Semantic-Aware Vulnerability Detection

Zhen Huang
School of Computing
DePaul University
Chicago, Illinois, USA
zhen.huang@depaul.edu

Marc White
School of Computing
DePaul University
Chicago, Illinois, USA
mwhite74@depaul.edu

Abstract—**Many vulnerability detection tools have been developed to detect vulnerabilities, but most of them only offer basic information for each detected vulnerability, which is insufficient for developers to reproduce and repair the vulnerability. We present an approach called SPDetect that uses safety properties to detect vulnerabilities and provides semantic information, including source code program expressions, about detected vulnerabilities. Each safety property defines the condition violated by one type of vulnerabilities. SPDetect uses symbolic execution to explore program paths and detect vulnerabilities by checking any violation of safety properties. To guide the symbolic execution to deep program paths that are more likely to contain vulnerabilities, we developed the novel technique of error path termination. We have designed and implemented a prototype of SPDetect for detecting vulnerabilities in C/C++ programs. Our evaluation of SPDetect on real-world programs shows that SPDetect can can detect vulnerabilities effectively and efficiently.**

Index Terms—**Software vulnerability, vulnerability detection, symbolic execution, safety property, program analysis**

I. INTRODUCTION

Vulnerability detection is essential to address software vulnerabilities, which exist in all kinds of software and are often exploited by real-world cyberattacks. As an example, the recent data breach of the private information of hundreds of millions of Facebook users [1], the LockFile ransomware attack [2], and the series of attacks on U.S. Federal government computer systems [3], [4] all took advantage of vulnerabilities.

For decades, a large number of tools have been developed to detect vulnerabilities [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16]. Some of them have been routinely used by major software vendors, in particular Google, Microsoft, and Adobe, on their software products.

After a critical vulnerability is detected by a vulnerability detection tool, the software vendor has to mitigate or repair the vulnerability in a timely manner, which usually involves reproducing, triaging the vulnerability, and developing the patch for the vulnerability. These tasks are often time-consuming and labor-intensive, albeit techniques have shown promising results in automated vulnerability mitigation and repair [17], [18], [19], [20], [21], [22], [23]. To accomplish these tasks rapidly, it requires vulnerability detection tools to provide as much information as possible on detected vulnerabilities. For example, the semantic information about the cause and effect of a vulnerability and a proof-of-concept exploit to trigger the vulnerability will be invaluable to software vendors.

However, most vulnerability tools can provide only limited information about the vulnerabilities they detect. This kind of information usually includes merely the location of the vulnerable code manifesting a vulnerability, typically the statement or instruction causing a software fault, the last step involved in the triggering of the vulnerability, and perhaps the call stack when the statement or instruction is being executed. It lacks high-level semantic information about the cause and effect of the vulnerability. With only the basic information, developers usually have to manually reproduce and debug the vulnerability to gather semantic information on the vulnerability, in order to fix the vulnerability correctly.

In this paper, we present our work that automatically detects vulnerabilities, produces a proof-of-concept exploit to reproduce each detected vulnerability, and provides semantic information, i.e. source code program expressions, involved in the vulnerability. We refer to our work as semantic-aware vulnerability detection.

Our work is inspired by the concept of safety properties used by Senx [21] to automatically generate vulnerability patches. A safety property is defined in terms of abstract expressions denoting the condition that will prevent a type of vulnerabilities from being triggered. The safety property for a buffer overflow vulnerability, for example, is defined as a check between the abstract expression denoting the range of the addresses accessed by a memory access and the abstract expression denoting the size of the buffer targeted by the memory access. A safety property is violated when a vulnerability corresponding to the safety property is triggered. To generate vulnerability patches, the abstract expressions in a safety property is automatically translated into different program expressions for different target programs.

While Senx generates patches for known vulnerabilities, our goal is to detect undiscovered vulnerabilities. To generate a patch for a vulnerability, Senx requires a user to provide an exploit for triggering the vulnerability. On the contrary, our work automatically produces a proof-of-concept exploit for each detected vulnerability.

Our work uses symbolic execution to explore the program paths and detect vulnerabilities by checking safety properties. It uses symbolic input to execute a target program and produces an exploit by concretizing the symbolic input that triggers the vulnerability. To generate program expressions most relevant to the vulnerability, it translates the abstract

expressions related to the corresponding safety property into program expressions.

During the symbolic execution, our work checks any violation of safety properties to determines whether a vulnerability is detected. When a safety property is violated, it deems that a vulnerability is detected and produces a concrete input, i.e. exploit, based on the path constraints leading to the code that violates the safety property.

One challenge of our work is to develop an efficient search strategy for the symbolic execution. The search strategy is critical for detecting vulnerabilities efficiently. It should guide the symbolic execution to focus on deep program paths that are likely to contain vulnerabilities. First, any program with a decently complex code base will have an infinite number of possible program paths, mainly due to the existence of loops whose number of iterations is determined by values derived from user input. As a result, it is preferable to give higher priority to the program path more susceptible to have vulnerabilities. Second, many programs expect a specific format for their input and they will terminate early if they consider an input invalid because the format of the input does not obey the expected format. Without any constraints on the symbolic input, the symbolic execution will spend most of the time producing inputs considered invalid by target programs and execute only shallow program paths.

