

國立交通大學

資訊科學與工程研究所

碩士論文

以導引式隨機測試方法探索軟體未規範實作功能

Resolving Unspecified Software Features
by Directed Random Testing

研究生：許立文

指導教授：黃世昆 教授

中華民國九十六年八月

以導引式隨機測試方法探索軟體未規範實作功能
Resolving Unspecified Software Features
by Directed Random Testing

研究生：許立文

Student : Li-Wen Hsu

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



Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

August 2007

Hsinchu, Taiwan, Republic of China

中華民國九十六年八月

以導引式隨機測試方法探索軟體未規範實作功能

學生：許立文

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

軟體測試是軟體開發過程中確保軟體品質最重要的步驟之一，因在軟體開發的過程中，我們無法保證程式不會發生錯誤。近年來動靜態程式分析工具的發展已相當成熟，而在 2005 年，更發展出了結合動靜態分析法的導引式隨機測試法。本論文實作了一個名為 ALERT，結合動靜態分析法的導引式隨機測試平台，以及其上的軟體未規範實作功能探索模組，以導引式隨機測試方法探索軟體未規範實作功能。藉由 SAT 模理論 (Satisfiability Modulo Theories) 的定理自動證明函式庫，分析特定的外部輸入所造成的程式流程，並以控制程式堆疊空間，進一步地操作使用未初始化變數的程式執行結果。本論文提出兩階段式執行測試方法：第一階段用動態分析工具分析原始程式，取得程式真實的執行資訊。第二階段使用導引式隨機測試方法，配合第一階段所收集的執行資訊，做分析及推理。推理所得的結果則為下一輪測試的輸入。我們不斷重複這兩階段的測試分析，直到找出錯誤或將程式所有執行路徑全數列舉完畢。我們將此一工具應用於尋找由未初始化變數所造成的程式未規範行為，成功地萃取出傳統程式分析方法不能找出的軟體行為。本論文提出的方法改善了現行導引式隨機測試方法中，因修改原始程式碼而造成測試時期和真正執行時期的差距，提升了測試的精確度。

關鍵字：軟體測試、導引式隨機測試、未初始化變數

Resolving Unspecified Software Features by Directed Random Testing

Student: Li-Wen Hsu

Advisors: Prof. Shih-Kun Huang

Institute of Computer Science and Engineering
National Chiao Tung University

Abstract

Testing is one of the most important phases of software quality assurance, for the process of software construction cannot guarantee the absence of bugs. Dynamic and static analysis tools are maturely developed in recent years. In 2005, the concept of concolic (combined word of concrete and symbolic) testing was proposed, which combines static and dynamic program analysis methods. In this thesis, we implement ALERT, a concolic testing framework and an Unspecified Software Feature (USF) Checker based on ALERT. By using automatic theorem prover library for satisfiability modulo theories, we can analyze and determine the inputs to direct program's execution along particular paths. With this mechanism, we can control the values in stack section. It can also be used to manipulate the values of uninitialized variables and to trigger specific behavior of the program. We present a two-phase testing algorithm in this thesis. In the first phase, we use dynamic analysis tool to retrieve real run-time information. In the second phase, we analyze the program by using concolic testing method with the data collected in the first phase. The result generated by the prover will be the input for the next testing run. This testing process iterates until a fault is found or all the program execution paths are enumerated. We use this tool to resolve unspecified program features caused by uninitialized variables. It successfully extracts the program behavior which cannot be found with traditional program analysis methods. The method in this thesis resolves the information lost problem caused by source code instrumentation in the process of testing and improves the accuracy of the test.

Keywords: Software Testing, Directed Random Testing, Uninitialized Variable

誌謝

首先要感謝我的父母，謝謝他們不辭辛勞地撫養我長大，並在我求學的路上提供一切所需。謝謝他們一路上支持我的興趣，沒有他們給我的這些，這篇論文不可能完成。

謝謝黃世昆老師費心地指導，是他提供實驗室的所有研究資源。跟老師學習的這三個學期以來，和老師的討論讓我在實作和理論，以及論文撰寫方面，都成長了不少。此外，老師也關心我的生活，願意陪我談心，幫助我解決遇到的困難。在念碩士的日子中，能碰到這個亦師亦友的好老師，真是太幸福了。還有要感謝馮立琪博士和孔崇旭博士在口試時給我寶貴的建議。

謝謝昌憲學長，和我討論實作還有論文寫作中每個細節的部份。

謝謝揚杰和彥佑在 ALERT 計畫中的努力與付出，特別感謝他們接受我霸道地推銷版本控制系統的使用，還有對於 commit message 的吹毛求疵。

謝謝泳毅、友祥還有其他 Software Quality Laboratory 的成員們，和你們一起在實驗室裡面和程式碼奮鬥的日子，我永遠不會忘記。還有謝謝揚杰和昌憲在口試前一天陪我校對投影片和論文至深夜，還有口試前友祥那罐拜過交大北大門土地公廟的仙草蜜。

