PAPER

Scaling Concolic Testing for the Environment-Intensive Program

Xue LEI[†a)], ***Member***, **Wei HUANG**[††], **Wenqing FAN**[††], ***and*** **Yixian YANG**[†], ***Nonmembers***

SUMMARY Dynamic analysis is frail and insufficient to find hidden paths in environment-intensive program. By analyzing a broad spectrum of different concolic testing systems, we conclude that a number of them cannot handle programs that interact with the environment or require a complete working model. This paper addresses this problem by automatically identifying and modifying outputs of the data input interface function(DIIF). The approach is based on fine-grained taint analysis for detecting and updating the data that interacts with the environment to generate a new set of inputs to execute hidden paths. Moreover, we developed a prototype and conducted extensive experiments using a set of complex and environmentally intensive programs. Finally, the result demonstrates that our approach could identify the DIIF precisely and discover hidden path obviously.

key words: *concolic testing, symbolic execution, taint analysis, environment-intensive*

1. Introduction

Symbolic execution is one of the common and important techniques in program static analysis. The key idea behind symbolic execution [1]–[3] is to emulate running the program with symbolic rather than concrete inputs and to represent the values of program variables as symbolic expressions over the symbolic inputs. As a result, all the execution paths of a program can be represented by these symbolic expressions. If these symbolic expressions can be solved by a constraint solver to generate inputs, these inputs will take the same path as the symbolic execution and terminate in the same way. In practice, there are two obvious problems with such symbolic execution: first, some system functions and encoding functions [4] cannot be represented as constraint expressions, and second, loops and recursion may result in an infinite number of paths if the termination condition for the loop or recursion is symbolic. Therefore, researchers proposed combining concrete and symbolic execution [5], which is the key elements of dynamic symbolic execution.

Dynamic symbolic execution has addressed the above problems with such a combination which is to generate inputs to explore all feasible execution paths. Consequently, dynamic symbolic execution gathered a lot of attention in recent years as an effective technique for generating high coverage test suites and finding deep errors in complex software applications [6]. Concolic testing [7] is one of the most representative instances of dynamic symbolic execution, where concolic stands for cooperative concrete and symbolic execution. The essence of concolic testing is to execute the program with an initial random or a given input, gather symbolic constraints on inputs at conditional statements along the execution [8], [9], negate the last condition, and then use a constraint solver to infer variants of the alternative path. Finally, concolic testing could explore all the paths of a target program in theory except the following circumstance. Assuming that the outputs of the function *localtime,10* decide the following branch condition, if we do not modify the system time manually in the runtime, maybe the alternative path will be omitted. In this paper, we use the term "data input interface function (DIIF)" to refer to system functions including at least one parameter property labeled "out". Usually, applications interact with the environment (e.g. the operating system, the user) by the DIIF in many ways: reading and writing files, checking file metadata such as file permissions and size, reading command-line arguments or environment variables, sending and receiving packets over the network, and so on.

In this paper, we propose a systematic approach to address this research problem. In particular, given a common binary, we aim to detect DIIF and modify their outputs automatically for exploring hidden paths. Our approach works by identifying all the outputs made by the DIIF as taint variables, keeping track of these taint variables flowing across the whole system, and symbolically computing values for program variables in form of symbolic logical formulas. Finally, with a constraint solver, the symbolic logic formula can be solved to find an input that would reverse a branch condition from true to false or vice-versa. To verify this idea, we prototyped our approach and evaluated it on the benchmark suite SGLIB [11]. In the experiment, our approach could execute those distinct paths and improve code coverage by identifying the DIIF precisely and specifying their output to corresponding values.

In summary, this paper makes the following contributions:

1. We propose a fine-grained analysis as a unified approach to detect and modify the outputs of the DIIF based on the binary instrumentation in the user mode.
2. We give an algorithm for identifying the outputs of the DIIF as taintdata automatically based on the DIIF array

Manuscript received February 4, 2015.
Manuscript revised May 18, 2015.
Manuscript publicized June 30, 2015.

[†]The authors are with Beijing University of Posts and Telecommunications, Beijing, China.
[††]The authors are with Communication University of China, Beijing, China.
a) E-mail: leixue@bupt.edu.cn

DOI: 10.1587/transinf.2015EDP7037

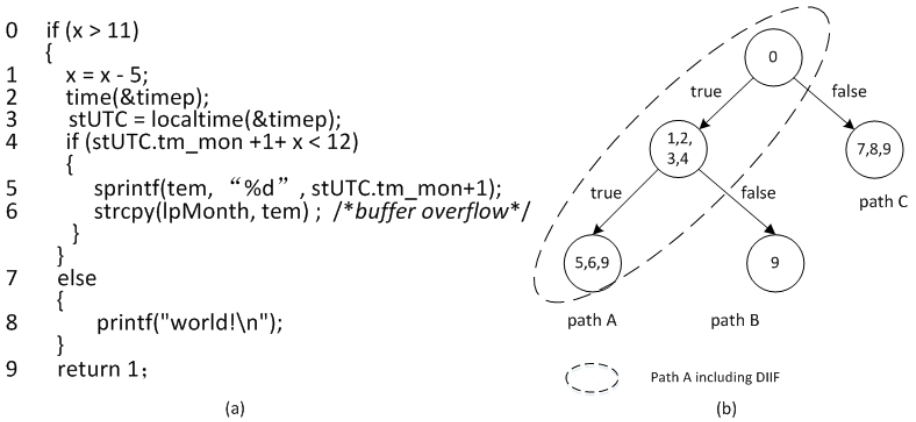

Fig. 1 A piece of pseudo code that calls a DIIF *localtime* is shown on the left (a). The right (b) illustrates a diagram of the path tree produced by symbolically executing the program shown on the left (a) from special inputs.

given only a stripped binary program.

3. We conducted our experiments with benchmark suite SGLIB, and demonstrated that our system is well suited for exploring code that must be accessed in a particular situation.

