similar vulnerabilities to VRSs in target programs.

By building upon identified vulnerabilities in the wild, we are able to extract any vulnerability as a standalone VRS. In other words we extract slices that are considered to be related to vulnerabilities from known vulnerable programs and then abstract the statements in the slice by eliminating local naming conventions used in variables and function names. Similarly we decompose the target program into abstract program slices. Now if the target program contains vulnerabilities similar to the vulnerable programs we have used to generate VRSs, then we will be able to identify those vulnerabilities. In other words, instead of comparing a target program with a vulnerable program, we compare program slices obtained from a target program with previously obtained VRSs.

In summary, our main contributions are

- We introduce a method which, by leveraging program slicing (Weiser and slicing, 1981), is capable of extracting succinct and self-contained vulnerability-relevant statements, VRS, from known vulnerable programs.
- We propose an efficient search technique, where by slicing the target program for all possible slicing criteria and then comparing a given VRS with the set of target program slices, can decide on the existence of vulnerabilities in the target program.
- To evaluate our approach, we developed a system called VulSlicer with two primary goals: **(i)** to extract VRSs from known vulnerable programs, and **(ii)** to search for vulnerabilities within target programs by decomposing the program and preparing comparable code segments with VRSs.
- We have been able to identify several new vulnerabilities, 3 of which have been assigned a CVE number.

The remainder of this paper is structured as follows: In Section 2, we present an overview of our proposed method. We discuss how the proposed method extracts VRS in Section 3, and how it slices target programs in Section 4. The vulnerability search method in target programs is presented in Section 5. Our implementation, and the evaluation results of VulSlicer, including comparing VulSlicer with prior work, are presented in Section 6. Section 7 studies related work. In Section 8 we discuss some limitation of VulSlier. Lastly, we conclude in Section 9.

2. Overview

In what follows, we first provide an overview of VulSlicer, discuss the challenges faced with VRS generation and search, and finally present a running example with a real-world vulnerability for better illustration and motivation.

2.1. VulSlicer's overview

VulSlicer works through three main stages to extract self-contained and succinct VRSs (i.e., vulnerability-relevant statements) from a known vulnerable program and detect new vulnerabilities based on them in a target program. Fig. 1 illustrates the high-level workflow of our proposed system. The first two stages prepare VRSs and target program slices which feed their resulting output as input to the search stage.

VRS Extraction: Initially, VulSlicer analyzes a given vulnerable program to find the vulnerability point, a program statement in a program where execution goes wrong, of a known vulnerability and identifies the required criterion (i.e., a set of variables and a program statement) for slicing the program and extracts the set of related statements that cause the vulnerability. Each vulnerable slice is then abstracted to generate generic vulnerability-relevant statements or VRS. This process is repeated for different known vulnerable programs. More details on this stage is presented in Section 3.

Target Program Slice Generation: VulSlicer analyzes each target program being tested for vulnerabilities and slices it based on all program statements. Every extracted slice is transformed into an abstracted form, which is then used in the search stage. More details on this stage is presented in Section 4.

VRS Search: After the first stage, there is a VRS database that contains a set of vulnerable slices stored in the form of abstracted code. In the second stage, all possible abstracted slices are extracted from a target code being tested. In this stage, the abstracted slices from the target code are searched in the VRS database. Given the number of target program slices and the number of VRS database entries, one of the key design points is the scalability of the search stage. Section 5 presents further detail on this stage.

2.2. Challenges

Challenges associated with VRS extraction and matching are summarized as follows:

Challenge 1 (C1): Lacking a well-defined functionality and proper interface (I/O)

The idea of extracting fragments of a program that preserves a specific vulnerability context is similar to code re-use (Caballero et al., 2010; Lin et al., 2017; Kolbitsch et al., 2010), where the focus is on extracting a fragment of a program that preserves a specific functionality. The significant difference here is that in code re-use, the functionality is well-defined, mainly based on the function's input and output pair. If the vulnerability and its effects are known, it is possible to identify the set of variables which should be monitored by the detection tools and understand how these variables would be affected when the vulnerability is triggered. The challenge is that the vulnerability is considered as an unknown for most tools; hence, the concept of functionality and functions may not be directly applied in the context of vulnerabilities.

Challenge 2 (C2): Unknown search boundaries

For any program, it is possible to partition the code (either binary or source code) into disjoint units at different granularities, i.e., functions are one of the more coarse-grained units while program instructions for binary code or program statements for source code are fine-grained. The challenge here is to find a program statement that we can use as the boundary, and select other program statements with respect to this boundary, in order to be able to compare them with the vulnerability definitions.

Challenge 3 (C3): Noncontiguous code fragments

Any extracted VRS must only contain parts of the program related to vulnerability and nothing more. More specifically, the challenge here is to decide which parts of the code should be extracted from a set of noncontiguous fragments, resulting in a self-contained fragment and, in turn, the expected vulnerable state without any irrelevant code statements.

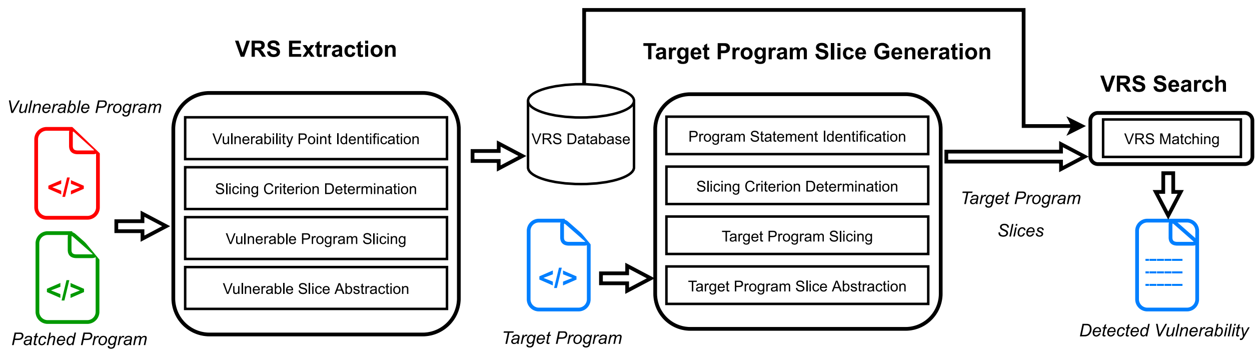
VRS Extraction
Vulnerable Program
Patched Program
Vulnerability Point Identification
Slicing Criterion Determination
Vulnerable Program Slicing
Vulnerable Slice Abstraction
VRS Database
Target Program
Target Program Slice Generation
Program Statement Identification
Slicing Criterion Determination
Target Program Slicing
Target Program Slice Abstraction
Target Program Slices
VRS Search
VRS Matching
Detected Vulnerability

Fig. 1. An overview of VulSlicer. VulSlicer accepts vulnerable programs and their patched versions for specific known vulnerabilities to build VRSs based on them. Further, every target program is passed to the slice generation module to produce program slices based on all its slicing criteria, where each slicing criterion is a combination of a program statement and its corresponding set of program variables. At the final step, VulSlicer searches the set of target program slices to find matches for VRSs.

2.3. Running example

To cover both aspects of VulSlicer, the running example we selected has two parts; the first part is a real-world vulnerability we use to extract a VRS from it and the second part is a real-world target program that we search to find a match with VRS. In what follows, we introduce these two samples and utilize them in the rest of the paper to describe the VulSlicer functionality.

2.3.1. Example vulnerability

For the sample vulnerability that we build a VRS based on, we make use of a critical vulnerability in the Linux kernel's infrared serial module, prior to version 5.1.6, identified as *CVE-2019-19543*. The vulnerable code related to this vulnerability is shown in Fig. 2.

