Testing C Programs for Vulnerability Using Trace-Based Symbolic Execution and Satisfiability Analysis

Dazhi Zhang, Donggang Liu, Wenhua Wang, Jeff Lei, David Kung, Christoph Csallner
Department of Computer Science and Engineering
The University of Texas at Arlington

Abstract

Security testing has gained significant attention recently due to the huge number of attacks against software systems. This paper presents a novel security testing method using *trace-based symbolic execution* and *satisfiability analysis*. It reuses test cases generated from traditional functional testing to produce *execution traces*. An execution trace is a sequence of program statements exercised by a test case. Each execution trace is symbolically executed to produce *program constraints* and *security constraints*. A program constraint is a constraint imposed by program logic on program variables. A security constraint is a condition on program variables that must be satisfied to ensure system security. A security vulnerability exists if there is an assignment of values to program variables that satisfies the program constraint but violates the security constraint. This assignment of values is used to generate test cases to uncover the security vulnerability. One novelty of this method is a test model that *unifies program constraints and security constraints* such that formal reasoning can be applied to detect vulnerabilities. Another novelty is *attribute-based* analysis that abstracts program variables and functions for effective and efficient symbolic execution. A tool named `SecTAC` has been implemented and applied to 14 benchmark programs and 3 moderate size open-source programs. The experiment shows that `SecTAC` quickly detects all reported vulnerabilities and 15 new ones that have not been detected previously. The merits of the proposed method are threefold. First, trace-based symbolic execution reduces the search space greatly as compared to conventional symbolic execution. Second, attribute-based analysis tracks more useful information about program variables and functions than previous methods, resulting in more effective detection of vulnerabilities. Third, it is efficient and effective as the experiment result indicates.

1 Introduction

Software security has gained significant attention in recent years due to the huge number of security attacks that exploit vulnerabilities in software. *Security testing* is becoming an active area of research, aiming at identifying software vulnerabilities effectively. Recently, many approaches have been proposed to detect vulnerabilities in programs [27, 15, 12, 6, 24, 1, 22, 7, 25, 5, 11, 13, 28].

Static analysis has been used to scan source code for errors that either crash a system or cause security problems [26, 16, 27]. Tools based on static analysis mainly check unsafe string functions and pointer operations. For example, function `strcpy(dst,src)` can cause a buffer overflow problem if the length of `src` is greater than the space allocated to `dst`. These static analysis tools use heuristics to determine if a security problem could occur; they usually approximate or even ignore runtime dynamics such as branch conditions and how buffer elements are visited. Thus, they are often imprecise, causing many false alarms.

Dynamic analysis examines program execution to detect security problems such as buffer overflow [14, 9, 10, 1, 22, 25]. These tools feed test data to a program and monitor its runtime behavior. A security vulnerability is detected if the behavior is considered abnormal, e.g., the program accesses a buffer outside its bounds. Although dynamic analysis tools can reduce false alarm rates, they require test inputs that actually cause security problems. This places a huge burden on testers. Dynamic analysis does not offer any help

in generating inputs that cause security problems. Our technique provides this help. We can generate new inputs that trigger security problems, even if the user-supplied inputs do not.

Dynamic symbolic execution has been used in many automatic test data generation tools for finding errors like assertion violations and bugs that crash a system or cause security problems [12, 24, 7, 5, 11, 13, 28]. These tools integrate static analysis and dynamic analysis; they do not need test inputs that can actually cause security problems. However, these tools try to exhaustively explore possible paths to find security vulnerabilities. Thus, they do not scale well to large and complex programs. Moreover, they are designed to detect some specific types of vulnerabilities such as buffer overflow. It is difficult, and sometimes impractical, to extend them for detection of new types of vulnerabilities.

In this paper, we propose a novel security testing approach using *trace-based symbolic execution* and *satisfiability analysis*. Trace-based symbolic execution avoids the search space explosion of conventional symbolic execution. In our approach, existing test cases are reused. Specifically, each test case is used to generate an execution trace, i.e., the sequence of exercised program statements. Symbolic execution is then applied to each execution trace to produce two predicates. The first predicate is called a *program constraint* (PC), which is updated when a statement of the trace is symbolically executed; it specifies a condition that program variables must satisfy after the execution of the statement. The second predicate is called a *security constraint* (SC), which is produced at certain points based on *security requirements* during the symbolic execution of the trace; it specifies the condition that program variables must satisfy to ensure the security of the given program execution. A security requirement is a restriction on program values used by operations that may cause security problems, such as buffer overflow caused by library function calls. A security vulnerability is detected if there is an assignment of values to program variables satisfies PC but violates SC, i.e., PC $\wedge\neg$ SC is satisfiable. We use the Yices [8] SMT solver to check the satisfiability of PC $\wedge\neg$ SC.

One novelty of our method lies in a test model that unifies program constraints and security constraints using logical expressions so that formal reasoning can be performed to detect security vulnerabilities. Hence, our approach can handle new types of vulnerabilities by simply formulating new security requirements for them. Another novelty is an attribute-based method for analyzing execution traces. Specifically, we treat program variables and functions as objects and introduce attributes in objects to effectively extract important information like buffer size, string length, and function return type. Compared with existing methods, this results in more precise and efficient symbolic execution. For example, previous methods do not track the relation between the strings in the same buffer, while our method does. Moreover, trace-based symbolic execution also makes it possible to test programs for vulnerability in parallel. This is because analyses on different execution traces are independent from each other. We can partition the test cases into a number of *disjoint* subsets and analyze these subsets in parallel. This cannot be directly achieved in dynamic symbolic execution based approaches since test cases that exercise different paths are generated online during path exploration. It is very difficult to split the testing task into "sub-tasks" at the beginning of testing. Certainly, tools like DART [12] and CUTE [24] can be modified to reuse existing test cases and only test the paths exercised in these test cases. However, in this case, they lose the benefit of automatically exploring program paths, and their symbolic execution on traces are not as effective as ours. Finally, `SecTAC` is open source and available at `http://ranger.uta.edu/~dliu/sectac`.

To evaluate the effectiveness of our approach, we implemented a tool named `SecTAC` (A Security Testing Approach for C programs) and applied it to 14 benchmark programs given in [19] and 3 open source programs. The benchmark programs were designed to evaluate buffer overflow detection tools by simulating historic real-world vulnerabilities in server programs. Compared with the results in [19, 29, 28], `SecTAC` can detect every reported vulnerability as long as the vulnerability exists in the execution traces tested in our experiments. In addition, `SecTAC` detected 6 previously unreported vulnerabilities in the 14 benchmark programs [1]. `SecTAC` also detected 9 vulnerabilities in the open-source programs that, to the best of our

[1]Two of them have been confirmed by the authors of [19] via personal communication. We are waiting for more responses.

knowledge, have not been reported previously.

The rest of this paper is organized as follows. In the next section, we explain our basic ideas. In Section 3, we overview the `SecTAC` design. In Section 4, we describe the `SecTAC` implementation. In Section 5, we present the experiment result. In Section 6, we review related work and compare `SecTAC` with existing methods. We discuss the limitations of `SecTAC` in Section 7 and draw some conclusions in Section 8.

2 Basic Ideas of Our Approach

Software systems must be tested to ensure that the required functionalities are correctly implemented. Unlike conventional software testing, our goal is to detect security vulnerabilities that exist in the software system. A program is said to be *vulnerable* if there is an execution path that can be exploited to compromise the security of the system. To detect such security vulnerability, we rely on a set of *security requirements* that must be satisfied by all execution paths of the program. An example of security requirements is that the length of the string copied to a buffer using function `strcpy` must not exceed the capacity of the buffer.

Testing for security vulnerabilities implies the generation of test cases that can effectively detect violations of security requirements. However, it is well known that effective test case generation is both difficult and time-consuming. Therefore, it is desirable to reuse the test cases that are already generated during conventional software testing. The merit of this is twofold. First, these test cases typically accomplish some required coverage criteria such as branch coverage. Second, the branches covered by the test cases are deemed important by the developer. Our goal is to provide a security testing method for software developers who have access to the source program and the test cases produced by traditional functional testing.

