國立交通大學

資訊科學與工程研究所

碩 士 論 文

CAST:自動化動態軟體驗證工具

CAST: Automatic and Dynamic Software Verification Tool

研 究 生：林友祥

指導教授：黃世昆　教授

中 華 民 國 九 十 八 年 六 月

CAST:自動化動態軟體驗證工具

CAST: Automatic and Dynamic Software Verification Tool

研 究 生：林友祥　　Student：You-Siang Lin

指導教授：黃世昆　　Advisor：Shih-Kun Huang

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Department of Computer and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer and Information Science

June 2009

Hsinchu, Taiwan, Republic of China

中華民國九十八年六月

CAST:自動化動態軟體驗證工具

學生：林友祥　　　　　　　　　　　　　　　　　　　　　　指導教授：黃世昆　教授

國立交通大學資訊科學與工程研究所碩士班

摘要

軟體測試是軟體工程用以確保軟體品質重要的一部分。此外，在程式中自動驗證特性是軟體測試的遠程目標。近年來，結合具體與符號執行(concolic 測試)成為一個眾所周知的方法用來路徑分支測試並且許多研究表明該方法可以結合全域檢查，來找出程式錯誤。在本論文中，我們提出 CAST 的規範語言，建立於 concolic 測試結合全域檢查的基礎上，可以描述各種規格檢查 C 語言程式的安全特性(從另一個角度來看，我們可以將此作為一種駭客攻擊，以獲得接近 exploit 的測試資料)。CAST 是一個自動和動態軟體驗證工具，主要包括樣式匹配，全域檢查和資料流分析，所以可以使我們的全域檢查比一般的 concolic 測試的更加靈活和複雜。

CAST: Automatic and Dynamic Software Verification Tool

Student: You-Siang Lin Advisor: Dr. Shih-Kun Huang

Institute of Computer Science and Engineering
National Chiao Tung University

ABSTRACT

Software testing is an essential part of software engineering for ensuring software quality. Furthermore, automatically verifying properties in programs is a long-time goal in software testing. In recent years, combining concrete and symbolic execution (concolic testing) becomes a well-known approach for branch testing and many researches indicate that the approach can combine with universal checks to find bugs. In this paper, we present the CAST specification language which can describe various kinds of specification for checking security properties of C programs (from another point of view, we can take this as a hack attack to attain test cases close to exploit) based on concolic testing with universal checks. CAST, an automatic and dynamic software verification tool, is mainly composed of pattern matching, universal check and data flow analysis such that we can make universal checks more flexible and complex than that general concolic testing uses.

誌謝

首先誠摯的感謝指導教授　黃世昆老師，老師悉心的教導使我得以一窺軟體品質領域的深奧，不時的討論並指點我正確的方向，使我在這些年中獲益匪淺。老師對學問的嚴謹更是我輩學習的典範。

三年裡的日子，實驗室裡共同的生活點滴，學術上的討論、言不及義的閒扯、讓人又愛又怕的宵夜、趕作業的革命情感........，感謝眾位學長、同學、學弟妹的共同砥礪(墮落?)，你/妳們的陪伴讓三年的研究生活變得絢麗多彩。感謝昌憲、彥佑、立文、揚傑學長們不厭其煩的指出我研究中的缺失，且總能在我迷惘時爲我解惑，也感謝昆翰、文健、彥廷、世宇同學的幫忙，恭喜我們順利走過這段研究生的日子。感謝秋如姊姊和文智姐夫時常對我的關心與不時的加油打氣，並且在我心情沮喪的時候帶我到處散心。感謝資工系踢球的小武、紅幫斬、修齊、僑生以及其他學弟們，在交大踢球的日子真的非常快樂，一起拿下大資盃、系際盃、風竹盃那段光榮日子，到老都值得回憶再三。真的很感謝大家，你們的好我會銘記在心一輩子。

最後，謹以此文獻給我摯愛的雙親。

Contents

List of Figures

List of Tables

1. Introduction

Software validation is the process of evaluating a software system or component in order to determine whether it satisfies specified requirements and it is a very hard problem. Usually, most validation is associated with traditional execution-based testing, that is, exercising code with manually-generated test cases.

Up to now software testing is the primary way to check the correctness and quality of software. Software testing is an essential part of software engineering and it not only costs United States economy tens of billions of dollars annually but also has high economic impacts because of inadequate infrastructure [1]. As software develops vastly and becomes more complex, we can't only rely on manpower to find bugs because of time-consuming and inefficiency. Automation is the lowest requirement and many approaches have been proposed since 70's [2]. Then, we implement a tool, CAST, which can automatically verify a c program with user-defined specifications depending on the power of concolic testing. In addition, combining concolic testing, pattern matching, dynamic data flow analysis and universal checks into CAST provides the ability to describe complex checks for testers who are eager to confirm whether the properties are violated or not.

In this paper, we show that CAST can be easily used to describe properties to automatically exploit wargames from our secure programming course.

1.1. Background

The property checking problem is to check whether a program satisfies a specified safety property. According to the different kinds of property, different techniques are applied and different policies are adopted. Static analysis is complete and fast but imprecise. Dynamic analysis is usually sound and precise but slow. Model checking try to be sound and complete but it can't be scalable. Concolic testing balances the advantage and the disadvantage of these techniques above. We describe these techniques more clearly in Section 2. There are two broad approaches which have been proposed, and we describe them as below.

1.1.1. Universal Checks

Universal checks [3] support several universal symbolic checks which are more powerful than concrete checks since a symbolic check checks all possible values for a given program, whereas a concrete check only checks a specific value. These checks, called “universal”, mean that if the check passes, there is no possible value that the symbolic input could cause the check to fail on this program path.

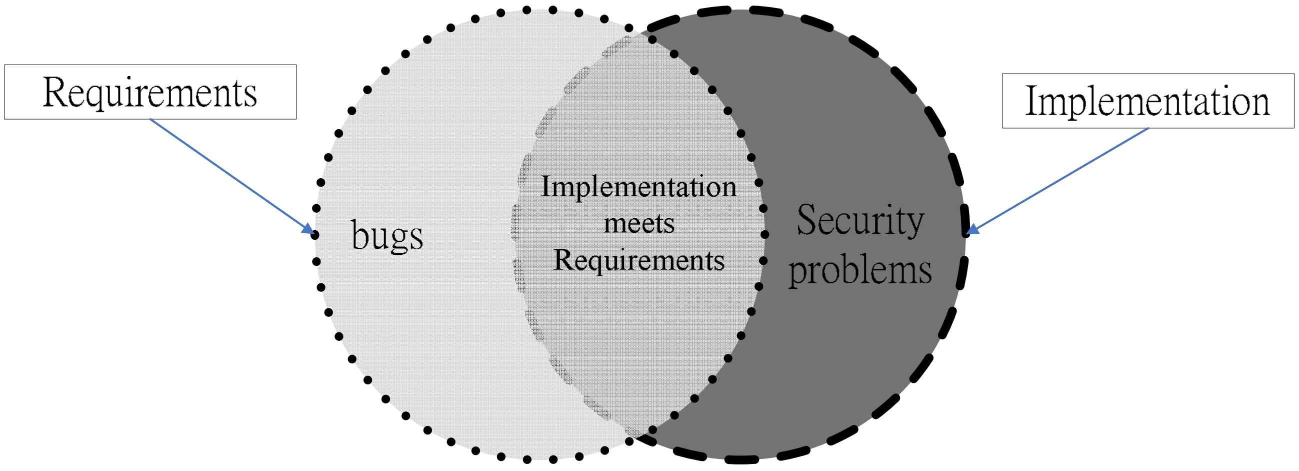


Figure 1: Testing to make sure that the implementation includes the features described in the specification will miss many security problems

As implementation fails to meet a particular requirement in testing, a bug is found, but even testing software functionally, it will miss many security problems because security problems are often not violations of the requirements. The intersection of requirements and implementation shown in Figure 1 [4] is usually fully under testing because concolic testing traverses each execution path of branch, but as we are eager to find bugs or security problems which are not violated requirements, we need additional constraints to guide constraint solver through specific execution path. In order for testing to be wide-ranging, one has to add additional universal checks which can assist us in verifying whether the program fits to our special requirement (specification) or cause some vulnerable errors on each program path.

1.1.2. Dynamic Data Flow Analysis

Dynamic data flow analysis (DDFA) is a useful technique that helps us to decide if a sensitive variable can be tainted from user inputs. DDFA usually maintains a map which records each variable’s address with its taint tag and set its taint tag when tainted variable is assigned to it. However, it encounters some problems with library function call and side effect described in Table 1.

Table 1: DDFA instruments codes for each instruction pattern in C.

Pattern	Instruction/*Instrumentation*	Difficulties

SSA	lhs = rhs1 ***op*** rhs2;
	taint_set_tag(&lhs, &rhs1, &rhs2);
		Easy.

Outer user-defined function call	*taint_push_in_arg*(&arg1);
	taint_push_in_arg(&arg2);
	ret = foo(arg1, arg2);
	taint_set_tag(&ret, taint_pop_out_arg());
		Easy.

Inner user-defined function call	int foo(int form1, int form2){
	  taint_set_tag(&form1, taint_pop_in_tag());
	  taint_set_tag(&form2, taint_pop_in_tag());
	…
	  taint_push_out_arg(&fooret);
	  return fooret;
	}
		Easy.

Library function call	strncpy(dest, src, n);
	taint_copy_buffer(dest, src, n);
		Manually instrument codes according to different library calls.

Side effect (global variable)	lhs = global;
	taint_set_tag(&lhs, &global);
		Imprecision when global variable was set in library call.

When we attempt to find a block of memory to insert shell code, DDFA direct us to find a series of tainted memory.

1.1.3. Static Program Analysis

Static program analysis is automatic, complete, and scalable but imprecise because of false positives. It simulates program running on the abstraction (such as data flow, syntax tree, etc) instead of really running the program on concrete values. Static verification, syntax parser, and traditional symbolic execution are static program analysis techniques. Using some approaches, static program analysis techniques can deal with path explosion problem that concolic testing encounters.

PREfix performs path-sensitive, interprocedural analysis and walks the abstract syntax tree (AST) to explore various execution paths [8]. Then the symbolic execution state is tracked in a virtual machine for checking errors. Avoiding a plenty of false alarms, PREfix adopts some heuristic analyses neither sound nor complete. It makes simplifying approximations that analysis can trace for huge codes of program with incomplete information.

PREfast performs simple, intraprocedural checks on code and it is a “fast” version of the PREfix tool [9]. Based on pattern matching in the abstract syntax tree of the C/C++ program, certain PREfast analyses find simple programming mistakes. PREfix and PREfast take complementary approaches for finding errors. PREfix performs a complete, path-by-path analysis of an entire program to track information across function boundaries. In contrast, PREfast is a lightweight tool that looks for errors locally.

Lint [10] , a c program checker like a compiler, does lexical and syntax analysis on the input text, constructs and maintains symbol tables, and builds trees for expressions. It is one of the earliest static C program checker that can detect many program errors.

LCLint [11] , an extension of Lint, supports different levels of specification and takes advantage of *abstract type* to check errors without affecting the correctness of programs

ESC/Java [12], an application of theorem proving, catches more errors than conventional static checkers such as type checkers and uses an automatic theorem-prover to reason about the semantics of programs, which allows ESC to give static warnings about many errors that are caught at runtime.

The Bandera Tool Set [13], an integrated collection of program analysis, abstraction, and other model-checking tool such as Spin, dSpin, SMV, and JPF, addresses the problem of state explosion, model construction, requirement specification, output interpretation.

Cyclone [14], like a type theory checker, performs a static analysis on source code, and inserts run-time checks into the compiled output at places where the analysis cannot determine that an operation is safe.

1.1.4. Dynamic Program Analysis

It is sound because it runs the program on concrete inputs, and doesn't have a lot of false alarms like static program analysis. The objective of dynamic program analysis is to determine the software quality of a system through collecting information and analyzing the results by running it.

