國立交通大學

網路工程研究所

碩 士 論 文

利用迴圈特性加速靜態與動態程式分析

RELEASE: Generating Exploits using Loop-Aware Concolic Execution

研 究 生：李秉翰

指導教授：謝續平 教授

中 華 民 國 九 十 八 年 八 月

利用迴圈特性加速靜態與動態程式分析

RELEASE: Generating Exploits using Loop-Aware Concolic Execution

研 究 生：李秉翰　　Student：Bing-Han Li

指導教授：謝續平　　Advisor：Shiuhpyng Shieh

國 立 交 通 大 學
網 路 工 程 研 究 所
碩 士 論 文

A Thesis

Submitted to Institute of Network Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

August 2009

Hsinchu, Taiwan, Republic of China

中華民國九十八年八月

利用迴圈特性加速靜態與動態程式分析

學生: 李秉翰　　　　　　　　　　　　　　指導教授: 謝續平 教授

國立交通大學網路工程研究所 碩士班

摘要

自動尋找軟體漏洞以及產生如何滲透軟體安全之過程爲當今軟體測試方法所迫切需求。實際/符號混和執行測試技術(concolic execution)爲符合此需求的新技術之一，其結合了實際執行測試的速度優點以及符號化執行測試的廣泛可測範圍。然而，此技術繼承了符號化執行測試的限制 –面對迴圈時，當迴圈執行次數與外部輸入值有相依性，此技術必須將每種可能的外部輸入值都執行過一次，進而造成效能嚴重降低，甚至退化成爲隨機測試。而迴圈是程式語言中大量使用的一種必要格式，這造成此技術面臨相當大的挑戰。在本論文中，我們提出一個新的實際/符號混和執行測試技術，稱爲：”迴圈感知實際/符號混和執行測試技術(*loop-aware concolic execution)*”。本新技術可精確分析迴圈相關變數，並減少軟體測試所需之時間。爲了展示此項新技術，我們開發了一套分析系統，稱爲：”RELEASE”。在本分析系統中，我們將此項新技術應用在分析緩衝區溢位漏洞，並產生如何滲透軟體安全之外部輸入值。

RELEASE: Generating Exploits using Loop-Aware Concolic Execution

Student: Bing-Han Li

Advisor: Dr. Shiuhpyng Shieh

Institute of Network Engineering
National Chiao Tung University

ABSTRACT

Automatically finding vulnerabilities and even generating exploits are eagerly needed by software testing engineers today. And for security issue, many testing software are usually lake of source code and symbol table information. Concolic execution is a novel technique, which takes advantage of the rapid executing speed of concrete execution and the wide testing coverage of symbolic execution, to find and understand software bugs, including vulnerabilities, with only analyzing machine code. However, a serious limitation of concolic execution inherited from symbolic execution is its poor analysis result with loops, a common programming construct. Namely, when the number of iterations depends on the inputs, the analysis cannot determine possible execution paths of the program. In this paper, we propose a new concolic execution technique, *loop-aware concolic execution*, for testing software and producing more precise analysis on loop-related variables with fewer execution steps. To demonstrate our technique, we developed a concolic analyzer, called RELEASE, and apply it to discover buffer-overflow vulnerabilities and generate exploits of software.

誌　謝

首先要感謝我的父母，辛苦的養育我，你們太偉大了，我會好好孝順你們的。也感謝妹妹在生活上給了我很多好意見，恭喜妳推甄上心目中的好學校－交大。

再來感謝我的指導教授，謝續平老師，在我的碩班生涯教導了我很多事，還讓我有機會出國到 UC Berkeley 待一個學期進修，使我的視野擴展許多。

感謝實驗室的學長姐，繼偉、桶哥、Stanley、經偉、Q 咪，在我碩一矇懂無知時，帶領著我。感謝實驗室同學，Vic、Toby、雨芊、Kuro，一起扶攜度過不同科目的轟炸。感謝實驗室學弟，梅子綠、花生、阿不拉、Fallman、Alan，幫忙扛計畫，讓我碩二時可以專心寫論文。感謝實驗室碩零新生，峰哥、江江江、修醇、嘉偉、芳瑜，讓實驗室更加熱鬧。感謝實驗室助理，明華、順瑩、季榆，平時多虧你們的照顧。

感謝子逸、佳純、Yellow 在我要到國外時給了很多寶貴意見。感謝 iCAST Term4 同學，煥宗、Vink、Jason、丁丁、王玨、軟啪啪、Ming、Tracy、Vic、Hsuany，以及在 Berkeley 認識的大家，大姐、Ili、Carol、Noral、Flora、Andy、清華男、Jack、Takumi、還有好多人，有了許多共同寶貴的回憶。

感謝大學死黨，肯尼、馬林、團長、羅淳蛙、KD、小班，永遠的好朋友！感謝成大新竹幫，F 龍、發哥、乙慈、姿容、校長、永澤、cc 實驗室、永生實驗室，平常給了我很多幫助。

感謝陪我度過這段時間的所有人。

謝謝大家！！！

Table of Content

List of Tables

List of Figures

1. INTRODUCTION

Automatically finding vulnerabilities and even generating exploits are eagerly needed by software testing engineers today. And for security issue, many testing software are usually lake of source code and symbol table information. Therefore, a technique to analyze executable program in just binary form is essential. Traditionally, software testing techniques can be classified into two categories, one is black-box testing, which does not utilize information inside the testing target and only concern about the relationship between input and output. The other is white-box testing, which looks deeper into the target and cares about the behavior inside. Extended and blended in recent researches, the two concepts gave birth to two advanced testing mechanisms called concrete execution and symbolic execution.

Concrete execution runs the target program with one set of specific inputs and detect whether the outputs satisfy the predefined requirement to find out the correspondence between inputs and outputs. Fuzz testing [1] (also known as *fuzzing*) is a technique of black-box testing using several times of concrete executions. Input format must be known by fuzz testing tool, and a valid input is picked for each execution. After each run, some strategies are taken by different fuzz testing tools. One is random type, the next valid input is randomly chosen; another is regional type, the next input is changed slightly from present one; and the other is feedback type, the next input is chosen by observing the previous outputs. There are several fuzz testing tools have been implemented, like MetaFuzz[2] and Peach[3]. Their speed of each run is very fast. But the most devastating problem of these tools is their poor test coverage. Without inner information of programs, fuzz testing tools can only explore the space of input set one by one. This strategy is very inefficient when the input space is large or even infinite.

