LoongChecker: Practical summary-based semi-simulation to detect vulnerability in binary code

Shaoxin Cheng¹, Jun Yang¹, Jiajie Wang², Jinding Wang¹ and Fan Jiang¹
¹Information Technology Security Evaluation Center, University of Science and Technology of China
Hefei, 230027, P. R. China
²China Information Technology Security Evaluation Center
Beijing, 100085, P. R. China
{ycheng, fjiang}@ustc.edu.cn, {yangjun1, jdwang}@mail.ustc.edu.cn
wangjj@itsec.gov.cn

Abstract—The automatic detection of security vulnerabilities in binary code is challenging and lacks efficient tools. This paper presents a novel semi-summary approach to statically detect potential vulnerabilities in binary code. The semi-summary approach simulates address related instructions accurately using value set analysis, and only traces data dependence on other instructions using data dependence analysis. We have implemented this approach on a tool called LoongChecker, and evaluate it on three real-world programs, and detect three known vulnerabilities and two zero-day vulnerabilities. The results show our approach is practical and can be applied to large real-world software.

Keywords—Semi-summary; static analysis; binary code; vulnerability detection; taint analysis; function summary

I. INTRODUCTION

As security has become an important issue these days, large efforts have been taken to develop automatic tools to find bugs and vulnerabilities [1-9]. There is a big demand for a practical tool, which uses flow-sensitive static analysis to detect vulnerabilities in binary code. In one respect, directly analyzing binary code has advantages over source code; in another respect, other technologies, such as dynamic tainting, fuzzing, and dynamic symbolic execution, may have problems when applying to large scale real-world programs.

Compared to source code, directly analyzing binary code benefits from the following aspects:

- To an end user, the source code of software is often not available. If not given the assurance of its safety, this software cannot be used in security-sensitive areas.
- Commercial off-the-shelf (COTS) components and DLLs lack source code, which may largely lower the accuracy of source code level analysis on applications using these components.
- Compiler errors or optimizers may cause a WYSINWYX phenomenon: “What You See Is Not What You eXecute”. Thus, source code analysis cannot find low-level problems.
- While source code level tools need to support dozens of kinds of languages, binary level ones only need to support several platform-specific ones.

Dynamic analysis of binary code, such as dynamic tainting [1, 10], has caught the attention of the security community for its effectiveness. However, the path coverage of dynamic technology is highly dependent on the inputs which drive the execution. If inputs are not well chosen, dynamic technologies may have low coverage and may thus miss important errors.

Currently, fuzzing [11-12] remains the main and efficient way in detecting vulnerabilities in binary code. Like dynamic analysis, it has low code coverage and is very inefficient in exploring paths. To hackers, discovering a single vulnerability in well-known software may become a great victory. However, to a software company or a security assurance department, no matter how many vulnerabilities have been detected, they need the confidence to assure that seldom will new vulnerabilities be discovered.

Recently, a powerful constraint-based technology called dynamic symbolic execution has been applied to both source code and binary code [2-7]. Symbolic execution is much more efficient than fuzzing, considering a simple example with a branch “if x == 50”; random fuzzing may need to take 2³² times to hit this branch, while symbolic execution needs only one. Though powerful, dynamic symbolic execution has the path explosion problem, which combits its application to large-scale programs (e.g., conclusion in SAGE [7]).

In contrast, flow-sensitive static analysis has high code coverage but may provide too many false positives. Previous work in static analysis of binary code focus mainly on information recovery (such as Reps’s work [13-14], and others [15-16]), or alias analysis [17-20]. Only few work have focused on bug finding or vulnerability detection [8-9]. To distinguish from them, our work does not aim at detecting vulnerabilities directly; instead, we want to combine it with dynamic symbolic execution, and thus both problems of the two can be solved.

Flow-sensitive static analysis is used widely in industrial level bug finding or vulnerability detection of source code. However, when applying to binary code, it may encounter the following challenges:

- **Lack of information.** The type and structure of variables, function information, logics of original programs have been totally lost.
- **Indirect control transfers.** Programs written with “switch” statements and designed in an object-oriented manner are prone to contain indirect jumps and indirect calls (or call them indirect control transfers), which cannot be statically determined.
- **Alias.** Alias problem is much more serious in binary than in source code, for all variables in source code are assessed by address in binary code.
- **False positives.** It is much easier to rule out false positives in source code than in binary code. In one respect, the logics of program are relatively straightforward to follow in source code, while understanding logics of binary code lacking so
much information is quite difficult. In another respect, developers of the software under test will be called to do this work, while in binary code, it is often the hackers’ work. These two make false positives hard to be ruled out statically. To overcome these challenges, we apply the following existing technologies:

- We make use of the decompile information from the state of the art industrial tool IDAPro [21] and Hexrays [22]. Moreover, we design a type inferring system to recover possible type information.
- We use an accurate Value Set Analysis (VSA) [23] to find out as many as possible addresses to solve both indirect control transfers and alias problems.
- We do not make static analysis the sole technology to detect vulnerabilities; instead, we combine it with other technologies. Currently, we’ve already combined it with fuzzing to convict the potential vulnerabilities we’ve found. In the future, we will combine it with dynamic symbolic execution.