We address the challenge with two techniques: *error path termination* and *input preconditions*. Both of them are for the purpose of avoiding shallow program paths that terminate program execution early.

Error path termination directs symbolic execution to skip the program branches that will lead to early program terminations, such as the ones due to invalid input formats. With static analysis, our work identifies error handling code in target programs and provides the information to the symbolic execution engine, which terminates any execution path reaching an error path.

The input preconditions enforce specific constraints on symbolic inputs to make the symbolic inputs conform to the common input formats expected by target programs. Our input preconditions, combined with the input preconditions proposed in the prior work [14], enables symbolic execution to bypass the format check code in target programs and proceed to deep program paths.

This paper makes the following main contributions:

- We propose an approach to automatically detect vulnerabilities and provide program expressions involved in each detected vulnerability. Our approach uses symbolic execution to execute target programs and detects vulnerabilities by checking for any violation of safety properties.
- We develop a technique called error path termination and two input preconditions to improve the performance of the symbolic execution in detecting vulnerabilities.
- We have implemented our approach in a prototype called SPDetect. We describe our design and implementation in the paper.
- Our evaluation on SPDetect shows that our approach can effectively and efficiently discover different types of real-world vulnerabilities.

The structure of paper is as follows. We present the background and related work in Section II. We discuss the motivation and challenges in Section III. Section IV illustrates a typical usage of our approach. We describe our design in Section V, present our evaluation in Section VI, and discuss the limitation of our work in Section VII. Finally we conclude in Section VIII.

II. BACKGROUND AND RELATED WORK

A. Safety Properties

Each safety property corresponds to a type of vulnerabilities. It abstracts the intrinsic condition that prevents a type of vulnerabilities from being triggered. It is defined as an abstract boolean expression that can be evaluated when translated to concrete program variables or expressions in a program. We describe the three types of safety properties currently supported by SPDetect.

Buffer overflows. A buffer overflow happens when a sequence of memory accesses traversing a buffer crosses the boundary of the buffer, i.e. the lower bound or the upper bound of the buffer. The safety property for buffer overflows involves two abstract expressions: a memory access and a buffer. A memory access corresponds to a pointer dereference or an access to an array element. A buffer refers to any bounded memory region, which may include arrays, structs, or class objects.

Bad casts. Many times a programmer mistakenly casts a pointer to a type that is incompatible with the object pointed to by the pointer. A bad cast occurs when such a pointer is used to perform memory access. The safety property for bad casts can detect bad casts for structs and objects.

Integer overflows. An integer overflow refers to the case when an arithmetic operation produces a value larger or smaller than what can be represented by the data type of the operation or the variable in which the result of the operation will be stored. An integer overflow vulnerability corresponds to two cases: 1) the result of an integer overflow is used as the allocation size for a memory allocation and 2) the result of an integer overflow is used as a predicate that controls the execution of a sensitive operation.

B. Vulnerability Detection

Vulnerability detection tools use a myriad of techniques including a variety of dynamic analysis, such as symbolic execution [14], fuzzing [7], and taint analysis [6], and different types of static analysis [12], [24]. In this section, we focus on symbolic execution, which is the basis of our work.

Symbolic execution runs target programs and abstracts program data as symbolic values during program executions so that it can automatically produce test cases to detect bugs [5]. The symbolic values constitute the program path constraints for each program execution, which are converted to concrete input data by using constraint solvers.

EXE [25] is a pioneering work in symbolic execution. It runs a target program with symbolic input and uses symbolic

constraints on the input to replace operations involving the data derived from symbolic input. When a conditional branch is encountered, it forks the execution to two executions: one follows the `True` branch if the predicate for the conditional statement can evaluate to a `True` value and the other one follows the `False` branch if the predicate can evaluate to a `True` value.

A main challenge of symbolic execution is path explosion [26]. When symbolic execution explores all possible execution paths of a program, branches in the program will cause the number of paths to grow exponentially and severely limit the scalability of symbolic execution. Many approaches have been proposed to address the issue of path explosion [27], [14], [28], [29], [30], [31], [32].

Like our error path termination, some techniques avoid executing uninteresting paths [14], [27], [29]. AEG [14] proposes the use of input preconditions to avoid exploring execution paths reached by invalid inputs. The input preconditions can considerably reduce the number of execution paths, at the expense of increased the cost of constraint solving. Our error path termination focuses on the effect of invalid inputs and does not increase the cost of constraint solving.

Chopped symbolic execution [27] allows users to manually specify uninterested parts of the target program code that should be skipped during execution. To ensure the integrity of symbolic execution, it lazily executes the skipped code when they are needed by the rest of the code. Our error path termination focuses on one particular type of uninterested code, error handling code, and automatically identifies error handling code in target programs.