1.1.5. Fuzz Testing (Fuzzing or Random Testing)

It is a software testing technique that provides random data to the inputs of a program and is often used for software development to perform black box testing because of its simple design, high speed, scalability, and free of preconceptions about system behavior. However, fuzz testing usually wastes much time to produce duplicate test inputs because of using pseudo-random number that depends on the seed value. Even if we use other random number generators to avoid the problem, it is better but still slow. [5]. The severest shortcoming is that there is no assurance of coverage for random testing [15]. For example, the probability that the conditional statement of the branch "*if*(**x == 5**){ …}" is 'true' to be executed is $1/2^{32}$ if x is a 32-bit integer input and is randomly generated.

1.1.6. Symbolic Execution

Symbolic execution [16] is useful in the validation of software and can be used to aid in the generation of test input and program proving. Traditionally, it doesn't execute a program with input values but run a program with symbolic values instead. The most common approach to use symbolic execution is to analyze a program by traversing its flow graph from an entry point and collecting a list of assignment statements and branch predicates in terms of symbolic values along a particular path. After gathering these constraints, a test case can be generated by solving the path condition. It fails to scale for large software due to the limitations of underlying theorem proofers and symbolic analyzers [17].

1.1.7. Concolic Testing

Concolic testing combines random testing, concrete and symbolic execution to iteratively generate test inputs to traverse all feasible paths and to partly overcome the limitations of each testing technique. It statically instruments codes according to each instruction and dynamically collects symbolic constraints. Based on dynamic methods for test input generation, Concolic testing doesn't have false positives. Based on symbolic execution, it generates concrete test inputs so that the coverage of concolic testing is better than random testing. The primary disadvantage of concolic testing is that it doesn't scale to large programs because of path explosion problem. The primary advantage of concolic testing is that it is better than pure symbolic execution because of the presence of concrete values, which can be used both to reason precisely about complex data structures and to simplify constraints for the sake of aiding the underlying constraint solver. It is unlike the traditional testing techniques only based on symbolic execution or static analysis.

EGT [18], EXE [7], DART [19], Hybrid [20], CUTE [6] and LATEST [21] perform concolic testing, especially EXE addresses byte-level memory model, Hybrid combines with random testing switch, LATEST abstracts function call to reduce the execution path with the same branch coverage in main function.

1.1.8. Static Data Flow Analysis

Static data flow analysis, a mature technique adopted by complier and some application such as cqual[22], CIL[23], helps compiler generate the optimized assembly code from source code and finds bugs like format string and command injection attack. By statically generating and analyzing AST (Abstract Syntax Tree), efficiency and scalability is its feature, but a large amount of false alarm is its disadvantage.

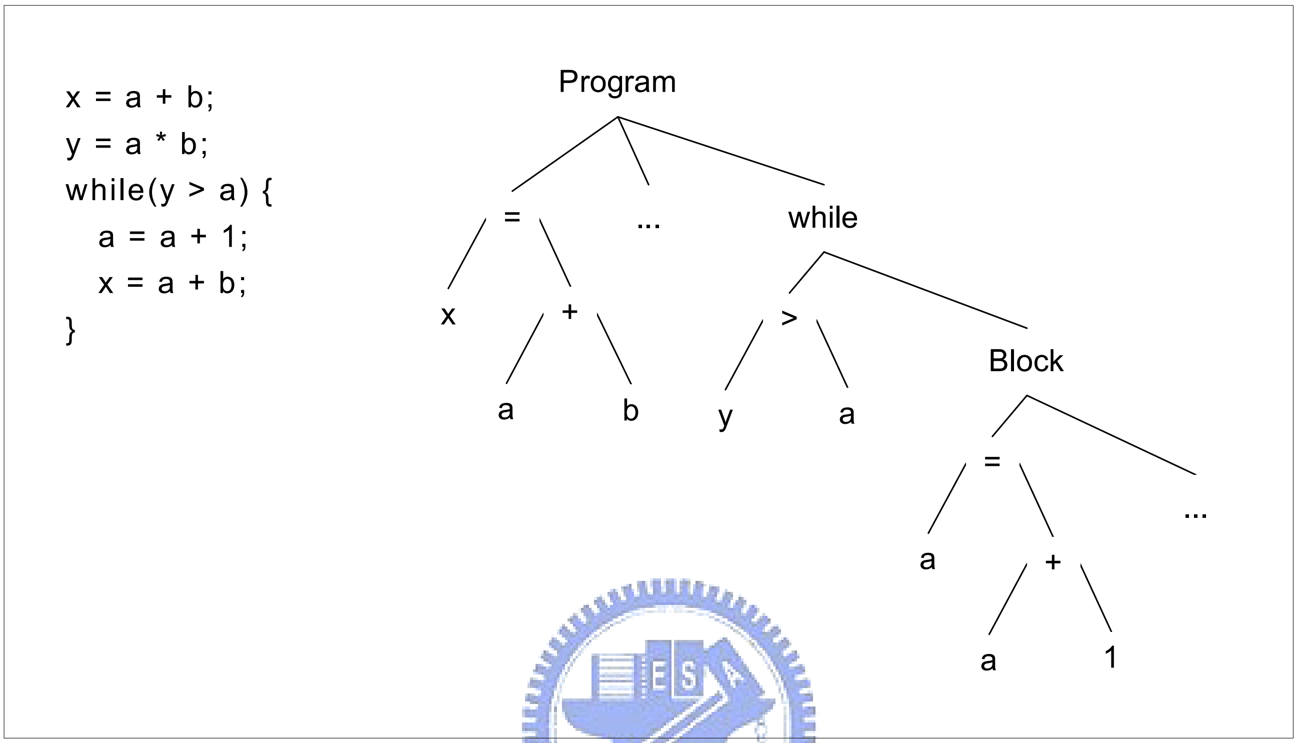


Figure 2: Static data flow analysis is based on abstract syntax tree.

1.2. Common Vulnerabilities

When a software contains a vulnerability, it may be exploitable. As long as a software is exploitable, it allows attackers from an authorized state to any unauthorized states to do harm to users. We introduce common vulnerabilities that cause wargames exploitable in section 3.

1.2.1. Buffer Overflow

Buffer overflow, the most notorious problem in software quality, is one of the OWASP Top 10 2004[25] and it causes many well-known attacks and worms shown in CERT/CC Advisories [26]. By overwriting memory fragments of the process, buffer overflow may control values of the instruction pointer, base pointer and other registers to direct the flow of the process and execute arbitrary codes. Buffer overflow contains stack overflow and heap overflow.

1.2.1.1. Stack Overflow

Local variables are allocated on the stack, along with parameters and linkage information that where to resume execution after a function returns. When data is written outside of the boundaries of the stack buffer, it is called stack overflow. Misuse of strcpy(), strcat(), gets(), sprint() or other analogous, self-implemented functions usually generates this kind of bug.

Figure 3 shows a simple code with the stack overflow caused by *gets* function and Figure 4 shows the graph of the stack. When our input length is over than 256, we can write the data of *b* and further the data of the return address and ebp to control the flow of the process and execute arbitrary code.

```c
#include <stdio.h>
#include <string.h>
void doit(int a)
{
    int b=0;
    char buf[256];
    gets(buf);
    printf("%s %d %d\n", buf, b, a);
}

int main()
{
    printf("Start doit\n");
    doit(10);
    printf("End doit\n");

    return 0;
}
```

Figure 3: An example of stack overflow.

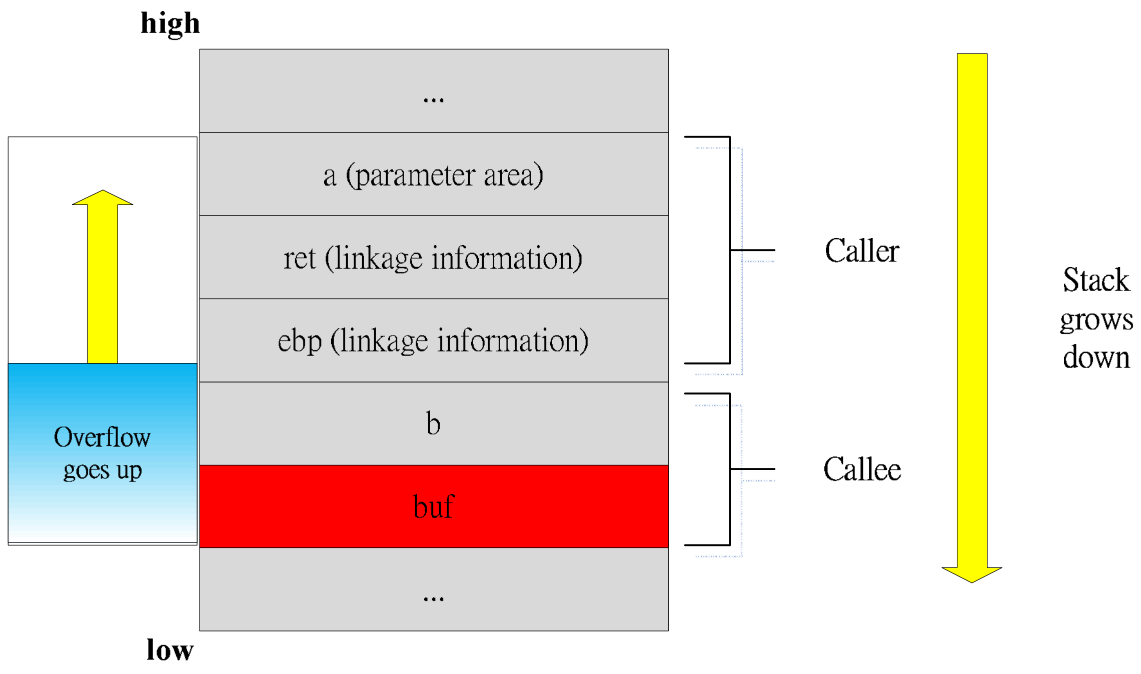


Figure 4: A simple illustration of stack segment in Figure 3.

1.2.1.2. Heap Overflow

Heap overflow is a kind of buffer overflow that occurs in the heap data area. The most difference with stack overflow is that heap overflow is hard to exploit with regard to the implementation of *malloc*() in C or *new* in C++.

1.2.2. Command Injection

The command injection attack is to inject and execute commands specified by the attacker in the vulnerable application. In most cases, it can be injected because of lack of correct input data validation, which can be manipulated by the attacker.

The following simple example accepts a filename as a command line argument, and displays the contents of the file. If the program runs with root privileges, the call to system() also executes with root privileges. If a user specifies a standard filename, the call works as expected. However, if an attacker passes a string of the form ";sh", then the call to system() fails to execute cat due to a lack of arguments and then execute sh with root privileges. Then, the attacker can do anything that a root can do.

```
#include <stdio.h>
#include <unistd.h>

int main(char* argc, char** argv) {
    char cmd[256] = "/usr/bin/cat ";

    strcat(cmd, argv[1]);
    system(cmd);

    return 0;
}
```

Figure 5: An example with command injection vulnerability.

1.3. Motivation

Software bugs or errors are usually introduced by some inputs and cause program not to behave as designers expect. If we obtain inputs which trigger bugs, we can traverse the program in a systematic way to observe what inputs affect. Finding inputs which trigger bugs is important, but the premise is how to define bugs. One common definition of a software error is a mismatch between the program and its specification. Even though the specification is perhaps incorrect and then it induces the testing to find a non-realistic error [5], our target is to help developers obtain the information of what cause violation of specification as soon as possible. Take divide-by-zero as an example. CUTE [6] uses assert function which should be manually inserted into the check point every time when the division happens. Even thought EXE [7] uses universal checking by means of CIL to automatically insert the predicate into the check point, we can't avoid writing an Ocaml module containing our analysis and transformation. As a consequence of the inconvenience, we want to design a C-like specification language to describe what we are eager to verify.

Even though we find bugs or errors in a program, there are often other hurdles that we have to overcome, especially whether or not the vulnerability exists in a program. For this reason, we make our specification language flexible and feasible enough to describe a bug which is more complicated than a bug found by CUTE and EXE and it approaches to the problem causing vulnerability.