The paper is structured as follows. The next section gives an overview of our approach. Section 3 describes details on the design and implementation of our approach. Section 4 presents the experimental results. Section 5 provides some additional discussion of our approach. Section 6 surveys related work and Sect. 7 concludes the paper.

2. Problem Statement and Our Approach

In this section, we formalize the problem of detecting and identifying taintdata automatically, and give a brief overview of our approach.

2.1 Problem Statement

Using concolic execution to generalize over observed program behavior is a powerful technique because it combines the strengths of dynamic and static analysis. However, given a program including the DIIF, if not handling such situations can lead to unexpected and be hard to diagnose errors. Next, we show this situation with a specific example.

Background: data input interface function. C standard library provides a number of functions for programs to interact with the environment, including a set of file descriptors, time functions, system information and so on. All these functions have a common feature: there is at least one argument which value is decided by environment rather than computed with initial inputs by the program. That is the reason why we refer these functions as the DIIF. In other words, functions with the aforementioned feature result that some codes in the target application are almost never executed by the regression suite, which then leads to a big hurdle preventing security researchers and practitioners from analyzing the target program based on concolic testing.

The challenge of concolic testing with DIIFs. In order to explain the impact of DIIFs, consider the example in Fig. 1. Figure 1 (a) shows a piece of pseudo code that calls a DIIF *localtime* of linux system. In this example, there are three paths starting at line 0. When path A is symbolically executed through the program, it covers the lines 0, 1, 2, 3, 4, 5, 6and 9; path B covers 0, 1, 2, 3, 4 and 9; and path C covers 0, 7, 8 and 9. Figure 1 (b) shows a diagram of the path tree produced by symbolically executing the program from special inputs. Concolic execution will generate some random inputs, say {x = 12} and execute the program both concretely and symbolically. Assuming that it is July now, when the program calls the DIIF *localtime* in line 3, the structure variable *stUTC* stores outputs after *localtime* is executed and the variable *stUTC.tm_mon* is assigned to six. The concrete execution will take the "else" branch at line 4 and the symbolic execution will generate the path constraint$(x>11)\wedge(x-5+stUTC.tm_mon+1 \geq 12)$ along the Path B. Concolic testing negates a conjunct in the path constraint and solves$(x>11)\wedge\neg(x-5+stUTC.tm_mon+1 \geq 12)$ to get the test input{x = 12,*stUTC.tm_mon* = 3}. Theoretically, this new inputs will force the program execution along a different execution path A which is framed by the dotted ellipse as illustrated in Fig. 1(b). Concolic testing repeats both concrete and symbolic execution on this new test inputs. However, it is noteworthy that during the actual execution, the output of the DIIF *localtime* is beyond our control and the variable *stUTC.tm_mon* is still six in a long time leading to the "if" branch at line 4 will not be taken. If we do not manually modify the value of the variable *stUTC.tm_mon* to three, codes in line 5 and line 6 will not be explored and a classic strcpy buffer overflow vulnerability in the Path A (line 6) will be ignored. Due to a set of DIIFs in linux system, it is a key challenge for concolic testing to achieve higher branch coverage.

2.2 Our Approach

In this section, we give an overview of our approach that identifying and modifying the output from the DIIF during

the runtime to avoid the program path missing. We first discuss the intuition behind it, outline the overall flow of our approach, and then explain how it applies to improve code coverage.

Intuition. The insight behind our approach is to address the DIIF problem, by identifying and modifying the output of DIIF during the program execution. For instance in the Fig. 1(b), if we make path A to be explored and if it is July which is the output of the DIIF localtime in a long time, the inputs must satisfy the path constraint $(x>11) \wedge (0 \leq stUTC.tm\_mon \leq 11, stUTC.tm\_mon \in N) \wedge (x + stUTC.tm\_mon<16)$, that is to say in the runtime, if the value of x is assigned to twelve, it needs us to dynamically modify the value of *stUTC.tm_mon* as three instead of six which is the output of the DIIF *localtime*. Hence, the DIIF is one of the key bottlenecks in concolic testing for those environment-intensive applications. We hope our work will spur discussions on the implications and applications of concolic testing. In outline, our approach proceeds as follows.

DIIF detection and modification. Based on this intuition, we propose fine-gained reverse analysis. Firstly, we summarize all the DIIF from C standard library and extract all the DIIF from the procedure linkage table(PLT) [12] for each target application. We construct the DIIF array, a list structure, to store the information for each DIIF. Secondly, our approach proposes linear-sweep instructions from the register EIP, which contains a 32-bit pointer to the next instruction to be executed [16], in order to detect a DIIF. If the callee is a DIIF, we further decide if its outputs need to be modified or not. As in Fig. 1(a), we first check the EIP while line1 is executed finding that the next instruction (as in line2) is a function call, and then we search the function name (*Time*) in the aforementioned DIIF array and get the information about its output of which the data type is "long". We mark the variable *timep* as an input and decide if we need to reassign the variable *timep* to the new value recorded in the DIIF array. After line 2 is executed and we process the DIIF *locatime* in line 3 in the same way. Finally, we record into a trace the details about how the inputs are propagated in the system, reverse the last branch condition and get the new inputs by a constraint solver. Note that the new inputs consist of the normal inputs and the DIIF inputs, the former denotes the user-provided input values and the latter represents the data used instead of the outputs from the DIIF during the next execution. Once the new inputs are computed we need to update the DIIF array with the corresponding DIIF inputs.

3. System Design and Implementation

In this section, we describe key aspects of our approach: first system overview and some infrastructure details (Sect. 3.1), then techniques for categorizing C standard library into a white-list for DIIFs and preprocessing the program PLT to construct a DIIF array (Sect. 3.2), analyzing DIIFs during the runtime according to the DIIF array (Sect. 3.3), and updating the DIIF array according to the execution results (Sect. 3.4).

