Scalable Static Detection of Use-After-Free Vulnerabilities in Binary Code

KAILONG ZHU, YULIANG LU, AND HUI HUANG

College of Electronic Engineering, National University of Defense Technology, Hefei 230037, China

Corresponding author: Yuliang Lu (cee401_nudt@126.com)

This work was supported in part by the National Key Research and Development Project of China under Grant 2017YFB0802900.

ABSTRACT The number of use-after-free vulnerabilities has been increasing rapidly in recent years, which is a serious threat to computer systems. However, few effective mitigations exist for large-scale binary code. In this study, the authors propose a scalable static approach for detecting use-after-free vulnerabilities in binary code. First, the use-after-free feature model is proposed to provide guidance for detection. Then, the binary code of the target program is converted to an intermediate representation, and CFGs (control flow graphs) are constructed. Finally, lightweight pointer tracking is performed to identify the use-after-free vulnerable point. Compared with state-of-the-art approaches, this approach uses function summaries rather than naive in-lining technique for the inter-procedural analysis in the vulnerability detection. Therefore, our approach has the ability to avoid redundant repeat analysis caused by the in-lining technique in the existing approaches and reduce the unnecessary performance overhead. The authors have implemented a prototype called UAFDetector and evaluated it using standard benchmarks and real-world programs. The experimental results show that this approach is effective in detecting use-after-free vulnerabilities in binary code and is more efficient and scalable than state-of-the-art static solutions.

INDEX TERMS Vulnerability detection, use-after-free, function summary, UAFDetector.

I. INTRODUCTION

When an object is deallocated in a program, the pointer to it becomes a dangling pointer. Any dereference of dangling pointers causes a use-after-free vulnerability that can be exploited by an attacker. Moreover, a double-free vulnerability is considered a special use-after-free vulnerability [1].

Fig. 1 shows the number of use-after-free and double-free vulnerabilities recorded in the National Vulnerability Database (NVD) [2] since 2009. These two classes of vulnerabilities are increasingly common. These vulnerabilities are present in various applications, including Microsoft’s and Google’s browsers, Windows and Linux operating system kernels, and OpenJPEG and LibPNG image processing libraries. Attackers can exploit these vulnerabilities to read or rewrite sensitive data [3]–[6] and can even hijack the program control flow to execute arbitrary code [7]–[9]. In the famous Operation Aurora attack, the attacker exploited several 0-day vulnerabilities, including use-after-free vulnerabilities, to attack the networks of Google and Adobe, causing serious economic losses [10].

Current solutions for use-after-free vulnerability detection fall into two categories: dynamic analysis and static analysis. Most solutions rely on dynamic analysis [7], [11]–[14], which generally have a high performance overhead and low code coverage. Static analysis does not suffer from such limitations. However, there are only a few studies [1], [15], [16]...
on static analysis for use-after-free detection, and they are not widely used in practice due to some important limitations. The solutions in [1] and [15] are only able to analyze the open-source programs. But in many practical situations, the original high-level source code is unavailable. The solution GUEB in [16] is a state-of-the-art static tool for detecting use-after-free vulnerabilities and has certain ability to detect vulnerabilities in binary code. In this approach, an in-lining technique is used for inter-procedural analysis and functions that are called many times need to be analyzed repeatedly. This approach introduces a lot of unnecessary overhead in analyzing large-scale programs. Therefore, this solution is not scalable enough and only targets small programs.

In this paper, we propose a scalable static approach for detecting use-after-free vulnerabilities in binary code. First, a use-after-free vulnerability feature model is presented through the analysis of a large number of samples, which provides a guide for detecting vulnerabilities. Then, considering indirect jumps, we construct more complete CFGs. Finally, we focus on pointer-related operations in the program to track each pointer’s states and use a finite state machine (FSM) to identify use-after-free vulnerabilities. The main advantages of our approach are as follows: (1) the function summaries technique is used for an inter-procedural analysis instead of in-lining, which avoids repeated analysis of functions and improves the scalability of the analysis; and (2) pointer aliases and indirect jumps are considered to enhance the completeness of the analysis.

Based on this approach, we implement a prototype called UAFDetector and use standard benchmarks [17] and real-world programs to evaluate it. The experimental results show that our approach can detect known vulnerabilities effectively and can discover new vulnerabilities in binary code. Compared with the existing tool GUEB [16], our approach has less overhead and more scalability.

Accordingly, we make the following contributions in this paper:

- Proposing a scalable static detection approach for detecting use-after-free vulnerabilities in binary code. This approach uses the function summaries technique in an inter-procedural analysis to reduce overhead and improve scalability, and it integrates a pointer alias analysis and CFG recovery to improve the completeness of detection.
- Implementing UAFDetector, which is a prototype for detecting use-after-free vulnerabilities in binary code. The prototype uses IDA Pro [18] to disassemble binary code and uses a binary analysis platform BinNavi [19] to translate assembly code into intermediate representation REIL. Then, based on the intermediate code and the CFGs provided by CFGConstructor [20], UAFDetector performs the detection procedure to identify use-after-free vulnerabilities.
- Evaluating the effectiveness and performance overhead of our approach using a standard benchmark (Juliet Test Suite [17]) and real-world programs. UAFDetector is able to detect use-after-free vulnerabilities in the benchmark with zero false positive rate and 2.39% false negative rate. In given real-world programs, it can discover most known vulnerabilities and an unknown vulnerability (CNNVD-201904-1451) despite some false positives and negatives. Furthermore, our approach has lower performance overhead and better scalability than the state-of-art tool GUEB [16].

The remainder of this paper is organized as follows: Section II presents the use-after-free feature model and introduces the challenges of detecting use-after-free vulnerabilities. The overview of the proposed method is described in Section III. Section IV describes the details of each part and takes a real-world vulnerability as an example to introduce the workflow. Section V presents the implementation details and evaluates the implemented prototype. The limitations, applicable conditions and improvement measures are discussed in Section VI. We summarize the related work in Section VII and provide our conclusions in Section VIII.

II. PROBLEM DESCRIPTION

A. FEATURES OF USE-AFTER-FREE VULNERABILITIES

Through the analysis of a large number of use-after-free vulnerabilities, we summarize the feature model of use-after-free vulnerabilities, as shown in Fig. 2. The model focuses on three types of pointer operations in the program:

- **FREE(p)**: frees the object pointed to by the pointer p. This type of operation changes the pointer p into a dangling pointer, such as the function `free` in C language.
- **USE(p)**: dereferences the pointer p. **FREE(p)** is a special dereference.
- **DESTROY(p)**: destroys the dangling pointer p through reassigning p to a new valid address or NULL.

Through observation, the path with a use-after-free vulnerability has the following features:

1) Creating dangling pointers. A use-after-free vulnerability is caused by dereferencing dangling pointers. Therefore, performing **FREE(p)** to create dangling pointers is a necessary condition for causing vulnerabilities.
2) Preserving the dangling pointer. If a dangling pointer is destroyed by **DESTROY(p)** before being dereferenced, no vulnerability will occur. Thus, when use-after-free vulnerabilities are discovered in practice, developers usually set the dangling pointer to NULL to prevent them.
3) Dereferencing a dangling pointer. Not all dangling pointers lead to use-after-free vulnerabilities because...
dangling pointers may not be dereferenced later. A use-after-free vulnerability occurs only when USE(q) is executed, where q is a dangling pointer or a dangling pointer’s alias.