Some techniques [31], [32] aim to prune similar execution paths that will not allow symbolic execution to reach more interesting execution paths. The challenge of these techniques is to define and identify similar execution paths effectively and efficiently.

III. PROBLEM DEFINITION

To aid developers in fixing vulnerabilities, simply reporting the location of the involved statements is often not enough. It is important for vulnerability detection tools to provide detailed information on each detected vulnerability. Particularly the program expressions involved in the manifestation of a vulnerability, *vulnerability-relevant code*, and an input to reproduce the vulnerability, *proof-of-concept exploit* or *exploit*, are critical information required for fixing the vulnerability.

For instance, typically a buffer overflow vulnerability is fixed in three different approaches: restricting the range of the memory access to the buffer, accommodating the buffer size based on the range of the memory access, and preventing the memory access to the buffer if the memory access will overflow the buffer. All fix approaches require the information on the statement allocating the buffer, the allocation size, the statement accessing the buffer, and the range of memory access.

We use the code in Listing 1 as a running example to illustrate such critical information required to fix vulnerabil-

```
1  int main(...) {
2    ...
3    if ((pszMailRoot = SysGetEnv(MAIL_ROOT)) == NULL)
     {
4      fprintf(stderr, "undefined variable:_%s\n",
         MAIL_ROOT);
5      return 1;
6    }
7    ...
8    FILE *pMailFile = fopen(szMailFile, "wb");
9    if (pMailFile == NULL) {
10     perror(szMailFile);
11     return 5;
12   }
13   int bRcptSource = strnicmp(szBuffer, "To:", 3) ==
       0 ||
14     strnicmp(szBuffer, "Cc:", 3) == 0 ||
15     strnicmp(szBuffer, "Bcc:", 4) == 0;
16   if (bRcptSource)
17     EmitRecipients(...);
18   ...
19 }
20
21 int EmitRecipients(...) {
22   ...
23   for (; pszCurr != NULL && *pszCurr != '\0';) {
24     char *pszAt = strchr(pszCurr, '@');
25     char szAddress[256] = "";
26
27     if (pszAt == NULL) break;
28     pszCurr = AddressFromAtPtr(pszAt, szAddress,
         sizeof(szAddress));
29     ...
30   }
31   ...
32 }
33
34 char *AddressFromAtPtr(...) {
35   ...
36   for (; (*pszEnd != '\0'; pszEnd++)
37     if (strchr("<>_\t,\":;'\r\n", *pszEnd) != NULL
       )
38       break;
39   int iAddrLength = (int)(pszEnd - pszStart);
40   strncpy(pszAddress, pszStart, iAddrLength);
41   ...
42 }
```

Listing 1: Example vulnerability, adopted from CVE-2004-2943 in `XMail`.

ities. The code is adopted from a real-world buffer overflow vulnerability in a mail server, `XMail`. The vulnerability allows an attacker to hijack the execution of the mail server.

The code shows the code in three functions, `main`, `EmitRecipients`, and `AddressFromAtPtr`, that are responsible of extracting the list of email addresses from the user input. The code allocates a buffer at line 25 in function `EmitRecipients` and the buffer overflow occurs at line 40 in function `AddressFromAtPtr` when the call to `strncpy` copies more data than expected into the memory buffer pointed to by the pointer `pszAddress`.

To fix this vulnerability, a developer needs the program expressions for the buffer allocation size, which is `256` as shown on line 25, and the range of memory access to the buffer, which is variable `iAddrLength` or expression `pszEnd − pszStart` on line 39, besides the location of the buffer allocation statement and the memory access statement.

A proof-of-concept exploit, the user input triggering the vulnerability, is also needed by the developer to reproduce the vulnerability and test the fix. The exploit for this vulnerability must cause the program execution to reach function `AddressFromAtPtr`, which is invoked by function `EmitRecipients` and function `main` in turn. We can see that the exploit has to starts with "To:", "Cc:", or "Bcc:" so that `main` will call `EmitRecipients`, as shown in line 13–15. To be able to call `AddressFromAtPtr`, the exploit must also contain a '@', checked at line 24 and 27. Finally to overflow the buffer, the exploit should not contain characters such as '<', '>', and '"', which are considered as separators, as shown on line 37–38.

IV. SPDetect

We design an approach, called SPDetect, to detect vulnerabilities and address the challenges presented in Section III. SPDetect symbolically executes the source code of a target program to identify vulnerabilities. It works on the LLVM bitcode of the target program. This section demonstrates a typical usage of SPDetect in detecting vulnerabilities.