Moreover, to distinguish our work from the previous ones, we make the following contributions:

- We propose a novel approach called semi-simulation to build summaries of binary code. The semi-simulation consists of an accurate VSA algorithm to trace variable addresses and an inaccurate Data Dependence Analysis (DDA) algorithm to trace data flow dependence. Thus, our approach is not only accurate in solving indirect control transfers and alias problems, but also efficient when applying to large programs. Moreover, the semi-simulation is based on summary, which can greatly speed up our analysis.
- We apply a lazy taint checking technology to detect potential vulnerabilities. We do not use taint lattice to trace whether a variable can be tainted by an unfiltered input; instead, we trace the possible data dependence from the entry point. In an early taint checking technology, most time will be wasted on tainting between registers; our approach only changes taint states (taint states in our approach are actually data dependence) at memory store points. Furthermore, accurate ordered results of tainting can be gained for we not only know whether a variable is tainted, but also know from which and which kind of source it is tainted, such as inputs from files, uninitialized parameters, and inputs from users interfaces. To those who want to use other lattices besides taint lattice, our approach is attractive, for we solely trace the data dependence, and thus extensions based on our work need little change.
- We implement our approach on a practical tool called LoongChecker and have already used it to verify three known vulnerabilities and discovered two zero-day vulnerabilities. The primary results not only show the efficiency but also the effectiveness of our tool.

In a word, we use a context-sensitive, flow-sensitive abstract interpretation and dataflow analysis, which is called semi-simulation, to build summaries, and then check the taint property based on summaries. Our approach is neither sound nor precise, however practical, to detect vulnerabilities in binary code.

The rest of this paper is organized as follows: Section II gives an overview of LoongChecker and an example which will be used several times through this paper. Section III describes the general approach, including the basis for our analysis, intraprocedural analysis, interprocedural analysis and taint checking. The core algorithms of VSA and DDA will be described in intraprocedural analysis. Section IV evaluates LoongChecker on three real world programs. Section V discusses the limitations of LoongChecker. Section VI presents related work. Section VII concludes.

II. OVERVIEW

This section gives an overall organization of the LoongChecker, the framework of which is shown in Figure 1. It is built upon two industrial level tools: IDAPro [21] and Binnavi [24].

A binary code, is first parsed by IDAPro to build Control Flow Graphs (CFGs) and then decompiled by Hexrays [22] to get decompile information (e.g., variable type and function information). Binnavi then uses IDAPython to import CFGs from an .idb file, which is generated by IDAPro, to a database. After that, it translates disassembled assembly code to REIL. Actually, our tool LoongChecker is running as a plugin of Binnavi. The translations from assembly to REIL and from REIL to eREIL are demand driven: it does not translate all the functions in a module at once; instead, it translates each function lazily.

A simple source code example is shown in Figure 2(a), and the corresponding eREIL of function foo is shown in Figure 2 (b). eREIL is almost the same with REIL, except some details suitable for static analysis [25]. For example, the “cmp” instruction of assembly code is translated into several lines of REIL to compute eflags. This is needed for dynamic simulation; however, to static analysis, we do not gain any benefit but lose the simplicity to collect constraints. Another example is the arithmetic instructions.

After eREIL is generated, LoongChecker uses intraprocedural analysis and interprocedural analysis to build summaries. It then checks taints on these summaries, and finally gives a list of ordered potential vulnerabilities to analyzers.

Consider a simple example in Figure 2. First, the main function is set as the entry point. The building summary process begins and semi-simulation is used to simulate the current function main. Semi-simulation backups the program state before each call instruction. When encountered a system call or a standard library call, semi-simulation replaces the current program state with the return state of library summaries (these
summarizes are manually simulated). When encountered a call to user defined functions, semi-simulation checks whether the summary of the callee has been built before replacing the current program state. Here, as the summary of foo has not been built, the process of building the summary foo begins. The process of building summary stops if all the summaries of user defined callees have been recursively built. The process of checking taint begins after summaries have been built. At each call site, the backup program state is replaced using variables from the summary of the callee. These functions are simulated manually. If the callee is not a user defined function, such as system or standard library functions, precondition is a list of parameters of the callee. These functions are simulated manually.

In the general approach (shown in Figure 3), we will only show the fundamental elements of our analysis. It mainly consists of two phases: a summary building phase to build a module summary based on intraprocedural analysis and interprocedural analysis and a taint checking phase to detect vulnerabilities based on summaries. In intraprocedural analysis, backward slicing is first performed on the CFG of a function, which selects instructions that should use VSA from those that only need to trace the data flow. In interprocedural analysis, most UIVs are resolved by top-down UIV replacements. Indirect control transfers are solved by performing intraprocedural analysis again [26]. Thus, a module summary from a specific entry point is built. A taint checker then checks all possible sinks based on summaries. Results are finally shown in an ordered manner.

Termination should be assured considering two factors: loops and mutual recursive or self recursive functions. To loops, a simple algorithm [18] is applied. To recursive functions, a break-replace summary-based approach [26] is applied.

### III. GENERAL APPROACH

In this section, a general approach will be presented. First of all, the definition of UIV and precondition will be given, for ease of description.

**Define UIV. Unknown Initial value.**

UIVs are used to represent memory blocks accessed but not initialized in the current function in previous work [18]. In our approach, we treat all unknown values as UIVs (e.g., registers). Furthermore, each UIV is represented using an uppercase symbol, e.g., UIV in eax is represented by EAX.

**Define Precondition. A map before a call instruction, mapping from UIV in callee’s summary to the current value of its relative variable in caller.**

If the callee is not a user defined function, such as system or standard library functions, precondition is a list of parameters of the callee. These functions are simulated manually.