Accordingly, the path $a \rightarrow e \rightarrow h \rightarrow i \rightarrow j$ in Fig. 2 contains a use-after-free vulnerability, as it satisfies the above three features, where node $a$ creates a dangling pointer $p$ and node $j$ dereferences it. However, the path $a \rightarrow e \rightarrow f \rightarrow g \rightarrow j$ has no use-after-free vulnerabilities because node $f$ destroys the dangling pointer $p$. In the path $a \rightarrow b \rightarrow c \rightarrow d$, the dangling pointer $p$ is retained but not dereferenced in the following nodes. Hence, this path also does not contain vulnerabilities.

B. CHALLENGES

Compared to detecting famous buffer overflow vulnerabilities [21], there are many challenges in detecting use-after-free vulnerabilities using a static analysis in binary code:

1) Incomplete CFG: The construction of a CFG is the basis of detecting use-after-free vulnerabilities. There are many binary analysis tools that provide the ability to construct CFGs, such as IDA Pro. This tool constructs CFGs from assembly code based on static analysis. This method cannot obtain indirect jump addresses, which leads to incomplete CFGs. Detecting use-after-free vulnerabilities using incomplete CFGs may result in false negatives.

2) Inter-procedural analysis: The operations FREE(p), USE(p) and DESTROY(p) related to use-after-free vulnerabilities may occur in different functions. Hence, vulnerability detection needs to support an inter-procedural analysis. The existing solution in [16] implements inter-procedural analysis by using a naive in-lining technique, which inserts callee functions directly into the caller function. This approach may be easier to implement but has a critical limitation that a function called many times will be analyzed repeatedly. The repeated analysis introduces a lot of redundancy overhead in analyzing large-scale programs. Therefore, the method has a high performance overhead and only targets small programs. An example will be shown in Section IV-B to further clarify the limitations of existing methods.

```c
1 int *p, *q;
2 p = malloc(size);
3 q = p;
4 free(p); // p and q are dangling pointers
5 p = NULL; // DESTROY(p)
...  
6 use(q); // this is use-after-free

FIGURE 3. Example of a pointer alias.
```

3) Alias analysis: Pointer alias is a difficult problem to solve in detecting use-after-free vulnerabilities. Fig. 3 shows an example of an alias. Two pointers, $p$ and $q$,

refer to the same memory block; they are then called aliases. The object pointed to by these two pointers is deallocated in line 4, and $p$ and $q$ become dangling pointers. DESTROY($p$) in line 5 destroys the dangling pointer $p$, while $q$ is still dangling. USE($q$) in line 6 leads to a use-after-free vulnerability. If a vulnerability detection lacks the alias analysis, this vulnerability will not be discovered.

III. OVERVIEW

Fig. 4 shows the overview of our proposed approach, in which the white modules represent the existing techniques and the gray modules represent our proposed parts. The input of the detection is the binary code of the target program, such as an executable file or a dynamic library. The outputs are error reports containing the sets of program locations involved in the use-after-free vulnerability. Our approach consists of two procedures: a pre-processing procedure and a detection procedure.

- Pre-processing procedure. The main function of this procedure is to convert binary code into an intermediate representation and construct the CFGs of functions in the program. First, we use a disassembler to transform the binary code into an assembly code. Then, this assembly code is translated into an intermediate representation by a translator. The CFGs containing indirect jumps are constructed from the intermediate representation by CFGConstructor, which combines dynamic and static analysis. Finally, this procedure outputs the intermediate code and the CFGs, which are taken as inputs of the detection procedure.

- Detection procedure. A scalable static analysis is performed to detect use-after-free vulnerabilities on the intermediate code and the CFGs. We use the function summaries technique for an inter-procedural analysis and a dedicated data flow analysis for the pointer alias. Finally, exploiting the above analyses, we track the pointer operations and transfers to identify use-after-free vulnerabilities and report program points that create and dereference dangling pointers.

IV. DESIGN

A. CFG CONSTRUCTION

Constructing CFGs is the basis of vulnerability analysis. Function summaries, alias analysis and pointer tracking in use-after-free vulnerability detection all depend on CFGs. It is important to construct complete and accurate CFGs from binary code.

There are two methods for constructing CFGs from binary code: dynamic analysis and static analysis. A dynamic method in [22] constructs CFGs by recording the program execution paths, which has high accuracy but low path coverage and efficiency. The static method, such as IDA Pro, directly analyzes the code to construct CFGs without running programs. This method has high code coverage but cannot
obtain indirect jump addresses, resulting in some code areas that cannot be reached.

We use a hybrid analysis CFGConstructor [20] that has been completed in our previous work to construct CFGs from binary code. First, a static analysis is used to obtain the basic control flow without indirect jumps by dividing and connecting basic blocks. Second, test cases generated by fuzz testing were exploited to run the target program, during which a dynamic binary instrumentation technique is used to obtain indirect jumps. Finally, the results of these two steps are integrated to construct more complete CFGs by adding indirect jumps to the basic control flow.

**B. FUNCTION SUMMARY GENERATION**

Detection of use-after-free vulnerabilities requires tracking the creation and dereference of dangling pointers, which may occur in different functions. Hence, it is necessary to support the inter-procedural analysis. For this issue, the existing solution GUEB [16] in-lines callee functions directly into the caller function. This solution has the serious limitation that a function called many times needs to be analyzed repeatedly.

An example of inter-procedural analysis is shown in Fig. 5. The (a) and (b) show the code and the control flow of the example. The results of the inter-procedural analysis using the in-lining approach in GUEB are shown in (c). The functions foo1 and foo2 are analyzed twice because the in-lining approach always analyzes the callee function in detail at each function call point. Although this example only has 3 functions, the total number of functions to be analyzed is 5 due to the redundant repeated analysis. The redundant analysis is more serious in analyzing large-scale programs. Therefore, the existing method has high performance overhead and is not suitable for analyzing large-scale programs.

We introduce function summary technique for the inter-procedural analysis to address this problem. First, important behaviors of each function are summarized with an intra-procedural analysis, such as creating, using and destroying dangling pointers. Then, when the analysis is enlarged to an inter-procedural analysis, each function call is replaced by its summary. Therefore, each function needs to be analyzed only once, even if it is called many times.

1) **GENERATING A SUMMARY FOR A SINGLE FUNCTION**

This section describes which functions should be summarized and how to generate a summary for a single function with an intra-procedural analysis. As described in Section II-A, three kinds of operations related to pointers are focused on in use-after-free detection. We consider generating summaries for the functions that have these three kinds of operations.

**a: DANGLING POINTERS CREATING FUNCTION**

If a function makes an external pointer dangling after the function is called, we call this function as a Dangling Pointers
Creating Function (DPCF). The external pointers mean the pointers that are not local variables in the function, generally including pointer-type arguments and global pointers. In static analysis, when a function satisfies the following conditions, a DPCF is identified.

1) The function uses an external pointer `ext_ptr`, that may be a pointer-type argument `arg_ptr` or global pointer `glb_ptr`.