Our target program is `XMail` [33], a popular mail server written in 33,542 lines of C++ source code. A user can use SPDetect to detect vulnerabilities in `XMail` in a few steps:

First, the user needs to build the LLVM bitcode from the C++ source code of the program. Second, she can create the safety properties corresponding to the vulnerability types to be detected or use the default safety properties. Third, she runs SPDetect on the bitcode and specifies the safety properties to be used and that a symbolic input should be used for the program. Fourth, SPDetect explores the program paths and detects the buffer overflow vulnerability illustrated in Listing 1. Lastly, SPDetect produces a concrete input for triggering the vulnerability and the detailed information about the vulnerability to help the user fixing the vulnerability, such as the involved program statements and expressions.

SPDetect executes the LLVM bitcode of the program. When it reaches each conditional statement in the program, it checks whether the predicate evaluated by the conditional statement is a symbolic value. If so, it uses a constraint solver to compute the satisfiability of the symbolic value and diverts program execution to the branches based on whether the symbolic value can be evaluated to `True`, `False`, or either.

When the program execution reaches the call to `strncpy` at line 40 and triggers the violation of the safety property for buffer overflows while executing `strncpy`, SPDetect creates a concrete input that satisfies the constraints that lead to the call. The input is 260 characters long and satisfies the requirements of starting with "To:", containing an '@', not containing any special characters such as '<', '>', and line feed.

Based on the violated safety property, it also translates the abstract expressions involved in the safety property into the corresponding program expressions. For example, it will translate `buffer_size` into `256` and `access_range` into

```
Safety Property:       buffer overflow
Access Statement:      line 40, AddressFromAtPtr
Access Range:          iAddrLength,
    AddressFromAtPtr
Buffer Size:           256, EmitRecipients
Buffer Allocation:     line 25, EmitRecipients
```
Listing 2: Semantic Information for CVE-2004-2943

`iAddrLength`. It will report that the buffer allocation occurred at line 25 and the memory access was invoked from line 40, as shown in Listing 2.

Using the input and the report generated by SPDetect, a developer can debug the program with the input and produce a patch that ensures `iAddrLength`, the length used by `strncpy`, is not larger than 256, the buffer size.

V. Design

In this section, we present the design of SPDetect, our approach to detect vulnerabilities and provide an exploit and program expressions denoting the cause and effect of each detected vulnerability.
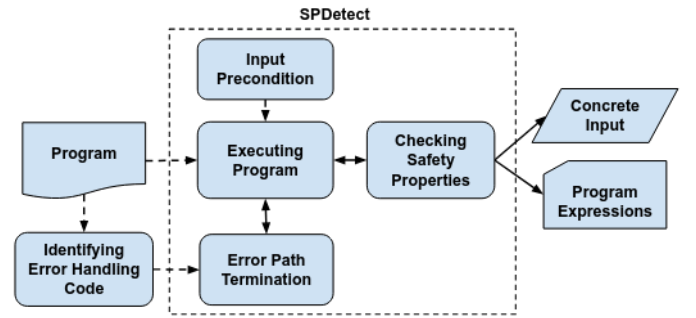

Fig. 1: Workflow of SPDetect.

A. Overview

SPDetect takes the source code of a program as input and uses symbolic execution to explore the program paths of the source code to detects vulnerabilities. When a vulnerability is detected, it produces a concrete input (exploit) to trigger the vulnerability and the program expressions representing the cause and effect of the vulnerability. Figure 1 shows the workflow of SPDetect.

The symbolic execution performed by SPDetect checks safety properties to determine whether a vulnerability is triggered. A safety property defines the condition when a type of vulnerabilities will not occur, in terms of abstract expressions. SPDetect translates the abstract expressions involved in a safety property into program expressions for the program being executed, and evaluates these program expressions during program execution to check whether a safety property is violated. If so, SPDetect considers a vulnerability is detected.

As vulnerabilities typically exist in deep program paths, SPDetect guides the symbolic execution to focus on deep program paths to improve the efficiency of vulnerability detection.

SPDetect leverages two techniques: error path termination and input precondition. Both of them allow the symbolic execution to avoid the shallow program paths that terminates program execution due to invalid inputs.

B. Checking Safety Properties

SPDetect checks the violation of safety properties during the symbolic execution of a target program in order to detect vulnerabilities. A safety property is defined for the purpose of detecting a type of vulnerabilities, such as buffer overflows, integer overflows, and bad casts.

Each safety property is associated with one ore more operations of the program. A safety property is checked when SPDetect executes its associated operations. For example, the safety property for buffer overflows is associated with all memory access because the safety property denotes that the range of a memory access should not exceed the size of the buffer targeted by the memory access. Therefore SPDetect checks the safety property for buffer overflows when executing each memory access.

SPDetect checks the violation of a safety property by evaluating the safety property. If the safety property is evaluated to be `True`, the safety property is not violated. If the safety property is evaluated to be `False`, the safety property is violated.

Because a safety property is defined in abstract expressions, SPDetect needs to translate these abstract expressions into corresponding program values to evaluate them. It follows the description in Table I to perform the translation.