Symbolic execution [4] is a very useful technique of white-box analysis. The basic idea is to symbolize each input variable of the program. After each statement is executed, variables in that statement will be symbolized also. Namely, those variables whose values depend on inputs will be formulated with input symbols their values would be constrained by branching conditions. This approach can be used to predict inputs according to specific path constraints. It is also important for considering path traversal. Although the testing coverage could be fully tracked, the effort to do symbolic execution is too large because the program does not execute directly on native machine. The testing scale of symbolic execution is limited.

Concolic execution is a new noun which was first presented in [5] and stands for cooperative concrete execution and symbolic execution. This novel technique takes advantage of the rapid executing speed of concrete execution and the wide testing coverage of symbolic execution. By representing inputs as symbolic variables and performing operations on values dependent on these symbolizations, mixing concrete and symbolic execution generates a set of specific inputs for single concrete execution. A set of execution path constraints (also called branch constraints) will be generated by each concrete execution. By inversing one of constraints in the set for each execution, concolic execution can be systematic and heuristic to explore all the paths in the testing program.

A serious limitation of concolic execution which inherited from symbolic execution is that it deals poorly with loops, a common programming construct. The testing situation is extremely stable when the number of iterations in a loop is constant. But it becomes problematic while the number of iterations depends on the inputs, since in principle loops should be executed any possible number of times to try to explore all possible paths of a program. In other words, in concolic and symbolic execution, the values of symbolic variables reflect only data dependencies on the inputs. Untraceable control dependencies such as loops could cause poor analysis result.

In this paper, we propose a new concolic execution technique, *loop-aware concolic execution*, for testing software and producing more precise analysis on loop-related variables with fewer execution steps. In loop-aware concolic execution, because the path constraints will be recorded, the data flow can be traced precisely no matter it was truncated by control flow or not. Based on the information of data and control flow, a backtrack method is used to connect the relationship between loop-related variables and input.

For applying it to detect buffer overflow vulnerabilities and generate related exploits of a program, we have built RELEASE, a prove-of-concept of this technique, using concolic analysis in x86 platforms with ELF binary format.

In summary, this paper makes the following contributions:

- We introduce loop-aware concolic execution, a new concolic execution approach with more information of loops, to enhance execution speed by eliminating unnecessary testing.
- We apply loop-aware concolic execution to an important security challenge, said buffer overflow vulnerabilities.
- We do not require high-level source code of the testing program; this restriction-relaxed approach is important to analyze any kind of applications, include important security applications.

- We evaluate our implementation, showing that it is effective and efficient at discovering vulnerabilities and generating exploits.

The rest of paper is organized as follows: Section 2 surveys related work. Section 3 motivates our loop-aware concolic execution with an example and then gives a detailed system overview. Section 4 shows the implementation of our approach and describes the main algorithm used to analyze loop dependencies. Section 5 gives our evaluation of our approach. The paper ends with a conclusion at Section 6.

2. RELATED WORK

This section discusses different approaches for doing software testing, and compares them with our approach.

Fuzz testing. Fuzzing is a black-box software testing technique that consists in feeding a program with *random* input data. A bunch of tools [2, 3] systematically generate testing inputs for maximum reach whole the input space. The most problem of fuzz testing tools is holding scarce inner information because they only concern about the pairs of inputs and outputs. A new type of fuzz testing technique called *white-box fuzzing* [6, 7] introduces symbolic execution to look inside the black-box, and it can be categorized as concolic execution which will be discussed later.

Static analysis. Static analysis technique is an important part of security analysis because they provide results without having to actually run a program, thus avoiding risks connected with the execution of malicious programs. Moreover, it can precisely predict the behavior of code statements in run time. Unfortunately, there are a number of challenges that have to be overcome by static analysis tools [8], like indirect calls. Also in [9], an approach combines symbolic execution and taint analysis is proposed to detect program vulnerabilities, that cannot resolve all indirect transfers of control, which is plenty used in executables.

Dynamic taint analysis. Taint-based approaches [10, 11] try to analyze the data dependences by tracking data flow from inputs or specific variables. These approaches are efficient but lack of control flow analysis, thus some analysis evasion mechanisms [11] are proposed to evade tainting track. Even worse, some of these mechanisms are commonly used in normal programming, like unlinear function *atoi()* and if-condition, they could cut the data flow if there is no control flow analysis.

Concolic execution. Concolic execution technique is a novel program analyzing and testing technique. Unlike above approaches, it traces both data flow and control flow of a program and can more precisely know data dependences than dynamic taint analysis. For programs with source code, there are several excellent research results [5, 12-16] which have been proposed. Especially in EXE[15], a useful tool called STP is implemented as path constraint solver. Similarly, DART[12] and its successor CUTE[5] also propose a mechanism to generate test cases from symbolic inputs, but they have troubles at pointer aliasing stated in [15]. Thus, our approach selects to use STP as concolic execution constraint solver. For programs without source code, it is more difficult to analysis because lot concepts of high-level language are loss after compiling to native machine code. It conducts that only few existing concolic execution approaches to analyze binary executables. One idea presented in [6] is only proposed and not implemented and evaluated. Another system, called Catchconv[17], is a powerful tool to analyze for integer conversion errors, but also struggles on loop. Third approach proposed in [18] is more closely related to ours, but our approach not need to know input grammar of testing programs.

Table 1 shows the comparison of our approach and other two loop-related works, which also focus on loop, and their inputs are binary executables.

	our approach RELEASE	**Static Detection [6]**	**Loop-Extended [8]**
Analysis method	concolic	symbolic	concolic
Loop modeling	loop counter	execution 3 times (Imprecisely)	trip count
Modeling capability	polynomial	linearity	linearity
Input grammar	not require	not require	require

Table 1: Comparison of our approach and other two loop-related works.

3. OVERVIEW

In this section, we first motivate our approach with an example showing the limitation of previous concolic execution, then give an overview of concolic execution, finally point out the core techniques of our loop-aware concolic execution.