2) There are operations, FREE(`ext_ptr`) or FREE(Alias(`ext_ptr`)), to free `ext_ptr` or its aliases in the function.

3) There are at least one path that does not contain DESTROY(`ext_ptr`) from the point where `ext_ptr` or its aliases are freed to the exit of the function.

A DPCF may have more than one such pointer. If `F` is a DPCF, Dangling(`F`) represents the set of pointers satisfying the above conditions. Because the static analysis in our approach is an over-approximation, all pointers in Dangling (`F`) are considered dangling after `F` is called. Therefore, the set of operations that free pointers in function `F` is considered as the summary of the function as follows:

\[ \text{Summary}(F) = \{\text{FREE}(\text{ext} \_ \text{ptr})|\text{ext} \_ \text{ptr} \in \text{Dangling}(F)\}. \]

In an intermediate representation, the offset relative to the stack base `BP` of a function is used to represent the argument, and the fixed offset is used to represent the global pointer.

In particular, the function `free` in C is a DPCF, which makes the first argument become dangling. Thus, \( \text{Summary}(\text{free}) = \{\text{FREE}(\text{arg}0)\} \). A simple example is shown in Fig. 6 to explain how to generate a summary for a single DPCF.

The function `foo` is a DPCF according to the above conditions. The arguments, `arg0` and `arg1`, are pointer-type arguments and `global_ptr` is a global pointer. These pointers are freed in lines 2, 3 and 4, respectively. There exists a path that does not contain DESTROY from FREE(`arg0`) and FREE(`global_ptr`) to the exit of the function. Thus, `arg0` and `global_ptr` are considered dangling pointers after `foo` is called. However, `arg1` is not a dangling pointer because it is redefined in line 5. Therefore, the summary of `foo` is \( \text{Summary}(\text{foo}) = \{\text{FREE}(\text{arg}0), \text{FREE}(\text{global} \_ \text{ptr})\}. \)

1 int foo(int* arg0, int* arg1){
2  free(arg0); // FREE(arg0)
3  free(arg1); // FREE(arg1)
4  free(global_ptr); // FREE(global_ptr)
5  arg1 = NULL; // DESTROY(arg1)
6  return 0;
7 }

\[ \text{FIGURE 6. Example of function summaries.} \]

b: DANGLING POINTERS DESTROYING FUNCTION

Similarly, we define the Dangling Pointers Destroying Function (DPDF). If a function assigns an external pointer to a new valid address or NULL after the function is called, we call this function as a DPDF. It requires the following conditions to be satisfied:

1) The function uses an external pointer `ext_ptr`.

2) The operations DESTROY(`ext_ptr`) exist in all paths from the entry to the exit of the function.

Suppose `F` is a DPDF, the set of operations that destroy pointers in the function is considered as its summary as follows.

\[ \text{Summary}(F) = \{\text{DESTROY}(\text{ext} \_ \text{ptr})|\text{ext} \_ \text{ptr} \in \text{Assigned}(F)\}. \]

\[ \text{Assigned}(F) \text{ denotes the set of all external pointers assigned in function } F. \]

c: DANGLING POINTERS USING FUNCTION

If a function dereferences an external pointer, we call this function as a Dangling Pointers Using Function (DPUF). It requires the following conditions to be satisfied:

1) The function uses an external pointer `ext_ptr`.

2) There are operations, USE(`ext_ptr`) or USE(Alias(`ext_ptr`)), to dereference `ext_ptr` or its aliases in the function.

3) There are at least one path that does not contain DESTROY(`ext_ptr`) from the entry of the function to the point where `ext_ptr` or its aliases are dereferenced. The summary of a DPUF `F` is as follows.

\[ \text{Summary}(F) = \{\text{USE}(\text{ext} \_ \text{ptr})|\text{ext} \_ \text{ptr} \in \text{Dereferenced}(F)\}. \]

\[ \text{Dereferenced}(F) \text{ denotes the set of all external pointers dereferenced in function } F. \]

As can be observed above, our analysis is conservative to discover as many use-after-free vulnerabilities as possible. If a function has any path where a dangling pointer is created and not destroyed, it is considered as a DPCF. And if a function has any path where a pointer is dereferenced before assigned, it is considered as a DPUF. However, only if all paths from the entry to the exit in a function contain DESTROY(`p`), the function is considered as a DPDF and `p` is considered destroyed. These three kinds of functions are not mutually exclusive. A function is probably a DPCF, a DPDF and a DPUF at the same time.

2) ITERATIVE ANALYSIS

The summary of a single function is already generated through the above procedure. However, summaries of all functions cannot be obtained by simple traversal. In the example in Fig. 7, obviously, foo2 is a DPCF with a summary \{FREE(`arg0`)\}. When foo1 calls foo2, the function call is replaced by the summary. Then, foo1 is also a DPCF. Therefore, iterative analysis is required to summarize all the DPCFs, DPDFs and DPUFs in the entire program.

We perform an iterative analysis along the backward direction of the call graph of the target program to generate summaries of the above three kinds of functions. Summarizing DPCFs is taken an example to explain how the iterative
K. Zhu et al.: Scalable Static Detection of Use-After-Free Vulnerabilities in Binary Code

1. void foo2(int* p) {
2.     free(p); // FREE(p)
3. }
4. void foo1(int* q) {
5.     foo2(q); // FREE(q)
6. }

FIGURE 7. Example of iterative analysis.

Input: the call graph and all functions of target program
Output: summaries of all DPCFs
1. Initialize Q with the DPCFs specified by the user
2. while (Q is not empty)
3.     Get and remove a function f from Q
4.     if f satisfies the conditions of DPCF
5.     Generate summary of f using the method in 4.2.1
6.     Add f to SDPCF
7.     Add all functions calling f to Q
8.     Output summaries of all DPCFs in SDPCF
9. End

FIGURE 8. The algorithm for generating summaries of all the DPCFs.

analysis works. Fig. 8 shows our algorithm for the iterative analysis. Q represents the work queue to be analyzed. SDPCF represents the set of DPCFs that have been summarized. At the beginning, the work queue Q is initialized with the DPCFs specified by the user, such as free. The algorithm gets and removes a function f from Q and determines whether it is a DPCF or not in each loop. If f is a DPCF, it will be summarized using the method in Section IV-B.1 and added to SDPCF. In addition, all functions calling f is added to the work queue Q. If f is not a DPCF, it will be ignored. The algorithm ends until Q is empty. It should be noted that we ignore recursive calls.

Obviously, the summaries of all DPDFs and DPUFs can be obtained by the same approach. With this knowledge, function calls can be handled easily in the inter-procedural analysis. If the callee of a function call belongs to the above three kinds of functions, the call is replaced by the summary of the callee. Otherwise, the call is ignored. Therefore, a function only need to be analyzed once in our method although it is probably called many times.