In our approach, we use existing test cases to generate execution traces. Each execution trace is a sequence of source code statements exercised by a test case. There are no loops in execution traces since a loop in the original program will be unfolded when it is exercised by a test case. We then symbolically execute each execution trace to determine whether it contains a security vulnerability. Symbolic execution of each trace produces two predicates. The first predicate is the *program constraint* (PC), which is updated during the symbolic execution of the trace; it specifies a condition that the program variables must satisfy. In other words, the program constraint specifies the possible values of variables at each point during the symbolic execution of the trace. The second predicate is the *security constraint* (SC), which is produced at certain points during the symbolic execution of the trace; it specifies a condition that program variables must satisfy to ensure the security of the software system. A security problem will occur when the values of some variables violate the security constraint. Testing C programs for vulnerabilities is therefore equivalent to determining whether at each point in the trace, there exists an assignment of values to program variables that satisfies PC but violates SC.

In the following, we will first explain the ideas of deriving program and security constraints at each point in the trace. We will then use a concrete example to demonstrate our approach.

Program constraints: The program constraint at a given point in the trace is determined by the program statements exercised to reach this point. These statements include declaration statements, assignment statements, branching statements, and library function calls; they impact the values of variables as follows:

- A declaration statement contains important information about the *type* and *size* of the declared program variable. These two pieces of information determine the initial program constraint on the variable. As an example, the declared size of a buffer or an array constrains the space available for holding data.
- An assignment statement constrains the value of its left expression to the result of its right expression.
- A branching statement in the program indicates that different execution paths could be taken under different conditions. However, our execution trace is produced by running the program under a real test case. We already know which execution path is taken by the test case. Hence, we can immediately determine a

condition expression that specifies a constraint between the involved variables. For example, if statement "`if(i>j)`" exercises the FALSE branch, we know that `i`$\leq$`j` is a constraint between `i` and `j`.

- A library function call restricts the range of its return value if it has one. For example, the return value of function `open` is always greater than or equal to -1. In addition, some library functions have side-effects (i.e., modifying the states in addition to returning a value) that also impose constraints on variables. For example, calling function `getcwd` will change the content of the buffer specified by the parameter.

According to the above rules, symbolically executing each statement produces an expression describing the constraint between the program variables involved in the statement. To distinguish it from the program constraint (PC), we call such expression the *program constraint conjunction* (PCC). Thus, the program constraint at any given point in the trace can be expressed as the conjunction of all PCCs produced so far.

Security constraints: Producing security constraints requires clearly-defined high-level security requirements, e.g., the length of the string copied to a buffer must not exceed the capacity of the buffer. A wide range of security vulnerabilities like buffer overflow, SQL injection, and format string, are caused by improper uses of operations such as `strcpy`, `sql.exec`, and `printf`. Correct uses of such operations can be expressed as security requirements, which can then be used to generate security constraints. For example, a security requirement for `strcpy` will be "*the length of the second argument must not exceed the capacity of the first argument*". If the trace includes a statement `strcpy(a,b)`, where `a` is a buffer and `b` is a string, we produce a security constraint: `a.space>b.strlen`, where `a.space` denotes the capacity of buffer `a` and `b.strlen` denotes the length of string `b`. We use first-order logic to express security constraints.

security-critical func.	security requirement
strcpy(dst,src)	dst.space>src.strlen
strncpy(dst,src,n)	(dst.space $\geq$ n) $\wedge$ (n $\geq$ 0)
strcat(dst,src)	dst.space>dst.strlen + src.strlen
malloc(size)	size>0
calloc(nmemb,size)	(size>0) $\wedge$ (nmemb>0)
getcwd(buf,size)	(buf.space$\geq$ size) $\wedge$ (size $\geq$ 0)
fgets(dst,size,f)	(dst.space $\geq$ size) $\wedge$ (size $\geq$ 0)
scanf(format, ...)	# formats = # parameters-1
printf(format, ...)	# formats = # parameters-1

Table 1: Security requirements for library function calls. "x.space" is the size of the memory allocated to x and "x.strlen" is the string length of x.

`SecTAC` can detect the violation of a security requirement as long as such requirement can be expressed as a condition that program variables must satisfy. In the current implementation, we support two kinds of security requirements: *pointer addition requirements* and *function parameter requirements*. The former is derived from a useful observation made in [17], i.e., the result of a pointer addition must point to the same original object. The latter is generated from *security-critical library functions*, i.e., the library functions whose parameters must satisfy a condition to ensure the security of a software system. For example, functions `strcpy` and `printf` are both security-critical library functions. We have selected 20+ library functions that are well known to be "insecure" and formulated their security requirements. Table 1 shows some of these functions and the corresponding security requirements.

Formulating security requirements for a given type of vulnerability requires time and effort. Such formulation is usually straightforward given some basic understandings about the vulnerabilities. In fact, it only takes us little effort to formulate security requirements for pointer operations and selected security-critical library functions in our experiment. We will study effective methodologies for formulating security requirements in the future. Once the security requirements for a given type of vulnerability are formulated, they can be easily added in `SecTAC` to detect such vulnerability in C programs.

```
1:   void foo(int a,char *s){
2:      char buf[10];
3:      if(a>0)
4:          strcpy(buf,s);
5:   }
```

Figure 1: A sample program

An example: Figure 1 shows a sample program, which copies the second argument `s` into a buffer, if the first argument is greater than 0. Assume that there is only one security requirement, i.e., the length of a

string copied to a buffer using function `strcpy` must not exceed the capacity of the buffer. Furthermore, we assume that both arguments are user inputs, meaning they can be any values that are not known in advance. Now, consider a test case that includes the call `foo(x,y)` with `x=1` and `y=''test''`. This test case generates an execution trace `(1,2,3,4)` of statement numbers. Although this test case does not trigger any security problem, we will demonstrate that our method can effectively find the vulnerability in the trace. Table 2 shows the result of symbolically executing this execution trace. The first column indicates the statement number, and the second and third columns give the program and security constraints at the respective statements.

As shown in the table, the PC at statement 1 is (MIN≤a≤MAX)∧(s.strlen≥ 0), where [MIN, MAX] defines the range of an integer number, which is usually machine dependent, and `s.strlen` is a symbolic value denoting the length of string `s`. This is because both `a` and `s` are user inputs, i.e., `a` can be any integer value and `s` can be any string. The security constraint at statement 1 is TRUE since the statement does not include any operation that may violate any security requirement. More specifically, it does not include a call to the `strcpy` function. Statement 2 is a declaration statement of a buffer; it sets the space of the buffer to `10`. We do not include this in the program constraint. Instead, we directly update the `space` field of the buffer, i.e., `buf.space=10`, for simplicity.

Line #	Program Constraint	Security Constraint
1	(MIN≤ a≤ MAX)∧ (s.strlen≥ 0)	TRUE
2	(MIN≤ a≤ MAX)∧ (s.strlen≥ 0)	TRUE
3	(0<a≤ MAX)∧ (s.strlen≥ 0)	TRUE
4	(0<a≤ MAX)∧ (s.strlen≥ 0)	s.strlen<10

Table 2: Program and security constraints for the execution trace $(1, 2, 3, 4)$

Statement 3 is a condition statement and the test case exercises the TRUE branch, which implies that `a > 0` must be TRUE. Thus, the program constraint changes from (MIN ≤ a≤MAX)∧(s.strlen≥0) to `(0<a`≤MAX)∧(`s.strlen`≥0), as shown in the third line of the table. A security constraint is produced at statement 4, as shown in the fourth line of the table. The reason is that function `strcpy` is associated with a security requirement, i.e., the string length of the second argument must be less than the space allocated to the first argument. As a result, we produce a security constraint: `s.strlen<buf.space`. Since `buf.space=10`, we have `s.strlen<10`.

A security vulnerability exists at a particular point in the trace if an assignment of values to program variables satisfies PC but violates SC, i.e., PC ∧¬ SC is satisfiable at this point. This means that there exists a test case that can reach this point and also violates the security requirements. At statement 4, we check the satisfiability of PC ∧¬ SC, i.e., (`0<a`≤ MAX) ∧ (`s.strlen`≥ 0)∧¬ (`s.strlen<10`). We use a theorem prover and find that `a=1` and `s=''012345678910''` satisfies PC∧¬ SC. Thus a test case can be generated to uncover the vulnerability.