In the general approach (shown in Figure 3), we will only show the fundamental elements of our analysis. It mainly consists of two phases: a summary building phase to build a module summary based on intraprocedural analysis and interprocedural analysis and a taint checking phase to detect vulnerabilities based on summaries. In intraprocedural analysis, backward slicing is first performed on the CFG of a function, which selects instructions that should use VSA from those that only need to trace the data flow. In interprocedural analysis, most UIVs are resolved by top-down UIV replacements. Indirect control transfers are solved by performing intraprocedural analysis again [26]. Thus, a module summary from a specific entry point is built. A taint checker then checks all possible sinks based on summaries. Results are finally shown in an ordered manner.

Termination should be assured considering two factors: loops and mutual recursive or self recursive functions. To loops, a simple algorithm [18] is applied. To recursive functions, a break-replace summary-based approach [26] is applied.
A. Basis

For convenience, we will first define an intermediate language called eREIL (extended Reverse Engineering Intermediate Language) [27].

1) eREIL

The language definition of eREIL is showed in Figure 4. The instruction set of eREIL can be divided into four groups: aop to perform arithmetic operations and bop to perform bitwise operations; dop to transfer data, including str and ldm; jop to transfer controls, such as call, jmp, and jl. eREIL transfers data from left to right. For example, “add t0, eax, t1”, takes t0 and eax as sources, and t1 as the target.

```
program ::= instr*
instr ::= aop var, var var | bop var, var, var | dop var, var | jop
var ::= reg< num | t_num < num | num
aop ::= add | sub | mul | div | mod | bsh
bop ::= and | or | xor
dop ::= ldm | stm | str
jop ::= call var | jmp var | cjmp var, var, var
```

Figure 4. Definition of eREIL.

2) Abstract Model

Define Value. All memory points and registers are treated as variables.

Define Base value. The base value is a UIV, which can represent the initial value of a stack pointer (e.g., esp or ebp).

For example, the first instruction of a function is “sub esp,4”. Using ESP as the base value, the value of esp after this instruction is ESP-4. Thus, all the addresses of local variables can be expressed by <ESP, offset>.

There are five kinds of variables in our abstract model:

- **Stack.** Variables in Stack region are those called local variables. As it’s not possible to decide the accurate value of esp or ebp statically, we refer to local variables by its offset to the current base value. Given a symbol to represent the base value, such as BASE, variables accessed by the form of “BASE-n” are treated as local variables of the current function; those by “BASE+n” are treated as parameters passed from the caller. A tuple <func, base, offset> is used to represent a local variable, in which func represents the current function.

- **Global.** Variables in Global region are called global variables, which are referenced directly by address. There are two ways to access a global variable: by an immediate number or by an offset to an immediate number. We use a tuple <addr, offset> to present a global variable and offset is set to 0 if not accessed by offset.

- **Heap.** Variables in Heap region are called heap variables. In c/c++, their addresses are often generated by a library function “malloc” or a “new” operator. Thus, their addresses cannot be statically decided. We use a tuple <id, offset> to represent a heap variable, in which id is used as an identifier to a heap variable.

- **Register.** In a high level view, registers are a set of temporary variables which perform computation for memory related variables. We represent register variable directly using its register name. For example, we use eax to represent a normal register eax; we use t1 to represent a temporary register t1 (in eREIL, we use, e.g., eax<8>, to indicate that the length of eax is 8 bit; for simplicity, we ignore the length in our description).

3) Constructing Type Inferring System

Define Value. The possible values that each variable may contain during the abstract interpretation are treated as a value in our type inferring system.

Define Type. A type is introduced to each value, to perform the execution correctly.

Our type system is weak typed, for type information has been lost on assembly level, that most types are inaccurate inferences of our analysis. There are five types of values:

- **Undefined value.** There are two kinds of undefined value: 1) in interprocedural analysis, the UIVs of the entry procedure cannot be replaced by any other value; 2) inputs from external environment, e.g., inputs got from library function scanf or fread, cannot be replaced by any other value either. To improve the accuracy, undefined value has several subtypes, such as Entry_Value, File_Input.

- **UIV.** As already mentioned, UIV is an unknown value in the current function and should be replaced during interprocedural analysis.

- **Address value.** Address value is a pointer, which may refer to all kinds of memory variables. An address value may become a function pointer, when used as the target operand of jop group instructions. In semi-simulation, an address value is represented by a value set.

- **Normal value.** A normal value is represented by a dependence list in semi-simulation.

- **Immediate value.** All the integers encountered in assembly are first treated as immediate values.

B. Intraprocedural analysis

Intraprocedural analysis is the core part of building summaries. During it, backward slicing is first used to divide the instructions of the function under analysis into two sets. Then semi-simulation uses VSA on one set and DDA on the other to summarize up the transformer of this function. The semi-simulation is introduced and implemented using the type inferring system we’ve constructed above.

1) Backward Slice

The algorithm of backward slicing (shown in Figure 5(a)) we’ve applied is quite simple. It traverses over the CFG of a given function func in a backward order. This is a work list based algorithm. Each block maintains a set S of memory related registers (we call S register set). In other words, these registers may contain memory addresses, and thus VSA is used to
accurately trace them on these instructions in semi-simulation. The \( r_e \) operand represents a register, a regular one or a temporary one \( t_e \). The \( e_p \) operand represents a register or a number. To \( jop \) group instructions, only the target address operand will be considered here, such as the first operand in "\( jmp \ \text{eax} \)", and the third operand in "\( jg \ \text{t}_1 \ \text{t}_2 \ \text{eax} \)".