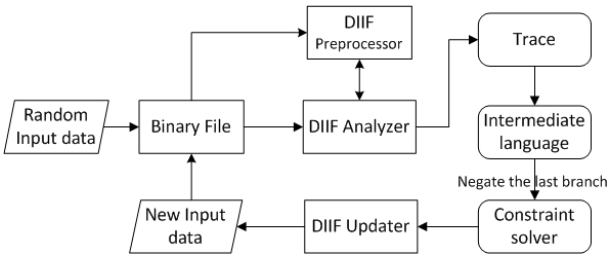

Fig. 2 System overview.

3.1 System Overview

An overview of the system architecture is shown in Fig. 2. Our system is based on the dynamic binary instrumentation tool Pin [13] which can facilitate instrumenting CPU instructions in a fine-grained manner. Therefore, we are able to instrument every CPU instruction and record a trace to perform program analysis on user space applications in Linux operating system with the x86 architecture. Besides, we use BAP [14] which is the successor of the binary analysis techniques developed for Vine [15] to analyze the trace file. In a nutshell, we implement our approach on top of Pin and BAP, the former is to record a trace while the latter is to process it.

Within the whole system, we build three components: DIIF preprocessor, DIIF analyzer, and DIIF updater. The DIIF preprocessor is premise and foundation to detect the DIIF in the runtime. It aims to summarize all the DIIFs of C standard library from their header files into a white-list. Before dynamic analysis of the target program, we extract DIIFs from PLT in the light of the white-list and create a DIIF array to record information about DIIFs of the program. To address the challenges posed by the presence of the DIIF, we need to find the function call of a DIIF by reasoning about the next instruction according to the states of CPU registers at the operating-system level. More precisely, DIIF analyzer firstly checks the instruction point EIP to identify a function call, and then compares the callee with the DIIF array to identify a DIIF and finally, marks the outputs of the DIIF as input variables. Furthermore, to make the application to explore the special path DIIF analyzer needs to replace the output of the DIIF as a particular value recorded by the DIIF array. Through analyzing the trace which is used to keeping track of path conditions and representing it with an intermediate language, we negate the last branch and get a set of new inputs by a constraint solver. Since a part of the new inputs are the DIIF inputs that are used instead of the outputs of the DIIF during the next execution, DIIF updater is responsible for maintenance and renewal the DIIF array with the corresponding DIIF inputs.

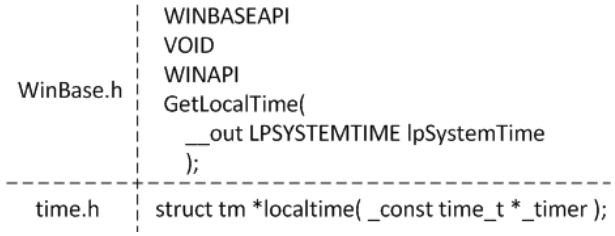

Fig. 3 An example of the definition of a DIIF.

3.2 DIIF Preprocessor

The DIIF preprocessor performs preparation for the subsequent dynamic analysis. Briefly speaking, the DIIF preprocessor consists of the following two steps: (1) compile all the DIIFs from C standard library into a white-list; (2) by comparison with the white-list extracts all the DIIFs from the PLT of the target program and record them into a list structure: the DIIF array. As a core part of the DIIF preprocessor, the DIIF array is a single list structure that stores information about all the DIIFs called by a program. Before dynamic analysis of an application, the first step is to extract DIIFs in the PLT of the application by comparison with the white-list.

The white-list for DIIFs. There is a common feature in the definition of Windows APIs: all the parameters of a function are labeled the input/output properties, as illustrated in Fig. 3. For instance there is only one parameter *lpSystemTime* of the function *GetLocalTime* and the property of this parameter is "out" shown on the left of the parameter type *LPSYSTEMTIME*. We define a DIIF as the function with at least a parameter labeled "out". In terms of the characteristic of the DIIF and with the aid of regular expression [17], we are able to category the DIIF from Windows APIs effectively. Unfortunately, there are no obvious characteristics for us to category DIIFs from the C standard library, such as the definition of *localtime* in Fig. 3. Since the C library function calls the windows APIs essentially, we extract DIIFs from the C standard library in the reference of DIIFs from the windows APIs manually in practice. However, this method is not perfect because some DIIFs form C library function in the linux OS does not correspond to the windows APIs. Therefore, this work needs more manual and engineering effort.

The DIIF array. The PLT is part of the executable text section, consisting of a set of entries. Each PLT entry is a short chunk of executable code for each external function that the shared library calls. Instead of calling the function directly, the code calls an entry in the PLT, which then takes care to call the actual function. When a function is called, the dynamic loader resolves its address and updates it into the global offset table (GOL). In a nutshell, the address for each external function is recorded in the PLT during the runtime. We are able to use the following command to get the information of the PLT and exact the address for each DIIF: *objdump -d -j .plt test*. As is shown in Fig. 4, it is a snippet of the PLT and we can see the address of the function *localtime* is 0x08048388.

```
08048388 <localtime@plt>:
 8048388: ff 25 38 97 04 08     jmp   *0x8049738
 804838e: 68 10 00 00 00        push  $0x10
 8048393: e9 c0 ff ff ff        jmp   8048358 <_init+0x30>
```

Fig. 4 PLT snippet of the previous example in the Fig.1 (b).

By comparison with the white-list for DIIFs aforementioned, we are able to choose and store the address for each DIIF in a single list for efficient lookup. Whenever a DIIF is retrieved, the information about the DIIF is written into a specially defined data structure as a new node of the DIIF array. In details, the purpose of the DIIF array is two-fold: (1) record the address, name, parameters labeled "out" for the DIIF; (2) indicate the DIIF whether its outputs need to be replaced and the replacement values. Similarly, the objects of the DIIF array are divided into two groups: one is the basic information for the DIIF, and the other is for the parameters labeled "out". The former consists of the address and name of the DIIF, while the latter consists of the number, the type size, the flag indicating a change, and the replacement value for each parameter labeled "out". In conclusion, the DIIF array serves as references for the DIIF analyzer mentioned in the Sect. 3.3 to tell us whether a function call belongs to the DIIF array. Then, for a DIIF located in that module, we need to mark its output accordingly. Furthermore, we determine if the outputs need to be modified in the light of the content recorded in the DIIF array.