The function summaries technique is used to analyze the example in Fig. 5 and compared with the in-lining technique in existing solutions. The results of the analysis are shown in Fig. 9. The number following “line” indicates the line number in Fig. 5. The function foo1 is first analyzed and summarized. Summary(foo1) = {Free(arg0)}. When foo2 is analyzed, the function call that calls foo1 at line 6 is replaced by foo1’s summary and foo1 does not need to be analyzed repeatedly. Similarly, foo2 also only needs to be analyzed once, although it is called twice in line 12 and line 14. Therefore, the total number of functions to be analyzed using our approach is 3 in this example, which is less than the total number using the in-lining technique. Our approach can effectively prevent redundant analysis in inter-procedural analysis. The advantage is more obvious when analyzing large-scale programs, which is shown in Section V-C.

C. ALIAS ANALYSIS
The purpose of an alias analysis is to determine which pointers point to the same memory address at a program point. As described in Section II-B, alias analysis is a challenge in use-after-free vulnerability detection, which affects the detection accuracy. Although there are many solutions for alias analysis in high-level languages, there are few solutions for binary code due to lack of information during compilation. Thus we need to create our own alias analysis to mitigate this problem. A dedicated data-flow analysis is performed to address alias analysis.

1) ABSTRACT STATE
To identify aliases, we focus on each pointer and the addresses it points to. The association between a pointer and memory addresses is represented by a pair \( (p, ADDR(p)) \), where \( p \) denotes a pointer variable and \( ADDR(p) \) denotes the set of memory addresses that \( p \) points to. In the intermediate representation, if the pointer is on the stack, it is represented as the offset relative to stack base \( BP \). Otherwise, if the pointer is a global variable, it is represented as a fixed offset.

The symbol \( addr_i \) is used to represent the memory address because the static analysis cannot obtain the specific value of the address. For each program point, \( AbsState \) contains all such pairs that associate pointers and addresses. If \( ADDR(p) \cap ADDR(q) \neq \emptyset \) in a point, \( p \) is considered an alias to \( q \).

2) STATEMENT TRANSFER FUNCTION
The assignment statement of the pointer that changes \( AbsState \) is defined as \( s : p = addr \). In the statement, \( addr \) represents a memory address, which may be the result returned by malloc, the address pointed by another pointer, an immediate value or NULL.
The values of AbsState before and after an assignment statement s are denoted by IN[s] and OUT[s]. Thus, the transfer function \( f_s \) of the statement s is

\[
\text{OUT}[s] = f_s(\text{IN}[s], p, \text{addr}) = \text{IN}[s] \leftarrow \{ p \mapsto \text{addr} \}.
\]

\( \text{IN}[s] \leftarrow \{ p \mapsto \text{addr} \} \) denotes the AbsState identical to \( \text{IN}[s] \), except that the address associated with p is addr.

3) BASIC BLOCK TRANSFER FUNCTION

The values of AbsState immediately before and immediately after each basic block B are denoted by \( \text{IN}[B] \) and \( \text{OUT}[B] \), respectively. Suppose a basic block B consists of assignment statements of pointers \( s_1, s_2, \ldots, s_n \), in that order. We derive the transfer function \( f_B \) of a basic block by composing the transfer function of the statements in the block. Thus, \( f_B = f_{s_n} \circ \ldots \circ f_{s_2} \circ f_{s_1} \). The relationship between the AbsState before and after the basic block is

\[
\text{OUT}[B] = f_B(\text{IN}[B]) = f_{s_n} \circ \ldots \circ f_{s_2} \circ f_{s_1}(\text{IN}[B]).
\]

The purpose of our data-flow analysis is to find all possible addresses pointed to by a pointer. Thus, the input of a basic block B is the union of all the output of its predecessors, i.e.,

\[
\text{IN}[B] = \bigcup_{P \in \text{predecessors}(B)} \text{OUT}[P].
\]

We use the iterative algorithm in [23] to solve these equations and obtain the least fixed point. For a pointer in each program point, the results of these equations contain all the addresses that the pointer may point to. The pointer aliases can be easily identified by exploiting the results. An example given in Section IV-E explains the detailed process of the alias analysis in use-after-free vulnerability detection.

D. POINTER TRACKING AND USE-AFTER-FREE VULNERABILITY CHECKING

According to the vulnerability feature model in Section II-A, the creation of dangling pointers is a necessary condition for use-after-free vulnerabilities. In the previous section, we have obtained the summaries of DPCFs, DPDFs and DPUFs. Supported by these summaries, the inter-procedural analysis is transferred to the intra-procedural analysis through replacing function calls with the summaries of callee functions. Therefore, we only need to perform an intra-procedural pointers tracking in the functions that call DPCFs to check use-after-free vulnerabilities. If a function calls a DPCF, it is called as a Calling Freeing Function (CFF).

A finite-state machine (FSM) for use-after-free detection is established according the vulnerability feature model, as shown in Fig. 10. The vulnerability checking maintains the state of each pointer and traces the state transition in CFFs. The pointer’s state is tracked forward along the CFGs from the entry of each CFF. The state of a pointer p is Start in the beginning and converted to Dangling when the pointer or its aliases are freed by FREE(p) or FREE(Alias(p)), otherwise the state remains the same when other operations are performed. In the Dangling state, dereferencing the pointer (USE(p)) shifts the state to a use-after-free defect. Moreover, the creation and dereference of the dangling pointer are recorded and output to the error reports. Note that FREE(p) is a special kind of dereference of pointer p (in Section II-A). Thus, double-free vulnerabilities can be detected by this FSM. Finally, if a dangling pointer is re-signed by DESTROY(p), the state is converted to End, and the pointer tracking for this pointer ends.

E. EXAMPLE

A real vulnerability, CVE-2015-5221, is taken as an example to introduce the workflow of our approach in detail. This vulnerability is a double-free vulnerability in an image manipulation software, Jasper. The key code associated with vulnerability is shown in Fig. 11. To explain the process of alias analysis in vulnerability detection, we modified the code to include more complex alias relationships.

```c
1  void jas_free(void *ptr) {
2      free(ptr); //FREE(ptr)
3  }
4  void jas_tvparser_destroy(jas_tvparser_t *tpv) {
5      ...
6      jas_free(tpv); //FREE(tpv)
7  }
8  int mif_process_cmpt(args[]) {
9      jas_tvparser_t *tpv1,*tpv2;
10     tpv1 = malloc(size);
11     tpv2 = tpv1;
12     ...
13     jas_tvparser_destroy(tpv1); //FREE(tpv1)
14     if(condition)
15         goto error;
16     ...
17     error:
18     jas_tvparser_destroy(tpv2); //FREE(tpv2)
19  }
```

FIGURE 11. A real use-after-free vulnerability.

- Generating summaries of the DPCFs. According to the algorithm in Fig. 8, the functions free, jas_free and
V. IMPLEMENTATION AND EVALUATION

A prototype called UAFDetector is implemented based on the proposed approach. This section describes the implementation details of the prototype and evaluates its effectiveness and performance overhead using Juliet Test Suite (JTS) and real-world programs.

A. IMPLEMENTATION

In order to analyze binaries from multiple architectures, we carry out vulnerability detection on the intermediate representation. We implemented the prototype based on Google’s binary analysis platform BinNavi [19], which is also employed in GUEB [16]. BinNavi can be combined with IDA Pro [18] to easily translate binaries from multiple architectures into an intermediate representation REIL. In the pre-processing procedure, IDA Pro is employed to transform the binary code into assembly code. Then, this assembly code is translated into REIL by BinNavi. The CFGConstructor is employed to obtain more complete CFGs.