1.4. Objective

Our objective is to make a specification language for convenience of defining many properties and verify it by our new concolic testing tool CAST to generate exploits automatically. In order to find a bug which violates some properties and has some specific flow relation in a program, we try to integrate universal checks, static code analysis and dynamic data flow analysis into one.

1.5. A Motivation Example

To illustrate why we need to combine universal checks with dynamic data flow analysis, consider the program in Figure 6.

```c
#include <stdio.h>

char Buffer[256];
int RespondHttpRequest(int Pos)
{
    int kk=0;
    char tmpCad[20]
    char *p1;
    int idx, j=0;

    fgets(Buffer, sizeof(Buffer), stdin);
    p1 = strstr(Buffer, "GET");
    if(p1 == NULL)
        p1 = strstr(Buffer, "Get");
    if(p1 == NULL)
        p1 = strstr(Buffer, "get");
    if(p1 != NULL){
        while(p1[j] != ' ' && p1[j]){
            tmpCad[kk++] = p1[j++];
        }
    }

    ...

    return 0;
}
```

Figure 6: An exploitable example of buffer overflow

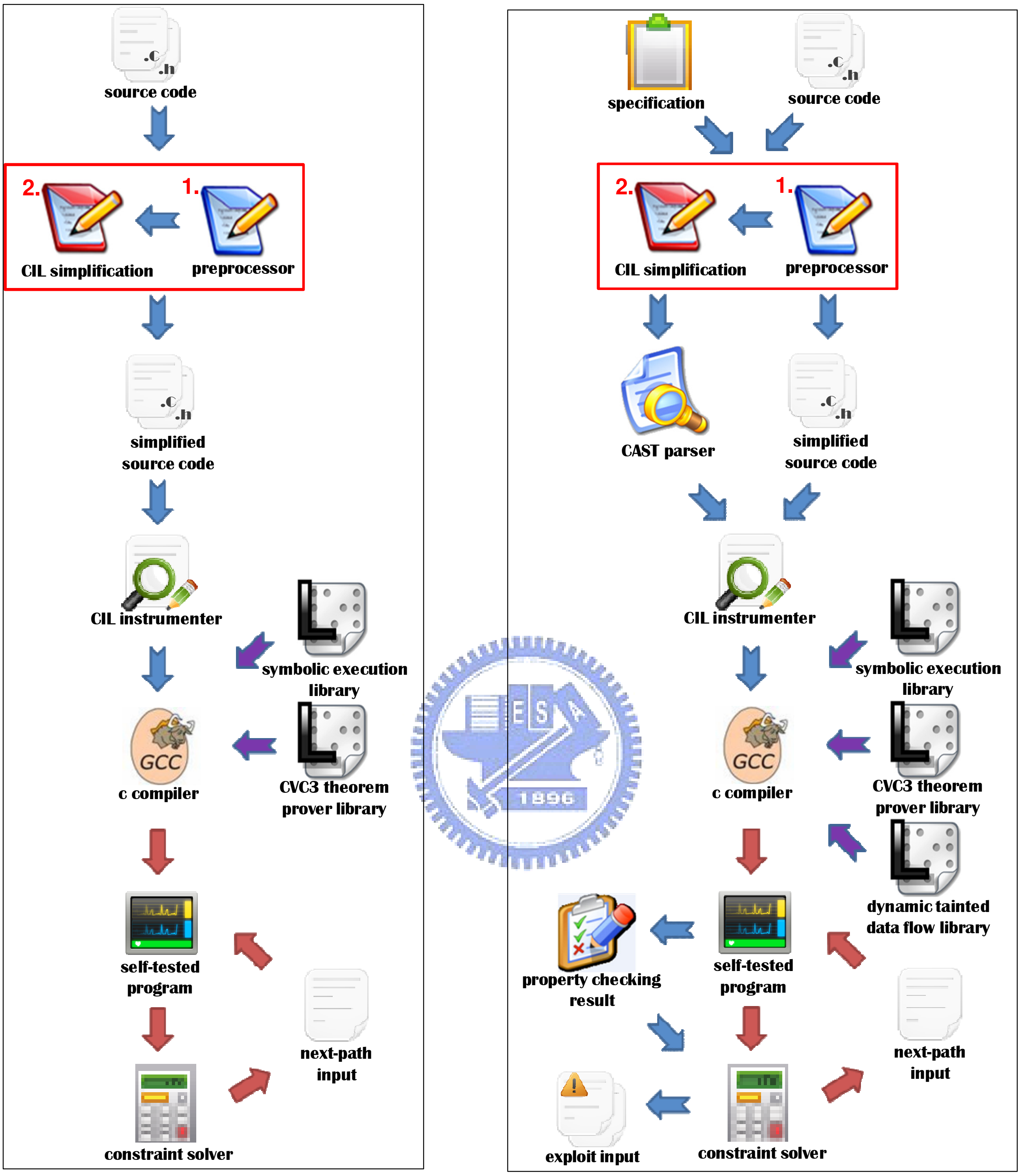
First, we describe the program's objective shortly. It is common to copy a string to a local buffer after parsing. Then a buffer overflow occurs, because programmer forgets to check if the size of the target buffer is big enough. Therefore, in Figure 6, the variable ***kk*** will be set as tainted by dynamic data flow analysis, and universal checks check if ***kk*** is greater than ***tmpCad***'s bound. Traditional concolic testing is usually applied to detect buffer overflow, but it doesn't know how to exploit it. However, we model a new object map and a byte-level symbolic execution to figure out the problem such that DDFA can check if the return address is tainted and if there are a lot of continuous symbolic values stuffed with shell codes.

2. Testing Sequential Programs with CAST Specification

In this chapter, we present an idea to combine concolic testing with a specification language for describing many kinds of properties and focus on testing sequential programs. Concolic testing [6] [17] can automatically explore all of feasible execution paths of sequential programs, where concolic denotes cooperative CONCrete and symbOLIC execution. And we define a specification language to instrument the program at the feasible place where we check security properties with the help of universal check. Our concolic testing tool, CAST modified from ALERT [27, 28], helps us explore all feasible paths, especially those paths causing vulnerability.

2.1. CAST Architecture

We can view our concolic testing tool, CAST modified from ALERT [27-29], as a program path explorer to traverse all path of the execution tree of a program and then check every property described in a specification on each path. The components and the flow chart of CAST are shown in Figure 7 and described in detail as below.



(a)ALERT (b)CAST

Figure 7: ALERT and CAST system architecture flow graph.

2.1.1. CIL Simplification

CIL (**C** **I**ntermediate **L**anguage) [23] is a tool that permits easy analysis and source-to-source transformation of C programs, then we uses CIL to instrument the C program for symbolic execution, universal checks and dynamic data flow analysis. In order to instrument the code conveniently, we use CIL modules to simplify the structure of the code and preserve its semantics. We simply describe the simplification steps and functions as below:

1. Simplified Control Flow (cilly --domakeCFG)

   We use this module to simplify the control flow structures described as below:

   A. Loop statement (while, for and do-while)

      Each loop statement will be changed into a `while(1)` statement with `if-else` statements, `goto` statements and `break` statements.

   B. Branch statement (`if`, `if-elseif-else` and `switch`)

      Each branch statement will be transformed into `if-else` statements.

   C. Predicate of if statement (`if`(A && B), `if`(A || B); A,B are C expression)

      Each predicate of if statement will be divided into many if-else statements that each predicate of them is a binary relation expression (such as `if`(x==y); x,y are variables).

2. Simple Three-Address (cilly --dosimplify)

   The `simplify.ml` module simplifies the expressions of a program to gives us a form of three-address code. Moreover, all `sizeof` and `alignof` forms are turned into constants. Accesses to arrays and structures whose address is taken are turned into "Memory" accesses and all index and field computations are turned into address arithmetic.

3. Simple Memory (cilly --dosimpleMem)

   The `simplemem.ml` module makes CIL lvalues that contain more than one memory access even further simplified via the introduction of well-typed temporary variables. After this simplification, all CIL lvalues involve at most one pointer dereference.

4. One Return (cilly --dooneRet)

   The `oneret.ml` module transforms each function body to have at most one return statement.

2.1.2. CAST Parser

CAST parser, written in Ocaml, reads specification and translates it into CIL instrumenter to statically analyze source codes for instrumenting specification check (a complex universal check) at appropriate position.

2.1.3. CIL Instrumenter

CIL instrumenter statically analyzes simplified source codes to instrument it with our dynamic analysis functions according to each instruction pattern. We show every kind of instruction with instrumented codes in Table 2.

Table 2: CIL instrumenter instruments three-address code.

Pattern	**Instruction**/*Instrumentation*

One-Address	*_sqPush*(para1, T_INT, (unsigned int)(& para1));
	_sqPush(para2, T_INT, (unsigned int)(& para2));
	_sqPush(para3, T_INT, (unsigned int)(& para3));
	function(para1, para2, para3)

Two-Address	*_sqPush*(para1, T_INT, (unsigned int)(& para1));
	_sqPush(para2, T_INT, (unsigned int)(& para2));
	_sqPush(para3, T_INT, (unsigned int)(& para3));
	LHS = **function**(para1, para2, para3);
	_sqPopReturnValue(T_INT, (unsigned int)(& LHS));

	LHS = RHS;
	_sqSymExec(T_INT, (unsigned int)(& LHS), OP_NOP,
	"RHS", (unsigned long long)RHS, T_INT, (unsigned int)(&RHS),
	"SQ_constant", 0, T_INT, 0, T_NON);

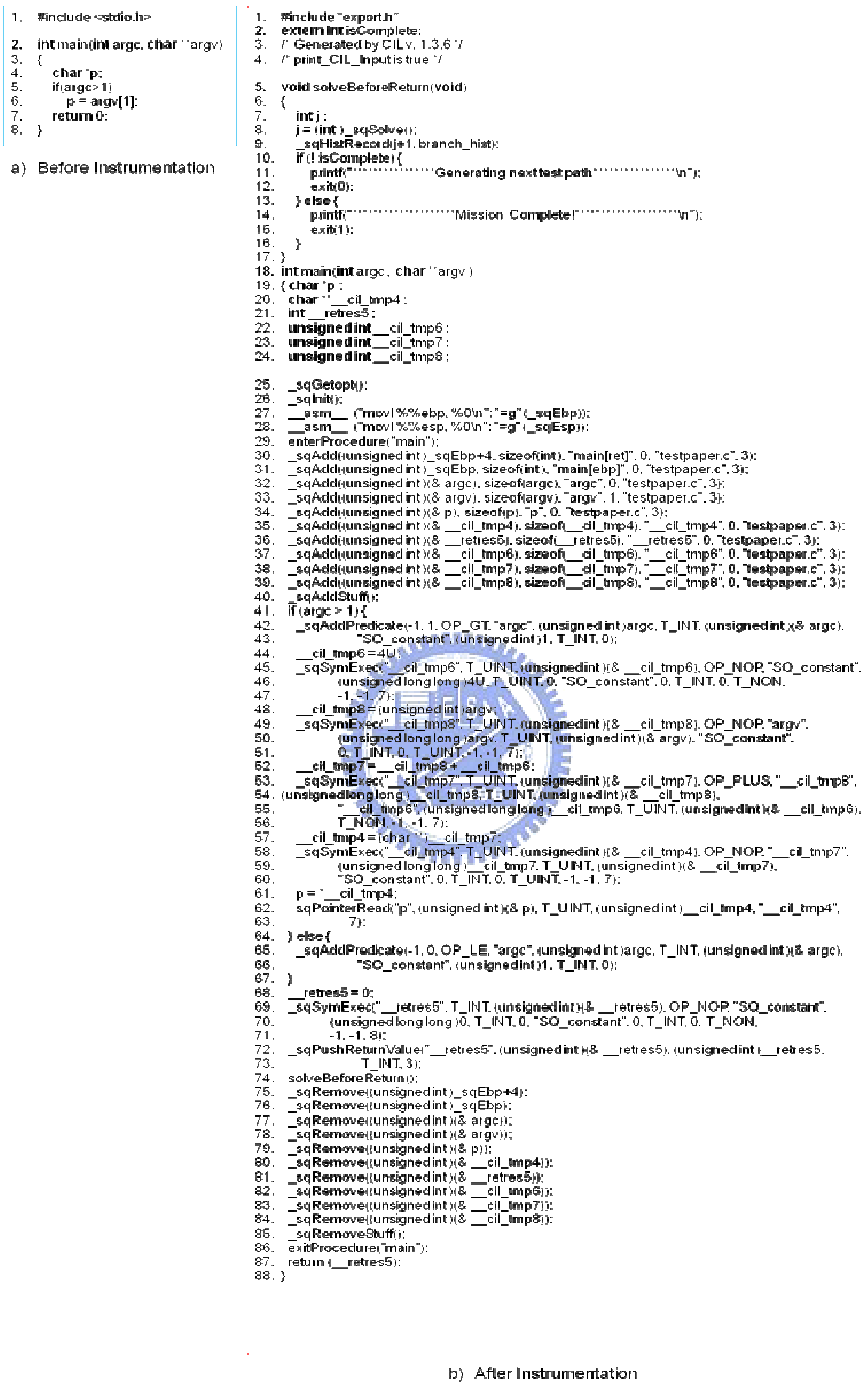
	LHS = (***long***)RHS; *_sqSymExec*(T_INT, (unsigned int)(& LHS), OP_NOP, "RHS", (unsigned long long)RHS, T_INT, (unsigned int)(&RHS), "SQ_constant", 0, T_INT, 0, ***T_LONG***);