TABLE I: Abstract Expression in Safety Properties

Expression	Description
$address()$	the memory address to be accessed
$buffer\_lower()$	the lower bound address of a buffer
$buffer\_upper()$	the upper bound address of a buffer
$buffer\_freed()$	Is the buffer freed?
$arg\_from\_input(F, i)$	Is the call argument #*i* for function F derived from the input?
$arg\_overflowed(F, i)$	Is the call argument #*i* for function F an overflowed integer?
$predicate\_overflowed()$	Is the predicate evaluated to an overflowed integer?
$cast\_struct()$	Is the pointer cast to a struct?

SPDetect produces a concrete input leading the program execution to the code causing the violation when it detects a violation of a safety property. It uses the Z3 constraint to create the concrete input that satisfies all the path constraints at the code. It also generates a report listing the program expressions involved in evaluating the safety property.

Assume we use SPDetect to detect vulnerabilities in the program presented in Listing 1. SPDetect will detect a violation of the safety property for buffer overflows when it executes the code of function `strncpy`, which is called from line 40. It then uses the constraint solver to find a solution that satisfies all the path constraints for the path reaching line 40.

SPDetect also translates the abstract expressions relevant to the safety property into source code program expressions in a report. We describe the report in details in Section VI-D.

C. Error Path Termination

We observe that many programs reject invalid inputs and terminate execution early. In other words, only shallow program paths are executed in the face of invalid inputs. Because these program paths are shallow, they have low probability to contain vulnerabilities.

One approach to address this issue is to ensure the inputs generated by symbolic execution are always in valid formats. The format specification may be translated into constraints that can be applied to symbolic inputs. But complicated format specifications will lead to complicated constraints that are challenging for a constraint solver to solve.

Instead, we choose to guide the symbolic execution to avoid executing shallow program paths that handles the errors caused by invalid inputs. This requires us to identify such program paths and direct the symbolic execution at runtime.

First, we use static analysis to identify error handling code because invalid inputs usually raise certain errors. We build our static analysis on Talos [18]. Talos uses heuristics to identify the set of basic error return values and then follows the propagation of errors in the call chain to expand the set of error return values. For each identified error handling code, Talos provides the function return value returned by the error handling code and the predicate evaluated by the conditional statement that governs the execution of the error handling code. We extend Talos to return the source code line of the start of the error handling code.

Second, our approach directs the symbolic execution to avoid executing error handling code. As error handling code is usually implemented in the form of a conditional statement such as an `if` statement with one of the branches leading to the error handling code, i.e error path, we terminate any execution path that reaches the start of an error path. This technique is called error path termination.

We use the code in Listing 1 to illustrate how this works. Before the symbolic execution for this program starts, SPDetect statically identifies that the `then` branches starting at line 4 and at line 10 are the start of error paths. As a result, SPDetect will terminate any execution path that reaches line 4 or line 10. This way SPDetect cut off the program path that propagate the error. Otherwise the symbolic execution will follow the error path to propagate the error to all the caller functions until the program terminates.

D. Input Precondition

As discussed in Section V-C, it is challenging for a constraint solver to solve complicated constraints. However, simple constraints can be solved efficiently and applying them on symbolic inputs can improve the overall performance of symbolic execution, as shown by prior work such as AEG [14].

This kind of constraints applied to symbolic inputs are called *input preconditions* because they are applied before the symbolic execution starts. AEG uses two kinds of input preconditions: *known length*, which ensures string inputs to have a fixed length, and *known prefix*, which ensures inputs to always start with a prefix.

As a complement of error path termination, we apply input precondition to guide the symbolic execution to deep program paths. This is mainly because a program does not necessarily treat all invalid inputs in the same way. Some invalid inputs can directly lead to error handling code, while less severe invalid inputs can be ignored silently.

Besides the input preconditions proposed by AEG, we developed two input preconditions, *infix* and *exclusion*, to specify the bytes that must be included in inputs and the bytes that must *not* be included in inputs, respectively.

We note that users do not need to examine the source code of the program to use these preconditions. All the three requirements are described in the document of the program [34] or in the RFC 5332 specification that specifies the Internet message format [35]. It is straightforward for a user to create these preconditions based on either one of these documents.

VI. EVALUATION

In this section, we evaluate the effectiveness and performance of SPDetect in detecting vulnerabilities. We evaluate SPDetect with five popular open-source C/C++ programs. First, we describe the setup of the environment of our experiments. Second, we measure the effectiveness of SPDetect with its result on detecting real-world vulnerabilities. We also measure its performance by the time it takes to detect these vulnerabilities. Third, we evaluate the effect of our error path termination, and the infix and exclusion input preconditions. Last, we demonstrate the benefits of providing semantic information for detected vulnerabilities with a case study.

A. Experimental Setup

We implement the prototype of SPDetect in C/C++ by extending KLEE [36], a state-of-art symbolic execution engine. We use the GiNaC symbolic computation framework [37] to evaluate safety properties. For constraint solving, we use the Z3 constraint solver [38]. We perform all our evaluations on a workstation equipped with a 32-core 2.2GHz AMD Ryzen Threadripper processor and 128 GB memory. The workstation runs 64-bit Ubuntu 20.04.