3.1 Motivation

Using concolic testing to observe the behavior of a program is a powerful technique because it combines the strengths of static and dynamic analysis. Static analysis techniques [9] analyze the executable without actually running it. The program code is analyzed by inferring over all the possible behaviors that may appear at run time, this action can show the whole control flow of the program. In contrast, dynamic analysis techniques, while unable to examine all the possibilities in one execution, can provide more accurate information about the data flow because the appropriate variables values in any particular execution states are known at run time, thus avoiding the imprecise analysis like aliasing problem [19]. However, concolic execution technique has a significant limitation in programs that contain loops. A specific example is illustrated to show the power of concolic execution and its limitation.

```
1 int username_check(char * name) {
2       int i = 0, len = 0;
3       char buf[100];
4       while (name[i] != '\0') {
5               i++; len++;
6       }
7       if (len < 5) {
8               printf("error: length is not enough.\n");
9               exit(-1);
10      }
11      for (i = 0; i < len; i++)
12              buf[i] = name[i];
13      buf[i] = '\0';
14      return check_duplication(buf);
15      }
```

Figure 1: An example function to illustrate the limitation of normal concolic execution.

Consider the function *username_check*, shown in Figure 1, which processes a name of a new user who wants to register. This function first gets the length of name by iteration on lines 4-6, and then checks the length limitation on lines 7-10. It will return an error message and exit right away when the length is too short. Finally, the context of name is copied to a buffer for next step checking.

There is an obvious vulnerability in this code to buffer overflow, but suppose we are using normal concolic execution technique to check for such weaknesses. For instance, a random input said string "in" is first generated, and then the program executed and exited on line 9. To explore another path to reach rest of the function, normal concolic execution tool notices that it should generate an input that can pass the check on line 7. However, a normal concolic execution tool does not have enough information to find out the relation between the variable *len* and the input *name* because *len* is not directly dependent on any bit of *name*. At this point, based on the faith of concolic execution, the tool will change the content of input *name* according line 4, and need to change input and re-execute four times to pass through line 7. In worst case, concolic execution is fall back to fuzz testing to generating random inputs for passing the checks with indirectly or unknown dependence on inputs.

To avoid analysis evasion and enhance the efficiency of analysis speed, we propose a new type of concolic execution, loop-aware concolic execution, to capture the dependences between loops and variables related with program inputs.

Throughout the paper we will use the running example given in Figure 1. Note that the example is in a C-like language for the sake of clarity and simplicity, while our analysis operates on binary code. In particular, we assume that our analysis will be operating on dynamically-linked x86 executable objects, formatted according to the Executable and Linking Format (ELF). We also assume that the analyzed executable follows a "standard compilation model": the executable has procedures, a global data region, a heap and a runtime stack; global variables are located at a fixed memory location; local variables of a procedure are stored at a fixed location in the frame stack of that procedure; the program follows the cdecl calling convention, and is not self-modifying.

3.2 Concolic Execution Overview

The process of concolic execution system could be divided into four steps: static analysis, initial input generation, concolic execution, path constraints solving. The following paragraphs give detail descriptions.

Step 1: Static analysis on binary executable. Concolic execution system does static analysis of the binary program and constructs control flow. The existence of this step depends on the analysis method. Our approach takes this one to remove some analysis overheads during runtime.

Step 2: Initial input generation. Concolic execution system needs to decide the program input for the first execution. It may randomly choose one or use external information like input grammar [18]. Our approach chooses the initial input statically. For next execution, the new input will base on the execution result of previous one to explore new execution path.

Step 3a: Concrete execution. The result of each operation will be simulated. Namely, every instruction are "virtually" executed or, in other words, real executed in a virtual environment. The first concrete execution help the system explore one execution path of the testing program. After that, concrete execution is performed to explore the branches of the path. In this step, concolic execution system is running symbolic execution simultaneously.

Step 3b: Symbolic execution. During concrete execution, the symbolic execution is also performed to symbolize input-related variables and gather path constraints. The initial symbols are program inputs. Because the data flow will be precisely tracked in the runtime, every variable (i.e., register, memory) which values are related to program inputs will be also symbolized. By taking this action, a path constraint can be recorded whenever there is a conditional branch which one of the comparing variables is symbolized.

Step 3c: Behavior observation. After each instruction executed, concolic execution system could observe the program executing behavior and analyze it. An executing behavior could be memory access, external device control (i.e., hard disk, network card, etc.), or even crashed exception throwing. Some applications profile the specific behavior of a program, like calculate clocks, to help optimize the performance.

Step 4: Solve path constraints for next execution input. After each execution in Step 3, concolic execution system will collect a set of path constraints. One set of path constraints is corresponding to one particular execution path. To explore a new execution path, a path traversal action that inverts one constraint in pervious set is taken. After that, new inputs will be generated by solving these new set of constraints, and a new execution starts to Step 3.

By concolic execution, all the execution paths in program would be explored. But a challenge is present when encounters loop: concolic execution needs to “unroll” loop (namely, execute loop block several times) for each input. This action is not efficient, and even worse, may cause the whole analysis process to retreat to only generate random inputs as a fallback plan.

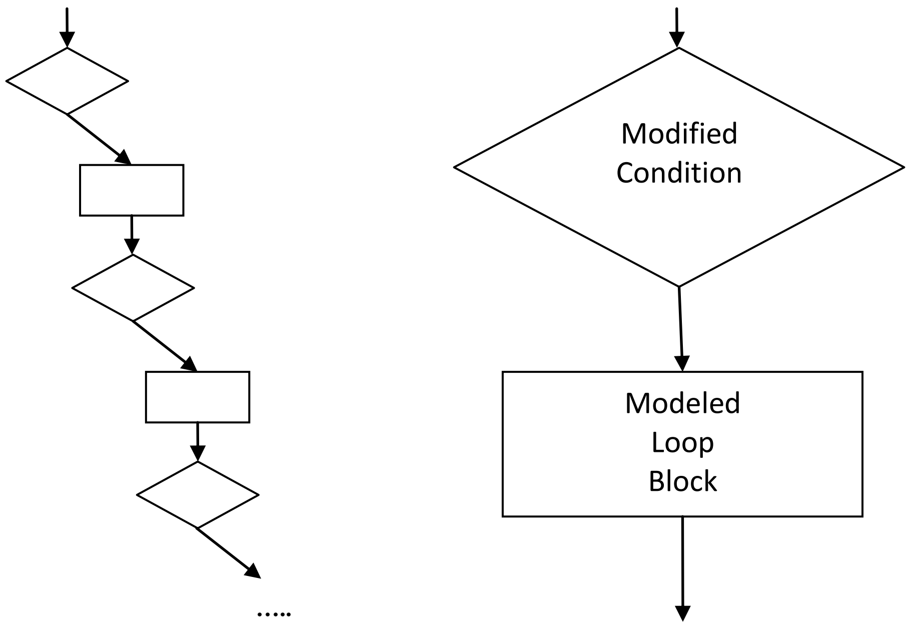


Figure 2: The different between traditional approach and ours. The left figure is a loop unrolling method using by traditional approach. The right one is a loop modeling method using by ours.