	LHS = *RHS;
	sqPointerRead((unsigned int)(&LHS), T_INT,
	(unsigned int)RHS, (unsigned int)(&RHS));

	*LHS = RHS;
	sqPointerWrite((unsigned int)LHS, (unsigned int)(& LHS), T_INT,
	(unsigned long long)RHS, (unsigned int)(& RHS));

Three-Address	LHS = Operand1 ***op*** Operand2; *_sqSymExec*(T_INT, (unsigned int)(& LHS), OP_PLUS, "Operand1", (unsigned long long)Operand1, T_INT, (unsigned int)(& Operand1), "Operand2", (unsigned long long)Operand2, T_INT, (unsigned int)(& Operand2), T_INT);
op: +, -, *, /, etc	

These instrumented functions trace data flow and collect symbolic constraints to dynamically make symbolic execution based on operands' type, operator, and operands' name got from object map by operand's address. In Figure 8, a simple C program is simplified to three-address codes with a lot of CIL temporary variables for the convenience of instrumentation. Because of CIL simplification, pattern matching of CAST to instrument according to a specification has some difficulties we discuss in section 2.3.1.



```
#include <stdio.h>

int main(int argc, char **argv)
{
    char *p;
    if(argc>1)
        p = argv[1];
    return 0;
}
```

a) Before Instrumentation

```
#include "export.h"
extern int isComplete;
/* Generated by CIL v. 1.3.6 */
/* print_CIL_Input is true */

void solveBeforeReturn(void)
{
    int j ;
    j = (int )_sqSolve();
    _sqHistRecord(j+1, branch_hist);
    if (! isComplete){
        printf("****************Generating next test path****************\n");
        exit(0);
    } else{
        printf("*******************Mission Complete!*******************\n");
        exit(1);
    }
}
int main(int argc , char **argv )
{ char *p ;
  char **__cil_tmp4 ;
  int __retres5 ;
  unsigned int __cil_tmp6 ;
  unsigned int __cil_tmp7 ;
  unsigned int __cil_tmp8 ;

  _sqGetopt();
  _sqInit();
  __asm__ ("movl %%ebp, %0\n":"=g" (_sqEbp));
  __asm__ ("movl %%esp, %0\n": "=g" (_sqEsp));
  enterProcedure("main");
  _sqAdd((unsigned int )_sqEbp+4, sizeof(int), "main[ret]", 0, "testpaper.c", 3);
  _sqAdd((unsigned int )_sqEbp, sizeof(int), "main[ebp]", 0, "testpaper.c", 3);
  _sqAdd((unsigned int )(& argc), sizeof(argc), "argc", 0, "testpaper.c", 3);
  _sqAdd((unsigned int )(& argv), sizeof(argv), "argv", 1, "testpaper.c", 3);
  _sqAdd((unsigned int )(& p), sizeof(p), "p", 0, "testpaper.c", 3);
  _sqAdd((unsigned int )(& __cil_tmp4), sizeof(__cil_tmp4), "__cil_tmp4", 0, "testpaper.c", 3);
  _sqAdd((unsigned int )(& __retres5), sizeof(__retres5), "__retres5", 0, "testpaper.c", 3);
  _sqAdd((unsigned int )(& __cil_tmp6), sizeof(__cil_tmp6), "__cil_tmp6", 0, "testpaper.c", 3);
  _sqAdd((unsigned int )(& __cil_tmp7), sizeof(__cil_tmp7), "__cil_tmp7", 0, "testpaper.c", 3);
  _sqAdd((unsigned int )(& __cil_tmp8), sizeof(__cil_tmp8), "__cil_tmp8", 0, "testpaper.c", 3);
  _sqAddStuff();
  if (argc > 1) {
    _sqAddPredicate(-1, 1, OP_GT, "argc", (unsigned int)argc, T_INT, (unsigned int)(& argc),
                 "SO_constant", (unsigned int)1, T_INT, 0);
    __cil_tmp6 = 4U;
    _sqSymExec("__cil_tmp6", T_UINT, (unsigned int )(& __cil_tmp6), OP_NOP, "SO_constant",
             (unsigned long long )4U, T_UINT, 0, "SO_constant", 0, T_INT, 0, T_NON,
             -1, -1, 7);
    __cil_tmp8 = (unsigned int )argv;
    _sqSymExec("__cil_tmp8", T_UINT, (unsigned int )(& __cil_tmp8), OP_NOP, "argv",
             (unsigned long long )argv, T_UINT, (unsigned int )(& argv), "SO_constant",
             0, T_INT, 0, T_UINT, -1, -1, 7);
    __cil_tmp7 = __cil_tmp8 + __cil_tmp6;
    _sqSymExec("__cil_tmp7", T_UINT, (unsigned int )(& __cil_tmp7), OP_PLUS, "__cil_tmp8",
(unsigned long long )__cil_tmp8, T_UINT, (unsigned int )(& __cil_tmp8),
             "__cil_tmp6", (unsigned long long )__cil_tmp6, T_UINT, (unsigned int )(& __cil_tmp6),
             T_NON, -1, -1, 7);
    __cil_tmp4 = (char **)__cil_tmp7;
    _sqSymExec("__cil_tmp4", T_UINT, (unsigned int )(& __cil_tmp4), OP_NOP, "__cil_tmp7",
             (unsigned long long )__cil_tmp7, T_UINT, (unsigned int )(& __cil_tmp7),
             "SO_constant", 0, T_INT, 0, T_UINT, -1, -1, 7);
    p = *__cil_tmp4;
    sqPointerRead("p", (unsigned int )(& p), T_UINT, (unsigned int )__cil_tmp4, "__cil_tmp4",
               7);
  } else {
    _sqAddPredicate(-1, 0, OP_LE, "argc", (unsigned int)argc, T_INT, (unsigned int)(& argc),
                 "SO_constant", (unsigned int)1, T_INT, 0);
  }
  __retres5 = 0;
  _sqSymExec("__retres5", T_INT, (unsigned int )(& __retres5), OP_NOP, "SO_constant",
          (unsigned long long )0, T_INT, 0, "SO_constant", 0, T_INT, 0, T_NON,
          -1, -1, 8);
  _sqPushReturnValue("__retres5", (unsigned int )(& __retres5), (unsigned int )__retres5,
                T_INT, 3);
  solveBeforeReturn();
  _sqRemove((unsigned int)_sqEbp+4);
  _sqRemove((unsigned int)_sqEbp);
  _sqRemove((unsigned int)(& argc));
  _sqRemove((unsigned int)(& argv));
  _sqRemove((unsigned int)(& p));
  _sqRemove((unsigned int)(& __cil_tmp4));
  _sqRemove((unsigned int)(& __retres5));
  _sqRemove((unsigned int)(& __cil_tmp6));
  _sqRemove((unsigned int)(& __cil_tmp7));
  _sqRemove((unsigned int)(& __cil_tmp8));
  _sqRemoveStuff();
  exitProcedure("main");
  return (__retres5);
}
```

b) After Instrumentation

Figure 8: A simple C example with CAST instrumentation.

2.1.4. Self-Tested Program

When making a program to test it by itself, we should figure out the problem of how to input a value generated from constraint solver. As long as a variable is symbolic, that means its tainted origin comes from user's setting (make_symbolic(&i, T_INT)) or ***stdin*** function like read(0, buf, size), fgets(buf , size , stdin), etc. If symbolic variable is set by user, we write its value into a file and read the file next time to make program to run next path. Otherwise, we write it into a file and pipe the file to the program next time to make program to run next path. Writing a shell script to automatically run self-tested program is easy to make all the process automatic.

2.2. Implementation

We show what CAST modifies ALERT to make it possible to test a vulnerable program and find exploits in this section. First, we simply describe symbolic execution applied to constraint solver; then secondly depict how to utilize symbolic execution to model a tested program into byte level precision. Thirdly, how to maintain an object map for recording sensitive data is shown in section 2.2.3. Fourth, the most important issue about symbolic pointer dereference shown in section 2.2.4 discuss how to query constraint solver for extra execution paths.

2.2.1. Symbolic Execution

Automatic theorem prover such as CVC3[30] we use in this paper, a tool to solve *Satisfiability Modulo Theories (SMT)* problems, is the kernel of symbolic execution such that symbolic execution is limited by the power of underlying theorem prover. We will explain how symbolic execution works in Figure 9 and turn it into CVC3 file format (.cvc) to experiment for finding why it fails without symbolic execution.

```
1   void testme(int i)
2   {
3        if(i>100)
4             i=i+1;
5   }
```

(a)A simple C program

```
1   i: INT;
2
3   ASSERT i > 100;
4   ASSERT i = i + 1;
5
6   QUERY  FALSE;
7   COUNTEREXAMPLE;
8   COUNTERMODEL;
```

(b)Wrong transformation from (a).

```
1   i1,i2: INT;
2
3   ASSERT i1 > 100;
4   ASSERT i2 = i1 + 1;
5
6   QUERY  FALSE;
7   COUNTEREXAMPLE;
8   COUNTERMODEL;
```

(c)Right transformation from (a)

```
1   i1,i2: INT;
2
3   ASSERT i2 > 100;
4   ASSERT i2 = i1 + 1;
5
6   QUERY  FALSE;
7   COUNTEREXAMPLE;
8   COUNTERMODEL;
```

(d)Ambiguous transformation from (b)

Figure 9: How to model a C program with symbolic execution.

When not using symbolic execution in Figure 9(a), we collect the constraint formula is (i>100 ***AND*** i=i+1) and then find that we can't make the difference between (i>100 ***AND*** i=i+1) that means (if(i>100) i=i+1;) and (i=i+1 ***AND*** i>100) that means (i=i+1; if(i>100);). Therefore, to verify this, we turn the formula (i>100 AND i=i+1) to CVC3 query language shown in Figure 9(b) and call theorem prover, then get the error message "symbolic.cvc:7: this is the location of the error". It, further, means that the solver doesn't know the formula is the case of Figure 9(c) (i_1>100 AND i_2=i_1+1) or Figure 9(d) (i_2>100 AND i_2=i_1+1), whereas after applying symbolic execution into Figure 9(a), we have the constraint formula (i_1>100 AND i_2=i_1+1) which is the interpretation of the path of the program in Figure 9(a) that input i passes through line 3 and line 4.