BinNavi also provides APIs for analyzing the binaries on the intermediate representation. In the vulnerability detection procedure, we implemented function summary generation, alias analysis and pointer tracking by utilizing the APIs. The UAFDetector maintains the state of each pointer and tracks the changes of the state according to the FSM in Fig. 10. Finally, the program locations involved in use-after-free vulnerabilities are reported.

The prototype is currently capable of analyzing binaries on x86 architecture. However, in theory, our approach can be applied to multiple architectures because it is based on intermediate representations.

B. EFFECTIVENESS

We used the standard benchmarks, Juliet Test Suite (JTS) [17], and real-world programs to evaluate the effectiveness of UAFDetector.

1) JTS

This benchmark is a collection of C/C++ programs with known vulnerabilities provided by the National Institute of Standards and Technology. We selected 1457 C/C++ programs with use-after-free or double-free vulnerabilities and compiled them into binaries. Then, UAFDetector is used to analyze these binaries and report the vulnerabilities. The analysis results are shown in Table 2. The columns in this table represent, from left to right, the type of vulnerabilities in the target programs, the language of the target programs, the number of target programs, the number of known vulnerabilities in the target programs, the number of vulnerabilities reported by UAFDetector, the number of false positives (FPs), the false positive rate (FPR), the number of false negatives (FNs) and the false negative rate (FNR).

As shown in Table 2, UAFDetector is capable of finding use-after-free or double-free vulnerabilities in JTS. UAFDetector correctly reports 2042 of the 2092 known vulnerabilities in 1457 target programs. The false negative rate is 2.39%, and the false positive rate is zero percent. We have shared the test binary programs and the experimental results at https://github.com/BinaryAnalysis/UAFDetector.

2) REAL-WORLD PROGRAMS

We further evaluate the effectiveness of UAFDetector using real-world programs. UAFDetector analyzes six popular programs with known vulnerabilities and outputs error reports. We manually determine whether each item in the report is true or false positive. The six programs analyzed are

| Line | Code | AbsState | Alias | FSM’s state |
|------|------|----------|-------|-------------|
| 10   | tvp1 = malloc(size) | (tvp1, {addr0}) | Alias(tvp1) = \phi | tvp1.state = Start |
| 11   | tvp2 = tvp1 | (tvp1, {addr0}), (tvp2, {addr0}) | Alias(tvp1) = tvp2, Alias(tvp2) = tvp1 | tvp1.state = Start, tvp2.state = Start |
| 13   | jas_tvparsr_destroy(tvp1) | (tvp1, {addr0}), (tvp2, {addr0}) | Alias(tvp1) = tvp2, Alias(tvp2) = tvp1 | tvp1.state = Dangling, tvp2.state = Dangling |
| 15   | jas_tvparsr_destroy(tvp2) | (tvp1, {addr0}), (tvp2, {addr0}) | Alias(tvp1) = tvp2, Alias(tvp2) = tvp1 | tvp1.state = Dangling, tvp2.state = UAF |

**TABLE 1.** Checking use-after-free in mif_process_cmpt.
TABLE 2. UAFDetector effectiveness evaluation results on JTS.

| Vulns type     | Language | # Program | # Known Vulns | # Reported Vulns | # FP | FPR | # FN | FNR |
|----------------|----------|-----------|---------------|-----------------|------|-----|------|-----|
| use-after-free  | C        | 157       | 182           | 179             | 0    | 0.00% | 3    | 1.65% |
| use-after-free  | C++      | 321       | 349           | 349             | 0    | 0.00% | 0    | 0.00% |
| double-free    | C        | 313       | 501           | 476             | 0    | 0.00% | 25   | 4.99% |
| double-free    | C++      | 666       | 1060          | 1038            | 0    | 0.00% | 22   | 2.08% |
| Sum            | –        | 1457      | 2092          | 2042            | 0    | 0.00% | 50   | 2.39% |

Jasper, an image processing/coding tool kit; OpenJPEG: a JPEG 2000 codec; Boolector: a satisfiability modulo theories solver; LibTIFF: a collection of tools for manipulations of TIFF images; LibPNG: the official PNG reference library; and GNU cflow: a tool for charting the control flow within programs. The experimental results are shown in Table 3. The columns in this table represent, from left to right, the name of the program, the version of the program, the number of known vulnerabilities in the program, the CVE identifiers of the known vulnerabilities, the number of vulnerabilities reported by UAFDetector, the number of true positives (TPs), the number of false positives (FPs), and the number of false negatives (FNs).

The results show that UAFDetector can successfully detect 5 of the 6 known vulnerabilities in the six programs and can discover a new vulnerability in GNU cflow despite some false positives and negatives. We have submitted the new vulnerability to the public vulnerability database and obtained a vulnerability identifier CNNVD-201904-1451. Thus, our approach is effective in discovering use-after-free vulnerabilities in binary code.

Although our approach can effectively detect use-after-free vulnerabilities in binary code, there are still false positives and false negatives. We analyze the reason of each incorrectness and propose corresponding improvements.

3) FALSE NEGATIVES

There are several reasons for false negatives. (1) The pointer alias analysis is not completely accurate. Considering the time cost, we analyzed only the pointer aliases of simple patterns, which cannot address complex situations, such as indirect aliases [24] caused by re-allocation. We will improve the accuracy of alias analysis by tracking the memory allocation or combining the local dynamic analysis in the next step. (2) According to the observations in [1] and [16], most of the use-after-free vulnerabilities are not sensitive to a particular loop iteration. Therefore, UAFDetector unrolls each loop at most once, similar to the solutions in [1], [16]. However, there are a few vulnerabilities that depend on executing a loop several times in our experiments. UAFDetector cannot detect the vulnerabilities in this case.

4) FALSE POSITIVES

The main reason for false positives is that our approach uses a path-insensitive analysis. A path-insensitive analysis has the advantage of low time overhead, but it is not completely accurate. Some paths that are considered accessible on the CFG in the static analysis cannot be executed dynamically. A typical example of false positives is shown in Fig. 12. Here, the function f2 returns 0 when it frees arguments and does not free arguments otherwise. The caller f1 checks the returned value and never uses the pointer if it is freed. However, our analysis considers a possible path from freeing argument (line 3) to using argument (line 12). A path-sensitive analysis using symbolic execution can ameliorate this problem. Because symbolic execution has a high overhead, we consider adding a lightweight local path-sensitive analysis to our approach to achieve the trade-off between performance and accuracy.

C. PERFORMANCE OVERHEAD

We evaluate the performance overhead of UAFDetector using real-world programs. The experiment was conducted on an Intel Core i7-6700 processor with 8 cores at 3.4 GHz and 16 GB of memory, running on a 64-bit Windows 7 operating system. Table 4 shows the test binary programs and the time overhead needed by UAFDetector to analyze them. The target programs contain dynamic libraries of different sizes: from
The experimental results show that UAFDetector can complete the analysis in a relatively short time for binary programs of different sizes. For example, it only takes 135.15 seconds for UAFDetector to analyze libmergedlo, which is a 50 MB binary in LibreOffice. Therefore, UAFDetector has the ability to analyze large-scale real-world programs.