3.3 Technique Overview

We propose a new concolic execution technique, *loop-aware concolic execution*, which enhances analysis speed by modeling the effects of loops. The effects include register value changing and memory access. Figure 2 illustrates the most difference between previous concolic execution and our loop-aware concolic execution technique. As previous concolic execution, it needs to execute *n* times to explore *n* states when it encounters a loop. In the worst case, *n* could be infinite (or, as large as input space). That makes previous technique not efficient to analyze loop construct in a program.

To enhance the analysis speed, our loop-aware concolic execution technique focuses on a specific loop construct which has only one break condition and no inner branch. These two limitations are used to simplify the modeling process. Later we will discuss how to modify the process from single break condition to apply multiple ones and deal with inner branches. The following three techniques are the core parts to model the effects of loops:

Dynamically locate specific loop. Before modeling loops, we need to find out the locations of them and make sure their properties are fit in our rules. Because the target program is in x86 executable format and may have indirect jump, this task cannot be complete in static analysis. To fully identify the loop blocks, this task should be also performed in runtime. During the execution time, the executed blocks will be precisely recorded and the re-executed block can be easily recognized.

Loop counter recognition. The definition of a loop counter is “a variable which value may change and must compare with other values in every iteration.” By verifying the break condition, we summarize that there are two types of comparison: variable compares with concrete value or variable compares with variable. For the former one, it is very easy to identify which parameter should be the loop counter. But for the latter one, it is difficult to tell which one should be the representative. In this case, our approach takes both of these two parameters as loop counters. The loop counters are symbolized as new symbols. The variables which is used in loop and related to loop counters will be also symbolized in the modeling technique.

Loop block re-sorting. For a loop block execution, the loop-executed process is not always stop in the end of block because the loop break condition may present in any place of whole loop sequence. The sequential symbolic execution is not suitable for the situation that the break condition is not in the end of loop block. In other words, the traditional way cannot model a loop with break conditions inside. To handle this problem, a loop block re-sorting technique is proposed. The basic idea of it is extends code before break condition and adjusts the loop block as a completely none-break-condition one. This technique helps symbolic execution can be applied in general case.

Loop Behavior Modeling. After getting the loop execution range, the next step is to model the effects of loops. By observing the behavior of program execution, the operations can be divided into three categories, called register value changing, memory access, and control flow director. For the first category, register value changing, it can be modeled as the second category by taking registers as additional memories. In the second one, memory access, it is composed of two actions: read and write. For applying to detect buffer overflow vulnerabilities, our approach checks the memory write destination address can access the target address (i.e., return address) or not. In order to achieve this goal, a secondary symbolic execution for modeling loop is applied. By using this technique, the loop-related variables can be precisely described their relationships of each other and the connection with loop counters.

4. IMPLEMENTATION

We implemented RELEASE for testing x86 ELF format binary executables. Figure 3 illustrates RELEASE's system architecture. At the highest level, RELEASE consists of a static analyzer and a dynamic analyzer which bases on a dynamic binary instrumentation (DBI) platform. In the following section, we will discuss each part of RELEASE deeper, followed by a detailed algorithm description.

4.1 Static analyzer

Static analyzer is the main part to extract information from binary program. By moving workload from dynamic analyzer to static analyzer, it can effectively mitigate plenty analysis overhead during the run-time. Our static analyzer is composed of a disassembler, a control flow constructor, and a static loop locator. The information in binary program is revealed step by step by running each of them.

Disassembler. RELEASE uses a disassembler to translate machine code to assembly code, then passes assembly code to control flow constructor for next step. To get other useful information of binary program is also very important, such as virtual address base, sections, and procedure linkage table (PLT). The virtual address base helps our system to handle the memory address relationship between real addressing and virtual one. The sections information let RELEASE know how to deploy each of them. By getting PLT, it makes our dynamic binary instrumentation subsystem can map the outer library function calls correctly. In our implementation, we choose to use two linux open source tools to achieve this goal: one is a disassembler called GNU objdump, and another one is an executable information revealer called GNU readelf, both of their versions are v2.18.93.20081009.