謝謝 FreeBSD.org 的所有 contributors，提供了如此好用的平台，並且無私地分享出來，讓我在實作上有穩固的基礎，還有謝謝在這段時間中幫忙處理我負責部份問題的 committers。

謝謝 cassie 給我的鼓勵，妳送我的布偶，一直在鍵盤邊靜靜地陪伴著我。

謝謝 chenpc 在熬夜時候陪我去便利商店買雞精提神，還有幫我在口試前從緊張的情緒中冷靜下來。

謝謝 chwong 在我心情煩悶的時候聽我嘮嘮叨叨，和我分享許多過來人的經驗。

謝謝 ericlin, hubert 還有建北電資的所有人，謝謝你們在專業技術上的幫助以及友情上的支持，我們是一輩子的好朋友。

謝謝 jserv 在深夜時分的問候和打氣，還有提供我 24 小時免費的技術諮詢。

謝謝一路上所有幫助過我的人，謹以此論文，獻給你們。

Contents

摘要	i
Abstract	ii
誌謝	iii
Contents	iv
List of Figures	vi
List of Tables	vii
1 Introduction	1
1.1 Background	1
1.1.1 Software Property Checking	1
1.1.2 Software Testing	2
1.1.3 Unspecified Software Features	3
1.2 Problem Description	4
1.3 Motivation	4
1.4 Objective	4
1.4.1 Concept	5
1.5 Simple Example	5
1.6 Thesis Synopsis	7
2 Related work	8
2.1 Static Program Analysis	8
2.1.1 Software Model Checking	8
2.1.2 Abstract Interpretation	11
2.1.3 Assertions and Hoare logic	11
2.2 Dynamic Program Analysis	11
2.3 Concolic Testing	11
2.4 Ad-hoc Techniques	12
2.5 Comparison of Program analysis	12

3	Design and Implementation	14
3.1	ALERT	14
3.1.1	ALERT System Architecture	14
3.1.2	ALERT Execution Logic	17
3.1.3	Program Simplification	17
3.1.4	Symbolic Execution	17
3.1.5	Theorem Prover Library	19
3.2	Unspecified Software Feature (USF) Checker	20
3.2.1	USF Overview	20
3.2.2	USF Execution Logic	21
3.2.3	Symbolic Value Propagation	22
3.2.4	Stack Map	22
3.2.5	Integration with GDB	24
3.2.6	Integration with ALERT	24
4	Experimental Results	27
4.1	Test1: Overlap of Integer Variables	27
4.2	Test2: Two Short Variables Flowing into One Integer Value	30
4.3	Test3: One Integer Variable Flowing into Two Short Variables	33
4.4	Summary	36
5	Conclusion	37
5.1	The Uncertainty Principle	37
5.2	Future Work	37
	References	39

List of Figures

1	The “Out-of-Specification” Input	5
2	A Program Uses Uninitialized Value and Has Data Overlap	6
3	Stack Transition	7
4	Abstract-Check-Refine Loop of Software Model Checking	10
5	ALERT System Architecture	15
6	ALERT Execution Logic	18
7	Program Simplification	19
8	Symbolic Execution Code	20
9	The USF Execution Logic	22
10	ALERT with USF Checker Execution Logic	23
11	Definition of struct StackObject	24
12	Algorithm of GDB Script	25
13	Algorithm of memstore()	26
14	Algorithm of memload()	26
15	Test1: Overlap of Integer Variables	28
16	Execution Paths of Test1	29
17	Stack Transition of Test1	29
18	Test2: Two Short Variables Flowing into One Integer Value	31
19	Execution Paths of Test2	32
20	Stack Transition of Test2	32
21	Test3: One Integer Variable Flowing into Two Short Variables	34
22	Execution Paths of Test3	35
23	Stack Transition of Test3	35

List of Tables

1	Comparison of Program Analysis	13
2	CUTE and ALERT (without USF Checker) Iterations of Test1	27
3	ALERT with USF Iterations of Test1	30
4	CUTE and ALERT (without USF Checker) Iterations of Test2	30
5	ALERT with USF Iterations of Test2	33
6	CUTE and ALERT (without USF Checker) Iterations of Test3	33
7	ALERT with USF Iterations of Test3	36
8	Comparison of UFS Checker and CUTE	36



1 Introduction

For the hardware manufacturing technique getting mature, the software begins to receive the attention and not just as the accessory of hardware anymore. Hence, software engineering becomes a profession [4].