Using Figure 2(b) as an example to go through, its CFG (shown in Figure 2(c)) consists of only three normal nodes except the \( \text{Entry} \) and \( \text{Exit} \) nodes. Each normal node contains the start and end of the line number.

The results of performing Algorithm 1 on normal nodes are shown in Table 1. As the example is relatively simple, the \( S \) of each block is \( esp \) or temporary register \( t_e \). All the instructions dealing with addresses are included in \( R \).

Supposing the current block is \( A \), Algorithm 1 first merges all register sets from children in \( \text{Merge} \_\text{Slice} \). The register set of \( B \) is \{ \( esp \) \}, and that of \( C \) is \( \emptyset \), thus \( S \) of \( A \) is \{ \( esp \) \}. Then, during \( \text{Backward} \_\text{Slice} \) of \( A \), each instruction of \( A \) will be traversed in a backward order. To "\( \text{stm} \ e_p, r_j, r_k \)" is an address; to "\( \text{str} \ r_j, r_k \)", \( r_j \) is an address if \( r_j \) is; to "\( \text{idm} \ r_j, e_p \)"; \( r_j \) is an address; to "\( \text{jop} \ r_j \)" or "\( \text{jop} \ e_p, e_q, r_j \)"; \( r_j \) is an address; to "\( \text{aop} \ e_p, e_q, r_j \)" or "\( \text{bop} \ e_p, e_q, r_j \)", \( e_p \) and \( e_q \) are addresses if \( r_j \) is.

\[
\text{Algorithm 1 Backward Slice (cont.)}
\]

<table>
<thead>
<tr>
<th>Backward_Slice(func)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ( W\leftarrow\emptyset )</td>
</tr>
<tr>
<td>2: ( b\leftarrow\text{exit block of} \ \text{func} )</td>
</tr>
<tr>
<td>3: ( W\leftarrow W\cup{b} )</td>
</tr>
<tr>
<td>4: while ( (W\neq\emptyset) )</td>
</tr>
<tr>
<td>5: ( b\leftarrow\text{the first node in} \ W )</td>
</tr>
<tr>
<td>6: ( W\leftarrow W\backslash{b} )</td>
</tr>
<tr>
<td>7: ( \text{Merge} _\text{Slice}(b) )</td>
</tr>
<tr>
<td>8: ( \text{Backward} _\text{Slice}(b) )</td>
</tr>
<tr>
<td>9: ( \text{foreach} \ b \text{'s child} \ c )</td>
</tr>
<tr>
<td>10: ( S\leftarrow S\cup S' ) /( S' ) is ( c )’s register set</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

\[
\text{Algorithm 1 Backward Slice} |

<table>
<thead>
<tr>
<th>Backward_Slice(b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>14: ( \text{foreach} \ b \text{'s instruction} \ i \text{ backward} )</td>
</tr>
<tr>
<td>15: ( \text{switch}(i) )</td>
</tr>
<tr>
<td>16: ( \text{stm} \ e_i, r_i )</td>
</tr>
<tr>
<td>17: ( S\leftarrow S\cup{r_i} )</td>
</tr>
<tr>
<td>18: if ( e_1 ) is register and ( e_1\in S )</td>
</tr>
<tr>
<td>19: then ( R\leftarrow R\cup{i} )</td>
</tr>
<tr>
<td>20: ( \text{str} \ r_j, r_k )</td>
</tr>
<tr>
<td>21: if ( r_j\in S ) then</td>
</tr>
<tr>
<td>22: ( S\leftarrow S\setminus{r_j} \cup {r_j} )</td>
</tr>
<tr>
<td>23: ( R\leftarrow R\cup{i} )</td>
</tr>
<tr>
<td>24: ( \text{idm} \ r_j, e_p )</td>
</tr>
<tr>
<td>25: ( S\leftarrow S\cup{r_j} )</td>
</tr>
<tr>
<td>26: if ( e_2 ) is register and ( e_2\in S )</td>
</tr>
<tr>
<td>27: then ( R\leftarrow R\cup{i} )</td>
</tr>
<tr>
<td>28: ( \text{jop} \ r_j: \text{jop} \ e_p, e_q, r_j )</td>
</tr>
<tr>
<td>29: ( S\leftarrow S\cup{r_j} )</td>
</tr>
<tr>
<td>30: ( \text{aop} \ e_p, e_q, r_j )</td>
</tr>
<tr>
<td>31: if ( r_j\in S ) then</td>
</tr>
<tr>
<td>32: ( \text{if} \ e_1 ) is register</td>
</tr>
<tr>
<td>33: ( \text{then} \ S\leftarrow S\setminus{r_j} \cup {r_j} )</td>
</tr>
<tr>
<td>34: if ( e_2 ) is register</td>
</tr>
<tr>
<td>35: ( \text{then} \ S\leftarrow S\setminus{r_j} \cup {r_j} )</td>
</tr>
<tr>
<td>36: ( R\leftarrow R\cup{i} )</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

Table 1. The results of Algorithm 1 on Figure 2(c).