3 SecTAC Design

The goal of `SecTAC` is to detect security vulnerabilities in a program. As discussed, `SecTAC` reuses existing test cases for achieving high coverage and reducing testing effort. Specifically, we extract the execution trace of the program under each test case and then analyze each execution trace to determine whether it contains a security vulnerability. Figure 2 shows the workflow of `SecTAC`.

`SecTAC` performs security testing through three steps, *preprocessing*, *symbolic execution*, and *satisfiability analysis*, as indicated in Figure 2. In preprocessing, we generate execution traces from existing test cases and prepare the symbol table for tracking the state of program variables; in symbolic execution, we analyze every execution trace to extract the program and security constraints at each point in the trace; and in satisfiability analysis, we find inputs that can detect security vulnerabilities.

Preprocessing: In this step, we first use the *transformer* to transform the source program into an inter-

mediate representation consisting of an operator and its two operands, i.e., the three-address code. To obtain execution traces, the *instrumenter* parses and inserts the trace-logging code into this transformed program. This transformed, instrumented program is compiled and then executed by the *program executor* using all test cases. The trace-logging code generates an execution trace for each test case.

The *symbol-table builder* constructs a *symbol table* for all program variables for effectively tracking the program constraints on them. In addition to the size and type information, each program variable is also associated with additional attributes. For example, for a pointer that points to a buffer, we introduce two attributes to track *which buffer and which position in the buffer it points to* so that we can test the out-of-boundary buffer access.

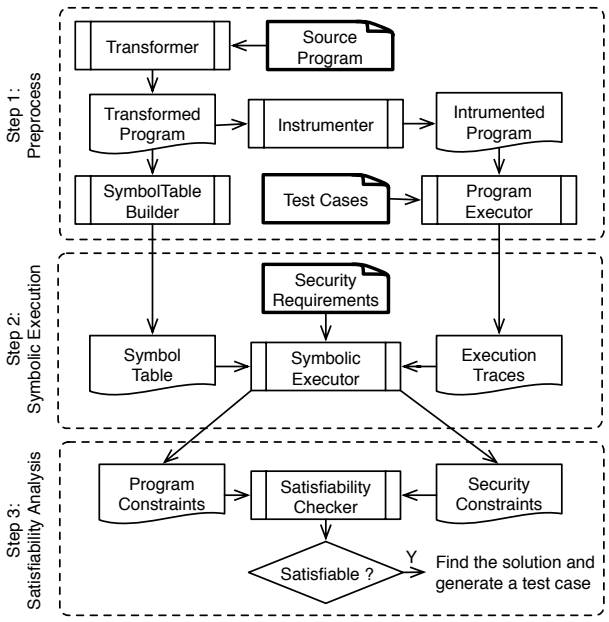

Figure 2: SecTAC Workflow

Symbolic execution: We use the *symbolic executor* to symbolically execute the trace to capture program constraints and check the pattern of each executed statement against the security requirements. Whenever a security requirement applies, e.g., a security-critical function call or a pointer addition statement is exercised, we generate a security constraint corresponding to such security requirement. The program and security constraints are predicates on the symbolic values of program variables and their attributes.

Satisfiability analysis: For each statement in the trace that generates a security constraint (SC), we get the program constraint (PC) at that statement and use a *satisfiability checker* to check if PC $\wedge\neg$ SC is satisfiable. If it is, a security vulnerability is detected. The solution given by the satisfiability checker is then used to generate test data to uncover the vulnerability. We express both program and security constraints using the SMT-LIB format [21] and use the Yices SMT-solver [8] as the satisfiability checker.

4 SecTAC Implementation

In this section, we describe the implementation of SecTAC in detail. Our discussion follows the workflow illustrated in Figure 2. During the discussion, we also use a concrete example to facilitate the understanding of the workflow. In this example, we apply SecTAC to the program shown in Figure 3, which receives a piece of data from the network and uses function DoParse() to process the data. To save space, we do not include the socket initialization and connection code.

```
typedef struct _Packet{
  char name[32];
}Packet;
DoParse(char * p){
   Packet pkg;
   strcpy(pkg.name, p);
}
void main(){
   char cmd[32];
   recv(sock, cmd, sizeof(cmd),0);
   if(strlen(cmd)>5) DoParse(cmd);
}
```

Figure 3: The sample program

4.1 Step 1: Preprocessing

The main tasks of preprocessing are (1) *generating execution traces* and (2) *constructing the symbol table*.

4.1.1 Generating Execution Traces

In SecTAC, the program is transformed by CIL [20], instrumented by the Java parser generator JavaCC, and executed under each test case to produce the corresponding execution trace. An execution trace is previously defined in Section 2 as a sequence of source code statements exercised by a test case. This definition facilitates the understanding of the basic ideas of our approach. However, SecTAC, an implementation of our approach, generates execution traces consisting of sequences of expressions and special marks. Expressions are either assignment statements or library function calls. Special marks are used to indicate: (1) function call entry and exit, (2) conditional branching, (3) parameter passing, and (4) returning of values to variables from function calls.

Figure 4 shows an example of the execution trace generated for the program given in Figure 3. This trace is produced by using a test case that sends a string, "testdata", to the program. The execution trace includes many "__cil_tmp" style variables introduced by CIL during code transformation. We will explain how SecTAC analyzes this trace line by line in Section 4.2.3. Declaration statements are not included in execution traces. The reason is that they are not "executed" by test cases. For example, the above trace does not include the declaration of cmd. However, declaration statements contain important information about the *type* and *size* of program variables. SecTAC handles declaration statements in the symbol-table builder, which we will discuss later.

```
1    @enter main;
2    __cil_tmp4=0;
3    __cil_tmp5=(unsigned long)(cmd)
                +__cil_tmp4;
4    __cil_tmp6=(char *)__cil_tmp5;
5    recv(sock, __cil_tmp6, 32, 0);
6    __cil_tmp7=0;
7    __cil_tmp8=(unsigned long)(cmd)
                +__cil_tmp7;
8    __cil_tmp9=(char *)__cil_tmp8;
9    tmp = strlen(__cil_tmp9);
10  @true (tmp > 5);
11  __cil_tmp10=0;
12  __cil_tmp11=(unsigned long)(cmd)
                +__cil_tmp10;
13  __cil_tmp12=(char *)__cil_tmp11;
14  @enter DoParse;
15  @parampass p=__cil_tmp12;
16  __cil_tmp3=0;
17  __cil_tmp4=0+__cil_tmp3;
18  __cil_tmp5=(unsigned long)(& pkg)
              +__cil_tmp4;
19  __cil_tmp6=(char *)__cil_tmp5;
20  strcpy(__cil_tmp6, p);
21  @exit DoParse;
22  @exit main;
```

Figure 4: The execution trace

4.1.2 Constructing the Symbol Table

The symbol table is used to track the state of program variables; it includes information about all program variables and user-defined functions in the trace. Specifically, the symbol-table builder parses the program and creates a *data object* for each program variable and a *function object* for each user-defined function. These objects include various attributes to track the state of program variables. Our symbolic execution is thus *attribute based*. This allows us to precisely capture the semantics of variables and functions to conduct more accurate reasoning. Next we describe the creation of the objects and discuss features added to address *pointer dependency* and *object locating*.

Data objects: The symbol-table builder creates a *data class* for every program variable type. A data class includes the size and type information as well as some other attributes about the data type it represents; an object of this class is created for each program variable of this data type. In SecTAC, we have a pre-defined class for each primitive type or primitive type with qualifiers. For example, we use classes Int and BCharacter for integers and characters declared in the program, respectively. For each composite type, we create a class using its type name. For example, for the data type _Packet in the code given in Figure 3, we create a class named _Packet as shown in Figure 5. We also have a pre-defined class Pointer for pointers, arrays, and buffers. All the above data classes are extended from a common *base class* BaseType that defines common attributes such as name, type, and symbolic value. It also includes a typesize field to

record the size of the memory allocated to the variable. For example, the `typesize` field of an `Int` object is 4 in a 32-bit computer.

Function objects: `SecTAC` also creates a class for each user-defined function to facilitate the trace analysis, i.e., help locate objects in the scope of any user-defined function. For example, for the program in Figure 3, `SecTAC` creates classes for user-defined functions `main()` and `DoParse()`. For every function class `f`, we create objects for the parameters to the corresponding function and the local variables declared in this function. These objects are the members of this function class `f`. Other statements in the function body are not included in class `f`.