In this example, the code could be exploited because there exists a *use-after-free* vulnerability in function `serial_ir_init_module`. Since there is already cleanup handling in `serial_ir_init` error path, there is no need to call `serial_ir_exit` again in `serial_ir_init_module`, otherwise the second call will trigger a use-after-free issue. In more detail, if the `result` variable is not zero, both functions enter the cleanup, hence the second cleanup uses the variable that is already freed. The following section explores the associate patch for this vulnerability and describes how VulSlicer extracts a VRS based on the patched and vulnerable code.

It is worth mentioning that by selecting the entire `serial_ir_init_module` function of the example in Fig. 2, as opposed to only selecting the highlighted statements in the function, does preserves the vulnerability context, but it still contains irrelevant program statements to the vulnerability. These irrelevant statements, i.e., line 22 code snippet in Fig. 2 make the vulnerability signature imprecise, which results in a false negative. This false negative is related to Challenge 3 (C3) we listed in Section 2.2.

2.3.2. Example target program

For the example target program, we used Linux 4.10, released on 19 Feb 2017. Among the vulnerabilities which existed in this particular version, there is a vulnerability identified with *CVE-2018-20836* that we concentrate on, in order to show how we slice the program and find a match with the example VRS we have. This example is illustrated in Fig. 3.

In the next section, we will discuss each step of the proposed approach in more detail.

```
 1 static int __init serial_ir_init(void){
 2   int result;
 3   [...]
 4   serial_ir.pdev = platform_device_alloc("serial_ir", 0);
 5   if (!serial_ir.pdev) {
 6     result = -ENOMEM;
 7     goto exit_driver_unregister;
 8   }
 9   [...]
10   return 0;
11   [...]
12 exit_driver_unregister:
13   platform_driver_unregister(&serial_ir_driver);
14   return result;
15 }
16 static void serial_ir_exit (void){
17   [...]
18   platform_driver_unregister(&serial_ir_driver);
19 }
20 static int __init serial_ir_init_module (void){
21   int result;
22   [...]
23   result = serial_ir_init (void());
24   if (!result)
25     return 0;
26   serial_ir_exit();
27   return result;
28 }
```

Fig. 2. There is a use-after-free vulnerability in `serial_ir_init_module`, since it calls the `serial_ir_exit` without any condition, if variable `result` is not equal to zero, a clean-up is already called with `exit_driver_unregister`. Precisely, function `platform_driver_unregister` is called twice once in line 13 and again in line 18, resulting to free a memory variable, which is already freed. Highlighted statements are related to the vulnerability which are used by VulSlicer.

3. VRS extraction

VulSlicer, at its core and as the name suggests, backward slices a given vulnerable program to isolate vulnerability-relevant statements or VRS. After obtaining a vulnerable slice, VulSlicer

```c
/* ---------- SMP task management ---------- */
static void smp_task_timedout (struct timer_list *t){
  [...]
  complete(&task->slow_task->completion);
}
static void smp_task_done (struct sas_task *task){
  if (!del_timer(&task->slow_task->timer))
    return;
  complete(&task->slow_task->completion);
}
```

Fig. 3. Snippet code from file `drivers/scsi/libsas/sas_expander.c` in the Linux version 4.10, we utilize this code as the example target program. There is a use-after-free where `smp_task_timedout()` will complete the task without any condition, meaning that the `task` variable will be freed after this call. While, `smp_task_done()` is called after the timeout and completes the task again, resulting to use the `task` variable after it has being freed.

abstracts it to remove dispensable parts, the information specific to the original vulnerable program, and generates an abstracted version of the vulnerable slice. The generalized slices are then stored in the VRS database. The rest of this section describes the process of VRS extraction.

3.1. Vulnerability point identification

VulSlicer finds the set of corresponding program statements that are related to a vulnerability. As we have mentioned in Section 2, for any known vulnerability, we need both vulnerable and patched versions of the program, further, we utilized the *diff* between these two versions and marked **modified** or **deleted** statements and the nearest statement before every **inserted** lines in the old vulnerable file. Utilizing *diff* file between vulnerable version and patched version of code to locate vulnerability points is an approach used extensively (Kim et al., 2017; Li et al., 2016; Du et al., 2019; Li and Paxson, 2017; Jang et al., 2012) which we leverage as well.

The output of this stage is a program statement that is assumed to be the vulnerability point (i.e., where the vulnerability is triggered) that will be used to determine the slicing criteria.

3.2. Vulnerable program slicing

VulSlicer utilizes the static program slicing (Weiser and slicing, 1981) technique, introduced originally to help programmers with fault isolation. More specifically, we leverage backward slicing to extract noncontiguous program statements that isolate parts of the program related to a specific vulnerability. The list of the notations and functions we used to describe the slicing approach is as follows:

- Function LoDV(`stmt`)[1] is used to obtain the list of defined (or modified) variables in a given statement.
- LoUV(`stmt`)[2] is used to obtain the list of used variables in a given statement.
- Function CFG(`stmt`) returns the CFG node that corresponds to `stmt`.
- Function STMT(`node`) returns a program statement from a node in CFG.

In order to backward slice a program, a slicing criterion `C=(V, ps)` is required, where `ps` is a program statement in

[1] List of Defined Variables.
[2] List of Used Variables.

Algorithm 1 Vulnerability Slicer takes a vulnerable program as P and a program statement as vp. At first, it generates CFG for the program P where each node of CFG represents a single program statement, then it finds a CFG node, CFG(vp)), corresponding node in CFG that contains the program statement vp. At the next step, it generates the PDG from the CFG, where PDG and CFG nodes are common. PDG-based slicing approach traverses the graph to find nodes that can reach directly or indirectly to the criterion node. After finding a set of nodes associated to the criterion node, VulSlicer reconstructs the slice from each node corresponding program statement using STMT(node).

```
1: function VulSlicer(P, vp)              ▷ P = [ stmt_1, ... , vp, ... , stmt_n]
2:     CFG ← CFGgen(P)
3:     PDG ← PDGgen(CFG)
4:     Criterion ← CFG(vp)                ▷ vp ∈ P
5:     Slice ← SLICEgen(PDG, Criterion)
6:     VRS ← []
7:     for node ∈ Slice do
8:         VRS ← VRS ∪ STMT(node)
9:     end for
10:    return VRS                         ▷ VRS = [ stmt_k, ... , vp]
11: end function
```

program `P`, and `V` is the set of variables in program. VulSlicer can obtain the corresponding CFG node, including the location of program statement, for `vp` by calling the CFG(`vp`). The CFG that VulSlicer uses is a single-statement block CFG, which each basic block corresponds to a single program statement.

There are various types of algorithms to compute backward static program slices, where we have employed the program dependence graph (PDG) (Ferrante et al., 1987; Ottenstein and Ottenstein, 1984) approach for slicing. A PDG is a directed graph that shares its node with CFG and has two kinds of edges, control, and data dependence edges. That means a PDG is constructed with subgraphs, control, and data dependence graph. We leverage LoUV(`stmt`) and LoDV(`stmt`) during PDG construction. Since both graphs can be constructed with CFG, we need to generate the CFG at first. After that, we construct PDG; by using the PDG-based slicing approach, the slicing criterion is identified with a node `ps` corresponding to a single program statement in the PDG. For backward slicing, it traverses the PDG to find all nodes that directly or indirectly reach the criterion node, which allows the program to be sliced in linear time. Compared to Weiser's algorithm (Weiser and slicing, 1981), and as explained in Tip (1994), this criterion is equal to `C = (V, ps)` where `ps` is the single-statement block CFG node, and `V` is the set of all variables defined or used at `ps` or `LoDV(ps) ∪ LoUV(ps)`. The slicing algorithm VulSlicer employs is presented in Algorithm 1.