For the critical software, testing is the most important part in the developing process. Minor errors in the developing process may cause large damages. To reduce the fault in the software, many techniques are developed for software testing and verification.

When designing the security-related mechanism, we usually use formal method to prove if the design is reliable through the verification process done by theoretical analyses. Moreover, the developing process lacks concrete execution verification through testing. During the testing process, since the full coverage is hard to archive, the fault in the program may cause deviation in the design and implementation. Non-syntax errors would lead the program to accept more inputs than the program is designed. And such errors may not be found and warned by the compiler or traditional program checker. This gap between design and implementation causes a critical security issue. We want to design a mechanism that can automatically discover these “out-of-specification” inputs, and locate the cause of the fault. In this thesis, we introduce this using uninitialized variables in the program.

1.1 Background

Research on the testing and verification in software engineering has been mounting steadily for a number of decades.

1.1.1 Software Property Checking

Property checking is the way to determine if the software is built right with respect to its specification. There are two approaches for property checking:

1. Testing

Find inputs and one or more execution paths that demonstrate a property violation.

It works when errors are easy to find and is inefficient to find a proof.

2. Verification

Find a proof that all execution paths of the program satisfy a property. It works when proofs are easy to find and is inefficient for finding errors.

The design of an algorithm of the program can be proved, but testing is the only effective way to find the implementation problem in the software.

1.1.2 Software Testing

Software Testing can be generally divided into three phases:

1. Unit Testing

Unit testing is also called component testing. Components may be simple entities such as functions or classes. All components are tested individually to ensure that everyone operates correctly. Each component is tested independently, without interaction with other components.

2. System testing

The main goal of this process aims at finding errors from unanticipated interactions between components and interface problems. It is also concerned with validating that the system meets its functional and non-functional requirements and testing the emergent system properties.

3. Acceptance testing

The last phases of the testing process before the system is accepted for operational use. The system is tested with “real” data from users rather than with simulated test data from testing engineers. Acceptance testing may reveal errors and omissions in the system specification because the real data reflect the users’ interactions with the system. Acceptance testing may also reveal specification problems that the system does not really meet the user’s needs or the performance is unacceptable.

We focus on unit testing, which is the base of all testing techniques. Before assembling all the components into one system, we must check all units are acting as we expect first.

There are three main testing mechanisms:

1. Functional Testing

Test if the unit contains all specified features, and is built right.

2. Random Testing

Inputting random control/data to the unit, for checking if it can tolerate the unexpected inputs. This testing can not be omitted in the processing because human-written test cases can easily contain blind spots. Random testing can avoid false negative result due to biased test cases.

3. Directed Random Testing

In 2005, Godefroid, Klarlund and Sen developed a new testing techniques, DART [15]. This work integrated the static analysis and dynamic analysis methods.

1.1.3 Unspecified Software Features

Unwanted or incomplete software behaviors can be called as software defects. Most software defects are caused by the fault in the implementation. Software defects can easily lead to trigger an unspecified software feature.

We can classify software implementation defects into two types:

1. Under-implementation

The implemented features are omitted and incomplete as specified in the specification. This kind of bug can be easily discovered by the traditional unit testing, or the acceptance testing. With the modern test-driven developing development process, every function specification should have a corresponding test case to guarantee the system is well implemented. Developers can claim the program implemented all the specification according to an all-pass unit testing result.

2. Over-implementation

Extra features not specified in the specification are implemented. The unwanted feature is surely a bug. However, to locate this kind of feature is relatively harder. Since the possible input data domain is infinitely large, it is hard to test all possible data. If an implementation contains this kind of bug, the program would accept the data which should be rejected by the design, and the behaviors of the program are not expected.

1.2 Problem Description

The most common bugs in the software are due to over-implementation. In this thesis, we want to develop a method to test the presence of unspecified software features caused by over-implementation. Following are three main types of software defects:

1. Use of uninitialized variables
2. NULL-pointer dereferences
3. Out-of-bounds array indexing

If software has these defects, it contains behavior not included in the original design. It will be easily used in malicious way, and in other words, has vulnerability.

In this thesis, we focus on the problems caused by using of uninitialized variables.

1.3 Motivation

An uninitialized variable is a variable that is declared but not set to a known value before it is used. Using an uninitialized variable triggers an undefined behavior in C and C++. In the common system implementation, uninitialized global variables are stored in BSS section and will be automatically initialized with all bits filled by 0's when program loading. On the other hand, uninitialized local variables are stored in the program's stack section, which is filled by some value, but not a predictable one. However, program stack in the memory will be reused, and uninitialized local variables are implicitly "initialized" by previous function calls.

As a result, if we can control what are leftover when the previous function call ends, we can indirectly control the value of uninitialized local variable. In this way, we can trigger many vulnerabilities caused by uses of uninitialized local variable, such as null pointer dereferences, memory corruption, authentication bypass and so on.