All function classes are extended from a common abstract base class `Function` that includes a `getObject` method, which can be used to locate the object representing a local variable or function parameter in the scope of a user-defined function given a name.

In C programs, the global variables or static variables declared in the file scope are not included in any function. To track these variables, `SecTAC` also constructs a `Global` class and a *file-scope* class for each file, and puts the variables in these classes accordingly.

```
class _Packet extends BaseType{
  Pointer name=new Pointer("Char","name",
                                    "32");
  public BaseType getObject(String name){
    if(name.compareTo("name")==0)
      return name;
    return null;
  }
  public BaseType getObject(int offset){
    if(offset<name.getsize())
      return name.getObject(offset);
    return null;
  }
}
```

Figure 5: Class for _Packet struct

Pointer dependency: It is possible that several pointer-type variables are declared and point to the same array. For example, we can declare "`char p[10]`" and define a pointer "`char *q=p+5`" in a C program. We know that both pointers `p` and `q` point to the same array. The only difference is that `p` points to the beginning of the array, while `q` points to the sixth element of the array. The pointer objects are said to be *related* or *dependent* if they point to the same array. Hence, `p` and `q` are related. We notice that the operation on a given pointer object may impact its related pointer objects. For example, if we copy a string of length 6 to `p`, then the string lengths of `p` and `q` become 6 and 1, respectively. If we immediately copy another string of length 4 to `q`, then the string lengths of `p` and `q` become 9 and 4, respectively.

To correctly analyze the impact of pointer operations on *related* pointer objects, the `Pointer` class also includes a `start` field and a `space` field. A pointer object uses `start` to record its starting position in the array, and `space` to record the size of the space from its starting position to the end of the array. Thus, we can determine how the operation on one pointer object can impact others. From the previous example, we know that the `start` fields of the objects for `p` and `q` are 0 and 5, respectively. If a string of length 6 is copied to `p`, then we immediately know that `q` is impacted and its string length should be 1.

Object locating: Object locating addresses how to determine the target object(s) of a program statement. For example, for statement "`i=j.id;`", we need to locate the objects created for variable `i` and the member `id` of the structure `j`. As discussed before, each function class provides a method `getObject` to locate the object created for variables in its scope given a name. However, when a member of a composite type variable, e.g., `j.id` in the above example, is referenced, we need to further locate the *member object* representing the member of this variable.

Every class created for a composite type variable (e.g., struct or array) has a method `getObject` to locate the member object given a *name* or an *offset*. If the name is given, locating the member object is straightforward. However, after the program is transformed by CIL, the member of a composite type variable is always referenced using the offset, i.e., the distance from the beginning of this variable in memory to the member we are looking for. For example, the reference of `name` in `strcpy` in the following code

```
struct Person{ int age; char name[12];};
Person students[20];
strcpy(students[5].name, "Mike");
```

will be transformed by CIL into the reference of `__cil_tmp6` as follows:

```
__cil_tmp2=0;
__cil_tmp3=4+ __cil_tmp2;
__cil_tmp4=80;
__cil_tmp5=__cil_tmp4+__cil_tmp3;
__cil_tmp6=(unsigned int)students+__cil_tmp5;
```

An array or struct object is able to locate its member object through the offset because it knows the size of the memory allocated to each member. In the above example, the symbolic value of `__cil_tmp5` is a constant and can be simplified to value 84. Thus we call the `getObject(84)` method of the `students` object. Since the size of an array element is 16, we know that the member we are looking for is in the `Person` object that represents the sixth element in the array. We then lookup in this `Person` object with offset 4, which returns its member object at offset 4, i.e., the `name` object. Note that the above method will not work when the offset is not an constant, e.g., it depends on external inputs. Fortunately, such usage rarely occurs in practice, and we did not see this kind of usage in experiments. If this usage does appear in the program, SecTAC simply gives a warning.

As shown in Figure 5, the object created for composite type variable `_Packet` in the program given in Figure 3 includes two `getObject` methods, one for locating the object given a name and the other for locating the object given an offset.

4.2 Step 2: Symbolic Execution

Once we have the execution trace and the symbol table, we start to analyze the execution trace statement by statement to capture the program and security constraints using symbolic execution (symbolic value propagation). In the following, we will first overview when to produce program and security constraints during the symbolic execution. We will then discuss the main algorithm for trace-based symbolic execution. Finally, we will illustrate our algorithm using the execution trace given in Figure 4.

4.2.1 Producing Program and Security Constraints

The program constraint will be updated when we are building the symbol table. Specifically, when we create an object for a program variable, we produce a program constraint conjunction according to the variable declaration information. For example, statement "`int i;`" leads to the creation of an `Int` type object `i`. Thus, we produce a program constraint conjunction (MIN $\leq$`i.sym`$\leq$ MAX), where `i.sym` is the symbolic value of `i`. The program constraint will also be updated when a statement in the trace is symbolically executed.

- If it is an assignment statement, the attributes of the object for the right part determines the attributes of the object for the left part. In this case, we directly update the attributes of the left object instead of updating the program constraint. However, a program constraint conjunction may be generated when we symbolically execute the right part of the statement, which we will show later.
- If it is a branch statement, we update the program constraint based on which branch is exercised. For example, a conditional expression "`@true i>j`" indicates that the TRUE branch is exercised. Thus, SecTAC generates a program constraint conjunction (`i.sym>j.sym`).
- If the statement calls a library function, we need to update the program constraint according to its semantics. If the return value of the library function is assigned to a variable, we generate a program constraint

conjunction according to this return type. Since some library functions have constraints on their return values, a program constraint conjunction that further restricts the range of the returned value is produced. For example, the return value of `fopen` is always greater than or equal to -1, which is different from the default range of its return type. In addition, some library functions have side-effects on their parameters. Some side-effects can be considered as equivalent to updating the object attributes, e.g., for `strcpy(dst,src)`, the `strlen` field (a symbolic value that denotes the string length) of the `dst` object is updated to that of the `src` object. Some side-effects, however, impose constraints on the involved parameters. For example, after calling `getcwd(buf,n)`, the `strlen` of `buf` is less than n if the length of the current path is less than n, and unchanged otherwise. In this case, we also generate a program constraint conjunction.

A program statement in the execution trace is said to be *security critical* if it may violate a security requirement. In the current implementation of `SecTAC`, any statement involving either a security-critical function or a pointer addition is a security-critical statement. `SecTAC` produces a security constraint, i.e., a first order logic expression, at every security-critical statement.

4.2.2 Algorithm for Symbolic Executor

We now describe the detail of the `SecTAC` symbolic executor as given in Figure 6. We first create a *stack* to keep track of the current function object, i.e., the active function object in use, which is always the one at the top of the stack. `SecTAC` then processes each statement in the trace according to the following rules: (1) if it is a function entry, `SecTAC` creates a new object of this function class and pushes the object into the stack; (2) if it is a function return, `SecTAC` pops an object from the stack; (3) If it is an assignment statement, `SecTAC` performs symbolic execution on the left and right expressions, and updates the symbolic values of the involved variables; (4) if it is a conditional statement, `SecTAC` produces a program constraint conjunction that captures which branch is exercised; (5) if it is a library function call, `SecTAC` processes it as shown in Algorithm 7. If the function is in the right part of an assignment statement, a new object is created according to its return type. If the function further limits its return value to a smaller range compared to its type, the program constraint on this object is updated. If the function also has side-effects, the attributes of the involved objects are updated accordingly, and the program constraint is also updated as needed. If the function is also a security-critical function, a security constraint is generated.

```
Input: trace (a file containing the execution trace)
 1: Stack S=[ ];
 2: while !eof(trace) do
 3:    stmt=getStatement(trace);
 4:    if stmt matches "@enter F" then
 5:       F f = new F(newname());
 6:       S.push(f);
 7:    else if stmt matches "left_expr = right_expr" then
 8:       left = symExpr(left_expr);
 9:       right = symExpr(right_expr);
10:       symbolicCopy(left,right);
11:    else if stmt matches "L(ParamList)" then
12:       processLibraryFunc(L,ParamList);
13:    else if stmt matches "@true condition" then
14:       obj = symExpr(condition);
15:       outputPCC(obj);
16:    else if stmt matches "@false condition" then
17:       obj = symExpr(¬ condition);
18:       outputPCC(obj);
19:    else if stmt matches "@exit F" then
20:       S.pop();
21:    end if
22: end while
```

Figure 6: Symbolic executor

`SecTAC` needs to locate the object given a variable name or an offset in a composite type variable. Given a name, `SecTAC` seeks the object in the current function object (i.e., the one at the top of the stack), the caller function object, the file-scope function object, or the global function object using `getObject` provided in every function object. In addition, given an offset and a composite type object, `SecTAC` locates the corresponding object using function `getObject` provided in every composite type object. We do not consider paddings used for memory alignment in a composite type variable.