As for the running example presented earlier, Fig. 4 shows the patch.[3] that resolves the vulnerability illustrated in Fig. 2 In fact, the last statement that is modified in the patch is `return result` at line 27 and the only variable involved in modified statements is `result`. Therefore, to extract a VRS, VulSlicer uses the criterion CFG(`returnresult;`). During slicing, if any node in PDG involves calling a function, e.g., `serial_ir_init` at line 23, VulSlicer does not slice within the callee function. Vulnerability-relevant statements are highlighted in Fig. 2.

3.3. Vulnerability slice abstraction

Vulnerability slice abstraction is an essential final step in order to eliminate any local naming conventions and program-specific information from the slice and therefore present the vulnerability more accurately. VulSlicer abstracts each output program slice in the following steps:

[3] The commit hash that patches this vulnerability is 56cd26b618855c9af48c8301aa6754ced8dd0beb.

```
@@ -773,8 +773,6 @@ static void serial_ir_exit(void)
static int __init serial_ir_init_module (void)
{
 -  int result;
    [...]
@@ -802,12 +800,7 @@ static int __init serial_ir_init_module(void)
 if (sense != -1)
      sense = !!sense;
 -  result = serial_ir_init();
 -  if (!result)
 -    return 0;

 -  serial_ir_exit();
 -  return result;
 +  return serial_ir_init();
}
```

Fig. 4. The chunk of code that patched use-after-free vulnerability identified with *CVE-2019-19543*. This chunk of code shows the underlying cause for the vulnerability. VulSlicer defines the slicing criterion based on it and generates VRS.

(1) **Grammar-based transformation.** VulSlicer employs Clang to parse each program, identify statements, and generate a CFG. Additionally, the AST generated by Clang is used to label statements with the compiler grammar. For example, a statement could be labeled as an *If* statement.
It further generates variants of available program abstraction. The most important one is utilizing a simple symbolic transitive replacement developed for compiler-level optimization. For example, as shown in Fig. 5, where a variable is defined to store the return value of a function, and it never changes in the slice, it replaces the variable with a return value of a function and removes the variable definition and assignment.

(2) **Arguments and variable names substitution.** VulSlicer replaces all the functions' arguments in the slice with a generic string, shown as *VAR*. For each variable or argument involved in the slice, we need to have a program statement that first defines them. For the variables, this statement is already present in the PDG; therefore, the corresponding PDG node that defines the variable for the first time is automatically selected and appears in the final slice. For the function arguments, the PDG does not have any nodes that contain those definitions; therefore if such arguments are present in the final slice, VulSlicer adds a program statement that defines the argument for the first time.

(3) **Variable type substitution.** VulSlicer replaces all the data types with a generic name *DATATYPE*.

(4) **Function name substitution.** VulSlicer replaces all the local function names with a generic name *CALL*. This means, during the parsing VulSlicer extracts all function names, and later it replaces them. It should be noted that when well known function names, known as C-standard functions (Standard, 2019) such as `free`(), `memcpy`(), or etc. from the `libc` library are used, the function names are kept unchanged and are not abstracted.

The abstracted version of the VRS for the running example vulnerability is presented in Fig. 5, which represents the sliced statements related to function `serial_ir_init_module`. Further, we leverage abstract syntax tree and compiler-level abstraction and replace lines 1–3 with only one line, line 3 in Fig. 5, meaning that the slice is further abstracted by using the function call itself rather than storing its return value in a new variable.

```
1 Defined DATATYPE VAR[VCONST];
2 VAR[VCONST] = CALL[FCONST]();
3 IF STMT (!VAR[VCONST]) {RETURN STMT[CONST]}
4 ELSE STMT {CALL[F2CONST];}
5 RETURN STMT VAR[VCONST];
```

Fig. 5. The VRS generated for the vulnerability *CVE-2019-19543*. The first three statements can be replaced by only one statement, meaning that instead of storing the result of `CALL[FCONST]()`, it can be used directly without storing it on `VAR[VCONST]` variable.

3.4. Preserving vulnerability context

VulSlicer uses program slicing to slice parts of the original code assumed to be related to a known vulnerability. The resulting slice, VRS, preserves the corresponding vulnerability context represented with a chain of statements. After extracting slices, an abstracted version of each slice is generated as described in 3.3.

This transformation is required to remove information and data that are only related to the original program and not the vulnerability context. The abstraction process targets the variable and local function names and is applied after slicing, so it cannot affect the slicing itself. Such an abstraction is necessary to detect similar slices in target programs with the same context and probably with some code modifications, e.g., variable names and function names. More specifically, this transformation is required to detect Type-2 clone slices.

4. Target program slicing

After a program is submitted to the system for analysis, VulSlicer generates a corresponding CFG and PDG for it. Then, it identifies all its program statements, `ps`, which results in determining all slicing criteria corresponding to each program statement and the set of its defined variables using LoDV(`ps`).

VulSlicer slices target programs using all of its program statements; therefore, it needs to get all possible slicing criteria concerning program statements. Since it needs both PDG and CFG for any target program, it generates these two graphs for all of the program's functions. Then, for a program statement (`ps`), it first generates the criterion as $\mathtt{C} = (\text{LoDV}(\mathtt{ps}), \text{CFG}(\mathtt{ps}))$ and extracts the corresponding slice by traversing the PDG. This means that each slice is a subset of PDG nodes, which contains the `ps` node and all nodes that directly or indirectly reach the `ps`. Last but not least, slices are abstracted using the same approach as noted in Section 3.3.

The target program slicing algorithm is shown as Algorithm 2. The input of this algorithm is a target program P, `TargetSlicer` generates all the possible independent, self-contained code fragments, each corresponding to a single program statement.

For instance as shown in example target *program* in Fig. 3, one of the slicing criteria is based on line 9, meaning that the program slice starts from line 9, traces backward, and selects program statements that have any effect on the corresponding node in PDG. VulSlicer repeats the same process for line 7 and line 8, and finally generates three distinct program slices for this function. VulSlicer abstracts all slices it extracts, for instance, the abstracted form of the target slice from line 9, is shown in Fig. 6, other slices are abstracted in the same way.

In the next section, we show how VulSlicer finds a match with the VRSs.

5. VRS search

Searching for a VRS in real-world and complicated programs is a challenging task. VulSlicer leverages the fact that VRS should not

Algorithm 2 The TargetSlicer takes, as input, program P and generates slices based on each program statement, ps. TargetSlicer function, which works like VulSlicer function in the vulnerability slicing algorithm, where at first it extracts all program statements and then defines the slicing criterion as the PDG node corresponding for the ps, then traverses the PDG to find nodes associated with criterion and constructs the target slice using STMT(node) for each selected node. If the resulting target slice, TS, is not already extracted, it adds it to the candidate set.

```
1: function TargetSlicer(P)                          ▷ P = [ stmt_1, ... , stmt_n]
2:     Candidates ← []
3:     CFG ← CFGgen(P)
4:     PDG ← PDGgen(CFG)
5:     for ps ∈ P do
6:         Criterion ← CFG(vp)                                          ▷ vp ∈ P
7:         Slice ← SLICEgen(PDG, Criterion)
8:         TS ← []
9:         for node ∈ Slice do
10:            TS ← TS ∪ STMT(node)
11:        end for
12:        Candidates ← Candidates ∪ {TS}
13:    end for
14:    return Candidates                      ▷ Candidates = [ slice_1, ... , slice_k]
15: end function
```

```
IF STMT (!CALL[FCONST]()) {RETURN STMT[CONST]}
ELSE STMT {CALL[F2CONST];}
```

Fig. 6. An abstracted slice from the target program shown in Fig. 3 that matches with VRS in Fig. 5.

match the entire program; instead, it generates program slices of the target program and then compares them with VRSs to find matches. Given that we have access to each program statement original location and the abstraction is applied on each statement, not the entire program, we sort the slice's statements based on their location at the beginning of the search process.