<table>
<thead>
<tr>
<th>b</th>
<th>S(before)</th>
<th>S(after)</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>B</td>
<td>( \emptyset )</td>
<td>esp</td>
<td>12, 15, 17, 18, 20, 21, 23, 24</td>
</tr>
<tr>
<td>A</td>
<td>esp</td>
<td>esp</td>
<td>3, 4, 5, 6, 8</td>
</tr>
</tbody>
</table>

\[
\text{Algorithm 2 Inferring Types} |

<table>
<thead>
<tr>
<th>Infer_Type(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ( \text{foreach} i \text{'s operand} e_i )</td>
</tr>
<tr>
<td>2: if ( e_i ) is a number</td>
</tr>
<tr>
<td>3: then ( \text{var} : t_i\leftarrow e_i )</td>
</tr>
<tr>
<td>4: else if ( e_i ) is a register</td>
</tr>
<tr>
<td>5: then ( \text{if} \ e_i ) is a UV then</td>
</tr>
<tr>
<td>6: ( \text{idm} \ {r_j} )</td>
</tr>
<tr>
<td>7: ( t_k\leftarrow E_j )</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

(a) The algorithm of backward slice. (b) The algorithm of inferring types.

Figure 5. The algorithm of backward slice and inferring types.

2) Structure of Summary

The structure of summary consists of three fundamental elements:

- **Input Table I.** The input table contains all UIVs, including all of the unknown memory variables and register variables.
- **Call Table \( \Gamma \).** The call table contains all invocations to callee in the current function. For later tracing use, each element \( E_i \in M \) in this table contains information to locate the call instruction. For simplicity, we use \( i \) to represent all the information here. \( M \) represents the precondition of the call instruction.
- **Return Memory \( \Sigma \).** In binary code, it’s hard to tell which memory variables and register variables in the return state of callee will be used by its caller, considering alias and passing parameters by registers. Thus, all the variables except the local variables with negative offsets will be included in the return state.

As we only present a general approach in this section, other issues, such as sink pattern, are not considered. Besides the three elements needed for summary, a variable tuple \( O<\Pi, A, N, H> \), is maintained during the semi-simulation of the current function. \( \Pi, A, N, H \) stand for the table which contains register, local, global and heap variables, respectively.

3) Inferring Types

The operands in assembly can either be a register or be a number. After building the type system, the algorithm of inferring different types is needed (described in Figure 5(b)).
In the algorithm, we use \( V_k : \tau_k \) to represent the value and type of the operand \( e_k \). For simplicity, accesses to a variable by address \( e_2 \) is simplified by \( \tau_2 \). For the same reason, adding a new variable to the variable table is simplified by \( \tau_2 \), which ignores the detail of which table among the four (\( \Pi, \Lambda, N, H \)) will \( \tau_2 \) be added to.

We give a partial order of the casting \( \Theta \) between two types: \( \alpha \mapsto \eta \mapsto \omega \mapsto \tau \) (e.g., if type \( \alpha \) meets type \( \eta \), the result will be type \( \omega \)). The symbol \( \varepsilon \) on two values here represents operations in the semi-simulation.

The algorithm first figures out all the immediate values and UIVs of the current instruction \( i \). Then, according to the type of \( i \)'s opcode, different actions will be taken. To “\( \text{str} \) \( e_1 \), \( e_2 \)”, it first tries to get the variable from \( O \) by address \( e_2 \). If not found, a new variable will be added to one of the tables \( \Pi, \Lambda, N, H \). It then assigns the value of the variable with \( e_1 \). To “\( \text{idm} \) \( e_1 \), \( e_2 \)”, it directly assigns the value of \( e_2 \) with the one of \( e_1 \). To “\( \text{ldm} \) \( e_1 \), \( e_2 \)”, it decides whether the variable with address \( e_2 \) exists in \( O \). If not, it ignores this operation. To the \( \text{aop} \) and \( \text{bop} \) group instructions, the casting between types is decided by a partial order. The “\( \text{bsh} \) \( e_1 \), \( e_2 \)\( e_3 \)” instruction is special, in which the type of \( e_3 \) is decided solely by \( e_2 \).

4) Semi-Simulation
Define semi-simulation. During the abstract interpretation, instructions related to address are simulated in an accurate way, while others are only cared for the possible data dependence.

We call it “semi-simulation” for the reason that instructions not related to address are not really simulated; instead, only the data dependence is traced.

We first give a framework of semi-simulation in Figure 6(a).

This is a forward work list based algorithm. Each block maintains a tuple \( O \) to record variables. We use an operator \( \Phi \) to represent the merging between two \( O \). Semi-simulation uses VSA to simulate the address manipulations, and uses DDA to trace the data dependence. VSA and DDA only differ in the operator \( \Phi \) and \( A \) in the framework. Below, we will introduce them respectively.

a) VSA
The Value Set Analysis (VSA) technology applied by Reps et al. [13-14] is mainly used to recover variable and structure information. Although they intend to find bugs [9] accurately based on their analysis platform, their approach is quite fine-grained and thus time consuming.

For practical purpose, we take a simple approach in implementing VSA. Each memory point is recognized as a variable, and memory points accessed by an offset to an address are grouped into a structure. We also do not recognize arrays, which may have little impact on our analysis.

The algorithm of VSA is shown in Figure 6(b). Each variable maintains a value set, which in the algorithm is represented as \( V_w \). We introduce a new operator \( \Xi \) to represent the \( \text{aop} \) or \( \text{bop} \) group operations. Since it is relative easy, we will not explain it.