2.2.2. Symbolic Name and Symbolic Byte Name

As ALERT, CUTE and CREST [31] adopt symbolic execution to query constraint solver, the problem of type size in C comes with it. In Figure 10, the variable ***a*** is equal to ***a4@a3@a2@a1*** (where "@" denotes bitvector concatenation, and we use little-endian order for multi-byte values) due to buffer overflow that sets ***a1***, ***a2***, ***a3***, ***a4*** to be symbolic inputs. ALERT collect symbolic expression (***buf[4]***_0 = ***a1***_0 && ***buf[5]***_0 = ***a2***_0 && ***buf[6]***_0 = ***a3***_0 && ***buf[7]***_0 = ***a4***_0) from line 13 to line 16 and try to query (***a***_0 = 5566) at line 17. Then it fails to generate inputs of the next path, because it doesn't make the equation of ***a***_0 and ***buf[7]***_0@***buf[6]***_0@***buf[5]***_0@***buf[4]***_0. CUTE and CREST use the same symbolic expression memory model to map &***buf[4]*** to ***a1*** at line 13 and to get ***a1*** from finding &***a*** in the map at line 17 to ask constraint solver if ***a1*** = 5566. Then, they can't generate next path inputs because the symbolic expression memory map model is not byte-sensitive enough to get the correct symbolic expression ***a4@a3@a2@a1***=5566. After running the case in CUTE and CREST, we make the conclusion as above.

```
1   #include <crest.h>
2   #include <stdio.h>
3
4   int main(void)
5   {
6       char a1, a2, a3, a4;
7       int a;
8       char buf[4];
9       make_symbolic_char(&a1);//make a1 as symbolic char input
10      make_symbolic_char(&a2);
11      make_symbolic_char(&a3);
12      make_symbolic_char(&a4);
13      buf[4] = a1;
14      buf[5] = a2;
15      buf[6] = a3;
16      buf[7] = a4;
17      if(a==5566)
18          fprintf(stderr,"a==5566\n");
19      else
20          fprintf(stderr,"a!=5566\n");
21  }
```

Figure 10: A C program with symbolic name problem.

Instead of traditional symbolic name, CAST makes all variable into symbolic byte name and check each byte of variable ***a*** by recording each bytes into symbolic expression memory map. Therefore, we can assert the formula (getByteExpr((char *)&a+3) @ getByteExpr((char *)&a+2) @ getByteExpr((char *)&a+1) @ getByteExpr((char *)&a) = 5566) to ask the correct symbolic expression (***a4_0@a3_0@a2_0@a1_0***=5566). EXE claims that they implement byte-level precision, but we can't get the program to test the case.

2.2.3. Object Map

CAST implements an object map like CRED[32], which dynamically records the information of all variables, especially ebp and return address of each function about its memory address and type size that help us make extra symbolic assignment constraints by _sqSymExec() and symbolic branch constraints by _sqAddPredicate() to query not only precise but also exploitable inputs. We add/remove the object information of every function into object map when program enter/exit each function. Further, adding sensitive data such as global offset table, return address, etc into object map provides a chance to calculate more exploitable paths. Avoiding getting unknown objects from buffer overflow, we fill every stack frame with stuff objects by sqAddStuff() to make sure that symbolic execution works as usual.

```
unsigned *sqEbp;

int testme(int a)
{
    asm("movl %%ebp, %0\n": "=g" (sqEbp));
    sqAdd((unsigned int) sqEbp+4, sizeof(int), "testme[ret]");
    sqAdd((unsigned int) sqEbp, sizeof(int), "testme[ebp]");
    sqAdd((unsigned int )(&a), sizeof(a), "a");
    sqAddStuff();

    a=a+1;

    sqRemove((unsigned int)_sqEbp +4);
    sqRemove((unsigned int) sqEbp;
    sqRemove((unsigned int)(&a));
16      sqRemoveStuff();
17
18      return a;
19  }
```

Figure 11: Instrumentation of inserting and removing object map.

GCC provides a large number of built-in functions for programmers to observe program, such as __builtin_frame_address() and __builtin_return_address(). When using these functions to get ebp and return address, we encounter some problems about its imprecision after calling some standard library. For the sake of getting precise ebp to solve the problem of imprecision, we write assembly code of the line 5 in Figure 11 such that an extra global variable sqEpb is necessary to record current ebp. Therefore, CAST can dynamically check the address from start to end of each frame to ensure that each byte is added into object map for symbolic byte name execution.

2.2.4. Symbolic Pointer Read/Write with Symbolic Value

One of the most notorious forms of attack to C program is buffer overflow which generates extra execution paths in which we are interested. Table 3 shows the cases that cause different kinds of buffer overflow with different exploitable levels of vulnerabilities. Here, we assume that variable ***a***, ***b[i]***, ****p*** and ***i*** are integer types. In the case (1), the variable ***a*** can be any value in the memory such that it has an influence on the branches directed by the variable ***a***. The case (2) means that we can write 4 bytes at any memory location with value ***a***. The most dangerous behavior of a program is the case (3) that users can write any value into any memory location, especially certain sensitive data like return address.

Table 3: The cases of buffer overflow cause different vulnerabilities to a C program.

Case	Pattern	Concrete	Symbolic	Exploitable
(1)Symbolic Buffer Index/Symbolic Pointer Reading	a = b[i];	B	i	possible
	a = *p;		p	possible
(2) Symbolic Buffer Index/Symbolic Pointer	b[i] = a;	b, a	i	possible

Writing	*p = a;	a	p	possible
(3)Symbolic Buffer Index/Symbolic Pointer Writing with Symbolic Assignment	b[i] = a;	b	i, a	achievable
	*p = a;		p, a	achievable

Since CUTE and CREST attempt to efficiently traverse all feasible execution paths in a program, extra paths are not handled after bugs execution shown in Table 3, in the mean time they also cannot achieve printf("GOAL") in Figure 12. ALERT and EXE, however, handle it as a normal access to any element of buffer, so they can achieve all branches except printf("sp is written by buffer overflow") at line 17 and line 19.

```
1   int main()
2   {
3       int i, sp=0, a[4]={1,2,3,4};
4       make_symbolic_int(&i);//make i symbolic
5       a[i]=10;
6       if(a[0]==10)
7           printf("GOAL");
8       if(a[0]==1)
9           printf("1");
10      if(a[1]==3)
11          printf("2");
12      if(a[2]==5)
13          printf("3");
14      if(a[3]==2)
15          printf("4");
16      if(a[4]==10)
17          printf("sp is written by buffer overflow");
18      if(sp==10)
19          printf("sp is written by buffer overflow");
20
21      return a;
22  }
```

Figure 12: A C program of buffer accesses with symbolic index.

Encountering a[i]=10 at line 5, EXE generates a big disjunction constraint (i == 0 && a[0] == 10) || (i == 1 && a[1] == 10) || (i == 2 && a[2] == 10) || (i == 3 && a[3] == 10). CAST generates a bigger disjunction constraint (i == 0 && a[0] == 10) || (i == 1 && a[1] == 10) || (i == 2 && a[2] ==

10) || (i == 3 && a[3] == 10 || … || (i==n && a[n]==nearest_return_address)) than EXE's constraint for the purpose that we can explore extra execution path. Thus, printf("sp is written by buffer overflow") at line 17 and line 19 can be achieved by CAST.

2.2.5. Use Object Map to Construct Shellcode Constraints

When a symbolic pointer assigned with symbolic value taints a return address, we want to find a continuous memory large enough to allocate shellcode and make return address to point the start of the shellcode.

```
void vc_mkShellCodeExpr(char *shellcode){
    vector<object>::iterator objmap_it;
    vector<object>::iterator omtmp_it; /* objmap temp iterator */
    for(objmap_it = objmap.begin(); objmap_it != objmap.end(); objmap_it++){
        omtmp_it = objmap_it;
        for(i=0; i< strlen(shellcode); i++){ // check if   there are continuous memory
            if(omtmp_it != objmap.end())
                andExpr &= (omtmp_it->expr = shellcode[i]);
            omtmp_it++;
        }
        orExpr |= (andExpr & retAddrExpr == objmap_it);
    }
    vc->assertFormula(orExpr); /* universal check for shellcode */
    vc->query(vc->falseExpr())); /* constraint solver check if it is valid */
}
```

Figure 13: Pseudo code of generating shellcode constraints.

2.2.6. Dynamic Tainted Data Flow for Function Call

When testing C programs by concolic testing, we must pass symbolic execution and tainted data flow to the parameter of function call. Instrumentation in Table 4 shows how to instrument a function when entering and exiting in order to make the equation of arg1 = para1 and LHS = ret_val.

Table 4: CIL instrumenter instruments functions code.

Case	**Instruction**/*Instrumentation*

(1)One-Address	*_sqPush*(arg1, T_INT, (unsigned int)(& arg1));
	_sqPush(arg2, T_INT, (unsigned int)(& arg2));
	_sqPush(arg3, T_INT, (unsigned int)(& arg3));
	function(arg1, arg2, arg3)

(2)Two-Address	*_sqPush*(arg1, T_INT, (unsigned int)(& arg1));
	_sqPush(arg2, T_INT, (unsigned int)(& arg2));
	_sqPush(arg3, T_INT, (unsigned int)(& arg3));
	LHS = **function**(arg1, arg2, arg3);
	_sqPopReturnValue(T_INT, (unsigned int)(& LHS));

(3)Function	int **function**(int para1, int para2, int para3){
	    __parameterToSymbolic((unsigned int)(& para1), T_INT);
	    __parameterToSymbolic((unsigned int)(& para2), T_INT);
	    __parameterToSymbolic((unsigned int)(& para3), T_INT);
	    …
	    _sqPushReturnValue("ret_val", (unsigned int)(& ret_val), (unsigned int)ret_val, T_INT);
	        return ret_val;
	}

2.2.7. Standard Library Testing by Using uclibc instead of glibc

A C program usually calls library function, but when it is tested by concolic testing, library function is not instrumented like case 3 in Table 4. So we cannot collect the correct symbolic execution and tainted data flow after calling library function. The approach proposed by ALERT[27] assumes that constraints are generated by our human logical reasoning without instrumentation such that it is fast in concolic testing but time-consuming for us to reason a library function. Instead of manually reasoning library, we instrument library with source code. We instrument *uclib* rather than *glibc* because uclibc is tiny and easy for debugging. Take function *strcmp* in string.h header file as an example, we use CIL to modify its name to *alert_strcmp* for each tested source code such that testing a C program calling standard library is just complied with uclibc object file compiled by gcc and instrumented by CAST.

2.2.8. Post-condition for standard library *fgets* and *read* from stdin

In the cause of testing programs with fgets and read from stdin, we apply the approach [27] to make post-condition constraints for fgets and read. Instead of instrumenting each stdin library function, we instrument each fgets and read from stdin with _sqPostFgets() and _sqPostRead for making symbolic byte name to each destination buffer.

2.2.9. Environment

Table 5 shows the environment and tools CAST uses.

Table 5: Environment of CAST

Operating System	ubuntu-7.04-desktop-i386
Memory	1.5G
CPU	Intel(R) Core(TM)2 CPU 6300 @ 1.86GHz
Compiler	gcc 4.1.2
Instrumenter	CIL
Theorem Prover	CVC3

2.3. Syntax of Specification

Check properties are expressed using observer concolic testing with universal checks. These provide a way to specify vulnerable properties of C programs based on syntactic pattern matching of C code. The definition of an observer concolic testing consists of a set of declarations, each defining an observer global variable, or a property. Figure 14 gives the grammar for specifying observer concolic testing.

Observer:	**DeclSeq**
DeclSeq:	**Declaration | DeclSeq Declaration**
Declaration:	**"GLOBAL"** CVarDef **| "PROPERTY" '{'** **Temporal** **"LOCATION" { #line num "filename" }** **"PATTERN" '{'** CStmt **'}'** **"PRIORITY" '{'** CIng **'}'** **"ASSERT" '{'** CCondExpr **'}'** **"SHELLCODE" '{'** CString **'}'** **'}'**
Temporal:	**"BEFORE" | "AFTER" | ""**
SVar :	**'$'Num**
Num :	**1~65535**

Figure 14: The grammar of the CAST specification language

Observer global variable is instrumented into codes to analyze and determine some properties. Each observer global variable may have any C primitive type, and is declared following the keyword **GLOBAL**, where the nonterminal CVarDef stands for any C variable declaration.