5.1. VRS and target slice matching

Each target program slice includes a subset of target program statements, while dependencies between statements are preserved, meaning that each slice is a projection of the target program. The number of slices for each target program is equivalent to the number of slices over all possible criteria constructed over program statements.

VulSlicer exploits the fact that both the target program's slice and VRSs are program slices stored in abstracted forms meaning they do not contain program-specific information. Therefore, the search problem turns into a problem of two (abstracted) slices comparison. Slice comparison is a classic problem in computer science (Horwitz, 1990), which has efficient solutions, like comparing slices' program dependence graphs in linear time (Horwitz and Reps, 1991), and deciding if two graphs are isomorphic. VulSlicer leverages a simple hash-based comparison (i.e., MD5) to compare two abstracted slices instead of comparing two PDG graphs, which is relatively faster.

As for the running example, we can find a match between the given VRS from stage 1, indicated as Fig. 5, and the target program slice from stage 2, depicted as Fig. 6, which is a previously use-after-free vulnerability, identified with *CVE-2018-20836*, in the Linux kernel. Where instead of defining a variable to store the return value of the function, the return value of the function is directly used.

One should note that if instead of using a pointer variable (i.e., `&task->slow_task->timer`) in the example, a non-pointer related variable was used in Fig. 6, then there would have been a false positive as the same abstracted slices are generated for this sample code in both cases. This is due to the fact that the abstracted slices are insensitive to the type of variables being used.

In the next section, we discuss how VulSlicer was implemented and discuss the evaluation results.

6. Implementation and evaluation

In this section we first present a summary of VulSlicer's implementation details in Section 6.1, and review the evaluation setup in Section 6.2. Furthermore, we answer the following research questions:

(I) Given a ground-truth dataset, how accurate is VulSlicer in identifying vulnerabilities? Evaluation results are discussed in Section 6.3.

(II) Is VulSlicer capable of detecting new and unknown vulnerabilities in real-world programs? This question is looked at in Section 6.4.

(III) How does VulSlicer compare to prior tools that are designed to detect vulnerabilities based on previously known vulnerabilities? This question is answered in detail in Section 6.5.

6.1. Implementation

VulSlicer is developed in a mixture of C++ and Python. The main parser and static analyzer module are developed based on Clang 8.0 and by utilizing its C++ binding (Clang, 2020).

We leveraged the *CFG* class available in Clang and fed *.C* files to get CFG objects, and further, we built both control and data dependence graphs using single-statement block CFG. Clang uses the abstract syntax tree (AST) of the target file to generate the CFG; therefore, CFG has access to compiler-level keywords, which we utilize to apply abstraction levels described in 3.3.

In order to generate slices, VulSlicer queries PDG by sending a node as a slicing criterion and finding all nodes related to the slicing criterion. Given that each node has the location and body of the program statement, VulSlicer can stitch them together to return a program slice. Finally, we store the abstracted version of slices. During the search phase, VulSlicer uses a hash-based (i.e., MD5) comparison and compares the hash of abstracted slices with that of VRS.

Furthermore, to collect and build a comprehensive dataset of open source projects, a GitHub parser was developed in Python 3.8 with 8K LoC. The resulting dataset containing all the known vulnerabilities collected from GitHub open-source C projects is publicly available on https://zenodo.org/record/6059924. Additionally, the Clang-based PDG generator and slicer source codes are publicly available on https://zenodo.org/record/6333819 as well.

6.2. Evaluation setup

In order to properly evaluate our implementation of VulSlicer, we collected a dataset of program source codes from GitHub as the resource for real-world and well-distributed (in terms of complexity and scalability) projects. More specifically, and by leveraging GitHub's API, we developed a crawler with which we crawled **26 117** C projects. The data was initially collected in November 2018 and was later updated in December 2019. Further, we crawled all the known CVE identifiers from the NVD database (NVD database, 2019) to prepare the appropriate input for the *GitHub Parser*.

By employing the developed GitHub parser, all commit messages for each project are analyzed, and commit messages which match the `CVE-\d{4}-\d{4,5}` regular expression are marked as commits that have patched a vulnerability with a specific CVE id in the program. Considering that the NVD database

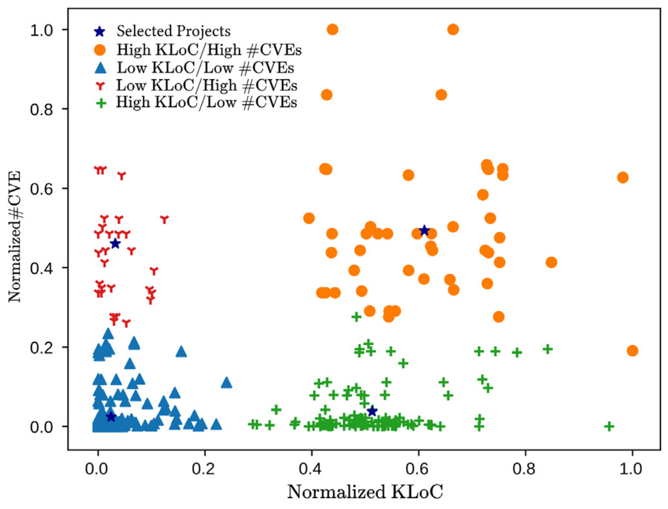

Fig. 7. Projects collected from GitHub clustered with respect to the number of known patched vulnerabilities, as noted in their logs and KLoC. For each cluster, we selected the cluster center's project for evaluating the VulSlicer for detecting new vulnerabilities indicated with ★. ● denotes projects with high KloC and high number of patched vulnerabilities, ▲ indicates projects with low KloC and low number of patched vulnerabilities, *Y* illustrates projects with low KloC and high number of patched vulnerabilities and finally + shows projects with high KloC and low number of patched vulnerabilities.

has all the meta-information for vulnerabilities, including CVE ids and involved project names, it is used to validate the CVE-related commits in the projects.

By removing duplicate (forked) projects and selecting only projects with at least three commits to patch vulnerabilities, we obtained **2719** projects. These projects have more than 14 million files and 3 billion lines of code in the latest crawl.

VulSlicer extracted **108 154** vulnerability patches. These patches are then further processed and patches that are not limited to only a unique vulnerability and may also include code refactoring or feature development,[4] patches that only mention the CVE id as comments without actually patching the code, and finally, patches that are larger than 2 MB in size are excluded, therefore we are left with **13 540** patches which are used to extract VRSs.

Although it is possible to run the VulSlicer against a wide range of projects, we focused on a limited set of candidate projects to evaluate our approach. This allowed us to explore different versions of these projects over time, considering each git commit hash as a new version. Furthermore, we investigated both known patched CVEs in the projects as well as silently patched bugs, where the commit message of the patch does not directly mention a CVE id that matched with our VRS database. Intuitively, known patched bugs are considered as a **ground-truth** dataset to provide a measure of accuracy for the proposed method.