B. Vulnerability Detection

We found these five programs from popular online vulnerability databases and exploit databases including CVE Details [39], BugZilla [40], and Exploit Database [41]. These programs belong to a variety of types, including email server, OCR tool, image processing tool, Internet data transfer tool, and data compression tool. Their size varies from 1,436 to 53,592 lines of source code. We list these programs in Table II. For each program, the column "#SLOC" is its number of C/C++ source code lines and the column "#Bitcode" is its number of LLVM bitcode instructions.

TABLE II: List of Evaluated Programs.

Program	Type	Version	#SLOC	#Bitcode
XMail	Email Server	1.27	53,592	38,856
gocr	OCR Tool	0.40	21,595	151,775
autotrace	Image Processing Tool	0.31.1	12,237	122,830
socat	Internet Data Transfer Tool	1.4	17,923	199,317
ncompress	Data Compression Tool	4.2.4	1,436	33,476

We run SPDetect on each of these programs to detect vulnerabilities. We use the DFS search strategy and the Z3 constraint solver. Table III lists the nine vulnerabilities that are successfully detected by SPDetect. The column "Vuln.Type" describes the type of each vulnerability and the column "Det.Time" presents the time for the first occurrence of SPDetect detecting each vulnerability, measured in seconds.

TABLE III: List of Detected Vulnerabilities.

Vulnerability	CVE#	Vuln. Type	Det. Time
XMail	CVE-2005-2943	buffer overflow	102
gocr-1	CVE-2005-1141	integer overflow	26
gocr-2	CVE-2005-1142	integer overflow	53
autotrace-1	CVE-2017-9166	buffer overflow	17
autotrace-2	CVE-2017-9169	buffer overflow	12
autotrace-3	CVE-2017-9182	integer overflow	9
socat	CVE-2004-1484	buffer overflow	30
ncompress-1	CVE-2001-1413	buffer overflow	34
ncompress-2	CVE-2006-1168	buffer overflow	305

Six of the vulnerabilities are buffer overflows and three of them are integer overflows. No bad casts are detected in this experiment. On average it takes SPDetect 65 seconds to detect a vulnerability. All the nine vulnerabilities are detected within 305 seconds.

C. Error Path Termination and Input Precondition

We conducted experiments to show the effect of our error path termination and input precondition on the performance of SPDetect in detecting vulnerabilities. We compare SPDetect the *prefix* and *length* input preconditions proposed by AEG, as well as KLEE. Like AEG, we set the maximum symbolic execution time to 10,000 seconds and terminate the symbolic execution after the maximum time is reached.

Our result is presented in Figure 2. Each bar represents the detection time when a particular technique is applied. The "Prefix + Length" bar shows the detection time when only the prefix and length input preconditions are applied. The "SPDetect" bar presents the detection time when our error path termination and input precondition are applied on top of the prefix and length input precondition. The "Concolic" bar presents the detection time with a concrete proof-of-concept exploit. The techniques that fails to detect a vulnerability within the time limit are shown as a bar of maximum time. As we can see, KLEE fails to detect any vulnerabilities within the specified time limit.

Two of the gocr vulnerabilities *cannot* be detected within the time limit by using only the prefix and length preconditions. With error path termination and input precondition, they can be detected in 26 and 53 seconds respectively. This
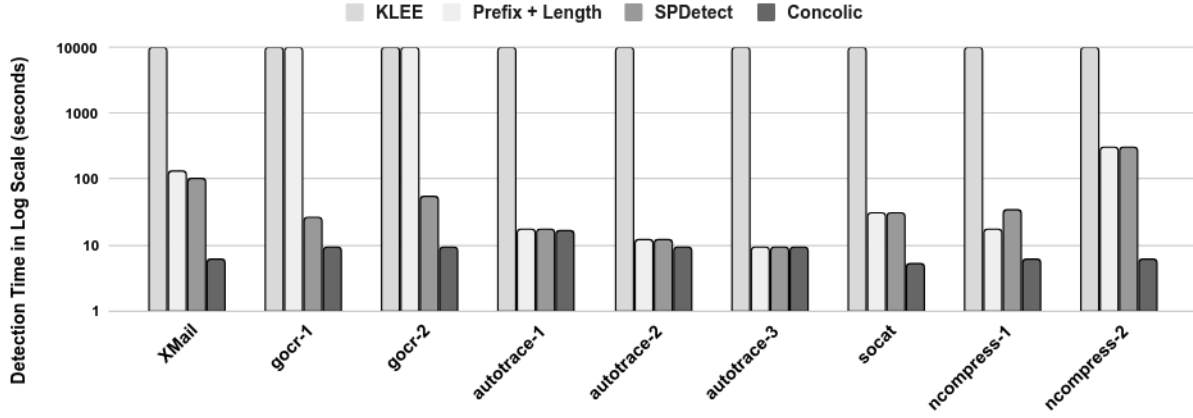
Fig. 2: Vulnerability Detection Time

shows the two preconditions are necessary for detecting some vulnerabilities.