Each property observes all program steps, if a pattern is matched, specifies what the property checks. The keyword **PROPERTY** is followed by up to five parts: a temporal qualifier, a priority, a location, a pattern, an assertion, and a shellcode. If more than one pattern matches, then CAST instrument the program with assertion at each matched point according to **Temporal** qualifier. **Temporal** qualifier is either **BEFORE** or **AFTER**. It specifies whether the assertion is checked before or after the source code that matches the pattern. The keyword **PRIORITY** is followed by a

C integer taken as the serious level to fix. The keyword **ASSERT** is followed by a C condition expression taken as a formula checked by universal checks. At each universal check point in the program execution, the observer checks the current program constraints formula with assertion formula. The keyword **PATTERN** is followed by a C statement that is matched against the program source code. The pattern is defined by the nonterminal **CStmt**, not a real C statement; it now only supports instruction list. Sometimes, there is no pattern to match, therefore the keyword **LOCATION** followed by a line number and a file name specifies where to be instrumented with universal checks. The keyword **SHELLCODE** is followed by a C string that is generated by Metasploit [33] to be executed after that any return address is overwritten. Finding if there are continuous symbolic memories to allocate this string is easy to query object map and make symbolic equation with each symbolic bytes and the shellcode.

2.3.1. Pattern Matching

During the execution of concolic testing we calculated a number of statistics on the performance of the pattern matching that resulted in the following interesting observations:

- Instruction set (a = b + c) dominates all others. In most cases they accounted for more than 85% of all instructions.
- Function call and other instruction sets (a = b op c, where op is not '+') are the other remaining.

Thus, in order for pattern matching to be efficient, CAST scans the program to match the less frequency instruction set, if any, of the pattern first. Then it sequentially searches the other instruction sets from the position of the matched instruction set backward and forward. In Figure 15, the steps of searching and matching the instruction follow the number in the middle of single arrow in increasing order. The multiplication of the expression like ($1*2) rarely appears in the source code; then we search it first. The dotted arrow line means it is unmatched, otherwise it is matched.

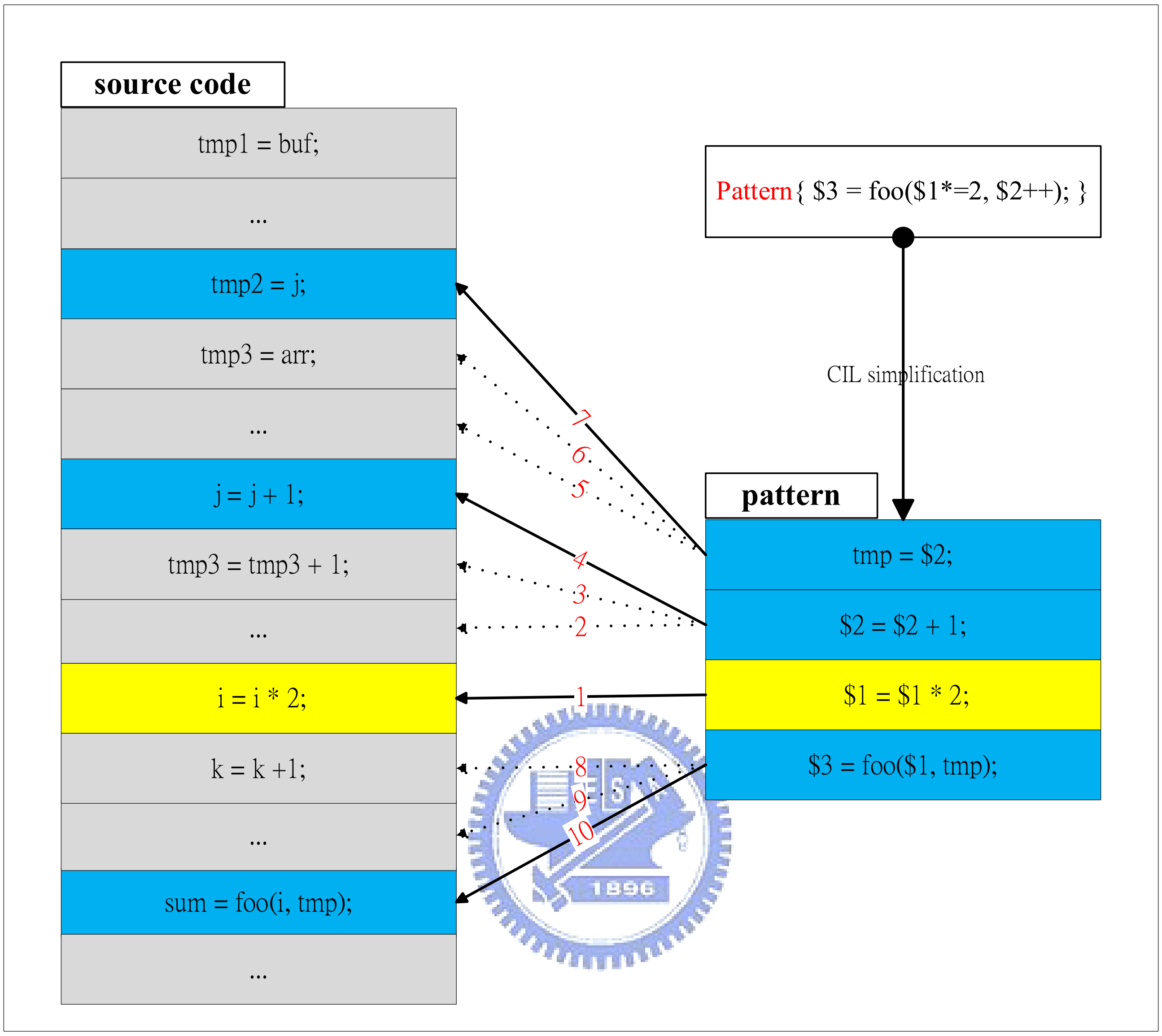


Figure 15: The method of efficient pattern matching.

As CIL simplifies source codes into three-address codes, pattern might as well be simplified such that pattern matching will be precise. As downside, CIL simplification increases times of pattern matching. On the other hand, it makes simple semantic pattern matching.

2.3.2. Universal Check (Specification Check Instrumentation)

Intuitively, at each point in the program execution, the observer checks the current program statement (i.e., AST node) being executed according to the **ASSERT** of each property. A universal check, used for checking specification assertion, we call it specificationCheck given in Figure 16.

The function specificationCheck instrumented by pattern matching can check complex logical expressions such as integer overflow. We will show more other specifications that check vulnerabilities and generate exploits in the section 5.

```
int main()
{
    int a, b, c;

    c = a + b;
    specificationCheck("(a>=0 && c<b) || (a<0 && c>b)”, "c a b",
"T_INT T_INT T_INT", 3, (long long)c, (long long)a, (long long)b,
&c, &a, &b);
    return 0;
}
```

Figure 16: Check integer overflow with universal checks.

2.4. Semantic of specification

The semantics of a trace property is given by running the concolic testing with the program. CAST accepts a trace property every time a pattern matches, the corresponding universal check is valid, and moreover, if shellcode is filled, then the string of the shellcode stuffs itself with the program memory.

2.4.1. Specification of Integer Overflow

Because our pattern matching is type-sensitive, type of each variable should be declared first. A post-condition of integer overflow in addition shown in **guard** block will be checked after each integer addition like the **pattern**.

```
svar int $1;
svar int $2;
svar int $3;

property {
    after
    pattern { $1 = $2 + $3; }
    guard { ($2>=0 && $1<$3) || ($2<0 && $1>$3) }
}
```

Figure 17: A specification checks integer overflow.

3. Experiment

We will discuss the specification of test cases that come from the course of secure programming in this section. These test cases, called wargame, are used for training students to understand the security problem of vulnerable programs.

3.1. Wargame 1 (Exploiting a Bug without ShellCode)

Wargame 1 is a simple C program that can be exploited by buffer overflow because of the misuse of ***fgets***(). Then it can enter an unauthorized region at line 29 without the correct password.

```
1   #include <stdio.h>
2   #include <string.h>
3   #include <unistd.h>
4   #include <sys/types.h>
5   #include <fcntl.h>
6   char pass[8];
7
8   int main(int argc, char **argv){
9
10    FILE *fp;
11    int i = 0, auth = 0;
12    char buf[8];
13
14    printf("Input passwd: ");
15    fgets(buf, 20, stdin);
16
17    if((fp = fopen("/home/wargame1/passwd", "r")) == NULL) {
18      printf("fopen error!\n");
19      return 1;
20    }
21    fgets(pass, sizeof(pass), fp);
22    pass[strlen(pass)-1] = '\0';
23
24    for( ; i < strlen(buf); ++i)
25      if(buf[i]<'a'|| buf[i]>'z')
26        return 1;
27   if(!strcmp(buf, pass))
28     auth = 1;
29   if(auth == 1 && buf[0] == '0'){
30     char fname[32];
31     uid_t uid = getuid();
32     sprintf(fname, "/home/wargame1/checkin/%u", uid);
33     open(fname, O_CREAT | O_WRONLY, 0000);
34   }
35   return 0;
36 }
```

Figure 18: Wargame 1 contains a buffer overflow.

After instrumenting fgets(), **auth** and **i** are set symbolic variables because of 19 bytes tainted by stdin from the address of buf to i shown in Figure 19.

Variable	Buf												auth				i			
Address	40	41	42	43	44	45	46	47	48	49	4a	4b	4c	4d	4e	4f	50	51	52	53

Figure 19: Simple memory allocation of wargame 1.

Concolic testing, however, is able to get all concrete values of a tested program such that it always generates an input with bounded values at line 24 and correct password at line 27. Avoiding to this, we write a specification to check the value of **auth**, **i** and **buf[0]** at line 23.

```
1   #include <stdlib.h>
2   #include <stdio.h>
3
4   svar int $1;
5   svar int $2;
6   svar int $3;
7
8   svar int i;
9   svar int auth;
10  svar char buf[8];
11
12  property {
13       before
14       location { #line 23 "wargame1.c" }
15       pattern { $1 = i > 20 && auth == 1 && buf[0] == 30; }
16       guard { i > 20 && auth == 1 && buf[0] == 30 }
17  }
```

Figure 20: Specification of wargame 1.

To enter line 29 should be assured that like the **guard** in Figure 20 and to check this should be at line 23 described by label **location**. The **pattern** is nothing important but not empty to make CAST to work well.