GitHub's API is designed to select projects based on various predefined tags, including the number of commits, the number of issues, update timing. There are no specific tags to sort projects based on the type or number of vulnerabilities or other security-related features. To obtain such sorting, we clustered all the crawled GitHub projects based on the number of known patched vulnerabilities noted in their logs and KLoC (i.e., kilo lines of code). Projects are either large or small with respect to KLOC and either include many or few patched vulnerabilities. In order to evaluate the effectiveness of our approach, we clustered all the crawled C projects with respect to KLOC and the number of security patches with *K-means* clustering algorithm, which results in four different clusters. The result of clustering is illustrated in Fig. 7. We selected the project at the center of each cluster as our test cases. More specifically, Table 1 lists the projects selected for evaluating the *VulSlicer* and the related information for each of them.

Table 1
Description of selected projects from GitHub to evaluate VulSlicer for detecting new vulnerability.

Cluster	Description	Selected	#CVEs	KLoC
●	High KLoC/High #CVEs	Linux kernel[a]	450	19 281
▲	Low KLoC/Low #CVEs	libgd	28	58
Y	Low KLoC/High #CVEs	samba	364	3159
+	High KLoC/Low #CVEs	libvirt	56	1791

[a]Will be referenced as kernel for the rest of the paper.

It should be noted that all experiments, including the VRS database generation and the search in target projects conducted on a desktop machine with Intel Core i7 CPU, 16 GB memory, and a 6 TB SSD drive, running Linux 4.16 with Debian Stretch.

6.3. VulSlicer accuracy

To evaluate the accuracy of VRSs for finding similar vulnerabilities, we need to count the number of detected vulnerabilities from our ground-truth dataset to measure *false positive* and *false negative* reports as general accuracy.

Our ground-truth dataset is constructed by exploring different versions of selected projects through time, as illustrated in Fig. 8. More specifically, or multiple versions of these projects, we can obtain various patches for known vulnerabilities and patches unrelated to known vulnerabilities, and we utilize them to observe the accuracy of VulSlicer's extracted VRSs. Since our dataset projects are collected from GitHub, we consider each *commit* as a new version. For a *commit*, that explicitly patches a vulnerability with assigned CVE, we remove the VRS which is extracted based on this CVE in version $i+1$ and run VulSlicer for a version before the *commit* (i.e., version i). If VulSlicer can detect the removed vulnerability, we consider it as a **true positive** since it is already patched, and we know the previous version is vulnerable. On the other hand, given a patched vulnerability in version $i + 1$, if VulSlicer fails to identify it in version i, then there is a **false negative**.

Alternatively, if we consider a commit that is not related to a CVE identifier and VulSlicer alerts as to the existence of a vulnerability, it is a **false positive**, and if it does not match the target slice with any VRS, then a **true negative**. We sample and analyze the same number of non-CVE related commits as the CVE commits to provide a balanced comparison and analysis. Although in a real life deployment the ratio of vulnerable code to non-vulnerable code would not be equal and that would affect the results. Nevertheless, we consider that discussion outside the scope of this work. Obtained results are summarized in Table 2.

6.4. Discovered vulnerabilities

In order to evaluate the practical impact of VulSlicer, we have checked all the program's slices for the four target projects. Among the bugs identified, several *true* vulnerabilities have been confirmed. Others have been either *silently* patched, that is, they are patched in the latest versions (latest version means the version with the latest commit for a GitHub project), but no known CVEs are assigned to them, or previously patched, meaning they have been patched with assigned CVEs in the latest versions. These two kinds of matches exist because our target projects are being actively updated. In more detail:

[4] We assume modifications to more than 2 *C* files could potentially indicate code refactoring or feature development and exclude such patches.

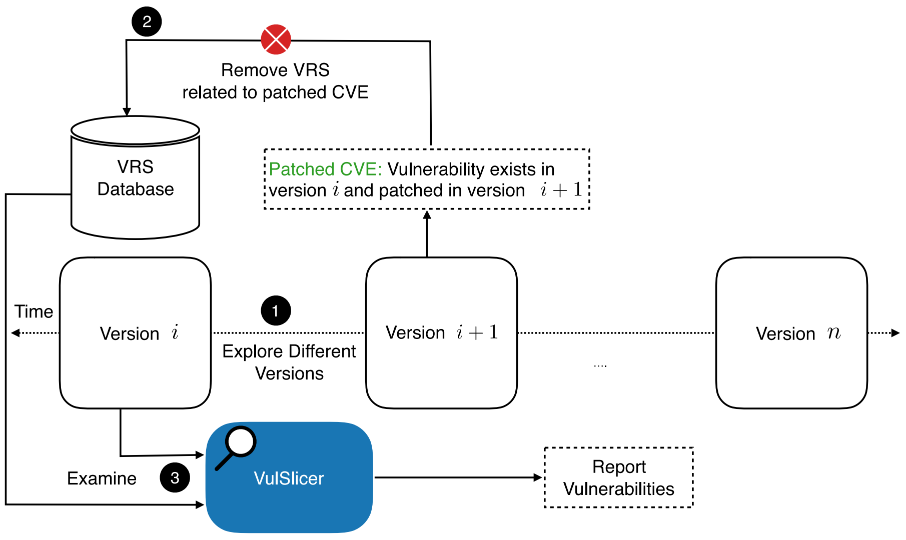
2
Remove VRS related to patched CVE
VRS Database
Patched CVE: Vulnerability exists in version i and patched in version $i+1$
Time
Version i
1
Explore Different Versions
Version $i+1$
Version n
Examine
3
VulSlicer
Report Vulnerabilities

Fig. 8. Steps to measure VulSlicer's errors: **(1)** Explore commits till a commit that indicates a patched vulnerability is found, we call the version with this commit as version $i+1$. **(2)** For each known vulnerability in version $i+1$, we remove the VRS corresponding to the patched vulnerability stored in our database, **(3)** and then run VulSlicer on version i, which we know that there is a vulnerability in, and then check if it can detect the mentioned vulnerability or not.

Table 2
VulSlicer's error for selected candidate projects. #Commits indicates the number of all commits we feed into VulSlicer, and #CVE are the commits that patch a vulnerability with an explicit identifier.

Project	#Commits	#CVE	#TP[a]	#FN[b]	#TN[c]	#FP[d]	Precision	Recall
kernel	440	220	162	58	212	8	95%	74%
libgd	26	13	2	11	13	0	100%	15%
samba	168	84	31	53	80	4	88%	37%
libvirt	86	43	6	37	36	7	46%	14%
Total	720	360	201	159	341	19	91%	56%

[a]Denotes all commits with a vulnerability patch in version $i+1$ and VulSlicer detects the vulnerability in the version i.
[b]Denotes all commits with a vulnerability patch in version $i+1$, but VulSlicer missed them in version i.
[c]Denotes all commits without a vulnerability patch, and VulSlicer does not report any match for the specif slice in version i.
[d]Denotes all reported matches where version $i+1$ and i are without a vulnerability patch but the old code mistakenly matches with at least one VRS.

libgd By examining Libgd project, we have identified 5 double-free vulnerabilities all in the `gdImage*Ptr` functions, two of them were previously identified and patched **CVE-2017-6362** and **CVE-2016-6912**. These two vulnerabilities exist in our ground truth dataset and correspond to the two *true positive* samples denoted in Table 2. LibGD developers patched the other three vulnerabilities we reported and assigned **CVE-2018-1000222** and **CVE-2019-6977** to these vulnerabilities in version 2.3.0, released on March 2020. The same three vulnerabilities also existed in the PHP source project

samba We have examined version 4.0, released in 2012, and found 37 vulnerabilities, while 31 of them, as denoted in Table 2 are already verified and patched, and considered as *true positive*, we have found six new vulnerabilities in version 4.9, released on 2018. Four of them are related to old files and dead codes, and two other vulnerabilities are reported.

libvirt We have examined version 1.0.1, released in 2012, and found 7 silently and previously patched vulnerabilities while we have found two new vulnerabilities in version 5.0.0 released in 2019, where both are patched as non-security-related bugs.