The `XMail` vulnerability can be detected with the prefix and length preconditions in 132 seconds. On the contrary, SPDetect detects it in 102 seconds with the help of the infix and exclusion preconditions. This indicates that they can improve the performance of vulnerability detection, although they add more complexity in constraint solving.

One of the `ncompress` vulnerabilities is detected with the prefix and length preconditions in 17 seconds. However it takes 34 seconds when the exclusion precondition is also applied. In this case, the overhead of the exclusion precondition outweighs its benefits.

For the three `autotrace` vulnerabilities, SPDetect can detect them in about the same time as the concolic executions using concrete proof-of-exploits. This demonstrates that the error path termination and the input precondition can remarkably improve the performance in vulnerability detection.

D. Benefits of Semantic Information

In this section, we illustrate the benefits of semantic information using our running example, shown in Listing 1, and the corresponding report generated by SPDetect, shown in Listing 2.

The report shows the type of the violated safety property, e.g. buffer overflow. It also lists the program expressions involved in the buffer overflow, including the location of the statement that accesses a buffer, the range of the access, the size of the buffer, and the statement that allocates the buffer.

Because the report shows the program expressions in the form of source code, a developer can directly use the program expressions for creating a patch for the buffer overflow. The report also shows the scope of each program expression to help the developer decide where in the code to apply the patch. For example, the access range is denoted by `iAddrLength` and valid in the scope of `AddressFromAtPtr` while the buffer size is denoted by `256`, an integer constant. A possible scope to apply the patch is `AddressFromAtPtr` because

both the access range and buffer size are valid in this scope, which are typically required to patch a buffer overflow. In contrast, it will be more difficult to apply the patch to function `EmitRecipients` because the access range is not valid in this scope.

VII. LIMITATION

SPDetect avoids symbolic execution on error handling code in order to focus on deep program paths because error handling code is relatively simple and shallow and thus unlikely to contain vulnerabilities. Nonetheless, SPDetect will miss vulnerabilities in error handling code.

Our prototype of SPDetect requires a user to manually define the input preconditions. While the user can examine the documentation on common input formats or program manuals to create input preconditions, it is possible to aid the user or automatically create input preconditions by analyzing program code or learning from program behavior, such as using clustering [42]. We plan to develop such technique as our future work.

VIII. CONCLUSION

We present our approach for detecting vulnerabilities using safety properties in order to provide semantic information about detected vulnerabilities. The approach uses symbolic execution to run target programs and detect vulnerabilities by checking any violation of safe properties. When the manifestation of a vulnerability violates a safety property, the approach generates a exploit to trigger the vulnerability and derives source code program expressions relevant to the vulnerability from the violated safety property. To improve the performance of our approach, we developed a technique to terminate error path and two input preconditions to guide the symbolic execution to focus on deep program paths. We evaluated our approach on five popular programs and it detects real world vulnerabilities effectively and efficiently.

REFERENCES

[1] "533 million Facebook users' phone numbers and personal data have been leaked online," https://www.businessinsider.com/stolen-data-of-533-million-facebook-users-leaked-online-2021-4, 2021.

[2] "PetitPotam Vulnerability Exploited in Ransomware Attacks," https://www.securityweek.com/petitpotam-vulnerability-exploited-ransomware-attacks, 2021.

[3] "Russian government hackers are behind a broad espionage campaign that has compromised U.S. agencies, including Treasury and Commerce," https://www.washingtonpost.com/national-security/russian-government-spies-are-behind-a-broad-hacking-campaign-that-has-breached-us-agencies-and-a-top-cyber-firm/2020/12/13/d5a53b88-3d7d-11eb-9453-fc36ba051781_story.html, 2020.

[4] "VMware Flaw a Vector in SolarWinds Breach?" https://krebsonsecurity.com/2020/12/vmware-flaw-a-vector-in-solarwinds-breach/, 2020.

[5] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[6] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software." in *NDSS*, vol. 5. Citeseer, 2005, pp. 3–4.

[7] 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, 2010, pp. 497–512.

[8] F. Yamaguchi, F. Lindner, and K. Rieck, "Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning," in *Proceedings of the 5th USENIX Conference on Offensive Technologies*, ser. WOOT'11. USA: USENIX Association, 2011, p. 13.

[9] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "Vulpecker: An automated vulnerability detection system based on code similarity analysis," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ser. ACSAC '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 201–213. [Online]. Available: https://doi.org/10.1145/2991079.2991102

[10] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 297–308. [Online]. Available: https://doi.org/10.1145/2884781.2884804

[11] F. Wu, J. Wang, J. Liu, and W. Wang, "Vulnerability detection with deep learning," in *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*, 2017, pp. 1298–1302.