3.2. Wargame 2 (Exploiting a Bug with Command Injection)

Wargame 2 can be exploited by command injection at line 46 because it calls **system**() without correct input data validation.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4   #include <errno.h>
5
6   #define BIGSIZ 15
7   char * Hmalloc (size)
8      unsigned int size;
9   {
10     unsigned int s = (size + 4) & 0xfffffffc; /* 4GB?! */
11     char *p = malloc (s);
12     if (p != NULL)
13        memset (p, 0, s);
14     else
15        printf("Hmalloc %d failed", s);
16     return (p);
17  } /* Hmalloc */
18
19  unsigned int findline (fbuf, siz)
20     char * fbuf;
21     unsigned int siz;
22  {
23     char * p;
24     int x;
25     if (! fbuf)    /* various sanity checks... */
26        return (0);
27     if (siz > BIGSIZ)
```

```
        return (0);
    x = siz;
    for (p = fbuf; x > 0; x--) {
        if (*p == '\n') {
            x = (int) (p - fbuf);
            x++;     /* 'sokay if it points just past the end! */
            return (x);
        }
        p++;
    } /* for */
    return (siz);
} /* findline */

void testcmd(char *p){
    char buf[24];
    strcpy(buf, "/usr/bin/cal \'");
    strcat(buf, p);
    strcat(buf, "\'");
    system(buf);
}

unsigned int insaved = 0; /* stdin-buffer size for multi-mode */

int main(int argc, char **argv)
{
    char **tav;//replace argv for test
    char *cp;
    int x;

    cp = argv[0];
    tav = (char **) Hmalloc (128 * sizeof (char *));
    tav[0] = cp;     /* leave old prog name intact */
    cp = Hmalloc (BIGSIZ);
    tav[1] = cp;     /* head of new arg block */
    fprintf (stderr, "Cmd line: ");
    insaved = read (0, cp, BIGSIZ);   cp[BIGSIZ-2] = '\n';
    x = findline (cp, insaved); //ok for cp
    if (x)
        insaved -= x;   /* remaining chunk size to be sent */
    cp = strchr (tav[1], '\n');//tav[1] ok
    if (cp)
        *cp = '\0';
    cp = tav[1];
    cp++;       /* skip past first char */
    x = 2;        /* we know tav 0 and 1 already */
    for (; *cp != '\0'; cp++) {
        if (*cp == ' ') {
            *cp = '\0';     /* smash all spaces */
            continue;
        } else {
            if (*(cp-1) == '\0') {
                tav[x] = cp;
                x++;
            }
        } /* if space */
    } /* for cp */
    argc = x;
    switch (tav[1][1]) {
        case 'x':
            if(tav[2]!=NULL)
                testcmd(tav[2]);
            break;
        default:
            errno = 0;
            printf("\033[1;33mnc -h for help\033[0m");
    }
    return 0;
}
```

Figure 21: Wargame 2 contains a command injection.

Concolic testing dynamically analyzes tainted data flow from **read**() to the argument of **system**() call. Thus, if the argument of system() is symbolic, how to make it to execute our desired system commands is easy to add a constraint formula into constraint solver to solve if the string equals to a illegal string (";", "&", "&&", "|", "||") followed with a command.

```
svar char* $1;

property {
     before
     pattern { system($1); }
     guard { $1[14] == '2' && $1[15] == '\'' && $1[16] == ';' &&
$1[17] == 's' && $1[18] == 'h' }
}
```

Figure 22: Specification of wargame 2 checks command injection.

Therefore, checking for wargame2 if it can be exploited with command injection, we make sure that it can execute `sh` after CAST generating an input fitting to our specification.

3.3. Wargame 3 (Exploiting a Buffer Overlfow Bug with ShellCode Injection)

Wargame3 contains a buffer overflow at line 34 in Figure 23. It copies the content from a global buffer *Buffer* to a local buffer *tmpCad*, but Buffer's size is greater than tmpCad's size such that it can cause buffer overflow.

```
1   #include <stdio.h>
2   #include <string.h>
3   #include <stdlib.h>
4
5   char  Buffer    [140];
6   int   SirveWeb = 0;
7   char *CabeceraHTTP = "HTTP/1.0 200 OK\nContent-Type:
    text/html\n\n<html><head><title>test</title></head><body><H1><Ce
    nter> HELLO</H1><HR></Center>";
8   char *FinalHTTP = "<br><hr></center></body></html>";
9   int   MaxPartidas;
10  int   PuertoWeb;
11  int   PuertoMus;
12  int   PosPrimeraPartida = -1;
13  int   NumPartidas;
14  unsigned *w3ebp;
15  void RespondeHTTPPendiente(int Pos)
16  {
17     int j, kk, faltan, tmp;
18     char tmpCad[64], *p1, *p2;
19     FILE *f;
20
21     Buffer[130] = 0;
22     p1 = strstr(Buffer, "GET");
23     if (p1 == NULL)
24        p1 = strstr(Buffer, "Get");
25     if (p1 == NULL)
26        p1 = strstr(Buffer, "get");
```

```
    if (p1 != NULL) {
        /* Bug position */
        j = 5;
        kk = 0;
        if (j < strlen(p1))
            while (p1[j] != ' ' && p1[j]){
                tmpCad[kk] = p1[j];
                kk++;
                j++;
            }
        tmpCad[kk] = '\0';
    }
    printf("Sirve Web [%s]\n", tmpCad);
    if (SirveWeb && strcmp(tmpCad, "/") && tmpCad[0]) {
        if (strstr(tmpCad, ".jar") || strstr(tmpCad, ".class"))
            p1 = "application/x-java";
        else if (strstr(tmpCad, ".jpg") || strstr(tmpCad, ".jpeg"))
            p1 = "image/jpeg";
        else if (strstr(tmpCad, ".gif"))
            p1 = "image/gif";
        else if (strstr(tmpCad, ".txt"))
            p1 = "text/plain";
        else
            p1 = "text/html";

        sprintf(Buffer, "./web/%s", tmpCad);
        f = fopen(Buffer, "rb");
        if (f != NULL) {
            sprintf(Buffer, "HTTP/1.0 200 OK\nContent-type: %s%c%c", p1, 10, 10);
            write(1, Buffer, strlen(Buffer));
            do {
                j = fread(Buffer, 1, 140, f);
                if (j)
                    write(1, Buffer, j);
            } while (j == 140);
        } else {
            strcpy(Buffer, "HTTP/1.0 404 FILE NOT FOUND\nContent-type: text/html\n\n<h1>404 FILE NOT FOUND.</h1>");
            write(1, Buffer, strlen(Buffer));
        }
        if (f != NULL)
            fclose(f);
    } else {
        write(1, CabeceraHTTP, strlen(CabeceraHTTP));
        sprintf(Buffer, "Puerto por el que escucha este servidor de partidas: %d <br> Numero maximo de partidas simultaneas: %
        strcat(Buffer, "<br>Listado de partidas en curso:<br><UL>");//problem global constant is not sqAdd in alert...
        write(1, Buffer, strlen(Buffer));
        write(1, FinalHTTP, strlen(FinalHTTP));
    }
}
int main(int argc, char **argv)
{
    int   j        , k, l, p;
    int   PonAyuda, PonAyudb;
    int   NuevoSocket;
    time_t    Ahora;
    time_t    Ahorb;


    PonAyuda = 0;
    MaxPartidas = 5;
    PuertoWeb = 0;
    PuertoMus = 0;
    k = 0;

    srandom(time(NULL));

    for (j = 0; j < argc; j++) {
        if (argv[j][0] == '-') {
            switch (argv[j][1]) {
                case 'w':
                    k = atoi(argv[j + 1]);
                    if (k > 0)
                        PuertoWeb = k;
                    break;
                case 'p':
                    k = atoi(argv[j + 1]);
                    if (k > 0)
                        PuertoMus = k;
                    break;
                case 'm':
                    k = atoi(argv[j + 1]);
                    if (k > 0)
                        MaxPartidas = k;
                    break;
                case 'd':
                    SirveWeb = 1;
                    break;
                default:
                    PonAyuda = 1;
            }
        }
    }

    k = read(0, Buffer, 140);
    if (k > 0)
        RespondeHTTPPendiente(j);
    else if (k == 0) {
        fprintf(stderr, "error: recv!!\n");
    }

    return 1;
}
```

Figure 23: Wargame 3 constains buffer overflow.

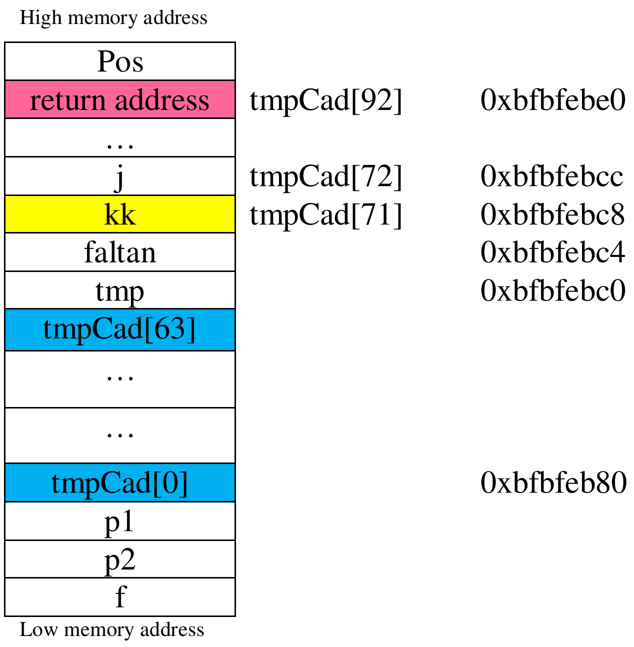


Figure 24: The memory map of wargame 3.

When the length of *Buffer* is greater than 71, kk will be tainted and tmpCad[kk] = p1[j] in the loop at line 32 in Fiugre 22 will cause symbolic pointer writing with symbolic value. Then, users can control the content of program's memory, especially return address. Because we model concolic testing to add a complex constraint composed of a large logic formula when encountering symbolic pointer dereference, the taint information of return address can be queried to constraints solver if it is valid to point the return address to the start address of the shellcode.

```
svar char* $1;

property {
     shellcode {
"\x33\xc9\x83\xe9\xf6\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x6
 7\xf9\x51\x6f\x83\xeb\xfc\xe2\xf4\x0d\xf2\x09\xf6\x35\x9f\x39\x
 42\x04\x70\xb6\x07\x48\x8a\x39\x6f\x0f\xd6\x33\x06\x09\x70\xb2\
 x3d\x8f\xfa\x51\x6f\x67\x8a\x39\x6f\x30\xaa\xd8\x8e\xaa\x79\x51
 \x6f"
     }
     pattern { tmpCad[$1] = $any; }
     guard { $1 >= 64 }
}
```

Figure 25: Specification of wargame 3 checks buffer overflow and exploit.

After all, we write a specification given in Figure 25 to check if the index of tmpCad is greater than its size 64, and find a large enough memory to allocate shellcode.

4. Related Work

KRYSTAL [34] uses a variant of symbolic execution, called universal symbolic execution to infer likely local data structure invariants for testing data structure such as binary search tree. CREST [31], an open source concolic testing tool for C, combines concolic testing with heuristic search strategies to achieve significant branch coverage on large software systems. Splat [35], a directed random testing tool guided by symbolic execution, detects buffer overflow by checking a symbolic length that may exceed the size of the symbolic prefix of the buffer. SAGE [36] , a tool for scalable, automated, guided execution, can test any file-reading program by symbolic execution and dynamic test generation. Active Property Checking [37] extends SAGE by checking whether the property (buffer overflow, null pointer dereference, etc) is satisfied by all program executions that follow the same program path, which we call universal checks. EXE [38], an effective bug-finding tool that automatically generates inputs that crash real code, can be view as a concolic testing tool with byte-level precision. RWset [39] not only generate high-coverage test inputs but also dynamically reduce the number of paths by discarding those that will produce the same effects as some previously explored path. KLEE [40] models file system for symbolic execution to automatically generate high-coverage tests for complex systems programs. Above works address the problem of finding more bugs and efficiently explore execution paths, but our work focus on exploring more execution paths when bug happens. After testing CUTE, CREST and reading EXE, we make a table as below. Other related works mention nothing about these.

Table 6: Comparison between CAST, CUTE, CREST and EXE

Tool	Symbolic Pointer Read (a = b[i])	Symbolic Pointer Write (b[i] = a)	Symbolic Pointer Write with Symbolic Value (b[i] = sym)

EXE	Yes-	Yes-	No
CUTE	No	No	No
CREST	No	No	No
CAST	Yes	Yes	Yes

BLAST, the Berkeley Lazy Abstraction Software verification Tool, is an automatic software model checker that uses counterexample-guided predicate abstraction refinement to verify temporal safety properties of C programs. The BLAST Query Language [41], based on the BLAST, can specify program verification tasks.

Table 7: Comparison between CAST and BLAST Query Language

	Execute program	Pattern matching	Ability of describing complex bugs	Type sensitive matching	Handle shellcode
BLAST	No	Weak	Low	NO	No
CAST	Yes	Strong	High	Yes	Yes

BLAST's pattern matching only focuses on single instruction without matching operator. It also doesn't care about the variable's type of pattern matching. It can't describe specification of wargames in Section 5 and can't find where to allocate shellcode in a program.

5. Conclusion

It is possible to use concolic testing to find exploit, but also need the knowledge to check where may cause vulnerabilities. Any exploit comprises a lot of complex constraint formula such that it is important to model concolic testing to handle special behavior of instruction or function. Concolic testing for generating exploits needs a strong constraint solver. Debugging a concolic testing tool is a hard work that we may need to model a architecture for this in the future.

6. Future Work and Discussion

6.1. Concrete Value Bound to Symbolic Value

We find in some test cases that CAST, using theorem prover, attempts to reach full branch coverage, but fails because some predicates are too complex to be solved.