3.3 DIIF Analyzer

The DIIF analyzer works by checking if the next instruction is a call instruction, identifying if the callee belongs to the DIIF array, and if so, marking the outputs of the callee as taintdata and modifying values of the outputs further if necessary. More precisely, we can get the machine code recorded in the instruction pointer register EIP(M_{EIP}) and decide whether it is a call instruction. If so, we compute the address of the callee, according to which we identify a DIIF and perform subsequent operation.

Identify the DIIF. The first step is to check each call instruction and detect a DIIF in the runtime. Generally, the DIIF is a system function from some dynamic link library such as libc.so and libutil.so in the linux/unix operating system. For the x86 platform, the machine code of the call instruction can be divided into three main groups [16] that are 1)E8:call near, relative, displacement relative to next instruction, 2)9A:call far, absolute, address given in operand, 3)FF: call near, absolute indirect, address given in r/m32. That is to say, for each machine code recorded in the instruction pointer register EIP, we only need to decide whether the opcode is one of the three and if so, we compute the address of the callee in further performance as the following formula:

$$I_{EIP} + M_{EIP} + len(EIP) = I_{DIIF} - >add \quad (1)$$

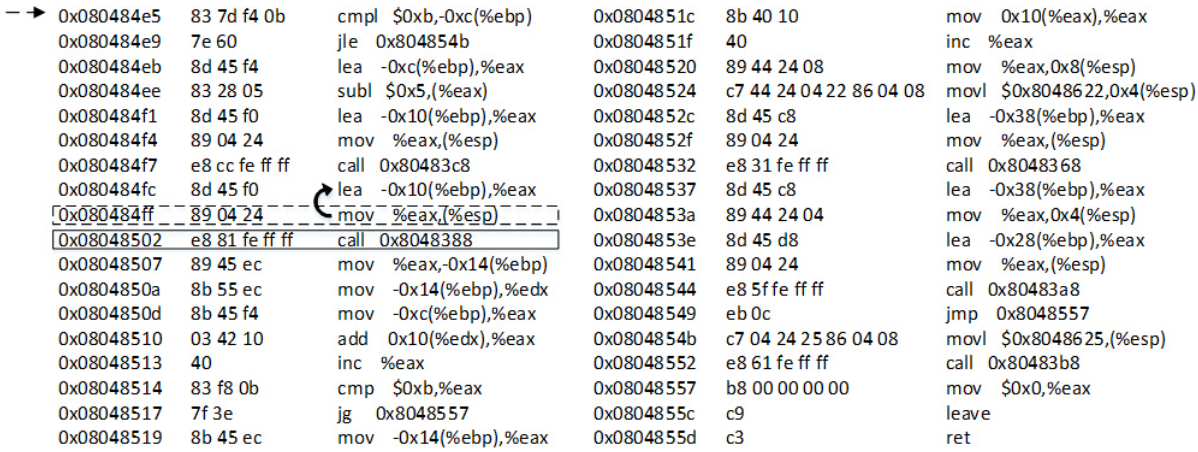

Fig. 5 The assembly code corresponding to path A of the previous example in the Fig.1 (b). The dotted arrow on the upper-left corner represents the beginning of the example.

Formula 1.The computation for the address of the DIIF. 1)I_{EIP}: the address recorded in *EIP*, 2)M_{EIP}: the machine code (offset) recorded in EIP, 3)*len(EIP)*:the length of the instruction, 4)$I_{DIIF} - >add$: the address of the DIIF.

For instance, in Fig. 5, the address recorded in EIP I_{EIP} is 0x08048502, the machine code in the EIP is "*e8 81 fe ff ff*", with off-the-shelf tools translating machine code to assembly code, it is easy to know that the M_{EIP} is "*ff ff fe 81*" and the length of instruction *len(EIP)* is five byte. Accord to the formula 1, the address of the DIIF $I_{DIIF} - >add$ is 0x8048388(0x08048502 + 5 + 0xfffffe81) as is shown in the Fig. 5.

Mark the output of the DIIF. Once a DIIF is identified, we are able to get the essential information about the output parameter of the DIIF, and then the question is how to mark the output of the DIIF. The reason why we focus on the output parameter of the DIIF is due to the definition of the constraint solver STP [18]: consists of a series of word-level transformations and optimizations that eventually convert the original problem to a conjunctive-normal form (CNF) formula for input to a high-speed solver for the satisfiability problem for propositional logic formulas (SAT) [19].In order to explain the process of marking the output parameters, consider the assembly code in Fig. 5 above, corresponding to path A of the previous example in the Fig. 1(b).

During the execution, if we observe that the instruction pointer (*i.e.*, EIP in x86 CPUs) is loaded with a call instruction (the machine code is *E8*) implied in a black solid bordered rectangle, and identify the callee is a DIIF *localtime* in the DIIF array, then we get the information about the parameters labeled "out". There is only one output parameter and its number ($I_{DIIF} - >No$) is one. Since pushing arguments onto the stack is in right-to-left order for C language, we are able to reversely deduce the push instruction for this output parameter, as implied by the black dotted bordered rectangle, and to obtain the address of the parameter (A_{No}) on the stack further, as implied by the curved arrow. Then, we set the memory region of which the address is A_{No} and size is $A_{No} + I_{DIIF} - >size$ as taintdata. We perform this backward search recursively until all the output parameters are obtained and marked.

Modify the output of the DIIF. Finally, the question is how to modify the outputs of the DIIF, if necessary. Furthermore, after marking an output parameter and in order to explore a new path, we decide whether its outputs need to be modified in the light of the variable $I_{DIIF} - >flag$. If so, when the instruction call is executed, we modify the memory region of the output parameter with the replacement value $I_{DIIF} - >replacement$. Note that if the output parameter is a structure type, we need to record information about each field of the structure in the DIIF array and if necessary, modify the value of each field one by one. A pseudocode description of this algorithm showing how the DIIF analyzer is performed to deal with the outputs of the DIIF is in algorithm1.