In Figures 6 and 7, `F` is the function class for a user-defined function; `L` is the name of a library function;

`newname()` returns a unique string that has never been used; `newobject(t)` creates an object of type `t`; `outputPCC()` outputs a program constraint conjunction (PCC); and `outputSC()` produces a security constraint (SC).

Symbolic Execution on Expressions: A critical part of the algorithm in Figure 6 is the symbolic execution on expressions, `symExpr()`. An expression `e` can be a constant number n, a character c, a constant string str, a variable v, a pointer dereference $*v$, a variable reference $\&v$, a dyadic operation $e_1\ op\ e_2$, a struct membership operation $v.m$, or a library function call.

```
Input: L (name of the library function)
       ParamList (list of function parameters)
 1: for every i-th parameter do
 2:    param[i]=symExpr(ParamList[i]);
 3: end for
 4: if Function L has a return value then
 5:    type = returnType(L);
 6:    obj = newobject(type);
 7: end if
 8: if Function L has restriction on the return value then
 9:    outputPCC(obj);
10: end if
11: if Function L has side effects then
12:    updates(L,param);
13:    outputPCC(param);
14: end if
15: if Function L is a security-critical function then
16:    outputSC(L,param);
17: end if
18: return obj;
```

Figure 7: processLibFunc()

The algorithm in Figure 8 shows the symbolic execution procedure on expressions. Given an expression `e`, the symbolic execution works as follows: (1) if `e` is a constant number or character, a new object of the class for such data type is created, and its symbolic value is set to this constant value; (2) if it is a constant string, a `Pointer` object is created, and its `strlen` field is set to be the length of this constant string; (3) if it is a variable, we will locate the corresponding object and return it; (4) if it is $*v$, we locate the object corresponding to v and return the object specified by the `point_to` field of this pointer object; (5) if it is $\&v$, we locate the object corresponding to v and create a `Pointer` object. We then set the `point_to` field of the newly created object to the object corresponding to v; (6) if it is $v.m$, we locate the object of v, then return its member object with the name m; (7) if it is $e_1\ op\ e_2$, we recursively perform symbolic execution on expressions e_1 and e_2. Based on the types of the returned objects, we take different actions as shown in Figure 8; (8) if it is a library function call, we handle it in the same way as we handle library function calls in Figure 6.

In Figure 8, `symOp()` concatenates two symbolic values with the operator. `symAddSpec()` processes the addition between a `Pointer` object `p` and an `Int` object `i` as follows:

- If `p` points to a buffer, we create a new `Pointer` object `obj` and set its `space`, `start`, and `strlen` fields based on `p` and `i`. Specifically, `obj.space` and `obj.start` are set to `p.space-i.sym` and `p.start+i.sym` respectively. `obj.strlen` is set to the following conditional expression:

  `((p.strlen≥i.sym) (p.strlen-i.sym) newsym)`

  This expression indicates that `obj.strlen` is set to `p.strlen-i.sym` if `p.strlen≥i.sym`, and a new symbol `newsym` otherwise. A program constraint conjunction is also produced for the new symbol `newsym`, i.e., `newsym`≥ 0. Finally, the new object `obj` is returned.
- If `p` points to a composite type object, e.g. array or struct, then we need to find a member object inside this composite object through offset `i`. In this case, we use the `getObject(i)` method in object `p.point_to` to locate and return the object.
- If `p` points to neither a buffer nor a composite type data, then it is just a pointer arithmetic. In this case, a new object will be created in a similar way as the first case. The only difference here is that the `strlen` field need not be set.

`SecTAC` determines the above three patterns through the *element type* of pointer and the *type cast* before it. As we discussed, `SecTAC` generates a security constraint for every pointer addition to check whether the

Statement	Program Constraint Conjunction	Security Constraint
declaration	cmd.strlen$\geq$0	TRUE
3	(newsym1 $\geq$ 0)	(0+0 $<$ 32)
5	((cmd.strlen $\geq$ 0) (cmd.strlen-0) newsym1)$\geq$ 0	(32 $\leq$ 32-0)
7	(newsym2 $\geq$ 0)	(0+0 $<$ 32)
10	((cmd.strlen $\geq$ 0) (cmd.strlen-0) newsym2)$>$5	
12	newsym3 $\geq$ 0	(0+0 $<$ 32)
20		32 $>$ ((cmd.strlen $\geq$ 0) (cmd.strlen-0) newsym3)

Table 3: Program and security constraints

result still points to the same original object.

Pointer analysis : We have already discussed the pointer dependency problem in the last subsection. In the following, we will discuss how we analyze related (or dependent) pointers in symbolic execution. Specifically, when we create an object for a buffer, we also include a number of links in this object through which we can locate all `Pointer` objects that operate on this buffer. Let us consider a particular pointer `p` that points to a buffer. When we update the object for this pointer, we will need to find the object for the original buffer this pointer points to and locate all `Pointer` objects that operate on this buffer. Let `q` be a `Pointer` object we find. We first check `p.start` and `q.start` to decide their relative positions in the buffer. There are two cases:

- If `q`'s position in the buffer is before that of `p`'s, we compare `q.strlen` with the distance between them. If `q.strlen` is larger than the distance, we have to update `q.strlen` accordingly; otherwise, nothing needs to be done.
- If `q`'s position in the buffer is after `p`'s, we compare `p.strlen` with the distance between their positions. If `p.strlen` is larger than the distance, we have to update `q.strlen` accordingly; otherwise, nothing needs to be done.

Input: e (*an expression*)
1: **if** e is a constant number n **then**
2: obj=new Int(n);
3: **else if** e is a constant char c **then**
4: obj = new BCharacter(c);
5: **else if** e is a constant string str **then**
6: obj = new Pointer();
7: obj.strlen = str.length;
8: **else if** e is a variable v **then**
9: obj=locateObject(v);
10: **else if** e is of form $*v$ **then**
11: obj =(locateObject(v)).point_to;
12: **else if** e is of form $\&v$ **then**
13: obj = new Pointer();
14: obj.point_to = locateObject(v);
15: **else if** e is of form $v.m$ **then**
16: obj = (locateObject(v)).getObject(m);
17: **else if** e is of form e_1 *op* e_2 **then**
18: o_1 = symExpr(e_1);
19: o_2 = symExpr(e_2);
20: **if** o_1 *op* o_2 matches "pointer + int" **then**
21: obj = symAddSpec(o_1, o_2);
22: **else if** o_1 *op* o_2 matches "pointer-pointer" **then**
23: obj = new Int();
24: obj.sym = symOp(o_1.index,'-',o_2.index);
25: **else**
26: obj = o_1.clone() ;
27: obj.sym = symOp(o_1.sym,*op*,o_2.sym);
28: **end if**
29: **else if** e is a library function call L(ParamList) **then**
30: obj=processLibFunc(L,ParamList);
31: **end if**
32: return obj;

Figure 8: symExpr()

4.2.3 Example of Symbolic Execution

Next we will show the result of applying our symbolic execution on the trace given in Figure 4. We process one statement in the trace at a time. Since the first statement is the function entry mark "`@enter main`", `SecTAC` creates a new object `main1` of the function class `main()` and pushes it into a stack that records the current active function object. `SecTAC` keeps a global counting variable to assure the name uniqueness of every new object.

Statement 2 is an assignment statement. `SecTAC` uses function `locateObject` to locate the object for the left expression by calling `main1.getObject("__cil_tmp4")`. Since the right expression is a constant 0, we create a new object of the pre-defined `Int` class, set its symbolic value to 0, and return it

as the object for the right expression. Based on the semantics of assignment statements, we then copy its symbolic value to the left object. Thus, the object for `__cil_tmp4` has the symbolic value 0.