Algorithm 3 Semi-Simulation

\[
\begin{align*}
\text{Semi\_Simulation}(\text{func} ) \{ & \\
1: & W \leftarrow \phi \\
2: & b \leftarrow \text{entry block of func} \\
3: & W \leftarrow W \cup \{b\} \\
4: & \textbf{while} (W \neq \phi) \{ \\
5: & \ b \leftarrow \text{the first node in } W \\
6: & W \leftarrow W \setminus \{b\} \\
7: & \text{Merge\_Simulation}(b ) \\
8: & \text{Semi\_Simulation}(b ) \\
9: & \textbf{foreach} b \text{’s child } c \{ \\
10: & W \leftarrow W \cup \{c\} \\
11: & \} \\
\} \\
\text{Merge\_Simulation}(b ) \{ & \\
12: & O \leftarrow \phi \\
13: & \textbf{foreach} b \text{’s parent } p \{ \\
14: & O \leftarrow O \cup \Phi O' \\
15: & // O’ is the tuple maintained by p \\
16: & \} \\
\} \\
\text{Semi\_Simulation}(b ) \{ & \\
17: & \textbf{foreach} b \text{’s instruction } i \{ \\
18: & \text{Infer\_Types}(i ) \\
19: & \} \\
\}
\]

(b) The algorithm of VSA.

Algorithm 4 VSA

\[
\begin{align*}
\Phi( O_1, O_2 ) & \{ \\
1: & \textbf{foreach} O_2 \text{’s variable } v_2 \{ \\
2: & \textbf{if} v_2 \in O_1 \text{ then } \{ \\
3: & \textbf{foreach} V_2 \text{’s value } u \{ \\
4: & V_1 \leftarrow V_1 \cup \{u\} \\
5: & \} \\
6: & \} \\
7: & \textbf{foreach} V_1 \text{’s value } u_1 \{ \\
8: & V_2 \leftarrow V_2 \cup \{u_1\} \\
9: & \} \\
10: & \textbf{return} V_1 \\
\}
\]

Algorithm 5 DDA

\[
\begin{align*}
\Phi( O_1, O_2 ) & \{ \\
1: & \textbf{foreach} O_2 \text{’s variable } v_2 \{ \\
2: & \textbf{if} v_2 \in O_1 \text{ then } \{ \\
3: & \textbf{foreach} L_2 \text{’s dependence } d \{ \\
4: & L_1 \leftarrow L_1 \cup \{d\} \\
5: & \} \\
6: & \} \\
7: & \textbf{foreach} L_1 \text{’s dependence } d_1 \{ \\
8: & L_2 \leftarrow L_2 \cup \{d_1\} \\
9: & \} \\
10: & \textbf{return} L_2 \\
\}
\]

(a) The framework of semi-simulation.

(c) The algorithm of DDA.

Figure 6. The algorithm of semi-simulation.

b) DDA
Unlike VSA, DDA only traces possible data dependences.

The algorithm is shown in Figure 6(c). Each variable maintains a list of possible dependences, which is represented as \( L_w \). We use \( C \) to represent an immediate value.

After introducing DDA, we have finished explaining all the algorithm of semi-simulation. For ease of understanding, the semi-simulation will apply to Figure 2(b). The results of performing semi-simulation on Node \( A \) are shown in Table 2.

155
The algorithm of translating a value set into a dependence list is easy, i.e. simply translating each UIV in each value to one dependence in a dependence list. If the value which represents an address is translated into a dependence list, accesses of this address will be ignored (e.g., “ldm t0, t1!” will not be simulated if t0 is a dependence list).

C. Interprocedural analysis

Considering the call from caller f to callee g, the interprocedural analysis involves two phases: constructing the precondition of each invocation and resolving g’s return state so that f can continue to build its summary.

The core part of both phases is how to resolve UIVs, while other issues are trivial.

1) Resolving UIVs

The most difficult part is mapping from UIV to its relative local variable, for mapping of other variables is straightforward. Now go back to the example in Figure 2(a). Suppose we have just simulated the call instruction at line 6. We are now before the first instruction in Figure 2(b). The current stack layout is shown in Figure 2(d). Using E1 and E2 to represent the initial value of the callee and caller respectively, we can clearly get the equation “E1=E2-420”. To be universal, we use H to replace “-420”, thus, E1=E2+H (**). We use <E1, O1> to represent local variables in the callee, <E2, O2> in the caller, respectively. If the two refer to the same address, another equation E1+O1=E2+O2 (***) can be gained. Replace E1 in ** using *, then we get O2=O1+H.

Thus, if we know the offset (H) of esp to its base value before entering the callee, the offset of each local variable in the callee can be easily mapped to the offset of the relative variable in the caller by adding it with H.

Resolving return state is simple. UIVs in the tuple O (ignore local variables with negative offsets) are replaced using their relative values in the caller.

2) Resolving addresses and indirect control transfers

We apply a break-replace algorithm [26] to solve these problems. We only introduce indirect calls here.

If a call instruction i’s target address A cannot be decided during semi-simulation, the current function f is broken into two pieces: one before the call instruction and one after. The semi-simulation begins after the call instruction as it is a new function f’. During interprocedural analysis, the target function g of i is decided if A can be resolved into an immediate value. After resolving the return state of g, we then replace All UIVs in f’ by the values resolved from g’s return state.

D. Taint Checking

Taint checking consists of two phases: replacing preconditions and checking sink points. Before taint checking, some basic elements should be included:

- **Source recognition.** In the process of building summaries, we have not mentioned the source recognition. However, source recognition happens in both intra- and interprocedural analysis. In intraprocedural analysis, a call instruction to an IO input function is recognized as a source point. Taints are brought in through parameters or the return value. In interprocedural analysis, the parameters of the...
entry point and other uninitialized variables are recognized as tainted.