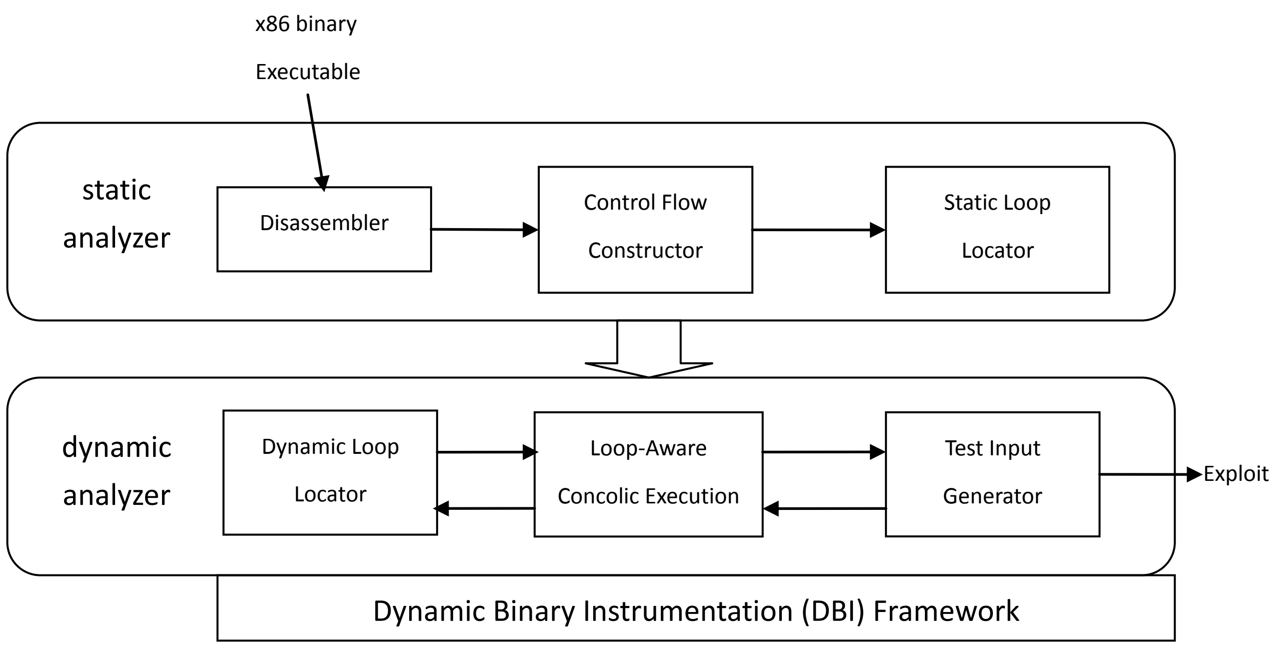
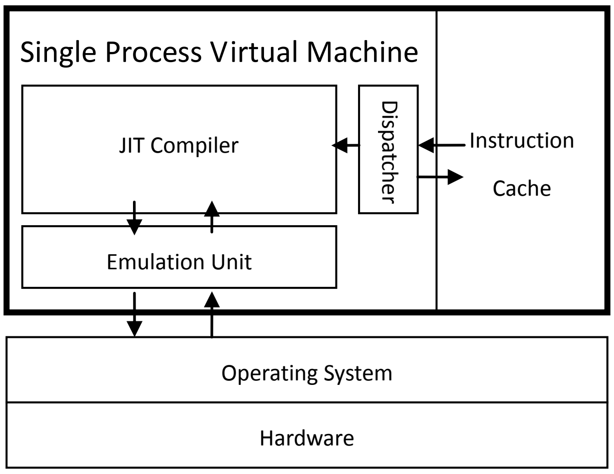


Figure 3: Illustrate of system architecture of RELEASE.

Control flow constructor. After analyzing each instruction, control flow constructor converts assembly code to its own data structure, and groups them as a "basic block," which contains a sequence of instructions and has single entry and single exit. RELEASE first divides basic blocks by using a linear scan method to search control-flow changing instructions like call, (un)conditional jump, and return. Then RELEASE does a different process than other frameworks, such as Valgrind[20], QEMU[21], and PIN[22]. Basic block is cut again by target addresses of direct call or jump and broken down to "atom block". The main idea different between other approaches is that RELEASE deals with direct control-flow changing instructions while static analysis rather than in run-time, thus make it fit in with our principle: "By moving workload from dynamic analyzer to static analyzer, it can effectively mitigate plenty analysis overhead during the run-time." To clarify and unify the definition, the phase "basic block" is used as "atom block" in the rest of paper.

Figure 4: DBI framework architecture



Static loop locator. By using control flow information, the “directly” loop location can be easily to identify. That means, by taking advantage of x86 instruction formats, the operand of direct conditional jump must be a relative offset. A relative offset is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit or 32-bit immediate value, which is added to the instruction pointer.

4.2 Dynamic Binary Instrumentation

Our dynamic binary instrumentation (DBI) framework has three components: a dispatcher, an emulator, and a just-in-time (JIT) compiler. The relationships between them are shown in Figure 4. The instruction cache inherits from control flow constructor described in Section 4.1. The dispatcher selects a target basic block to JIT compiler for execution. JIT complier fetches each instruction in the selected basic block and emulates their behaviors. In the emulation unit, an emulated memory system is implemented for storing the emulation results. By using this DBI framework, both instructions of normal program and user instrumentation can execute correctly.

4.3 Dynamic analyzer

Based on concolic execution, loop-aware concolic execution does concrete and symbolic execution simultaneously. As a new approach applied on concolic execution, a serial of methods is proposed to analyze program behavior. The names of these methods are dynamically loop finding (DLF), loop counter recognition (LCR), loop block re-sorting (LBR), and loop behavior modeling (LBM). Their functionalities are described in Section 3.3. The following paragraphs illustrate their implementations in RELEASE system.

Dynamic loop locator. Dynamic loop locator is the implementation of DLF. While RELEASE system running, the executed flag of executing basic block will be marked. Dynamic loop locator will check the executed flag of new fetched basic bock is marked or not to identify the control flow is a loop or not.

Loop-aware concolic execution. By modeling the loop behavior, we apply our analysis method, LCR, LBR, and LBM, to discover buffer overflow vulnerabilities. RELEASE system uses a recurrence relation solver, called PURRS[23], on LBM to solve the modeling problem. This solver has the capability to calculate polynomial recurrence relation. For instance, a polynomial recurrence relation equation is like:

$$x(n) = x(n-1) + n^2 + 3n + 2$$

And the solved equation is:

$$x(n) = \frac{1}{3}\mathrm{n}^3 + 2\mathrm{n}^2 + \frac{11}{3}\mathrm{n} + \mathrm{x}(0), \text{ for n >= 0}$$

Test input generator. To solve the path constraints gathered in concolic execution, RELEASE system uses a path constraint solver, called STP. The result answer is generated as the next execution input.

5. EVALUATION

We performed a series of experiments with RELEASE to evaluate the effectiveness of loop-aware concolic execution based on our proposed techniques. By applying it to discover buffer overflow vulnerabilities, RELEASE will generate the corresponding exploits to show the correctness of our scheme. All experiments were run on a Ubuntu 8.10 system, equipped with a 2.5 GHz AMD Athlon 64 X2 Dual processor and 3.25 GB RAM. All executables were obtained using the gcc 4.3.2 compiler using its standard configuration.

5.1 Evaluate static analysis

Our approach analyzes a program in two phases: static and dynamic. In static phase, we evaluated the time consuming for different program size. As Figure 5, the result shows the analysis time is in a direct ratio with file size.