Algorithm 1: Our DIIF analyzer algorithm

input: M_{EIP}: machine code recorded in the register EIP

```
1  I_op ← HextoAsm(M_EIP);
2  if I_op ∧ I_call then                    //match the opcode FF
3  |  I_DIIF ← Look-table(I_op, T_DIIF);
4  |  if I_DIIF ≠ Ø then
5  |  |  while(I_DIIF->No ≠ end) do
6  |  |  |  TaintData ← A_No + I_DIIF->size;
7  |  |  |  MarkOutput(TaintData);
8  |  Execue(I_op);
9  |  if I_DIIF ≠ Ø ∧ I_DIIF->flag == 1 then
10 |  |  ModifyOutput(A_No, I_DIIF-> replacement);
11 else
12 |  Execue(I_op);
```

3.4 DIIF Updater

The DIIF updater is responsible for maintenance and renewal of the DIIF array at the end of each execution. During the execution, for each instruction that creates or propagates the marked data, we write a record into a trace with the dynamic binary instrumentation tool Pin. Meanwhile, BAP provides users with a functionality to perform analysis on the execution trace, lifts the assembly code in the execution trace to BAP intermediate representation, and then generates the weakest precondition from the intermediate representation. It is important that BAP interacts with STP, a SMT solver, to solve the satisfiability of the weakest precondition formula. More precisely, in order to generate a new input for a new path, we negate the last condition of the path constraint in the BAP IR file which is then translated to the weakest precondition formula. Finally, we use STP to solve this formula and obtain the inputs that lead the target program into a directed path, if any exists. The inputs could generally be divided into two categories: the normal inputs and the DIIF inputs, the former is used as the new input data to the next execution and the latter is recorded in corresponding node of the DIIF array. In summary, at the end of each execution, we update the DIIF array containing the flag ($I_{DIIF} -> flag$) and the replacement ($I_{DIIF} -> replacement$) of the concrete DIIF accessed in the last execution. When the next execution is performed, the aforementioned two variables are the decisive factors to lead the target program to explore a new path.

4. Evaluation

The following sections present our experimental results of our approach, by evaluating it with a benchmark suits. We first give a summary of the experimental results, describe the environment in which we conducted our experiments. Then, we present details of the experimental results on the benchmark suit.

4.1 Environment Setup

Our approach is written in a mixture of C and Python and consists of three major components: DIIF preprocessor, DIIF analyzer, and DIIF updater. The DIIF preprocessor is written in Python while the other two components are written in C. We choose PIN as our dynamic binary instrumentation tool, and add about 5500 lines of code to implement our algorithm. Moreover, we choose BAP as our static analysis tool for handling the trace and used STP for constraint solving. We evaluate our algorithms on a 4GHz Intel(R) Core 5 Duo Linux workstation with 4GB of RAM running Ubuntu 12.04. We measure the effectiveness of our approach using a set of ten samples included in the SGLIB, which is usually used by researchers to evaluate their tools [20]–[22]. The benchmarks are open-source and we use the source code to verify our results, but the system does not use the source code or source-level information such as debugging symbols. Since samples in the SGLIB rarely involve DIIFs, we add a couple of DIIF for each benchmark to verify the correctness of each DIIF and our DIIF preprocessor. In addition, a number of DIIFs in a single binary file can highlight the problem of hidden paths and validate rationality and feasibility of our algorithm.

We have analyzed about 700 header files (while there are 1037 header files in total of C standard library in linux) and founded 103 DIIFs. To test the performance, stability, and reliability of our algorithm, we choose typical DIIFs with different data types of parameters labeled "out", including reading and writing files, checking file metadata such as file permissions and size, reading command-line arguments or environment variables, sending and receiving packets over the network, and so on. We add DIIFs in each benchmark of SGLIB by three forms below.

```
form1:                  form2:                  form3:
if(DIIF_1)              if(DIIF_2)              ....
{                       {                       DIIF_3;
  printf("DIIF_1\n");     printf("DIIF_2\n");   ....
  exit (0);             }
}
```

For example, when we add the form1 at the beginning of the program, there is only one path increased. However, when we only add the form2 at the beginning of the program, the number of paths is double. If we add the two forms by the order DIIF_1 and DIIF_2, the number of paths is *(2N+1)*, *N* is the path number of the original program. By the reverse order, the number of paths is *2(N+1)*. In the real-world applications, the DIIF may not control a branch and we use form3 to verify the correctness of our DIIF analyzer. Obviously, the number of new branches caused by DIIFs is related to the form and insertion point in the context of the benchmark. In the paper, we chose a number of DIIFs added randomly in the context of each benchmark to be close to the real-world programs. In Table 1, we summarize the basic information of these benchmarks before and after modification, including basic block and path number, which are

Table 1 The basic information of each benchmark before and after modification.

Beachmarks	Before adding DIIFs		After adding DIIFs		
	BBL	path	DIIF	BBL	path
dllist	77	850	5	82	882
listsort	48	216	7	55	701
list insert sort	20	562	3	23	1125
list insert sort1	29	469	10	39	727
array bin search	26	471	6	32	535
array sort	41	569	11	62	1974
array sort1	34	793	18	52	1742
hash	48	521	9	57	651
rbtree	110	804	12	122	939
queue	34	609	20	54	2650

easy to get with the off-the-shelf static analysis tools such as IDApro. The fourth column lists the DIIF number added for each benchmark.