Linux kernel Linux Kernel is a very dense project with respect to KloC. We have examined version 3.4 LTS that contains all kernel and driver files, updated on December 2013,[5] and found 162 silently and previously patched vulnerabilities. Most of these patched bugs existed multiple times in different parts of the code.

6.5. Deep comparison with prior works

The main goal of VulSlicer is to improve an extracted VRS from known vulnerabilities in a way that it finds similar unknown vulnerabilities more accurately in other target projects. We have examined the accuracy of VulSlicer in Section 6.3. There are prior works that extract vulnerability signatures and search them in target programs. The three most similar works to VulSlicer, designed for detecting vulnerable code clones at scale, are *VUDDY* (Kim et al., 2017) and *ReDeBug* (Jang et al., 2012) and Vulpecker's (Li et al., 2016), where the first one uses vulnerable functions as a search pattern, while the second defines a line-based pattern on a given *diff* file with program statement-level granularity. Vulpecker leverages multiple search algorithms to decide if each entry of the database can be found in a target program. The Vulpecker's public GitHub repository contains a database of mapping CVEs and its related diff file(s). Given that search algorithms are not accessible in the repository, it is not possible to reproduce their results. Likewise, as VCCFinder (Perl et al., 2015) is one of the recent and practical approaches that can decide if a *diff* file patches a vulnerability in open-source projects, we opted to compare our work and discuss the results.

To compare VulSlicer with VUDDY, ReDeBug, and VCCFinder we designed a similar experiment, as explained in Section 6.3, and measured the same error for all of them. We reused the VUDDY vulnerability database found at VUDDY vulnerability database (2019) and regenerated the ReDeBug database by implementing it. The database used in VCCFinder is also not available, yet the code (VCCFinder Code, 2021) of this project is used to regenerate a database for the set of vulnerabilities used in the experiment. Most importantly, given that the obtained VUDDY database covered vulnerabilities up to mid-2017, we rolled back ReDeBug and

[5] This version can be found by commit hash 2cc64b5655da65bbb6a760a722b5ab1f53f92cf7.

Table 3
Comparing VulSlicer with prior work targeting the selected four projects (Linux kernel, libgd, libvirt and samba) to evaluate their results for 360 commits with vulnerabilities and 360 commits without a vulnerability.

Tool	#Reports	#FP	#FN	#TN	#TP	Recall	Precision
VulSlicer	220	19	159	341	201	56%	91%
VCCFinder	226	165	299	195	61	17%	27%
ReDeBug	199	68	229	292	131	36%	65%
VUDDY	190	39	209	321	151	42%	79%

VulSlicer databases to that time frame for a fair comparison which results in having **360** vulnerabilities in all of the databases. We used ReDeBugs default similarity metrics (n = 4, c = 3, where n is the number of lines per window, and c is the amount of context). We employed the highest abstraction level for VUDDY (abstraction level = 4) where formal parameters, local variables, data types, and function calls are abstracted; with this level of abstraction, VUDDY can detect Type-2 vulnerable code clones. We used 10k commits of the four projects, excluding the 720 commits used in the experiment, as the training set to train the linear SVM model as described in the paper, and choose similar metrics (cost = 1 and weight W = 100) to predict the state of vulnerability-related commits.

Table 3 summarizes results of this experiment. In this experiment, for VulSlicer, ReDeBug, and VUDDY, we search for a vulnerability in the code of projects. On the other hand, in the case of VCCFinder, we examine the commit itself using the SVM model to decide if it patches a vulnerability or not.

The most important result of this experiment is the false negative(**FN**) results. VUDDY and ReDeBug, are designed to find exact syntactic clones of previously known vulnerabilities. By removing the pattern extracted based on the vulnerability, they fail to detect the same vulnerability in an older version, which means they can find clone vulnerabilities very well, but when the clone changes, for example, by statement re-ordering VUDDY or syntactic changes for ReDeBug, they cannot detect it. The same discussion is valid for VCCFinder, while this tool is designed to predict if a given commit is related to vulnerability or not based on extracted metrics of commits. Consequently, it can categorize commits not contacting a patch very well. While VulSlicer is designed to extract a self-contained vulnerability signature; therefore, it is not dependent on an exact patch to detect each vulnerability as it abstracts a vulnerable code while it preserves the semantic of code by utilizing slices. While VulSlicer outperforms the other tools, the low *recall* means there are almost 40% of vulnerabilities without similar enough samples that we can employ to detect them.

Further, in the evaluation step, we leveraged commits that contain approved vulnerabilities, and we selected some other commits which do not indicate that they patch a vulnerability. However, silently patch vulnerabilities might still exist; therefore, some false positive results might still be related to vulnerabilities, but since there is no direct reference, we consider them false positive results.

Last but not least, and in terms of performance comparison, ReDeBug uses a similarity measurement to report a clone which makes it slow with respect to the other three techniques. While VUDDY and VulSlicer utilize fast comparison methods (i.e., hash-based comparison). The setup time VUDDY requires to prepare a candidate function is mostly, as mentioned in the paper, because of parsing. VulSlicer needs to slice each candidate target after parsing, which results in a time increase compared to VUDDY, although the VulSlicer parser works faster than VUDDY. It should be noted that creating the initial vulnerability database is an offline process for both VUDDY and VulSlicer that would not affect the performance of the proposed techniques when deployed. That means VulSlicer needs 838 s on average (range 593 to 1069 s) to prepare each VRS, compared to VUDDY that needs 1031 s on average to generate a function-based vulnerability signature. On the other hand, the VCCFinder training phase needs about 3.5 h (about 12 600 s) for a training set of size 2000 commits. However, at the final step of deciding if a code contains a vulnerability or not, both VulSlicer and VUDDY need a fraction of time to compare two hashes, and VCCFinder can predict if a commit contains a patch for the vulnerability in a similar amount of time. On the contrary, ReDeBug searches the target program, which takes 1030 s on average for each query. All timing experiments were conducted on our server in order to obtain comparable results.

7. Related work

There are several techniques (Shoshitaishvili et al., 2016; Cha et al., 2012; Peng et al., 2018; Stephens et al., 2016; Boonstoppel et al., 2008; Ramos and Engler, 2015) introduced to detect vulnerabilities without using known vulnerabilities. These techniques are considered outside the scope of this work.

Techniques that search for a specific vulnerability signature in the program could be studied toward two important aspects. How a vulnerability signature is defined and how the signature is searched within a code to detect new vulnerabilities. In what follows, we will review related works for the two noted aspects:

7.1. Vulnerability signature extraction

Accurate vulnerability signatures are the key to identifying complex and subtle vulnerabilities, although the automatic generation of such signatures is getting harder while newly discovered vulnerabilities are getting more complicated and involve most parts of the programs (Li and Paxson, 2017). As a remedy, one of the active branches of vulnerability detection research is the automatic generation of vulnerability signatures.

There are also techniques (Brumley et al., 2008) that identify the vulnerability in a program by comparing it to its pre-patched version. ReDeBug (Jang et al., 2012) analyzes the security patches of known vulnerabilities, and it defines a vulnerability signature based on code statements directly elicited from the patch concerning the file boundary as `diff` chunks boundaries are defined on files. VUDDY (Kim et al., 2017) leverages the single patch to extend the vulnerability signature for the function level degree. Although both of these approaches preprocess the known vulnerabilities to remove irrelevant data, they only focus on immediate patch chunks of code, leading to missing code fragments related to the vulnerability out of defined boundaries. VUDDY is not capable of selecting noncontiguous parts of codes.