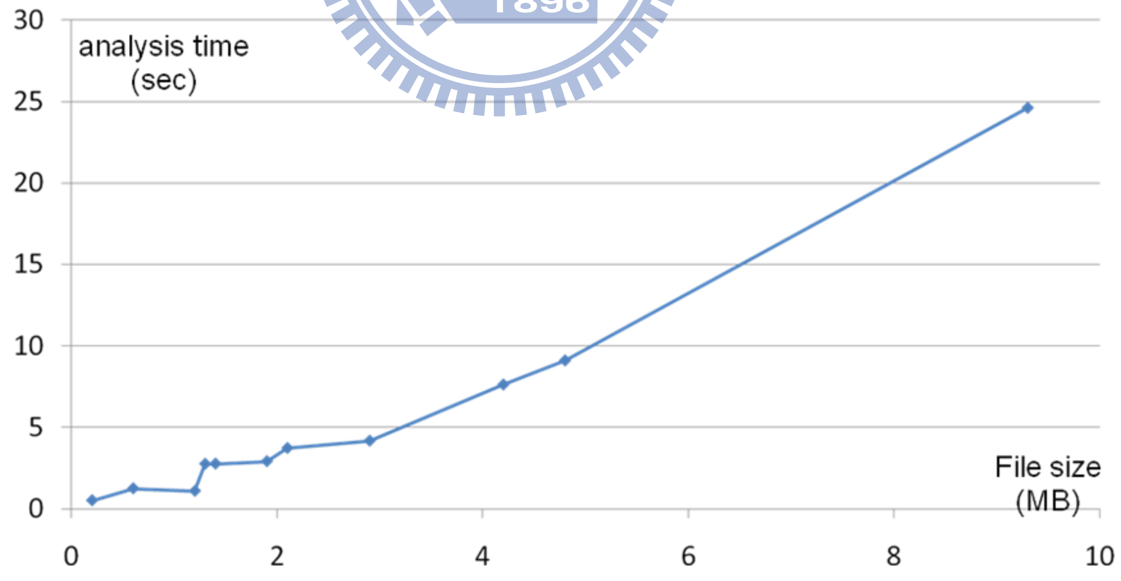


Figure 5: Evaluated result of static analysis time consuming.

5.2 Evaluate dynamic analysis

In dynamic phase, we evaluated the time consuming for different number of loops. Furthermore, we discuss in two different situations: loops are input-related or not. A loop called input-related is that one of instructions in the loop block is related with input. Figure 6 is an evaluated program with one loop and the loop is not input-related. Figure 7 is another evaluated program with two sequential loops and not input-related. The other evaluated programs which amount of loops is from three to ten are in the same format. Figure 8 shows an evaluated program with three loops and the loops are input-related. Figure 9 is an evaluated program with inner loop and the loop is not input-related, and Figure 10 is another program with inner loop and the loop is input-related. Figure 11 illustrates when loops are not input-related, the analysis time of RELEASE is a little high than a normal concolic execution. But when loops are input-related, RELEASE keeps a *O(n)* relation in analysis time, and a normal concolic execution raises to *O(n^2)*. Figure 12 shows the comparison result.

```
int main(int argc, char *argv[]) {

    int i;
    char buf[1024];

    for (i = 0; i < argc; i++)
        buf[i] = 1024;

    return 0;
}
```

Figure 6: An evaluated program with one loop and the loop is input-related.

```
int main(int argc, char *argv[]) {

    int i, j=1024;
    char buf[1024];

    for (i = 0; i < j; i++)
        buf[i] = 1024;

    for(i = 0; i < j; i++)
        buf[i] = 1024;

     return 0;
}
```

Figure 7: An evaluated program with two sequential loops and the loops are not input-related.

```
int main(int argc, char *argv[]) {

    int i;
    char buf[1024];

    for (i = 0; i < argc; i++)
        buf[i] = 1024;

    for (i = 0; i < argc; i++)
        buf[i] = 1024;

    for (i = 0; i < argc; i++)
        buf[i] = 1024;

    return 0;
}
```

Figure 8: An evaluated program with three loops and the loop is input-related.

```
int main(int argc, char *argv[]) {

    int i, j;
    char buf[1024];

    for (i = 0; i < 1024; i++)
        for(j = 0; j < 1024; j++)
            buf[i+j] = 1024;

    return 0;
}
```

Figure 9: An evaluated program with inner loop and the loop is not input-related.

```
int main(int argc, char *argv[]) {

    int i, j;
    char buf[1024];

    for (i = 0; i < argc; i++)
        for(j = 0; j < argc; j++)
            buf[i+j] = 1024;

    return 0;
}
```

Figure 10: An evaluated program with inner loop and the loop is input-related.

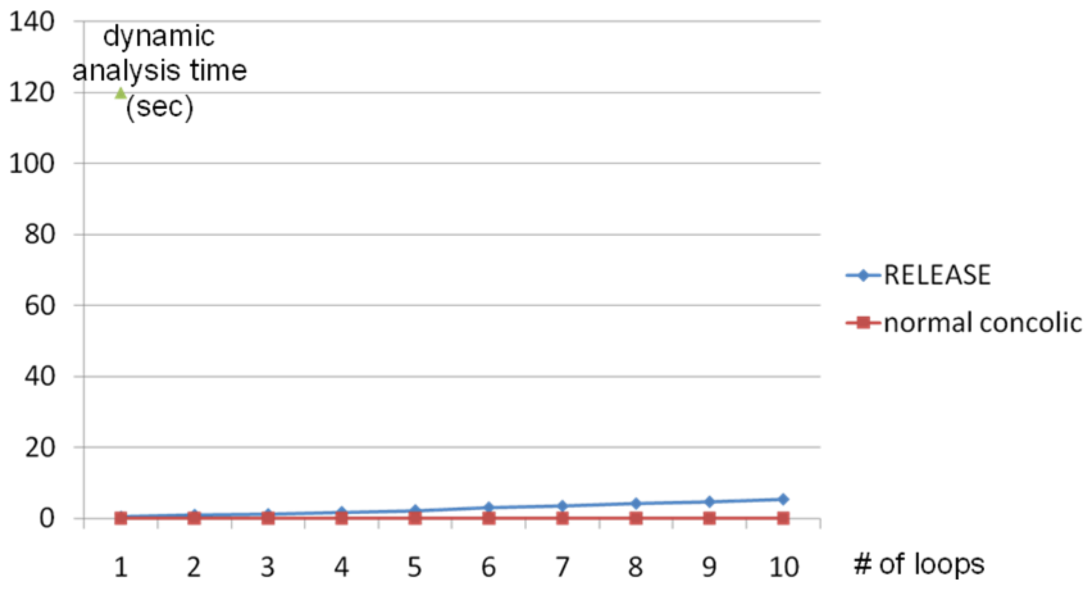


Figure 11: Evaluated result of dynamic analysis time consuming.