4.2 Evaluation on a Benchmark Suite

Because of differences in input assumptions and search heuristics, it can be difficult to fairly compare concolic execution tools on an end-to-end basis. Therefore, we ran modified benchmarks to illustrate the improvement of our algorithm over the baseline concolic execution approach. More specially, this section measures how our algorithm performs compared to the baseline approach on three metrics: (1) how much more instruction coverage it gets, (2) how much time consumes it reaches the maximum instruction coverage, and (3) how much more paths it explores. The results of the evaluation are summarized in Table 2 and described in more detail in the remainder of the section.

In Table 2, we show individual results from running our algorithm and the baseline approach on the ten benchmarks. Each column consists three sub-columns: the number of paths explored, the time (T) to reach the maximum coverage and the maximum coverage (*cov*).

In our experiment, we randomly added a certain amount of DIIFs for each benchmark and manually made the outputs from the DIIF as the key issue for an alternative branch. As illustrated in Fig. 6, the execution paths with our algorithm are shown with solid lines, whereas dashed lines show the execution paths with the base line approach. By contrast with the base line approach, there are more execution paths varying from 12 to 81. The reason why the execution path increases is that in the runtime, we modify the outputs of the DIIF and change the executive flow to explore the hidden paths. In the real-world applications, the DIIF may not control a branch, that is to say, there may be no branch following the DIIF. Hence, the number of the DIIF is not absolutely relative with the increase of the execution path.

As is shown in Fig. 7, the average coverage increase is 4.78% and it shows the coverage increases for each benchmark. The maximum increase is up to 11.4% while the number of DIIFs of the benchmark is not the most one. There are three reasons why the coverage cannot achieve 100%. The first one is from the limitation of the STP which will be discussed in Sect. 5. The second one is that we also neglected the property of the return values of some functions, and although the return value is the key factor deciding the following branch such as the exception handler. At last, there are some dead code. Nevertheless, our algorithm identified all the DIIFs precisely and reached higher coverage.

In order to reach higher coverage, it is inevitable to sacrifice certain time. The time can be broken into two parts: the runtime for instrumentation for each sample in the user-mode, and the runtime for new inputs generation. The former is the key factor to decide the time difference between our algorithm and the baseline approach. The time to reach the maximum coverage of our approach varies from 69.3 seconds to 206.5 seconds, as is shown in in Fig. 8, depending on the number of BBL, the number of DIIFs calling and other factors. Due to the binary instrumentation tool Pin working in the user mode, the efficiency of our algorithm is obviously improved comparasion with the whole system emulator TEMU. We calculated time consumption as fol-

Table 2 Individual results from running our algorithm and the baseline approach on the ten samples.

Beachmarks with DIIFs	The baseline approach			Our algorithm		
	path	*T(sec)*	*cov(%)*	*path*	*T(sec)*	*cov(%)*
dllist	79	185	82	101	201	86.3
listsort	164	48.1	86.5	194	69.3	90.1
list insert sort	150	125.2	92	162	129.6	94.2
list insert sort1	166	105.6	72.1	247	126.2	83.5
array bin search	166	83.9	89.6	201	96.8	91.3
array sort	142	122.8	91.2	207	155	94.2
array sort1	109	165.3	85.3	168	206.5	92
hash	148	105	76.5	186	113	81.2
rbtree	94	169	82.7	149	194	88.5
queue	122	137	92.5	186	191.2	96.9

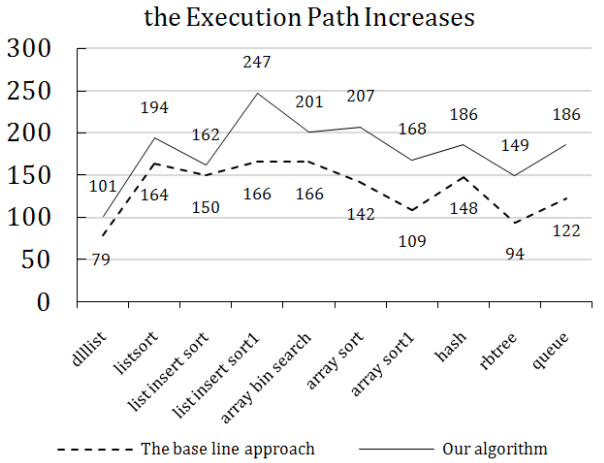

Fig. 6 The number of execution path comparison between the baseline approach and our algorithm.

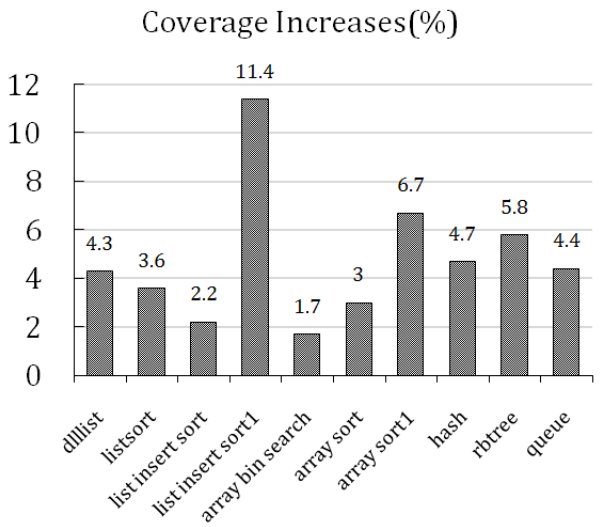

Fig. 7 Coverage comparison between the baselineapproach and our algorithm.

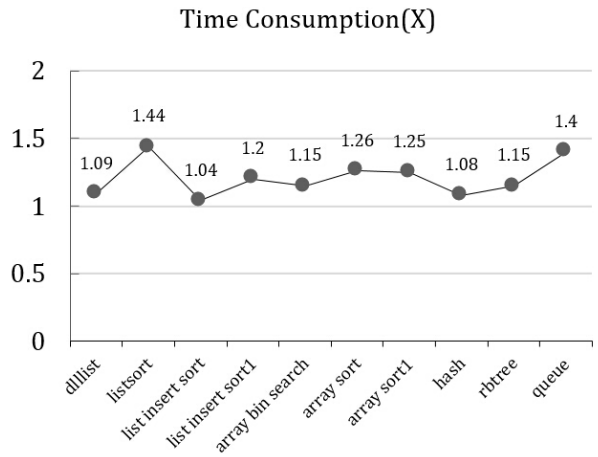

Fig. 8 Time consumption comparison between the baseline approach and our algorithm.

lows: Ran each program to reach the maximum coverage and recorded the time and calculated the time consumption as T_{our}/T_{base}. As can be seen from the results, our algorithm gives more time consumption on each sample, which is due to the instruction match in the DIIF analyzer detailed in Sect. 3.3.