We use GUEB [16] and our approach to analyze the above binaries in real-world programs to compare their performance overhead. The time in Table 4 is the average of ten repeated analyses.

The results show that UAFDetector can detect vulnerabilities by analyzing only some of the functions in the program in detail because it only focuses on use-after-free vulnerability-related functions (DPCFs, DPDFs, DPUFs and CFFs) and ignores other functions. Moreover, the number of functions to be analyzed by UAFDetector is approximately 70.8% lower than that by GUEB.

The reason is that UAFDetector uses function summaries instead of the in-lining technique used by GUEB to achieve inter-procedural analysis. As shown in the example in Fig 5, the in-lining method needs to repeatedly analyze the functions that are called multiple times, which results in a lot of redundant overhead. However, the function summaries method used by our approach does not have this limitation. In this method, all function calls are replaced by the summaries of callee functions. Thus, functions are analyzed only once, even if they are called many times.

**VI. LIMITATION AND FUTURE WORK**

Although UAFDetector has been demonstrated to be able to detect the use-after-free vulnerabilities in the large-scale binaries, it still has important limitations. We briefly describe the limitations and applicable conditions of our system.

First, UAFDetector utilizes static analysis to detect vulnerabilities without the ability of dynamic analysis. Therefore, its analysis results are not completely correct. As discussed in the Section V-B, the incomplete pointer alias analysis and the imprecise loop handling may cause false negatives. In addition, UAFDetector uses the path-insensitive analysis. This method may cause false positives.

Second, we employ some existing tools to implement UAFDetector, such as IDA Pro, BinNavi and CFGConstructor. The prototype depends on the assumption that the employed tools convert the binary code into the intermediate representation and obtain the CFGs correctly. However, this assumption is not always satisfied. For example, the code transformation methods and the code confusion methods may prevent IDA Pro from disassembling the binary or obtaining the CFGs correctly. In addition, IDA Pro may fail to identify the function boundaries in some cases. These problems are the basic problems widely existing in many binary analysis tools.
applications, such as the detection of code clone [25], [26] and the detection of malware [27]–[29]. Our prototype cannot detect the vulnerabilities correctly in the above cases.

In addition, the impact of using different compilers or compilation settings on binary analysis tools is often discussed. Many existing binary analysis tools rely on assumptions about specific compilers and compilation settings [25]. UAFDetector carries out the analysis on the intermediate representation and focuses on the semantics of the binaries rather than the structure of code. Therefore, different compilers or compilation settings do not impact our system in general.

Although our static analysis approach can provide good results from a scalability point of view, it is not as accurate as dynamic approaches. Therefore, our analysis is suitable as a first level vulnerability detection step. Our results containing the sets of program locations involved in the use-after-free vulnerability can provide basic knowledge for further vulnerability detection. For example, our approach can point out which parts of the target program must be stressed for popular fuzzing techniques.

The approach would be further improved in our future work. First, a lightweight local path-sensitive analysis could be added to reduce the number of false positives. Second, we will consider more patterns of pointer aliases, such as indirect aliases [24], to improve the completeness of the alias analysis and reduce the number of false negatives. In addition, a fuzzing technique will be utilized as the further analysis to avoid false positives.

VII. RELATED WORK

This section summarizes the related research on protecting against use-after-free vulnerabilities from two aspects: vulnerability detection and vulnerability mitigation.

A. USE-AFTER-FREE DETECTION

Most existing use-after-free detection solutions depend on dynamic analysis. CETS [30] inserts a runtime check when the program is compiled. When a pointer is referenced, CETS checks whether the object pointed to by this pointer is still allocated to find the dangling pointer. However, this solution lacks robustness in its prototype implementation, which causes a large number of complex programs to be unable to compile using this method. AddressSanitizer [13] is a popular runtime detection tool. It can also dynamically detect the illegal use of pointers in programs. However, like CETS, AddressSanitizer uses instrumentation at compiling and requires the source code, which limits the practical application of the tool. Valgrind [12] and Purify [11] detect vulnerabilities by checking whether the dereferenced pointer points to valid memory. This approach is unable to detect dangling pointers that point to an object that has reused the memory. To address this problem, Undangle [7] uses a dynamic analysis approach called early detection to protect against use-after-free vulnerabilities. This method combines taint analysis and pointer tracking to effectively identify unsafe pointers that are created but not used, which improves the completeness of vulnerability detection. However, this method relies on an execution trace analysis, which has a high performance overhead and cannot analyze large-scale programs. Clause et al. [14] presents a dynamic technique for detecting invalid memory access. Their approach taints both the objects and the corresponding pointer using the same taint mark. The taint marks are propagated and checked every time a pointer is referenced. If the taint marks of the object and the pointer differ, then the illegal access is reported. The approach is able to work on binary code but requires hardware support to achieve an efficient taint analysis.

Although dynamic analyses achieve a high detection accuracy and few false alarms, the method requires inserting dynamic runtime checks and has a high runtime overhead and a high memory overhead. In addition, it is difficult to generate input that can execute vulnerable paths, which leads to low code coverage in dynamic analyses.

Static analyses do not suffer from the above limitations. However, there are very few studies on static analyses for use-after-free vulnerability detection. UAFChecker [1] uses classic static analysis techniques, including taint analysis and symbolic execution, to detect use-after-free vulnerabilities in C/C++ code. This paper does not discuss the performance overhead of symbolic execution and taint analysis, and this method is not suitable for binary code. Tac [31] is a machine learning-guided static use-after-free vulnerability detection framework. It learns the correlations between program features and use-after-free vulnerability-related aliases by using a support vector machine and exploits this knowledge to improve the precision of the alias analysis. Nevertheless, the approach is not yet sound and requires a large number of marked training samples. The most important limitation of the above methods is that they can only analyze the open-source programs. However, the source code for a large number of applications is unavailable.

The closest approach to ours is GUEB [16], which uses static analysis to detect use-after-free vulnerabilities in binary code. The solution uses a dedicated value set analysis to track heap operations and address transfers. Then, it exploits these results to statically identify use-after-free vulnerabilities and extract the subgraph for each vulnerability. The main difference between this solution and ours is that GUEB uses a naive in-lining technique for inter-procedural analysis. The functions that are called many times need to be analyzed repeatedly, which results in a high overhead for analyzing large programs. UAFDetector uses function summaries instead of an in-lining technique to improve the efficiency and scalability of vulnerability detection.

B. USE-AFTER-FREE MITIGATION

Instead of detecting use-after-free vulnerabilities, some studies focus on how to prevent the exploitation of vulnerabilities.

Cling [32], Diehard [33] and Dieharder [34] are safe memory allocators that are designed to make the exploitation of use-after-free vulnerabilities harder. These allocators restrict memory reuse by using more address space or randomizing
the memory allocation. These solutions can effectively prevent exploitation of use-after-free vulnerabilities and have acceptable overhead. However, attackers can bypass these mitigations by using “heap spraying” or “heap fengshui”-like attacks.

