Poster: Dynamic Taint Analysis of Concurrent Program Based on Symbolic Execution

Yu Hao*, Xiaodong Zhang*, Zijiang Yang† and Ting Liu*,IEEEauthorrefmark2
*Ministry of Education Key Lab for Intelligent Networks and Network Security
Xi'an Jiaotong University, Xi'an, Shaanxi 710000, China
Email: {yhao,xdzhang}@sei.xjtu.edu.cn, tingliu@mail.xjtu.edu.cn
†Department of Computer Science, Western Michigan University, Kalamazoo, MI 49008, USA
Email: zijiang.yang@wmich.edu

***Abstract*—With the advent of multicore processors, there is a great need to write concurrency programs to take advantage of parallel computing resources. However, the undetermined execution of the concurrency programs poses a huge challenge to current dynamic malware analysis how to guarantee the program is secure under the same input. In our work, a dynamic taint analysis of concurrent program (DTAC) is proposed to systematically detect tainted instances on all possible executions under a given input, by introducing the symbolic execution to guide dynamic analysis. Symbolic analysis infers alternate interleavings of an executed trace and computes thread schedules that guide future executions. Dynamic analysis explores new execution traces that drive future symbolic analysis. Then, the analysis can identify whether there is abnormal behavior within these executions by tracing the data flow. A prototype is developed for multithreaded C programs, built upon LLVM, KLEE and Z3. The primary experiments show that our method can find all possible tainted instances in the demo case and can systematically analyze real concurrency programs in SPLASH2 and PARSEC.**

I. Introduction

Multithreaded programming is a key technique to unleash the full potential of present and future generations of parallel computing systems based on the use of multi-core processors. However, the intrinsic nondeterminism of parallel execution invalid most dynamic malware analysis techniques which monitor the execution of the program and identify whether there is malicious behavior in this execution. In this paper, we select dynamic taint analysis (DTA), one of most useful dynamic malware analysis technique, to study how it fails to deal with concurrency programs and how to upgrade.

Figure 1 shows the demo program P where the thread main() creates one thread foo(). There is two shared variables x and y (all are accessed by all threads). The input sensdata is sensitive information. We aim to detect whether there is the information leak in the demo program. It is equals to detect all variables are tainted by the input and whether they are leaked. Since there is only line 5 would leak the information in this demo program, we need to verify whether the variable y sent at line 5 is relevant to the input sensdata. Four possible executions of demo program are shown in Figure 2. The number 1 and 2 at the top respectively represent the threads main and foo. Within each node we give the line numbers. Besides, node 8 with T/F means the branch turns true/false. All the tainted nodes are marked red. For ease of understanding we illustrate the

```
0  void main(int sensdata){
1      y = 0;x = 0;
2      int data = sensdata;
3      create(foo, NULL);
4      x = data;
5      send(y);
6  }
```

```
7  void *foo(){
8      if (x != 0) {
9          y = x;
10     }
11 }
```

Fig. 1. A multithreaded program with shared variables x,y

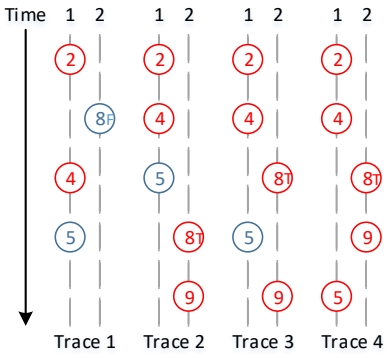

Fig. 2. Four valid executions of the program in Figure 1

example at the source code level and assume each statement is atomic, although our implementation is at the LLVM bytecode level.

The further analysis shows that: 1) in Trace 1, y is irrelevant to x since line 9 would not be executed; 2) in Trace 2 and 3, y sent at line 5 is irrelevant to x and the input sensdata, since line 9 is executed after line 5; 3) sensdata would be sent out at line 4 only in Trace 4. When we analyze the demo program with DTA, the malicious behavior would only be detected if the program is executed as Trace 4; otherwise, we will miss this malware.

Figure 3 depicts the insufficiency of applying DTA on multithreaded programs. By monitoring a particular execution under a given input, existing DTA techniques have two severe shortcomings. When a variable x is declared not tainted, a future execution under the same input but with different thread schedule may contradict the current result. When a variable x is declared tainted, although it is true, propagation evidence for such tainting cannot be easily reproduced. Unlike

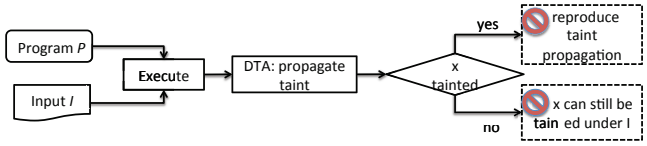

Fig. 3. Existing DTA techniques fail in concurrency programs

in sequential programs where a re-execution under the same input is sufficient, deterministically replaying an execution is a major challenge for multithreaded programs[1].