```
#define MAX_NUM 100
void testme(int stu_id)
{
    int num=0, i=0;
    for(; i<stu_id; i++)
        num=num+1;
    if(num>MAX_NUM)
        ...
    else
        ...
}
```

Figure 26: Concrete values bound to symbolic values

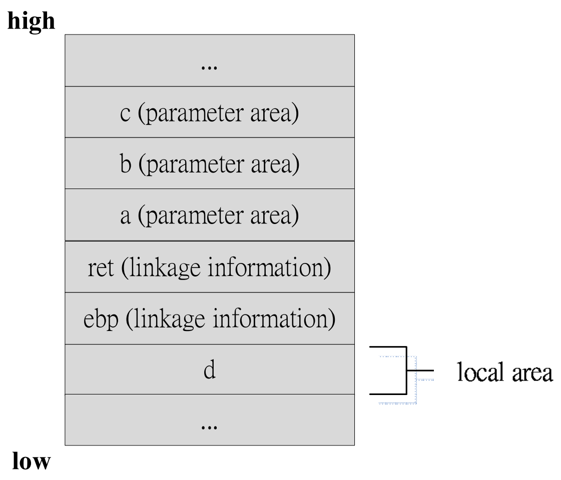
In Figure 26, controlled under the symbolic value ***stu_id,*** the concrete value ***num*** can make the branch ***if***(num>MAX_NUM) true or false, but since it is concrete, CAST don't collect it to be solved by theorem prover and don't know how to collect it with the relation to symbolic value ***stu_id*** to the solver. Figuring out this problem will help us to explore more and more branches that cause hard-found errors.

6.2. The Influence of Alignment of gcc Compiler

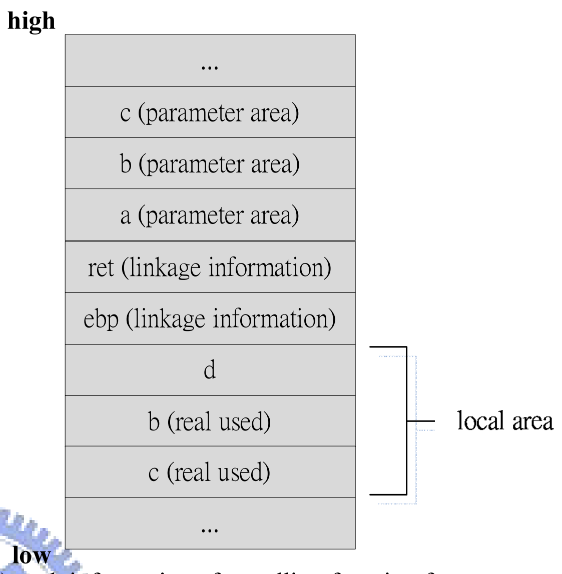
Because we try to make specification describing undefined behavior in C language, some things are hard to predicate and confuse us how to implement CAST.

```
void foo(int a, char b, short c)
{
      char d[16];
      ...
}
```

(a)function foo with different size parameters



(b)stack information after calling function foo without alignment



(c)stack information after calling function foo with alignment

Figure 27: The influence of gcc alignment on function parameter.

Figure 27(c) show that **gcc** applies alignment mechanism to a function with different size parameters and the memory addresses of parameter *b* and *c* at local area are different from those in Figure 27(b) so that if buffer *d* is overflow, it is hard to overwrite the value of b and c after alignment.

6.3. Source Level Testing without the Details of Low Level Implementation

```
void foo(char *src)
{
        int i,j;
        char tmp[64];
        for(;src[j]!=’\0’;)
            tmp[i++]=src[j++];
}
```

(a)normal local identifiers without any type qualifier

```
void foo(char *src)
{
        register int i,j;
        char tmp[64];
        for(;src[j]!=’\0’;)
            tmp[i++]=src[j++];
}
```

(b)normal local identifiers with **register** type qualifier

```
void foo(char *src)
{
        volatile int i,j;
        char tmp[64];
        for(;src[j]!=’\0’;)
            tmp[i++]=src[j++];
}
```

(c)normal local identifiers with **volatile** type qualifier

```
void foo(char *src)
{
        int i,j;
        char tmp[64];
        for(;src[j]!=’\0’;){
            tmp[i]=src[j];
            i++; j++;
        }
}
```

(d)normal local identifiers with the same semantic of (a)

Figure 28: The same semantic instruction with different assembly codes after gcc compiler.

It is common that compiler optimizes some variables from memory into register to make the program execution faster, but sometimes this optimization causes some errors to happen. Therefore, certain type qualifiers are designed to avoid it. The case in Figure 28 that gcc optimizes variable i and j into register in (a), (b) and (c) only except (d) confuses us why type qualifier fails in (b). Then, even though buffer *tmp* is overflow to overwrite the memory of variable i and j, it doesn’t have any influence on the value of i and j in *for* loop. In conclusion, source level testing can find bugs without the information about complier constructing a program but it is hard to exploit bugs accurately (false positive).

6.4. How to Debug a Debug Tool

The biggest problem in CAST is how to debug rather than how to implement. A huge of temporary variables are generated after CIL simplification, a lot of constraints are collected after program execution, and names of these variables are so similar that to differentiate and to trace back is tough. When encountering that a set of constraint formula should be *invalid* to generate the next path inputs but *valid*, we must manually trace and filter hundreds of constraint formula to find the minimum set of constraint formula causing *valid*. Delta Debugging [42] and Locating Causes of Program Failures [43] have an excellent idea about automatically finding the minimum codes causing errors. Applying these techniques will help us debug CAST quickly for extending CAST to be scalable.

References

[1] G. Tassey. The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology RTI Project Number 7007.011, 2002.

[2] R. S. Boyer, B. Elspas and K. N. Levitt. SELECT—a formal system for testing and debugging programs by symbolic execution. In Proceedings of the International Conference on Reliable Software, Los Angeles, CA, 21–23 April 1975; 234–245.

[3] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill and D. R. Engler. EXE: Automatically generating inputs of death. In Proceedings of the 13th ACM Conference on Computer and Communications Security, 2006.

[4] B. Chess and J. West. *Secure Programming with Static Analysis.* Boston, MA, USA: Addison-Wesley Professional, 2007,

[5] C. Kaner, J. L. Falk and H. Q. Nguyen, *Testing Computer Software.* John Wiley & Sons, Inc. New York, USA, 1999.

[6] K. Sen, D. Marinov and G. Agha. CUTE: A concolic unit testing engine for C. In Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005.

[7] C. Cadar, P. Twohey, V. Ganesh and D. Engler. EXE: A system for automatically generating inputs of death using symbolic execution. Technical Report CSTR 200601, Stanford, 2006.

[8] J. Larus, T. Ball, M. Das, R. DeLine, M. Fahndrich, J. Pincus, S. Rajamani and R. Venkatapathy, Righting software. *IEEE Software,* vol. 21, pp. 92-100, 2004.

[9] N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density. In Proceedings of the 27th International Conference on Software Engineering, 2005.

[10] S. C. Johnson and inc Bell Telephone Laboratories, *Lint, a C Program Checker.* Bell Telephone Laboratories, 1977,

[11] D. Evans, J. Guttag, J. Homing and Y. M. Tan, LCLint: A tool for using specifications to check code. In Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering, p.87-96, December 06-09, 1994.

[12] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe and R. Stata. Extended static checking for java. In Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, June 17-19, 2002.

[13] J. Hatcliff and M. Dwyer. Using the bandera tool set to model-check properties of concurrent java software. In Proceedings of the 12th International Conference on Concurrency Theory, p.39-58, August 20-25, 2001.

[14] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney and Y. Wang. Cyclone: A safe dialect of C. In Proceedings of the General Track: 2002 USENIX Annual Technical Conference, p.275-288, June 10-15, 2002.

[15] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold Co. New York, USA, 1990.

[16] P. Coward and B. Polytech. Symbolic execution systems-a review. *Software Engineering Journal 3(6),* pp. 229-239, 1988.

[17] K. Sen. Concolic testing. In Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, 2007, pp. 571-572.

[18] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In Proceedings of SPIN Workshop, 2005.

[19] P. Godefroid, N. Klarlund and K. Sen. DART: Directed automated random testing. In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, June 12-15, 2005.

[20] R. Majumdar and K. Sen. Hybrid concolic testing. In Proceedings of the 29th international conference on Software Engineering, p.416-426, May 20-26, 2007.

[21] R. Majumdar and K. Sen. Latest: Lazy dynamic test input generation. Technical Report UCB/EECS-2007-36, EECS Department, University of California, Berkeley, 2007,

[22] U. Shankar, K. Talwar, J. S. Foster and D. Wagner, Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium,* 2001,

[23] 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 11th International Conference on Compiler Construction, p.213-228, April 08-12, 2002.

[24] W. Chang, B. Streiff and C. Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In Proceedings of the 15th ACM conference on Computer and communications security, October 27-31, 2008.

[25] OWASP Top 10 2004. http://www.owasp.org/index.php/Top_10_2004

[26] CERT/CC Advisories. http://www.cert.org/advisories/

[27] C. F. Yang. Resolving constraints from COTS/Binary components for concolic random testing. Master thesis, NCTU, 2007.

[28] Y. L. Yen. Target directed random testing for feasible state generation. Master thesis, NCTU, 2007.

[29] Li-Wen Hsu. Resolving unspecified software features by directed random testing. Master thesis, NCTU, 2007.

[30] C. Barrett and C. Tinelli. (2008, CVC3. *LECTURE NOTES IN COMPUTER SCIENCE*

[31] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. Presented at 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008).

[32] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In Proceedings of the 11th Annual Network and Distributed System Security Symposium, 2004.

[33] Metasploit Shellcode, 2009. http://www.metasploit.com/shellcode/

[34] Y. Kannan and K. Sen. Universal symbolic execution and its application to likely data structure invariant generation. In Proceedings of the 2008 International Symposium on Software Testing and Analysis, 2008.

[35] R. G. Xu, P. Godefroid and R. Majumdar. Testing for buffer overflows with length abstraction. In Proceedings of the 2008 International Symposium on Software Testing and Analysis, 2008.

[36] P. Godefroid, M. Y. Levin and D. Molnar. Automated whitebox fuzz testing. In Proceedings of the Network and Distributed System Security Symposium, 2008.

[37] P. Godefroid, M. Y. Levin and D. A. Molnar. Active property checking. In Proceedings of the 7th ACM International Conference on Embedded Software, 2008.

[38] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill and D. R. Engler. EXE: Automatically generating inputs of death. ACM Transactions on Information and System Security, 2008.

[39] P. Boonstoppel, C. Cadar and D. Engler. RWset: Attacking path explosion in constraint-based test generation. In Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, 2008.

[40] C. Cadar, D. Dunbar and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. USENIX Symposium on Operating Systems Design and Implementation, 2008.

[41] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala and R. Majumdar. The blast query language for software verification. In Proceedings of the 11th International Static Analysis Symposium, 2004.

[42] A. Zeller, Yesterday, my program worked. Today, it does not. Why?, Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering, p.253-267, September 06-10, 1999.

[43] A. Zeller, U. des Saarlandes and G. SaarbrOcken, Isolating cause-effect chains from computer programs, Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering, November 18-22, 2002.