[12] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," in *Proceedings of the 25th Annual Network and Distributed System Security Symposium*, vol. abs/1801.01681, 2018.

[13] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 259–269.

[14] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," *Communications of the ACM*, vol. 57, no. 2, pp. 74–84, 2014.

[15] A. Aumpansub and Z. Huang, "Detecting software vulnerabilities using neural networks," in *ICMLC 2021: 13th International Conference on Machine Learning and Computing, Shenzhen China, 26 February, 2021- 1 March, 2021*. ACM, 2021, pp. 166–171. [Online]. Available: https://doi.org/10.1145/3457682.3457707

[16] Z. Huang and X. Yu, "Integer overflow detection with delayed runtime test," in *ARES 2021: The 16th International Conference on Availability, Reliability and Security, Vienna, Austria, August 17-20, 2021*, D. Reinhardt and T. Müller, Eds. ACM, 2021, pp. 28:1–28:6. [Online]. Available: https://doi.org/10.1145/3465481.3465771

[17] Z. Huang and G. Tan, "Rapid Vulnerability Mitigation with Security Workarounds," in *Proceedings of the 2nd NDSS Workshop on Binary Analysis Research*, ser. BAR '19, February 2019.

[18] Z. Huang, M. DAngelo, D. Miyani, and D. Lie, "Talos: Neutralizing Vulnerabilities with Security Workarounds for Rapid Response," in *Proceedings of 2016 IEEE Symposium on Security and Privacy*, ser. S&P 2016, May 2016, pp. 618–635.

[19] Z. Huang, T. Jaeger, and G. Tan, "Fine-grained program partitioning for security," in *Proceedings of the 14th European Workshop on Systems Security*, ser. EuroSec '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 21–26. [Online]. Available: https://doi.org/10.1145/3447852.3458717

[20] S. M. S. Talebi, Z. Yao, A. A. Sani, Z. Qian, and D. Austin, "Undo workarounds for kernel bugs," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2381–2398.

[21] Z. Huang, D. Lie, G. Tan, and T. Jaeger, "Using safety properties to generate vulnerability patches," in *2019 IEEE Symposium on Security and Privacy (SP)*, May 2019, pp. 539–554.

[22] R. Shariffdeen, Y. Noller, L. Grunske, and A. Roychoudhury, "Concolic program repair," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 390–405.

[23] P. Muntean, M. Monperrus, H. Sun, J. Grossklags, and C. Eckert, "Intrepair: Informed repairing of integer overflows," *IEEE Transactions on Software Engineering*, vol. 47, no. 10, pp. 2225–2241, 2019.

[24] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 595–614.

[25] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, Dec. 2008. [Online]. Available: https://doi.org/10.1145/1455518.1455522

[26] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, may 2018. [Online]. Available: https://doi.org/10.1145/3182657

[27] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar, "Chopped symbolic execution," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 350–360. [Online]. Available: https://doi.org/10.1145/3180155.3180251

[28] H. Wang, T. Liu, X. Guan, C. Shen, Q. Zheng, and Z. Yang, "Dependence guided symbolic execution," *IEEE Transactions on Software Engineering*, vol. 43, no. 3, pp. 252–271, 2017.

[29] D. A. Ramos and D. Engler, "Under-Constrained symbolic execution: Correctness checking for real code," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 49–64. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ramos

[30] Y. Li, Z. Su, L. Wang, and X. Li, "Steering symbolic execution to less traveled paths," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages amp; Applications*, ser. OOPSLA '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 19–32. [Online]. Available: https://doi.org/10.1145/2509136.2509553

[31] S. Guo, M. Kusano, C. Wang, Z. Yang, and A. Gupta, "Assertion guided symbolic execution of multithreaded programs," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 854–865. [Online]. Available: https://doi.org/10.1145/2786805.2786841

[32] P. Boonstoppel, C. Cadar, and D. Engler, "Rwset: Attacking path explosion in constraint-based test generation," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 351–366.

[33] "XMail Mail Server," http://www.xmailserver.org/, 2021.

[34] "XMail Reference Doc," http://xmailserver.org/xmaildoc.htm, 2021.

[35] "Internet Message Format," https://datatracker.ietf.org/doc/html/rfc5322, 2021.

[36] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*. San Diego, CA: USENIX Association, Dec. 2008.

[37] "GiNaC is Not a CAS," http://www.ginac.de/, 2021.

[38] "The Z3 Theorem Solver," https://github.com/Z3Prover/z3, 2021.

[39] "CVE Details," http://www.cvedetails.com, 2021.

[40] "BugZilla," http://bugzilla.maptools.org/, 2021.

[41] "Exploit Database," https://www.exploit-db.com/, 2021.

[42] Z. Huang and D. Lie, "Ocasta: Clustering configuration settings for error recovery," in *Proceedings of 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN '14, June 2014, pp. 479–490.