DangNULL [4], FreeSentry [35] and DangSan [36] prevent the exploitation of use-after-free vulnerabilities by using pointer invalidation. These solutions insert runtime checks during compilation to track per-object pointers and invalidate the pointers once the object is freed. When the dangling pointer is referenced, pointer invalidation crashes the program to prevent the attacker from exploiting the vulnerability. Despite a number of optimizations, the approach still has a high performance and memory overhead and is not widely applied in practice.

Overwriting virtual table pointers is the most widely used technique to exploit use-after-free vulnerabilities. VTGuard [37], SafeDispatch [38] and VTV [39] protect against this exploitation technique to mitigate use-after-free vulnerabilities. These mitigations have low performance overhead because they only focus on protecting the virtual tables. However, these approaches are ineffective when attackers target other pointers rather than virtual table pointers.

Many vulnerabilities are caused by incorrect memory management by programmers using C/C++. Therefore, some efforts have been made on safe languages to avoid known memory corruption vulnerabilities by modifying language constructs. These languages use garbage collection instead of explicitly freeing memory to reduce the harm of dangling pointers. There are a number of safe languages [40], [41] that are as close to the C/C++ language as possible. Although they try to maintain compatibility with C/C++ programs, it still requires a lot of effort to translate existing projects in these languages.

VIII. CONCLUSION

Use-after-free vulnerabilities caused by dangling pointers are an increasingly serious threat to computer systems. While a number of mitigations are proposed to address this problem, few of them are sufficiently practical for large-scale programs. In this paper, we propose a scalable static approach that combines CFG construction, alias analysis, function summaries and pointer tracking to detect use-after-free vulnerabilities in binary code. Our approach uses the function summaries technique instead of the in-lining technique for the inter-procedural analysis and avoids repeated analysis problem in existing approach. We have implemented a prototype called UAFDetector to detect use-after-free vulnerabilities in binary code and evaluated it using standard benchmarks (JTS) and real-world programs.

The experimental results show that our approach can effectively find use-after-free vulnerabilities. UAFDetector achieves a low false negative rate (2.39%) and a zero false positive rate in the JTS. Moreover, despite some false positives and false negatives, UAFDetector finds most known vulnerabilities and one unknown vulnerability in the real-world programs provided. Compared with the existing static analysis solution GUEB for detecting use-after-free vulnerabilities, our approach can reduce the redundant overhead caused by repeated analysis. In our experiments, the number of functions to be analyzed in UAFDetector is approximately 70.8% less than that in GUEB. Therefore, our approach has a lower overhead and better scalability for detecting use-after-free vulnerabilities in large-scale programs.

REFERENCES

[1] J. Ye, C. Zhang, and X. Han, “Poster: Uafchecker: Scalable static detection of use-after-free vulnerabilities,” in Proc. ACM SIGSAC Conf. Comput. Commun. Secur., Nov. 2014, pp. 1529–1531.
[2] National Vulnerability Database. Accessed: Apr. 10, 2019. [Online]. Available: https://nvd.nist.gov.
[3] M. Conti, S. Crane, and L. Davi, “Losing control: On the effectiveness of control-flow integrity under stack attacks,” in Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur., Oct. 2015, pp. 952–963.
[4] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee, “Preventing Use-after-free with dangling pointers nullification,” in Proc. Netw. Distrib. Syst. Secur. Symp., San Diego, CA, USA, Feb. 2015, P. 1–5.
[5] F. J. Serna. (2012). The Info Leak Exploit on Software Exploitation. Black Hat, NY, USA. Accessed: Apr. 10, 2019. [Online]. Available: https://docs.huluoo.com/blackhat-usa/2012/BH_US_12_Serna_Infol_Era_Slides.pdf
[6] W. Xu, J. Li, and J. Shu, “From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel,” in Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur., Oct. 2015, pp. 425–441.
[7] J. Caballero, G. Grieco, M. Marron, and A. Nappa, “Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities,” in Proc. Int. Symp. Softw. Test. Anal. (ISSTA), 2012, pp. 133–143.
[8] N. Elhage. (2011). Virtunoid: A kvm Guest—Host Privilege Escalation Exploit. Black Hat, NY, USA. Accessed: Apr. 10, 2019. [Online]. Available: http://media.blackhat.com/bh-us-11/Eihage/BH_US_11_Eihage_Virtunoid_WP.pdf
[9] X. Fan, Y. Sui, X. Liao, and J. Xue, “Boosting the precision of virtual call integrity protection with partial pointer analysis for C++,” in Proc. 26th ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA), 2017, pp. 329–340.
[10] K. Zetter. (2010). Hack of Google, Adobe Conducted Through Zero-Day IE Flaw. Accessed: Apr. 10, 2019. [Online]. Available: https://www.wired.com/2010/01/hack-of-adob/
[11] R. Hastings and B. Joyce, “Purify: Fast detection of memory leaks and access errors,” in Proc. Winter Usenix Conf. Jan. 1992, pp. 125–136.
[12] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” ACM SIGPLAN Notices, vol. 42, no. 6, pp. 89–100, 2007.
[13] K. Serebryany, D. Bruening, and A. Potapenko, “AddressSanitizer: A fast address sanity checker,” in Proc. USENIX Annu. Tech. Conf., Jun. 2012, pp. 309–318.
[14] J. Clause, I. Doudalis, A. Orso, and M. Prvulovic, “Effective memory protection using dynamic tainting,” in Proc. 22nd IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE), Nov. 2007, pp. 284–292.
[15] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, “Frama-C: A software analysis perspective,” Formal Aspects Comput., vol. 27, no. 3, pp. 573–609, May 2015.
[16] J. Feist, L. Mounier, and M.-L. Potet, “Statically detecting use after free on binary code,” J. Comput. Virol. Hacking Techn., vol. 10, no. 3, pp. 211–217, Aug. 2014.
[17] T. Boland and P. E. Black, “Juliet 1.1 C/C++ and Java test suite,” Computer, vol. 45, no. 10, pp. 88–90, 2012.
[18] Interactive DisAssembler. Accessed: Apr. 10, 2019. [Online]. Available: https://www.hex-rays.com/ BinNavi. Accessed: Apr. 10, 2019. [Online]. Available: https://github.com/google/binnavi
[19] K. Zhu, Y. Lu, and H. Hang, “Construction approach for control flow graph from binaries using hybrid analysis,” J. ZheJiang Univ. (Eng. Sci.), vol. 53, no. 5, pp. 829–836, 2019.
Exploit Mitigation Improvements

K. Johnson and M. Miller. (2012).

E. van der Kouwe, V. Nigade, and C. Giuffrida, “DangSan: Scalable Use-After-Free Detection” in Proc. USENIX Secur. Symp. USENIX Secur., Aug. 2014, pp. 941–955.

D. Grossman, M. Hicks, and T. Jim, “Cyclone: A type-safe dialect of C++ Users J., vol. 23, no. 1, pp. 112–139, 2005.

G. C. Necula, S. McPeak, and W. Weimer, “CCured: Type-safe retrofitting of legacy code,” ACM SIGPLAN Notices, vol. 37, no. 1, pp. 128–139, Jan. 2002.

J. Feist, “Finding the needle in the heap: Combining binary analysis techniques to trigger use-after-free,” Ph.D. dissertation, Dept. Cryptogr. Secur., Univ. Grenoble Alpes, Grenoble, France, 2017.