- **Sink pattern.** Sink points are sometime difficult to recognize. To a dangerous system or standard library function, the recognition is trivial (simply by its function name). However, some dangerous functions are translated into several lines of assembly code by a compiler or optimizers. For example, the function `strcpy` in Figure 2(a) is often translated into several lines of assembly code. Even manually written code may contain “dangerous points”. We design several sink pattern search engines to find potential dangerous points.

- **Library simulation.** It is easy to trace into library functions, for our analysis is based on binary code and we already have the binary code of libraries. However, we have not analyzed library functions for the following reasons: first, we only care about taints, while most libraries, except ones related to IO, will not introduce new taints or remove old ones; second, our analysis only traces data dependencies and thus is not precise. In other words, simulations according to api documents will be more precise than our analysis.

1) Replacing and Checking

After building summaries, each function summary contains an input table, a map of preconditions at each call site, and a return state.

Replacing preconditions and checking taints can be performed simultaneously in a top-down manner. Beginning at the entry point, all the preconditions of each callee of the entry point are replaced using values from the entry point. Recursively repeating this process, all the preconditions of each callee on CG will be replaced by values passing from its caller. At the same time, a taint checking is performed immediately after each replacement at sink points. For example, at a sink point `strcpy` in Figure 2(a), its parameters will be checked whether they are tainted or not. As one parameter `b` is tainted in `main` by `scanf` at line 15, this sink point will be reported as a stack overflow vulnerability. Another potential error is an integer overflow at line 17. From the return state of `foo`, we can infer that the return value of `foo` depends on `a` and `g`. As `a` is tainted by `scanf` at line 14, an integer overflow may happen by setting `a` to a specific value. In this example, `a + 100` will overflow to 1 if setting `a` to `0xFFFFFFFF9C`.

IV. EVALUATION

In this section, we will evaluate our tool LoongChecker on three real world applications. Before that, some implementation details will be presented.

A. Implementation Details

We have implemented our approach on a tool LoongChecker, which consists of more than 30k LOC in java. There are several details that we should pay attention to:

- **Taint checking.** We eliminate taint checking on call sites with the same precondition. This optimization saves nearly 20% of taint checking time (50% in most).

- **Calls between libraries.** As LoongChecker is based on Binnavi, which saves all information in database, dealing with a call to another DLL file is straightforward: simply loading the information of that DLL from database.

- **Vulnerability pattern.** There are two important patterns. The first one is the buffer overflow pattern: 1) there is a loop, 2) there is an iterator, which compares itself with a const number, 3) there are write operations from the iterator to the target, 4) the begin address of the iterator is tainted. The second one is the integer overflow caused buffer overflow pattern: 1) the length parameter of “malloc” or “new” is a tainted variable with a constant, 2) dangerous points or buffer overflow patterns use the return value of “malloc” or “new” as the target address.

Table 3. The evaluation on three real world programs.

<table>
<thead>
<tr>
<th>Module</th>
<th>Serenity player</th>
<th>Fox Player</th>
<th>Kingsoft Office Writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry</td>
<td>Sub_401270</td>
<td>_tmainCR</td>
<td>__tmainCR</td>
</tr>
<tr>
<td>Func</td>
<td>86</td>
<td>274</td>
<td>1264</td>
</tr>
<tr>
<td>MaxDep</td>
<td>6</td>
<td>19</td>
<td>22</td>
</tr>
<tr>
<td>ASMIns</td>
<td>5.0K</td>
<td>20.0K</td>
<td>48.0K</td>
</tr>
<tr>
<td>eREILIns</td>
<td>26.5K</td>
<td>103.8K</td>
<td>237.1K</td>
</tr>
<tr>
<td>SumIns</td>
<td>49383ms</td>
<td>34188ms</td>
<td>65906 ms</td>
</tr>
<tr>
<td>CheckT</td>
<td>15ms</td>
<td>93ms</td>
<td>94ms</td>
</tr>
<tr>
<td>Memory1</td>
<td>19.6M</td>
<td>33.0M</td>
<td>77.1M</td>
</tr>
<tr>
<td>ValNum1</td>
<td>1/8</td>
<td>1/27</td>
<td>0/42</td>
</tr>
<tr>
<td>Detected1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

B. Verifying Known Vulnerabilities

The results of the evaluation are shown in Table 3. For our analysis is summary based, the function number, ASM instructions and eREIL instructions are all counted for only once. The max depth is counted when the callee’s summary has not been built. Thus the real counts are much larger than these. Below, we will explain the three vulnerabilities respectively.

- **Serenity player** [28]. The vulnerability is discovered in `sub_404870`. Using our tool, we find that the taint is introduced in by `fopen` at `0x40497F`. Then it is used as the source of `sscanf` at `0x4049AC`. For the length of the source is not limited, any length of the source can be copied to the target, and thus this causes a stack overflow.

- **FoxPlayer** [29]. The vulnerability is discovered in `sub_40DE00`. The taint is from `ReadFile` at `0x40DFEE`. Like Serenity, it is used as the source of `sscanf` at `0x40E01C`, and for the same reason, it causes a stack overflow.

- **Kingsoft Office Writer** [30]. The vulnerability is discovered in `sub_4864E8C0`. The taint is introduced from another library `numberfmt` in `sub_485E9CB0`. The taint is then passed to `sub_4862B7F0`. From `0x48523DF1` to `0x48523DFE`, this piece of code matches the buffer overflow pattern: 1) it is a loop, 2) `cx` is an iterator, and it compares itself with 0, 3) there is a write from `cx` to `[edx+eax]`, 4) `eax` is tainted, which is tainted from the library `numberfmt`. If inputs from the relative file section does not contain 0, a stack overflow will take place.