Analyzing the usage of function and type names as signatures to find similar vulnerabilities in API symbols is introduced in Yamaguchi et al. (2011), which also presents the idea of *vulnerability extrapolation*. An enhanced version of vulnerability extrapolation introduced in Yamaguchi et al. (2012) utilizes AST trees to generate signatures with structural code information and is granted as the generalized version of the vulnerability extrapolation technique. It should be noted that vulnerability extrapolation is the technique that is mostly considered as a solution for the vulnerability search, yet in Yamaguchi et al. (2012), the idea of having inter-procedural vulnerability signatures is introduced. Some other approaches focus mostly on the semantics of vulnerabilities, i.e., inferring search vulnerability signatures for taint-style vulnerabilities introduced in Yamaguchi et al. (2015). A similar approach that uses already known security issues instead of patched vulnerabilities is JoanAudit (Thomé et al., 2017), which computes a slice based on the vulnerable sources and then searches the slice in a target program. Security slices determined by JoanAudit are similar to VRSs that VulSlicer generates. The core idea of using program slicing is common in both approaches, yet VulSlicer leveraged this technique to search in target programs as well.

A different branch of security research has focused on modeling vulnerabilities. Such models cannot necessarily be used as vulnerability signatures for source code. The most comprehensive approach is code property graph (Yamaguchi et al., 2014), which merges abstract syntax trees, control flow graphs, and program dependence graphs and generates vulnerability models for various kinds of (expert-knowledge-defined) vulnerabilities.

Finally, some other approaches focus on extracting vulnerability signatures from binary and executable codes (Chandramohan et al., 2016; Egele et al., 2014; Eschweiler et al., 2016; Hu et al., 2017; Pewny et al., 2014, 2015) for noncontiguous signatures and Byteweight (Bao et al., 2014) for intra-procedural signatures. Although the signatures could be defined to be inter-procedural, selecting a proper boundary is very difficult in binary programs.

7.2. Vulnerability search problem

Scanning the source code to find vulnerabilities has a long history in software, especially for auditing code by programmers. Therefore, there exist some practical code scanners which primarily utilize simple, well-known vulnerability signatures such as the boundary of arrays to detect buffer-overflows, e.g., PScan (2019) and RATS (2019), or they use manually defined vulnerability signatures (Engler et al., 2001). The idea of using program slicing to find vulnerability-related predicates is proposed in Dashevskyi et al. (2019). In this work, the program slicing approach is modified, introduced as *thin slicing*, which can explore predicates related to security issues in one project.

Vector comparison and *exact matching* are two main categories of approaches for a search problem. It is clear that the first category mainly focuses on the accuracy of the comparison algorithm, and the second category focuses on the efficiency of the search algorithm. The core idea of using program slicing to extract a more minimal part of a program that can help to find more specific code clones studied in Danicic et al. (2005).

Among proposed methods, VulPecker (Li et al., 2016) is a novel work that selects a different comparison algorithm based on the nature of vulnerability itself. In Yamaguchi et al. (2014) the search problem is addressed with the graph mining technique. Since the generated vulnerability signature is represented with the code property graph, there is a need to transform any target program to code property representation and then traverse it to find vulnerability-related graphs.

As mentioned before, vulnerability extrapolation (Yamaguchi et al., 2012) is also a helpful technique that focuses on extrapolating a signature in target programs and, based on the results, decides on whether there is a vulnerability. Although the search method itself is function-level based and works with a single vulnerability as input.

Finally, the search methods that focus on scalability change the search problem domain to a query problem, i.e., targeting Java programs (Livshits and Lam, 2005) and targeting C programs (Shankar et al., 2001) with predefined vulnerability formulas. More recent approaches in this category are VUDDY (Kim et al., 2017) and ReDeBug (Jang et al., 2012). VUDDY also utilized a hash function to generate simple hashes of signatures to boost up the search algorithm. However, both of these approaches ignore the vulnerability point identification and use either function-level vulnerability signatures (Kim et al., 2017) or file-level vulnerability signatures (Jang et al., 2012) with the program's statement granularity.

8. VulSlicer limitations

In what follows, we note some limitations VulSlicer has, which could be considered for further improvements or as future research questions.

- VulSlicer leveraged *Clang*, a wide-used and sophisticated compiler, as our core parser and CFG/PDG generator; therefore, it is limited to its supporting language-specific features of programs.
- VulSlicer requires the source code for both the vulnerable set of programs to extract the VRS and the target program to search the VRS within it.
- The abstracted form of slices may lead to false positive results in some cases, as explained in Section 5.1. This can be resolved by applying a more accurate approach, i.e., transforming slices to an intermediate representation, is a straightforward approach for abstraction.

9. Conclusion

This work presented a novel technique to obtain succinct and self-contained VRSs and detect new vulnerabilities at scale by slicing target programs and matching slices with VRSs. Given the source code of vulnerable programs, we employed the vulnerability slicing technique for automatically extracting self-contained and noncontiguous parts of the program that preserve a vulnerability context.

We developed a prototype called VulSlicer to analyze the proposed technique. Moreover, we elaborated on the selection of several test case projects for the evaluation. Last but not least, we discussed how VulSlicer could generate more acculturate VRSs, and compared to other similar approaches, it can reduce both false positive and negative results. Finally, we explained how VulSlicer successfully detects new vulnerabilities in target programs.

CRediT authorship contribution statement

Solmaz Salimi: Conceptualization, Methodology, Software, Validation, Investigation, Data curation, Writing – original draft, Writing – review & editing, Visualization. **Mehdi Kharrazi:** Methodology, Investigation, Writing – original draft, Writing – review & editing, Visualization, Supervision, Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Alomari, H., Stephan, M., 2020. Srcclone: Detecting code clones via decompositional slicing. In: ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020. pp. 274–284.

Bao, T., Burket, J., Woo, M., Turner, R., Brumley, D., 2014. BYTEWEIGHT: learning to recognize functions in binary code. In: Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22. 2014, pp. 845–860.

Boonstoppel, P., Cadar, C., Engler, D., 2008. Rwset: Attacking path explosion in constraint-based test generation. In: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. pp. 351–366.

Brumley, D., Poosankam, P., Song, D., Zheng, J., 2008. Automatic patch-based exploit generation is possible: Techniques and implications. In: 2008 IEEE Symposium on Security and Privacy (S & P 2008), 18-21 May 2008. Oakland, California, USA, pp. 143–157.

Caballero, J., Johnson, N., McCamant, S., Song, D., 2010. Binary code extraction and interface identification for security applications. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd 2010.

Cha, S., Avgerinos, T., Rebert, A., Brumley, D., 2012. Unleashing mayhem on binary code. In: IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012. San Francisco, California, USA, pp. 380–394.

Chandramohan, M., Xue, Y., Xu, Z., Liu, Y., Cho, C., Tan, H., 2016. Bingo: cross-architecture cross-os binary search. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016, pp. 678–689.

2020. Clang: a C language family frontend for LLVM. https://clang.llvm.org/ (accessed December 1st, 2020).

2019. CVE: Vulnerabilities by year. https://www.cvedetails.com/browse-by-date.php (accessed December 1st, 2019).

Danicic, S., Fox, C., Harman, M., Hierons, R., Howroyd, J., Laurence, M., 2005. Static program slicing algorithms are minimal for free liberal program schemas. Comput. J. 48 (6), 737–748.

Dashevskyi, S., Brucker, A., Massacci, F., 2019. A screening test for disclosed vulnerabilities in FOSS components. IEEE Trans. Softw. Eng. 45 (10), 945–966.

Du, X., Chen, B., Li, Y., Guo, J., Zhou, Y., Liu, Y., Jiang, Y., 2019. Leopard: identifying vulnerable code for vulnerability assessment through program metrics. In: Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019, pp. 60–71.