In the experiment, we found that the hidden paths of a program should not only contain the feasible paths, but also include the infeasible paths by some unsound test executions as is shown below.

```
1 localtime(&tm);
2 if(tm ->tm_mon > 11 | tm ->tm_mon < 0)
  {/*the line 3 and 4 will not be explored
   because the unsound range of tm ->tm_mon.*/
3  printf("it is a vulnerability");
4  return -1;
  }
5 else
  {
   /*buffer overflow*/
6  strcpy(lpMonth, sprintf(tem,"%d",tm->tm_mon+1));
  }
```

Usually, the function localtime(*&tm*) ensures the value of tm.mon will be between 0 and 11. Therefore, the path A(covers 1, 2, 3, 4) will not be explored during the actual execution. In our approach, if we taint the variable $tm->tm\_mon$, the constraint solver will generate the test input $\{tm->tm\_mon = 12\}$. During the next execution, we modify the variable $tm->tm\_mon$ as 12 and then path A will be accessed. Hence, our approach is also aid to explore infeasible paths and detect potential vulnerabilities.

5. Limitations and Future Work

Limitations. The first limitation is that the complete support of STP. It is well known that STP does not support floating point and pointers. The limitation of STP leads that sometimes it takes a long time without yielding a meaningful result and we have to discard a set of constraints if STP runs out of memory or exceeds a five-minute timeout or constraint solving. Therefore, it is indeed a bottleneck of our approach, as well as all the concolic testing. The second limitation is that the binary instrumentation tool Pin which is only working in the user mode. Unlike the whole-system symbolic executors such as S2E [23] or BitBlaze [24] that can execute both user and kernel code while our approach execute the user code. Therefore, our approach cannot process the system calls that are common in larger and more complicated programs. The main advantages of our approach are that less time spent and lower state restoration cost by avoiding analyzing kernel code. The third limitation, and the most obvious, is the summary of DIIFs. In this paper, we only regarded the function with at least a parameter property definitely "out" as the DIIF. Factually, a couple of functions with uncertain property of the parameters (short for DIIF_U), maybe the property is "out_opt", "inout" and "inout_opt". The uncertainty of the latter three property is truly a headache for the DIIF analysis and the level of engineering effort required is a major reason we decided not to go on with the precisely analysis in this paper.

Future work. Our approach is a step forward in increasing coverage of concolic testing, but obviously, much more works remains to be done. A more interesting future direction is to extend our approach to precisely detect the DIIF_U with data flow analysis. At a high level, it should be possible to monitor and modify the return value of the function since a lot of potential vulnerabilities exist in exception handling which is often ignored and not instantly accessible. Therefore, we would like to extend our detection strategy to cope with this potential bugs in our future work.

6. Related Work

In this section, we briefly describe some related work on other approaches to cope with data from the environment. Binary instrumentation is a technique to insert extra code into a binary that monitors the instrumented program's behavior. Pin is a dynamic binary instrumentation framework for the IA-32 and x86-64 instruction-set architectures that enables the creation of dynamic program analysis tools [13]. Many binary analysis platforms create their Pintools and construct concolic testing to explore vulnerabilities, such as BAP [14], MAYHEM [25], MergePoint [26] and so on. With Pin during the runtime we can calculates register liveness information to trace the executed path.

KLEE [27] leveraged several years of lessons from previous tool EXE [18] and it is capable of automatically generating tests that achieve high coverage on a diverse set of complex and environmentally-intensive programs. KLEE is implemented as a virtual machine for the Low Level Virtual Machine [28](LLVM) assembly language and it needs special compiling for each target program. Therefore, it is not suitable for binary file without source code. For those functions interact with the environment, KLEE summarizes

parts of the DIIF and rewrites them based on the standard C library. In practice, this improvement does not give large aggregate coverage increase, but is required to reach the last (tricky) bit of code in many applications with already high coverage.

Analogously, S2E which is a new symbolic execution platform for analyzing the properties and behavior of software systems and developed by École Polytechnique Federale de Lausanne (EPFL), implements consistency models on both kernel-mode and user-mode binaries to address the problem. In nature, the core of S2E is QEMU [29] which makes it scale even to large, proprietary, real-world software stacks on Mac OS X, Microsoft Windows, and Linux. The S2E proposes a complete working model SC-SE(Strictly Consistent System-level Execution) to implement instrumentation for the environment while ours can selectively implement instrumentation for DIIFs.

Our approach uses fine-grained taint analysis to locate data interacting with the environment, which differs from the two approaches above in essence. The fine-grained taint analysis is actually a dynamic taint analysis technique, which is proposed to solve and analyze many other security related problems. Many systems detect exploits by tracking the data from untrusted sources and other systems make use of this technique to analyze how sensitive information is processed by the system. For finding hidden paths of environment-intensive programs in user mode, our approach is more simple and efficient.

7. Conclusion

Since concolic testing is frail and insufficient to an environment-intensive program, in this paper we present a fine-grained analysis as a unified approach, identifying DIIFs automatically based on the DIIF array of an application, to explore hidden paths in environment-intensive program. Moreover, we developed a prototype to implement our algorithms and evaluated it with the benchmark suite SGLIB. The experimental results demonstrated that our system is well suited for exploring code that must be accessed in a particular situation.