(Sequential loops and not input-related)

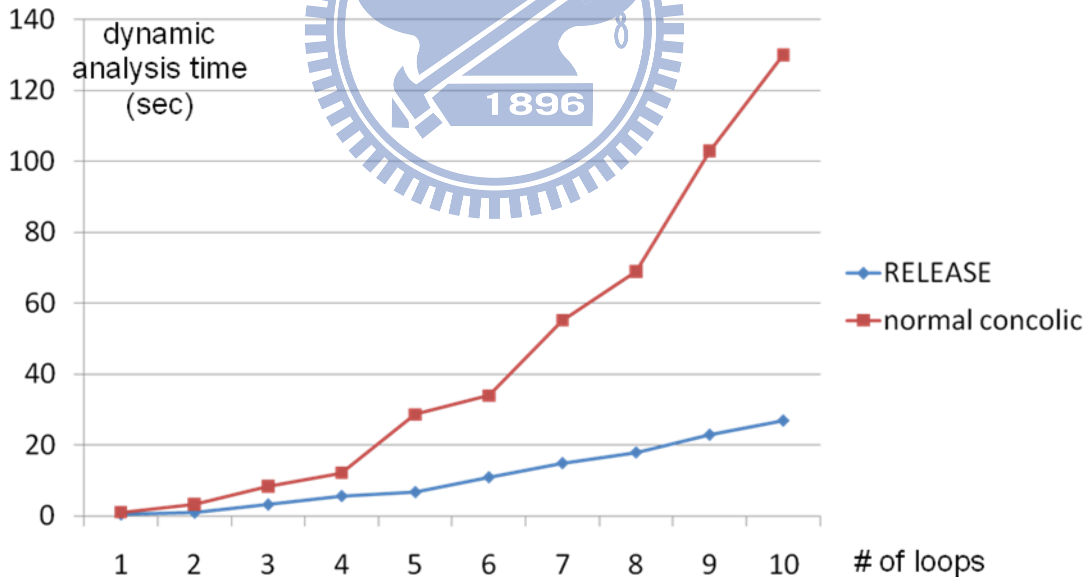


Figure 12: Evaluated result of dynamic analysis time consuming.

(Sequential loops and input-related)

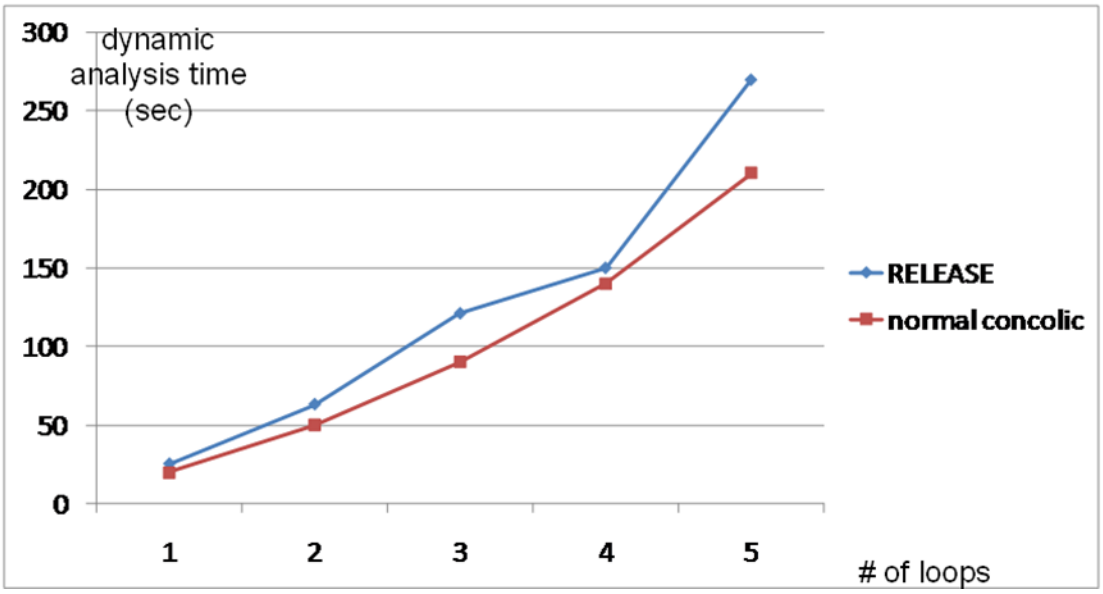


Figure 13: Evaluated result of dynamic analysis time consuming.

(Inner loops and not input-related)