In order to find all tainted variables, one intuitive solution is to examine all possible execution traces. However, there are two major challenges: 1) the user has no control over the scheduling of threads in most current environments and few DTA technique supports thread scheduling; and 2) it is infeasible to explicitly check all thread interleavings. The number of possible interleavings of a multithreaded program with n threads each executing at most k steps can be as large as $(nk)!/(k!)^n \geq (n!)^k$, a complexity that is exponential in both n and k[2].

In this paper, we proposed a dynamic taint analysis of concurrent program (DTAC) to systematically detect tainted instances on all possible executions under a given input, by introducing the symbolic execution to guide dynamic analysis. We also develop the prototype of DTAC for multithreaded C programs, which is built upon LLVM, KLEE and Z3.

II. Dynamic Taint Analysis of Concurrent Program

In order to address the challenges of taint analysis for multithreaded programs, we develop a synergistic taint analysis framework as depicted in Figure 4, which has three components to analyze and guide program executions instead of just observing an execution as in existing DTA techniques. The new components include a predicative symbolic predicative analysis component, an alternate branch search component, and a guided execution component (for dynamic analysis). The entire framework forms a synergistic loop, where the symbolic analysis and the dynamic analysis feedback each other, to guarantee the systematic and complete traversal of all program behaviors for a given test input.

Specifically, in the symbolic analysis component, we capture valid partial order of a given execution trace using a quantifier-free first-order logic (FOL) formula and conduct a form of predictive analysis to infer new program behaviors. In the alternate branch search component, we detect the branch sequences that are not yet visited by the previous executions, and compute a thread scheduling that enables the execution with new branches. In the guided execution component, we enforce the newly computed thread scheduling in executing the program. Such execution, in turn, leads to new execution traces to be analyzed by the symbolic analysis. The loop terminates when no new distinctive program path is found.

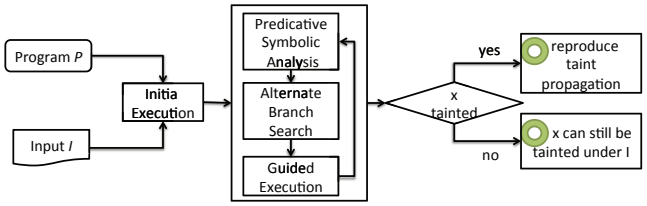

Fig. 4. Framework of DTAC)

TABLE I
Accuracy Comparison

Prog.	LOC	#Input	#DTA	#DTAC	Delta
pfscan	998	4	34	36	3
lunc	1182	3	16	50	34
luc	1401	3	154	179	25
fft	1482	3	582	631	49
radix	1547	2	71	76	5
swarm	2286	2	224	224	0
total	1483	3	180	199	19

III. Experiment and Discussion

We firstly analyze the dome program in figure 1. Four execution traces and all possible tainted variables are found as shown in Figure 2. The information leak in Trace 4 is also detected. Moreover, we can replay the execution to demonstrate how sensdata is transferred to y and sent out.

Then, an empirical study is conducted on 7 programs from two concurrency program benchmark SPLASH2 and PARSEC, in which the inputs have been added. As shown in Table I, LOC is lines of code, #Input is the number of the input, #DTA is the number of tainted variables detected with DTA, #DTAC is the number with DTAC, and Delta is the number of the improvement of DTAC. In total, there are 116 new tainted variables have been detected in these programs with DTAC, about 10.7

IV. Conclusion and Future Work

We present a dynamic taint analysis for concurrency programs by introducing symbolic execution into taint analysis. DTAC is able to offer a systematic and complete coverage of concurrency programs. We also show how to detect the information leaking in concurrency programs which are missed in current taint analysis techniques, with DTAC.

References

[1] X. Zhang, Z. Yang, Q. Zheng, P. Liu, J. Chang, Y. Hao, and T. Liu, "Automated testing of definition-use data flow for multithreaded programs."

[2] Z. Tian, T. Liu, Q. ZHENG, E. Zhuang, M. Fan, and Z. Yang, "Reviving sequential program birthmarking for multithreaded software plagiarism detection," *IEEE Transactions on Software Engineering*, 2017.