S. Alrabaee, L. Wang, and M. Debbabi, “SIGMA: A semantic integrated graph matching approach for identifying reused functions in binary code,” Digit. Invest., vol. 12, pp. S61–S71, Mar. 2015.

W. Wang, Y. Li, X. Wang, J. Liu, and X. Zhang, “Detecting Android malicious apps and categorizing benign apps with ensemble of classifiers,” Future Gener. Comput. Syst., vol. 78, pp. 987–994, Jan. 2018.

S. Alrabaee, P. Shirani, and L. Wang, “FOSSIL: A resilient and efficient system for identifying FOSS functions in Malware binaries,” ACM Trans. Privacy Secur., vol. 21, no. 2, pp. 1–34, 2018.

W. Wang, M. Zhao, Z. Gao, G. Xu, H. Xian, Y. Li, and X. Zhang, “Constructing features for detecting Android malicious applications: Issues, taxonomy and directions,” IEEE Access, vol. 7, pp. 67602–67631, 2019.

S. Nagarakatte, M. M. K. Martin, and J. Zhao, “CETS: Compiler enforced temporal safety for C,” ACM SIGPLAN Notices, vol. 45, no. 8, pp. 31–40, 2010.

H. Yan, Y. Sui, S. Chen, and J. Xue, “Machine-Learning-Guided typestate analysis for static Use-After-Free detection,” in Proc. 33rd Annul. Comput. Secur. Appl. Conf., Dec. 2017, pp. 42–54.

P. Akritidis, “Cling: A memory allocator to mitigate dangling pointers,” in Proc. USENIX Secur. Symp., Aug. 2010, pp. 177–192.

E. D. Berger and B. G. Zorn, “DieHard: Probabilistic memory safety for unsafe languages,” ACM SIGPLAN Notices, vol. 41, no. 6, pp. 158–168, Jun. 2006.

G. Novark and E. D. Berger, “DieHarder: Securing the heap,” in Proc. 17th ACM Conf. Comput. Commun. Secur. (CCS), 2010, pp. 177–192.

Y. Younan, “FreeSentry: Protecting against Use-After-Free vulnerabilities due to dangling pointers,” in Proc. Netw. Distrib. Syst. Secur. Symp., San Diego, CA, USA, Feb. 2015, pp. 1–15.

E. van der Kouwe, V. Nigade, and C. Giuffrida, “DangSan: Scalable Use-after-free detection,” in Proc. 12th Eur. Conf. Comput. Syst. - EuroSys, Jun. 2017, pp. 405–419.

K. Johnson and M. Miller. (2012). Exploit Mitigation Improvements in Windows 8. Black Hat USA. Accessed: Apr. 10, 2019. [Online]. Available: https://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf

D. Jung, Z. Tatlock, and S. Lerner, “SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks,” in Proc. Netw. Distrib. Syst. Secur. Symp., San Diego, CA, USA, Feb. 2014, pp. 1–15.

C. Tice, T. Roeder, and P. Collingbourne, “Enforcing forward-edge control-flow integrity in GCC & LLVM,” in Proc. 23rd USENIX Secur. Symp. USENIX Secur., Aug. 2014, pp. 941–955.

B. M. Padmanabhi and H. B. K. Tan, “Auditing buffer overflow vulnerabilities using hybrid static–dynamic analysis,” IET Softw., vol. 10, no. 2, pp. 54–61, 2016.

L. Xu, F. Sun, and Z. Su, “Constructing precise control flow graphs from binaries,” Univ. California, Davis, CA, USA, Tech. Rep., 2009. [Online]. Available: https://pdfs.semanticscollor.org/8a80f0bd173ec7420478e4b96a3264e2e10dafa0c0.pdf

A. V. Aho, M. S. Lam, and R. Sethi, “Machine-independent optimizations,” in Compilers: Principles, Techniques, and Tools, 2nd ed. Boston, MA, USA: Addison-Wesley, 2006, pp. 601–607.

J. Feist, “Finding the needle in the heap: Combining binary analysis techniques to trigger use-after-free,” Ph.D. dissertation, Dept. Cryptogr. Secur., Univ. Grenoble Alpes, Grenoble, France, 2017.

S. Alrabaee, L. Wang, and M. Debbabi, “BinGold: Towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (SFGs),” Digit. Invest., vol. 18, pp. S11–S22, Aug. 2016.

S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi, “SIGMA: A semantic integrated graph matching approach for identifying reused functions in binary code,” Digit. Invest., vol. 12, pp. S61–S71, Mar. 2015.

W. Wang, Y. Li, X. Wang, J. Liu, and X. Zhang, “Detecting Android malicious apps and categorizing benign apps with ensemble of classifiers,” Future Gener. Comput. Syst., vol. 78, pp. 987–994, Jan. 2018.

S. Alrabaee, P. Shirani, and L. Wang, “FOSSIL: A resilient and efficient system for identifying FOSS functions in Malware binaries,” ACM Trans. Privacy Secur., vol. 21, no. 2, pp. 1–34, 2018.

W. Wang, M. Zhao, Z. Gao, G. Xu, H. Xian, Y. Li, and X. Zhang, “Constructing features for detecting Android malicious applications: Issues, taxonomy and directions,” IEEE Access, vol. 7, pp. 67602–67631, 2019.

S. Nagarakatte, M. M. K. Martin, and J. Zhao, “CETS: Compiler enforced temporal safety for C,” ACM SIGPLAN Notices, vol. 45, no. 8, pp. 31–40, 2010.

H. Yan, Y. Sui, S. Chen, and J. Xue, “Machine-Learning-Guided typestate analysis for static Use-After-Free detection,” in Proc. 33rd Annul. Comput. Secur. Appl. Conf., Dec. 2017, pp. 42–54.

P. Akritidis, “Cling: A memory allocator to mitigate dangling pointers,” in Proc. USENIX Secur. Symp., Aug. 2010, pp. 177–192.

E. D. Berger and B. G. Zorn, “DieHard: Probabilistic memory safety for unsafe languages,” ACM SIGPLAN Notices, vol. 41, no. 6, pp. 158–168, Jun. 2006.

G. Novark and E. D. Berger, “DieHarder: Securing the heap,” in Proc. 17th ACM Conf. Comput. Commun. Secur. (CCS), 2010, pp. 177–192.

YULIANG LU was born in China, in 1964. He received the B.Sc. (Hons.) and M.Sc. degrees in computer application from Southeast University, China, in 1985 and 1988, respectively. He is currently a Professor with the National University of Defense Technology, Hefei, China. His areas of research interests are in computer application and information security.

KAILONG ZHU was born in China, in 1991. He received the B.Sc. and M.Sc. degrees in cyberspace security from the National University of Defense Technology, Hefei, China, in 2014 and 2017, respectively, where he is currently pursuing the Ph.D. degree. His areas of research interests are in information security and program verification.

HUI HUANG was born in China, in 1987. He received the B.Sc. and Ph.D. degrees in information security from the National University of Defense Technology, Hefei, China, in 2012 and 2017, respectively. His areas of research interests are in information security and program verification.