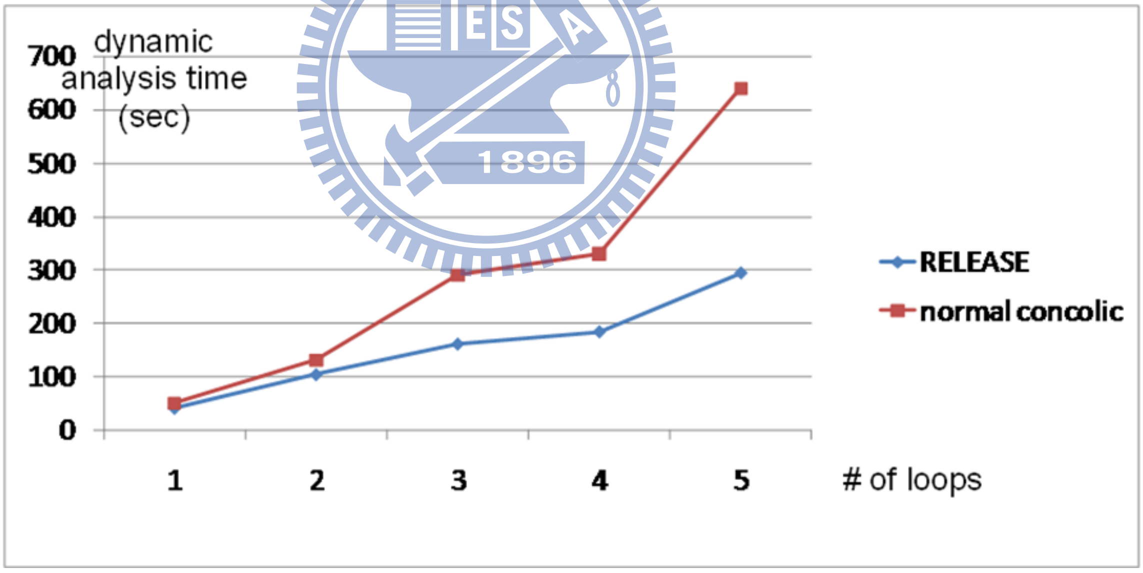


Figure 14: Evaluated result of dynamic analysis time consuming.

(Inner loops and input-related)

6. CONCLUSION

We propose loop-aware concolic execution, a new concolic execution approach with more information of loops, to enhance execution speed by eliminating unnecessary testing. Our approach applies on binary executables in x86 platforms with ELF binary format and does not need source code; this restriction-relaxed approach is important to analyze any kind of applications, include important security applications. We apply loop-aware concolic execution to an important security challenge, said buffer overflow vulnerabilities. Besides generating exploits, our approach can also point out which part of code is vulnerable; it helps software developers patch their programs quickly. These results makes loop-aware concolic execution is much better than other normal concolic execution.

7. REFERENCES

[1] B. P. Miller, L. Fredriksen, and B. So, "An Empirical-Study of the Reliability of Unix Utilities," *Communications of the Acm,* vol. 33, no. 12, pp. 32-43, Dec, 1990.

[2] D. A. Molnar, and D. Wagner, *SmartFuzz and MetaFuzz,* Technique Report, EECS Department, University of California, Berkeley, 2009.

[3] M. Eddington. "Peach Fuzzing," http://peachfuzzer.com/.

[4] J. C. King, "Symbolic Execution and Program Testing," *Communications of the Acm*, vol. 19, no. 7, pp. 385-394, 1976.

[5] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-13), Lisbon, Portugal, 2005, pp. 263-272.

[6] A. Lanzi, L. Martignoni, M. Monga, and R. Paleari, "A Smart Fuzzer for x86 Executables," in Proceedings of the Third International Workshop on Software Engineering for Secure Systems, 2007, p. 7.

[7] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation, Tucson, Arizona, USA, 2008, pp. 206-215.

[8] T. Reps, G. Balakrishnan, J. Lim, and T. Teitelbaum, "A next-generation platform for analyzing executables," in Proceedings of the Third Asian Symposium on Programming Languages and Systems (APLAS'05), Tsukuba, Japan, 2005, pp. 212-229.

[9] C. Marco, F. Viktoria, B. Greg, and V. Giovanni, "Static Detection of Vulnerabilities in x86 Executables," in Proceedings of the 22nd Annual Computer Security

Applications Conference on Annual Computer Security Applications Conference (ACSAC '06), Shanghai, China, 2006, pp. 269-278.

[10] J. Newsome, B. Karp, and D. Song, “Polygraph: Automatically Generating Signatures for Polymorphic Worms,” in Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P '05), Oakland, California, USA, 2005, pp. 226-241.

[11] J. Newsome, and D. Song, “Dynamic Taint Analysis: Automatic Detection, Analysis, and Signature Generation of Exploit Attacks on Commodity Software,” in Network and Distributed Systems Security Symposium, 2005.

[12] P. Godefroid, N. Klarlund, and K. Sen, “DART: Directed automated random testing,” in Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI '05), Chicago, Illinois, USA, 2005, pp. 213-223.

[13] C. Cadar, and D. Engler, “Execution generated test cases: How to make systems code crash itself,” in Proceeding on the 12th International SPIN Workshop on Model Checking of Software, San Francisco, California, USA, 2005, pp. 2-23.

[14] R.-G. Xu, P. Godefroid, and R. Majumdar, “Testing for buffer overflows with length abstraction,” in Proceedings of the 2008 international symposium on Software testing and analysis (ISSTA '08), Seattle, Washington, USA, 2008, pp. 27-38.

[15] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, “EXE: Automatically Generating Inputs of Death,” in Proceedings of the 13th ACM conference on Computer and communications security (CCS '06), Alexandria, Virginia, USA, 2008, pp. 322-335.

[16] C. Cadar, D. Dunbar, and D. Engler, “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs,” in USENIX

Symposium on Operating Systems Design and Implementation (OSDI '08) San Diego, California, USA, 2008.

[17] D. A. Molnar, and D. Wagner, *Catchconv: Symbolic execution and run-time type inference for integer conversion errors,* Technique Report No. UCB/EECS-2007-23, EECS Department, University of California, Berkeley, 2007.

[18] P. Saxena, P. Poosankam, S. McCamant, and D. Song, *Loop-Extended Symbolic Execution on Binary Programs,* Technique Report No. UCB/EECS-2009-34, EECS Department, University of California, Berkeley, 2009.

[19] W. Landi, and B. G. Ryder, "Pointer-induced aliasing: a problem taxonomy," in Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '91), Orlando, Florida, USA, 1991.

[20] N. Nethercote, and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI '07), San Diego, California, USA, 2007, pp. 89-100.

[21] F. Bellard, "QEMU, a fast and portable dynamic translator," in Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference (ATEC'05), Anaheim, California, USA, 2005, p. 41.

[22] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI '05), Chicago, Illinois, USA, 2005, pp. 190-200.

[23] R. Bagnara, A. Zaccagnini, and T. Zolo, *The Automatic Solution of Recurrence Relations. I. Linear Recurrences of Finite Order with Constant Coefficients.* ,

Technique Report No. Quaderno 334 (2003), Department of Mathematics, University of Parma, Italy, 2003.