Statement 3 is also an assignment statement. We use `locateObject("__cil_tmp5")` to locate the object For the left part. For the right part, it is a pointer addition where the pointer points to buffer `cmd`. We thus create a `Pointer` object `obj` that points to the same buffer as the `cmd` object and set its `space`, `start`, and `strlen` fields as follows. The `space` field is set to `cmd.space-__cil_tmp4.sym`. Since the symbolic values of `cmd.space` and `__cil_tmp4` is 32 and 0 respectively, the symbolic value of `obj.space` is set to `32-0`. Similarly, the symbolic value of `obj.start` is set to `0+0`. According to `symAddSpec()`, the `strlen` field of `obj` is set to:

`((cmd.strlen` ≥ 0`) (cmd.strlen-0) newsym1)`.

The newly created object `obj` is then used to update the object for the left part, i.e., `__cil_tmp5`. In addition, we also generate a program constraint conjunction on `newsym1`, as given in the table. Since the right part is a pointer addition, we also generate a security constraint based on our *pointer addition security requirement*: `cmd.start+__cil_tmp4.sym < cmd.space`, which is actually `0+0<32` (i.e., TRUE).

Statement 4 is an assignment statement, and we simply copy the attribute values of the right object to that of the left object.

Statement 5 calls `recv()`, which is a security-critical function according to our security requirement. Thus we generate a security constraint: `32`$\leq$`__cil_tmp6.space`, i.e., the number of bytes to receive should not exceed the size of the receiving buffer. Since the symbolic value of `__cil_tmp6.space` is `32-0`, we have a security constraint `32`$\leq$`32-0` (i.e., TRUE) as shown in Table 3. Since `recv()` has side-effects, we also capture its program constraint. Specifically, according to the specification of `recv()`, the receiving buffer `__cil_tmp6` is not null terminated. Thus, the `strlen` field of the receiving buffer, which is copied from `__cil_tmp5`, should be greater than or equal to 0. Thus, we generate a program constraint conjunction as shown in Table 3.

Statements from 6 to 8 have the same patterns as statements from 2 to 4 and are processed in the same way.

Statement 9 involves another library function `strlen` which retrieves the length of a given string. Thus, we create an `Int` object and set its symbolic value to `__cil_tmp9.strlen`. This object is then used to update the symbolic value of the left object.

Statement 10 is a branch condition with the `@true` mark. Thus we generate a program constraint by replacing the involved variables with their symbolic values, as shown in Table 3.

Statements from 11 to 14 have the same patterns as statements from 1 to 4 and are processed in the same way. Statement 15 is an assignment statement with mark `@parampass` to simulate parameter passing. The left part is the formal parameter of a user-defined method, and the right part is the actual parameter. This mark tells `SecTAC` to skip the *current* function object (callee) when locating the object for the right part of the statement, i.e., the actual parameter. Once we locate the objects, we process this statement in the same way as other assignment statements.

Statements 16 and 17 are both assignment statements and are processed in the same way as others.

Statement 18 is an assignment statement where the right part indicates that we are visiting a member inside a *struct* object through an *offset*. `SecTAC` recognizes this pattern from the *type* of the `Pointer` object for `&pkg` and the *type cast* before it. It implies that the pointer which points to a `|_Packet|` object is type-casted to an address. Thus, for expression `&pkg`, `SecTAC` creates a new `Pointer` object and sets its `point_to` field to object `pkg`. Then, for the right part, `SecTAC` calls the `getObject(0)` method of `pkg` to find the object at offset `0`, where `0` is the symbolic value for `__cil_tmp4`. It will return the object that represents the `name` buffer inside `pkg`.

Statements 19 is another assignment statement and is processed in the same way as others.

Statement 20 calls a library function `strcpy`. Based on its semantics, `SecTAC` updates the `strlen` field of the object for `__cil_tmp6` using that of the object for `p`. Based on the pointer dependency, the objects for `__cil_tmp5` and `pkg.name` are `__cil_tmp6`'s related `Pointer` objects that point to the same buffer. Thus, the `strlen` fields of these two objects are also updated. In addition, this function is also a security-critical function. We thus have `__cil_tmp6.space>p.strlen`, which produces a security constraint as shown in Table 3.

The next two statements makes `SecTAC` pop the two function objects from stack, and the symbolic execution stops here.

4.3 Step 3: Satisfiability Analysis

Finally, the program and security constraints are expressed in SMT-LIB [21] format, which is recognized by many SMT solvers. Table 3 shows the program and security constraints for the statements in the trace given in Figure 4 that either produce a security constraint or a program constraint conjunction. We use the SMT solver Yices [8] to check the satisfiability of PC $\wedge\neg$ SC for each SC and the PC at the same point in the trace.

For each security constraint, `SecTAC` combines its negation with the program constraint at the same statement in the trace. They are represented in first-order logic and form an input file to the SMT solver Yices [8]. Suppose that we are trying to check if a security vulnerability exists at statement 20. We then retrieve the PC at statement 20 as follows:

```
(cmd.strlen>=0) AND (newsym1>=0) AND
(((cmd.strlen>=0) (cmd.strlen-0) newsym1)>=0) AND
(newsym2>=0) AND (newsym3>=0) AND (((cmd.strlen>=0) (cmd.strlen-0) newsym2)>5)
```

We found that PC $\wedge\neg$ SC is satisfiable at statement 20. The SMT solver Yices gives a solution, from which we generate a test case that sends the program a 32 bytes long string whose last character is not null. This test case causes `strcpy` to overflow the `name` buffer in `pkg`. A buffer overflow vulnerability is thus detected.

Note that the PC at a given point in the trace may include a large number of conjunctions. In this case, checking the satisfiability of PC $\wedge\neg$ SC could be very expensive. However, we note that a lot of PC conjunctions are actually irrelevant to SC since they only involves variable symbols that do not impact SC. Removing these irrelevant conjunctions will not change the result of satisfiability analysis. We thus use only *SC-dependent* PC conjunctions to save the cost. Two conjunctions are said to be *directly related* if they include at least one common variable symbol. Then, starting from an empty S, we first identify all PC conjunctions that are directly related to SC and put them in S. We then *repeatedly* check every PC conjunction and add it into S if it is SC-dependent, i.e., directly related to at least one conjunction in S. We stop when there are no more SC-dependent PC conjunctions. Let PC$'$ be the conjunction of all conjunctions in S. We only need to check the satisfiability of PC$'$ $\wedge\neg$ SC instead of PC $\wedge\neg$ SC.

4.4 False Negatives and Positives

Given a security vulnerability in a program, `SecTAC` can detect such security problem if (1) the vulnerability is modeled by one of the security requirements, (2) an execution path that can trigger such security problem is exercised by one of the test cases, and (3) the theorem prover for satisfiability analysis can correctly find a solution if PC $\wedge\neg$ SC is satisfiable. In other words, there will be false negatives if one of the above three conditions is false. Similarly, `SecTAC` will generate a false positive if the theorem prover returns a solution when PC $\wedge\neg$ SC is not satisfiable. In our experiments, we did not find any false positive.

Program	LOC	Input	LOT	Time(mm:ss)	#KnownBugs	#FoundBugs	#FP	Remark
Bind 1	1116	www.cnn.com	539	00:01	1	1	0	
Bind 2	1306	cnn.com	1117	00:01	1	1	0	
Bind 3	380	default	365	00:01	1	1	0	
Bind 4	645	www.nbc.com; www.cnn.com	162	00:01	1	2	0	1 new bug
Sendmail 1	537	default	6207	00:02	6(5)*	6	0	1 new bug
Sendmail 2	791	default	5509	00:03	1	1	0	
Sendmail 3	416	default	2534	00:03	1	2	0	1 new bug
Sendmail 4	485	default	1379	00:03	4	4	0	
Sendmail 5	622	default	6669	00:03	3	3	0	
Sendmail 6	390	default	129	00:01	1	1	0	
Sendmail 7	929	default	2145	00:03	2	2	0	
Wu-ftp 1	503	/tmp/aa	79	00:01	4	4	0	
Wu-ftp 2	744	/tmp/test.c	106	00:01	1	2	0	1 new bug
Wu-ftp 3	689	/tmp/aa	399	00:01	6	8	0	2 new bugs
nullhttpd-0.5.1	2328	50 test cases	12447	08:07	1	2	0	1 new bug
lancer	4261	50 test cases	118657	49:18	0	4	0	4 new bugs
bftpd-2.3	5766	10 test cases	65027	11:42	0	1	0	1 new bug

* According to the BAD marks in the program, there are 6 bugs in the trace. However, we found that one of them is not a bug.