Acknowledgments

We thank Jing An and Quanchen Zou for suggestions and help related to the experiments and previous papers. This work is supported by National Science Foundation of China (No.61121061) and Ministry of Education-China Mobile Research Foundation (No. MCM 20130411). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Natural Science Foundation of China or other supporters.

References

[1] L.A. Clarke, "A program testing system," Proc. 1976 annual conference, ACM New York, NY, USA, pp.488–491, 1976.
[2] J.C. King, "Symbolic execution and program testing," Commun. ACM, vol.19, no.7, pp.385–394, 1976.
[3] C. Cadar, P. Godefroid, S. Khurshid, C.S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," Proc. 33rd International Conference on Software Engineering, Honolulu, HI, pp.1066–1071, 2011.
[4] J. Caballero, P. Poosankam, S. McCamant, D.B. ć, and D. Song, "Input generation via decomposition and re-stitching: Finding bugs in malware." Proc. 17th ACM conference on Computer and communications security, Chicago, IL, USA, pp.413–425, 2010.
[5] E. Larson and T. Austin, "High coverage detection of input-related security faculties," Proc. 12th conference on USENIX Security Symposium, USENIX Association Berkeley, CA, USA, vol.12, p.9, 2003.
[6] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," Commun. ACM, vol.56, no.2, pp.82–90, 2013.
[7] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," Proc. 10th European Software Engineering Conference (ESEC) and ACM SIGSOFT Symposium on the Foundations of Software Engineering, Lisbon, Portugal, vol.30, no.5, pp.263–272, 2005.
[8] P. Pongsin, Scaling Concolic Execution of Binary Programs for Security Applications, Ph.D. Thesis, Carnegie Mellon University, Pittsburgh, USA, 2013.
[9] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM New York, NY, USA, vol.40, no.6, pp.213–223, 2005.
[10] "localtime," available at https://msdn.microsoft.com/en-us/library/bf12f0hc.aspx
[11] M. Vittek, P. Borovansky, P.-E. Moreau, "A Simple Generic Library for C, in Reuse of Off-the-Shelf Components," Proc. 9th International Conference on Software Reuse, Turin, Italy, pp.423–426, 2006.
[12] "Dynamic Linking and Loading," available at http://www.iecc.com/linker/linker10.html
[13] S. Berkowits, "PIN," available at https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool#UserManual, 2012-6-13.
[14] D. Brumley, I. Jager, et al., "The BAP handbook," available at http://bap.ece.cmu.edu/doc/bap.pdf, 2014-4-10.
[15] D. Brumley, et al., "Vine: The BitBlaze Static Analysis Component," available at http://bitblaze.cs.berkeley.edu/vine, 2008-12.
[16] Guide, Part, "Intel 64 and IA-32 Architectures Software Developer Manuals," available at http://download.intel.com/design/processor/manuals/253668.pdf, vol.2A, pp.387–390, 2011-5.
[17] "Regular Expression," available at http://msdn.microsoft.com/en-us/library/az24scfc(v=vs.110).aspx, 2014-7
[18] C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill, and D.R. Engler, "EXE: automatically generating inputs of death," Proc. 13th ACM Conference on Computer and Communications Security, ACM New York, NY, USA, pp.322–335, 2006.
[19] V. Ganesh and D.L. Dill, "A decision procedure for bit-vectors and arrays," Proc. 19th international conference on Computer aided verification, Springer-Verlag Berlin, Heidelberg, pp.519–531, 2007.
[20] K. Sen and G. Agha, "CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools," Proc. 18th international conference on Computer Aided Verification, Springer-Verlag Berlin Heidelberg, pp.419–423, 2006
[21] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M.A. Alipour, and D. Marinov, "Comparing non-adequate test suites using coverage criteria," Proc. 2013 International Symposium on Software Testing and Analysis, New York, NY, USA, pp.302–313, 2013.
[22] T. Chen, X.-S. Zhang, C. Zhu, X.-L. Ji, S.-Z. Guo, and Y. Wu, "Design and implementation of a dynamic symbolic execution tool for Windows executables," Journal of Software: Evolution and Process. vol.25, no.12, pp.1249–1271, 2013.

[23] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," Proc. 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ACM New York, NY, USA, vol.39, no.1, pp.265–278, 2011.
[24] D. Song, et al., "BitBlaze: Binary Analysis for Computer Security," available at http://bitblaze.cs.berkeley.edu/, 2008-12.
[25] S.K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," Proc. 2012 IEEE Symposium on Security and Privacy, IEEE Computer Society Washington, DC, USA, pp.380–394, 2012.
[26] T. Avgerinos, A. Rebert, S.K. Cha, and D. Brumley,"Enhancing symbolic execution with veritesting," Proc. 36th International Conference on Software Engineering, ACM New York, NY, USA, pp.1083–1094, 2014.
[27] C. Cadar, D. Dunbar, and D.R. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," Proc. 8th USENIX Conference on Operating Systems Design and Implementation, USENIX Association Berkeley, CA, USA, vol.8, pp.209–224, 2008.
[28] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," Proc. international symposium on Code generation and optimization: feedback-directed and runtime optimization, IEEE Computer Society Washington, DC, USA, pp.75, 2004.
[29] "QEMU," avaliable at http://www.qemu.org/, 2015-4-27.

Xue Lei was born in 1986, is currently a PhD candidate in Information Security Center, Beijing University of Posts and Telecommunications, China. She received her master degree from Beijing University of Posts and Telecommunications, China, in 2011. Her research interests include information security, program analysis.

Wei Huang was born in 1983, is currently an instructor in School of Computer Science, Communication University of China. He received his PhD degree from Beijing University of Posts and Telecommunications, China, in 2010. His research interests include information security.

Wenqing Fan was born in 1982, is currently an instructor in School of Computer Science, Communication University of China. He received his PhD degree from Beijing University of Posts and Telecommunications, China, in 2010. His research interests include information security, program analysis.

Yixian Yang was born in 1961, is currently a professor in Beijing University of Posts and Telecommunications, China. His research interests include information security, cryptography.