Egele, M., Woo, M., Chapman, P., Brumley, D., 2014. Blanket execution: Dynamic similarity testing for program binaries and components. In: Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014, pp. 303–317.

Engler, D., Chen, D., Chou, A., 2001. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In: Proceedings of the 18th ACM Symposium on Operating System Principles, SOSP 2001, Chateau Lake Louise, Banff, Alberta, Canada, October 21-24, 2001, pp. 57–72.

Eschweiler, S., Yakdan, K., Gerhards-Padilla, E., 2016. Discovre: Efficient cross-architecture identification of bugs in binary code. In: 23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016.

Ferrante, J., Ottenstein, K., Warren, J., 1987. The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 9 (3), 319–349.

Horwitz, S., 1990. Identifying the semantic and textual differences between two versions of a program. In: Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990, pp. 234–245.

Horwitz, S., Reps, T., 1991. Efficient comparison of program slices. Acta Inf. 28 (8), 713–732.

Hu, Y., Zhang, Y., Li, J., Gu, D., 2017. Binary code clone detection across architectures and compiling configurations. In: Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22-23, 2017, pp. 88–98.

Jang, J., Agrawal, A., Brumley, D., 2012. Redebug: Finding unpatched code clones in entire OS distributions. In: IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012. San Francisco, California, USA.

Kim, S., Woo, S., Lee, H., Oh, H., 2017. VUDDY: A scalable approach for vulnerable code clone discovery. In: 2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017. pp. 595–614.

Kiriansky, V., Waldspurger, C., 0000. Speculative buffer overflows: Attacks and defenses, CoRR.

Kolbitsch, C., Holz, T., Kruegel, C., Kirda, E., 2010. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In: 31st IEEE Symposium on Security and Privacy, S & P 2010, 16-19 May 2010. Berleley/Oakland, California, USA, pp. 29–44.

Li, F., Paxson, V., 2017. A large-scale empirical study of security patches. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017, pp. 2201–2215.

Li, Z., Zou, D., Xu, S., Jin, H., Qi, H., Hu, J., 2016. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In: Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016, pp. 201–213.

Lin, Y., Meng, G., Xue, Y., Xing, Z., Sun, J., Peng, X., Liu, Y., Zhao, W., Dong, J., 2017. Mining implicit design templates for actionable code reuse. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017, pp. 394–404.

Livshits, V., Lam, M., 2005. Finding security vulnerabilities in java applications with static analysis. In: Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005.

Newsome, J., Song, D., 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA.

2019. Nvd database. https://www.cvedetails.com/browse-by-date.php (accessed January 1st, 2019).

Ottenstein, K., Ottenstein, L., 1984. The program dependence graph in a software development environment. In: Riddle, W., Henderson, P. (Eds.), Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania, USA, April 23-25, 1984. ACM, pp. 177–184. http://dx.doi.org/10.1145/800020.808263.

Peng, H., Shoshitaishvili, Y., Payer, M., 2018. T-fuzz: Fuzzing by program transformation. In: 2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 2018. San Francisco, California, USA, pp. 697–710.

Perl, H., Dechand, S., Smith, M., Arp, D., Yamaguchi, F., Rieck, K., Fahl, S., Acar, Y., 2015. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. Denver, CO, USA, October 12-16, 2015.

Pewny, J., Garmany, B., Gawlik, R., Rossow, C., Holz, T., 2015. Cross-architecture bug search in binary executables. In: 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015. pp. 709–724.

Pewny, J., Schuster, F., Bernhard, L., Holz, T., Rossow, C., 2014. Leveraging semantic signatures for bug search in binary programs. In: Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014, pp. 406–415.

2019. PScan: A limited problem scanner for c source files. http://deployingradius.com/pscan (accessed December 1st, 2019).

Ramos, D., Engler, D., 2015. Under-constrained symbolic execution: Correctness checking for real code. In: 24th USENIX Security Symposium, USENIX Security 15, Washington, D.C. USA, August 12-14, 2015. pp. 49–64.

2022. Rapid7 report: Unauthenticated configuration export vulnerability (cve-2019-1663). https://www.rapid7.com/blog/post/2019/02/28/cisco-r-rv110-rv130-rv215-unauthenticated-configuration-export-vulnerability-cve-2019-1663-what-you-need-to-know/ (accessed December 27st, 2020).

2019. RATS: The rough auditing tool. https://security.web.cern.ch/security/recommendations/en/codetools/rats.shtml (accessed November 1st, 2019).

Shankar, U., Talwar, K., Foster, J., Wagner, D., 2001. Detecting format string vulnerabilities with type qualifiers. In: 10th USENIX Security Symposium, August 13-17, 2001. Washington, D.C. USA.

Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Krügel, C., Vigna, G., 2016. SOK: (state of) the art of war: Offensive techniques in binary analysis. In: IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016. pp. 138–157.

2019. Standard C library functions table. https://www.ibm.com/docs/en/i/7.3?topic=extensions-st{and}ard-c-library-functions-table-by-name (accessed September 1st, 2019).

Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G., 2016. Driller: Augmenting fuzzing through selective symbolic execution. In: 23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016.

Thomé, J., Shar, L., Bianculli, D., Briand, L., 2017. Joanaudit: a tool for auditing common injection vulnerabilities. In: E. Bodden, W. Schäfer, A. van Deursen, A. Zisman, (eds.) Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017, pp. 1004–1008.

Tip, F., 1994. A Survey of Program Slicing Techniques. Centrum voor Wiskunde en Informatica Amsterdam.

2021. VCCFinder code. https://github.com/hperl/vccfinder (accessed September 1st, 2021).

2019. Vuddy vulnerability database. https://github.com/squizz617/vuddy (accessed November 1st, 2019).

Weiser, M., slicing, P., 1981. Program slicing. In: Proceedings of the 5th International Conference on Software Engineering, San Diego, California, USA, March 9-12, 1981. IEEE Computer Society, pp. 439–449.

Xue, H., Venkataramani, G., Lan, T., 2018. Clone-slicer: Detecting domain specific binary code clones through program ing. In: Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation, FEAST '18, pp. 27–33.

Yamaguchi, F., Golde, N., Arp, D., Rieck, K., 2014. Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014. pp. 590–604.

Yamaguchi, F., Lindner, F., Rieck, K., 2011. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In: 5th USENIX Workshop on Offensive Technologies, WOOT'11, August 8, 2011, San Francisco, CA, USA, Proceedings. pp. 118–127.

Yamaguchi, F., Lottmann, M., Rieck, K., 2012. Generalized vulnerability extrapolation using abstract syntax trees. In: 28th Annual Computer Security Applications Conference, ACSAC 2012, OrlandO, FL, USA, 3-7 December 2012. pp. 359–368.

Yamaguchi, F., Maier, A., Gascon, H., Rieck, K., 2015. Automatic inference of search patterns for taint-style vulnerabilities. In: 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015. pp. 797–812.

Solmaz Salimi is a Ph.D. candidate in the field of Computer Engineering working at S4lab, Sharif University of Technology. Before that, she obtained her M.Sc. degree in computer engineering from Iran University of Science and Technology, Tehran, Iran, in 2015. Her research interests focus on software security, including software program analysis, vulnerability detection, analysis and exploitation techniques.

Mehdi Kharrazi received his B.E. in E.E. from the City College of New York, New York, in 1999, and M.Sc. and Ph.D. in E.E. from NYU Tandon, Brooklyn, NY, in 2002 and 2006. He is currently an Associate Professor with the Department of Computer Engineering, Sharif University of Technology, Tehran, Iran. His current research interests include software, system, and network security and is the director of Safety and Security in Software and Systems Laboratory (S4Lab).