Table 4: Experimental Results. "LOC" represents the number of lines of the code; "Input" represents the program input we use; "LOT" represents the number of lines in the execution trace exercised by the test case; "Time" represents the time that our tool used; "#KnownBugs" is the number of previously reported vulnerabilities in the execution trace; "#FoundBugs" is the number of vulnerabilities found by `SecTAC`; and "#FP" is the number of false positives.

5 Experiments

To evaluate the effectiveness of our approach, we developed a tool named `SecTAC` and applied it on 14 benchmark programs [19], two open source http server programs, `nullhttpd-0.5.1` and `lancer`, and an open source ftp server program `bftpd-2.3`. We used their latest versions in our experiment. The benchmark programs represent various kinds of memory corruption vulnerabilities in certain versions of the **Bind**, **Sendmail**, and **Wu-ftp** programs. They have been used to evaluate the effectiveness of many buffer overflow detection tools [19, 29, 28, 23]. For each of these programs, there is a buggy version and a fixed version. We used the buggy version in our experiment. Our results show that `SecTAC` can detect every reported vulnerability as long as the vulnerability exists in the traces. In addition, `SecTAC` also detected six vulnerabilities in the benchmark programs, four vulnerabilities in `nullhttpd-0.5.1`, four vulnerabilities in `lancer`, and one vulnerability in `bftpd-2.3` that, to the best of our knowledge, have not been reported previously. Next, we will report our findings in detail.

Table 4 summarizes our experimental results. The first 14 rows show the result of evaluating `SecTAC` on the 14 benchmark programs [19]. The numbering for each program in Table 4 is identical to what was used in [19]. As shown in the last column of the table, we found new vulnerabilities in Bind 4, Sendmail 1, Sendmail 3, Wu-ftp 2, Wu-ftp 3, `nullhttpd-0.5.1`, `lancer`, and `bftpd2.3` programs.

Test inputs: For each buggy benchmark program version, [19] either provided a specific input file as the test data or hard-coded the values of some variables in the program to trigger the vulnerability. However, a major merit of `SecTAC` is that it can detect vulnerabilities under test cases from functional testing that do not trigger any vulnerability. Hence, in our experiments, whenever it is possible, we use test inputs that exercise paths containing the reported vulnerabilities but do not trigger any of them. Only when it is impossible to find a test case exercising the known vulnerable path without triggering the vulnerability, we use the test input provided in [19]. For the sake of presentation, we call a test case *normal* if it does not trigger any

Program	Test Input that Triggers the New Vulnerability	Location of the New Vulnerability	Remarks
Bind 4	www.cnn.com; www.nbc.com	ns-lookup-bad.c:277	*nsp* out-of-bound
Sendmail 1	default	crackaddr-bad.c:460	*buflim* out-of-bound
Sendmail 3	default	mime1-bad.c:212	*infile* out-of-bound
Wu-ftp 2	a 200 bytes long string for argv[1]	call_fb_realpath.c:94	*strcpy* buffer overflow
Wu-ftp 3	/a...a (48 a's)/aa	realpath-2.4.2-bad.c:269	*where* out-of-bound
Wu-ftp 3	/a...a (48 a's)/aa	realpath-2.4.2-bad.c:257	*strcpy* buffer overflow

Table 5: New Vulnerabilities in the benchmark programs

vulnerability. We call the test cases provided in [19] *default* in the table.

For http server programs, we randomly generate 50 normal http requests. For the ftp server program, we manually generate 10 test cases that include basic ftp commands such as "ls", "get", and "put". We use GCC bounds checking extension to monitor the program execution. These test cases do not trigger any out-of-boundary operation. Next we describe the test input to every benchmark program tested in the experiment.

In the Bind 1 program, a vulnerability occurs when a negative value is passed as the third argument of `memcpy`, which causes the program to copy a huge amount of data to a buffer. In [19], a constant string "$sls.lcs.mit.edu$" is hard-coded as the second argument of `strcpy` to achieve this. `SecTAC` can detect this vulnerability easily. However, to make our experiments more illustrative, we use string "$www.cnn.com$" as the normal test data under which the program runs normally. Similarly, for the Bind 2 program, we use string "$cnn.com$" as the normal test input instead of the original hard-coded input string "$sls.lcs.mit.edu$" that crashes the program. The Bind 3 program does not check the buffer space when calling `memcpy`. The provided test case is a file `s3.in` whose content is "9283721". However, we notice that as long as its content is not "0", the vulnerability always occurs and crashes the program. Thus we just use the original test case. The Bind 4 program uses `sprintf` without boundary checking. A string of 1072 bytes long is provided in [19] as the input to trigger the vulnerability. We do not use this input. Instead, we use a normal test input as given in Table 4.

The test input to each Wu-ftp program is a string that represents a path. For the Wu-ftp 1 program, the original test case in [19] is "$/tmp/$" followed by 24 'a's. This is carefully designed to trigger the buffer overflow caused by `strcpy`. For the Wu-ftp 2 program, the original test case is also a specific complex path with 9 subdirectories, which triggers the vulnerability caused by `strcat`. For the Wu-ftp 3 program, the length of the input path is made more than 47 to trigger the vulnerability caused by `strcpy`. In our experiments, we use normal test inputs. Specifically, for Wu-ftp 1 and Wu-ftp 3, we use a normal test input "$/tmp/aa$" that does not trigger the vulnerability. For Wu-ftp 2, we also use a normal test input "$/tmp/test.c$", which is the path of an existing file and does not trigger the vulnerability.

Most of the vulnerabilities in the Sendmail programs are caused by out-of-bound pointer operations. These operations are usually in a loop in the program where the pointer is increased by 1 for each looping. As a result, in the execution trace, the out-of-boundary operation of a pointer only occurs when a test case can actually trigger the vulnerability. In other words, the execution trace under a normal test case does not contain the vulnerability. Thus, we use test cases provided by the benchmark programs in our experiments.

Performance: We did the experiments on a 2GHz Core 2 Desktop running Ubuntu-8.10 Linux operating system. We let the JVM use a maximum of 1G heap memory during our experiments. The fifth column of Table 4 shows the execution time of `SecTAC` for analyzing all traces for each program. The execution time is the sum of the times needed for trace-based symbolic execution and satisfiability analysis. We can see that `SecTAC` can quickly analyze C programs for vulnerability.

New vulnerabilities: In addition to the known bugs, `SecTAC` also detected six new vulnerabilities in the 14 benchmark programs as shown in Table 5. Test cases that trigger these vulnerabilities can be directly derived from the solutions given by the satisfiability checker Yices [8] in `SecTAC`. Notably, we detected a

vulnerability that is previously considered to be safe. The authors of [19] explicitly commented the line 257 of file `realpath-2.4.2-bad.c` in the Wu-ftp 3 program as a safe call. However, our experiment shows that it is not. As shown in Table 5, when the length of a directory name is long enough, the `strcpy` function at line 257 will overflow the destination buffer whose size is only 46 bytes.

For the `nullhttpd-0.5.1` program, `SecTAC` found three buffer overflow vulnerabilities at line 143 of file "http.c". The attacker can overflow three different buffers in this line of code. In addition, it also found a new vulnerability at line 58 of file "config.c", where the program uses `snprintf` to copy a string variable `config.server_base_dir` and a constant string "$/bin$" to buffer `server_bin_dir`. However, the space allocated to `server_bin_dir` is 255. If the string length of `config.server_base_dir` is 255, the buffer is not null terminated and the string "$/bin$" cannot be copied to the buffer, causing a configuration error. Lines 59 to 61 in the same file have the same vulnerability. For the `lancer` program, `SecTAC` found four buffer overflow problems in "handler.c" and "host.c". These problems have the same pattern: the author declared a buffer with the size of `n`, and used `strncpy` to copy at most `n-1` non-zero characters to the buffer. However, it is possible that the buffer is not null-terminated and cause buffer overflow. For the `bftpd-2.3` program, `SecTAC` detected that the buffer of "bu_host" can be not null-terminated whose content comes from an external input. We reported this vulnerability to the author of the program and a new version was released subsequently to fix this bug.