We discover the other two vulnerabilities in a similar way with [30], which are both discovered by the buffer overflow pattern.

1 Memory here represents the max memory consumption, including memory taken up by Binnavi during analysis.
V. DISCUSSION

Although LoongChecker has shown that it is powerful to detect vulnerabilities, several limitationsumber its ability.

- **Convicting Vulnerabilities.** It is the most serious problem. Xu et al.’s work [31] shows a suitable way to solve this problem. Currently, we use a fuzzing tool to randomly generate inputs and manually choose those who can reach a given dangerous point. Then we manually trace the parameters of this point back to its source to see whether it is controllable. This process is quite inefficient and makes our work of convicting vulnerabilities difficult. In future, we need a dynamic tool to trace data flow from source to sink point dynamically, for manually tracing is both boring and heavy handed. Our tool already provides a call string from source to sink. However, it provides all the possibilities which may be hard to follow in dynamic debugging. We also need a convicting tool like [31], which can provide a detailed source. In practice, most times, even if you have found the source of a sink point in memory, you still cannot decide whether it is controllable from its final source (e.g., from a file).

- **False positive.** As aforementioned, false positive is the biggest problem of flow-sensitive static analysis. Currently, we are trying two ways to mitigate this problem. First, we collect simple constraints on inputs and dangerous points. In other words, the current return state will be replaced by a map from a constraint to its relative return state. Also, constraints will put upon dangerous points, so that only tainted inputs which satisfy the constraint of a dangerous point will be reported. We do not solve these constraints in a constraint solver, and instead, we solely compare them in a simple way. Second, we prune branches which may be useless. It is hard to decide whether a branch is use or useless, and we make it configurable so that users can give a branch pattern which they believe useless. Both the two have not been implemented in LoongChecker currently. As our final target is to combine our current work with dynamic symbolic execution, false positives can be largely pruned by dynamic symbolic execution.

- **Library simulation.** We only simulate some IO and memory related library functions. In most situations, it is enough; sometimes, however, this may cause big problems. Consider a map which is implemented by a C++ library. We do not simulate the library operations on this map, such as “put” and “remove”. Thus, a key-value pair added into this map is ignored by our analysis and if the value appears again later, we treat it as an undefined value.

- **Entry point.** For a more accurate analysis, the entry point should be well chosen. If choosing a quite upper function as the entry point, our analysis may provide lots of false alarms, for largely widening of results. Moreover, for message based applications, such as most GUI applications, the control flow is hard to trace; choosing an upper function may soon lead to the termination of the analysis. However, if choosing a bottom function, the results may be useless for lacking of taint sources. We leave the entry point configurable too.

VI. RELATED WORK

There are two pieces of work most similar to our work: Guo’s [18] and Reps’ [9, 13-14, 32-33]. Although we use a summary structure similar to Guo’s, ours still differs from theirs in the following aspects:

- **SSA.** They use a SSA (Static Single Assignment) form. SSA can ease analysis in several ways: input recognition, expression propagation and ease of use-define analysis [16]. However, recognizing input is simple; Expression propagation is less useful for we only use DDA for most instructions; although use-define information plays an important role in our analysis, our analysis needs only one pass of analysis, while there will be two if adding SSA, plus the time of translating out of SSA.

- **UIV.** They do not treat registers as UIVs. This will make the analysis quite inaccurate if some parameters are passing by registers. In contrast, we treat all unknown values as UIVs.

- **Address parameter.** They assume that there are no address parameters. This assumption makes their work quite limited in practice. Moreover, this is the most difficult part of summary based approaches. We deal with indirect jumps, indirect calls and variables with unknown addresses, which are all caused by address parameters, using a break-replace algorithm [26].

Reps’ work focuses on the recovery of variables and structures, in order to provide a platform for further analysis. We differ from them in the following ways:

- **VSA.** They use intervals to approximate array-like structures and improve efficiency, while we do not recognize arrays and use a simple implementation of VSA. We find this inaccuracy have little impact on detecting vulnerabilities.

- **Predicts.** In their recent work [9, 34], they use a meet-over-all-paths (MOP) solution instead of a maximal-fixedpoint (MFP) one to increase the accuracy. Their approach is similar to ESP [35] and Fischer et al.’s [36], both of which are considered not practical by us. ESP selects paths related to the properties they are interested in. It is useful when checking a specific property, but may encounter path explosion problem when checking many. Fischer et al.’s, however, may be too expensive to refine predicates.

Summary-based approaches have long been used in static analysis [37-40]. While most of these approaches are available in source code, few work have been done in binary code [8, 18, 34, 41]. Cova et al.’s work [8] is based on Kruegel’s [41], using a symbolic execution technology which ignores alias information.

VII. CONCLUSION

This paper presents a novel semi-simulation approach to statically detect potential vulnerabilities in binary code. A tool called LoongChecker has been implemented and evaluated on real world programs. The results are promising. In the future, we’ll develop an assistant dynamic tool and apply our approach to more real world programs.
ACKNOWLEDGMENTS

This research was supported in part by the Fundamental Research Funds for the Central Universities of China (No. WK0110000007).

REFERENCES

[29] FoxPlayer exploit. www.exploit-db.com/exploits/11333/