6 Related Work

This section reviews existing security testing approaches and compare them with our approach. The method in [15] detects buffer overflow vulnerabilities using existing test cases. This method inserts checking code into the source program as assertions to check if string library functions are properly used. They do not perform symbolic analysis and ignore branch conditions, causing many false alarms. The predictive testing approach in [18] inserts assertion statements into the source program and uses a combination of concrete and symbolic execution on the given test inputs to discover assertion violations. This method uses concrete values for expressions like library function calls. Our trace-based symbolic execution is more precise in handling the library functions. In addition, `SecTAC` also addresses the pointer dependency problem.

Dynamic symbolic execution is used in many test data generation tools for security testing [12, 24, 7, 11, 5, 13]. DART [12, 11] and CUTE [24] can automatically generate test cases. However, they use concrete values for complex constraints that they cannot handle. Many possible paths are ignored. The coverage is often not as good as the test cases that are carefully designed in traditional testing. `SecTAC` can take advantage of previous test effort for high coverage. In addition, these tools overlook useful information about variables and functions such as pointer dependency and function return type. In contrast, `SecTAC` uses attribute-based analysis; it treats program variables and functions as objects and introduces attributes like buffer size, string length, and function return type in objects to achieve more accurate analysis. Moreover, these tools do not scale well to large and complex programs. Our tool is more scalable since it avoids the search space explosion problem and can perform testing in parallel.

EXE [7] and KLEE [5] were developed to achieve high branch coverage. They only detect memory overflow vulnerabilities. `SecTAC` features a novel test model that unifies program constraints and security constraints for detecting vulnerabilities; it can detect a wider range of attacks than EXE and KLEE. As one example, EXE and KLEE cannot detect security vulnerabilities caused by pointer dependency. Nevertheless, `SecTAC` can take advantage of the high coverage achieved by EXE and KLEE. Specifically, we can reuse the test cases produced from EXE or KLEE to uncover vulnerabilities in the program.

SPLAT [28] improves DART by introducing a length attribute in each buffer. It also represents a fixed-length prefix of the buffer elements symbolically. Other buffer elements are represented using concrete values during execution. The limitation is that when the program visits a buffer element beyond the prefix, their

symbolic execution becomes *concrete*. SecTAC treats *related* buffer elements (e.g., those belonging to the same string) as a single object and generates new objects only when a particular buffer element is visited. This greatly improves the precision and reduces the cost.

SAGE [13] also employs trace-based symbolic execution with satisfiability analysis. However, SAGE works on the binary level; a lot of useful information in the source code is not directly available for facilitating the analysis. For example, in the binary level, it is very difficult to extract information about the declared buffer size and the pointer dependency unless the debug information is available, in which case it is no longer "binary" level. This means that SAGE cannot handle pointer dependency. Similarly, we found that the methods in [4, 23] also work on binary code and thus cannot handle the pointer dependency issue. In addition, SAGE is still not available for public evaluation. In contrast, SecTAC is open source and available at `http://ranger.uta.edu/~dliu/sectac`.

7 Limitations and Suggestions

SecTAC has a number of limitations. First, we must have the test cases ready before doing the security testing. The effectiveness of SecTAC depends on the completeness of the existing test cases. In fact, the branch coverage of the test cases determines the number of paths that our method can check. We plan to address this limitation by complementing SecTAC with systematic path exploration techniques. For example, after checking the existing test suite for security vulnerabilities, we could use concolic execution techniques to generate additional paths, which we then can also check for vulnerabilities with SecTAC.

Second, the size of an execution trace for large complex programs may be huge, e.g., millions of lines of statements. Analyzing a large execution trace can cause many problems. For example, the program constraint at a given point in the trace may consist of millions of conjunctions, making it infeasible for a theorem prover to do the satisfiability analysis efficiently. As another example, it may be the case that a large number of statements in the trace generate security constraints. As a result, SecTAC may invoke the SMT solver very frequently, which can slow down security testing significantly. We plan to improve SecTAC by managing program and security constraints more efficiently, e.g., by using BDDs [2, 3].

Third, although a wide range of security requirements can be modeled by improper uses of certain operations, a generic and effective method is needed to model security requirements in complex scenarios. As one example, our approach will fail if a security problem only occurs when multiple execution traces are involved. How to effectively model such security requirement is a challenging problem that we would like to investigate in the future.

8 Conclusion and Future Work

In this paper, we proposed an approach for testing the security of C programs using trace-based symbolic execution and satisfiability analysis. We developed a tool named SecTAC to demonstrate the effectiveness of our approach. We evaluated this tool on 14 benchmark programs and 3 open source programs. The result shows that our tool quickly identified every reported vulnerability in the traces and also found 15 new vulnerabilities. In conclusion, our tool is effective and efficient in testing the security of current software systems since test cases from traditional software testing can be reused to reduce the testing effort and perform more effective symbolic execution.

We are interested in the following directions in the future. First, although our approach can handle multi-threaded programs as long as the test cases are available, it only analyzes a specific combination of the traces generated by different threads. We propose to identify the trace for each thread and seek effective ways to combine them to improve the detection of security vulnerabilities in multi-threaded programs. Second, we

will also seek solutions to further improve the efficiency of `SecTAC` and conduct more experiments on large and complex programs to evaluate our approach.

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 340–353, 2005.

[2] S. B. Akers. Binary decision diagrams. *IEEE Transaction on Computers*, C-27(6):509 – 516, June 1978.

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, C-35(8):677–691, 1986.

[4] J. Caballero, S. McCamant, A. Barth, and D. Song. Extracting models of security-sensitive operations using string-enhanced white-box exploration on binaries. Technical report, EECS Department, University of California, Berkeley, 2009.

[5] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex system programs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[6] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In *Proceedings of the International SPIN Workshop on Model Checking of Software*, 2005.

[7] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: automatically generating inputs of death. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 322–335, 2006.

[8] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proceedings of the Computer-Aided Verification Conference (CAV)*, pages 81–94, 2006.

[9] G. Fink, C. Ko, M. Archer, and K. Levitt. Towards a property-based testing environment with applications to security-critical software. In *Proceedings of the 4th Irvine Software Symposium*, pages 39–48, 1994.

[10] A. Ghosh, T. O'Connor, and G. McGraw. An automated approach for identifying potential vulnerabilities in software. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 104–114, 1998.

[11] P. Godefroid. Compositional dynamic test generation. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, 2007.

[12] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[13] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed Systems Security (NDSS)*, pages 151–166, 2008.

[14] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, pages 125–136, 1992.

[15] E. Haugh and M. Bishop. Testing c programs for buffer overflow vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, pages 123–130, 2003.

[16] Secure Software Inc. The rough auditing tool for security (RATS). www.securesoftware.com.

[17] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the International Workshop on Automated Debugging*, 1997.

[18] P. Joshi, K. Sen, and M. Shlimovich. Predictive testing: amplifying the effectiveness of software testing. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 561–564, 2007.

[19] T. Leek M. Zitser, R. Lippmann. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 97–106, 2004.

[20] G. C. Necula, S. McPeak, S.P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the International Conference on Compiler Construction*, pages 213–228, 2002.

[21] S. Ranise and C. Tinelli. The satisfiability modulo theories library(smt-lib). `www.SMT-LIB.org`, 2006.

[22] M. Ringenburg and D. Grossman. Preventing format-string attacks via automatic and efficient dynamic checking. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 354–363, 2005.

[23] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 225–236, 2009.

[24] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 263–272, 2005.

[25] E. C. Sezer, P. Ning, C. Kil, and J. Xu. Memsherlock: an automated debugger for unknown memory corruption vulnerabilities. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 562–572, 2007.

[26] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, page 257, 2000.

[27] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, pages 3–17, 2000.

[28] R. Xu, P. Godefroid, and R. Majumdar. Testing for buffer overflows with length abstractions. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 27–38, 2008.

[29] M. Zhivich, T.Leek, and R. Lippmann. Dynamic buffer overflow detection. In *Proceedings of the Workshop on the Evaluation of Software Defect Detection Tools*, 2005.