1.4 Objective

Our main goal is to systematically check whether the program would accept "out-of-specification" inputs. We use directed random testing method to check if we can manipulate the undefined behavior cause by using uninitialized variables. This kind of bug is not

easy to discover through traditional testing mechanism, but can be done effectively with concolic and directed random testing.

1.4.1 Concept

Figure 1 shows the relation of “implementation-acceptance” domain and the “specification-acceptance” domain. The inputs in “implementation-acceptance” but not in “specification-acceptance” are bugs, and may cause a security hole. Our goal is to find the possible input which can cause the wrong behavior of the program.

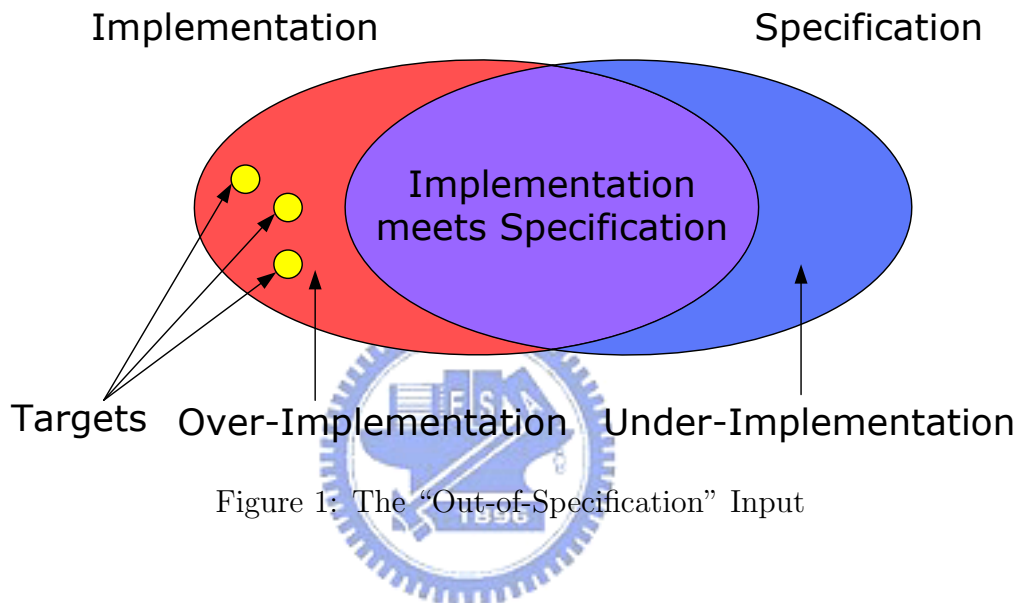


Figure 1: The “Out-of-Specification” Input

The “bug” is the deviation between implementation and design. The concept of “under-implementation” means that the specifications are not fully implemented. It is easy to detect under-implementation through test-driven process. However, the “over-implementation” is relatively harder to detect, since the domain of possible input is universally large, and cannot be fully tested.

1.5 Simple Example

Figure 2 is a simple code snippet with an uninitialized local variable:

The integer variable `data` in the `danger()` function is uninitialized, and it has the same address as `i` in `func1()`.

The corresponding stack transition is shown in Figure 3. In the transition diagram, we can find that the variable `data` in function `danger()` and `i` in `func1()` use the same address. The value of variable `data` will be implicitly “initialized” by the value

```
1 #include <stdio.h>
2
3 void func1(int arg)
4 {
5     int i = arg;
6     printf ("func1(): i == %d\n", i);
7 }
8
9 void func2(int arg)
10 {
11     printf("func2(): arg == %d\n", arg);
12 }
13
14 void danger(int dummy)
15 {
16     int data;
17     printf ("danger(): data == %d\n", data);
18     if (data == 5)
19         printf("access permit!\n");
20     else
21         printf("access deny!\n");
22 }
23
24 int main(int argc, char *argv[])
25 {
26     int n;
27
28     scanf("%d", &n);
29
30     if (n < 10)
31         func1(n);
32     else
33         func2(n);
34     danger(n);
35
36     return 0;
37 }
```

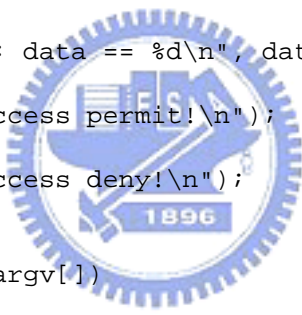


Figure 2: A Program Uses Uninitialized Value and Has Data Overlap

of `i`, whose values are from `func1()`'s argument, `arg`. We traced back to the caller, and can find that this argument is from outer input, and can be manipulated from the external.

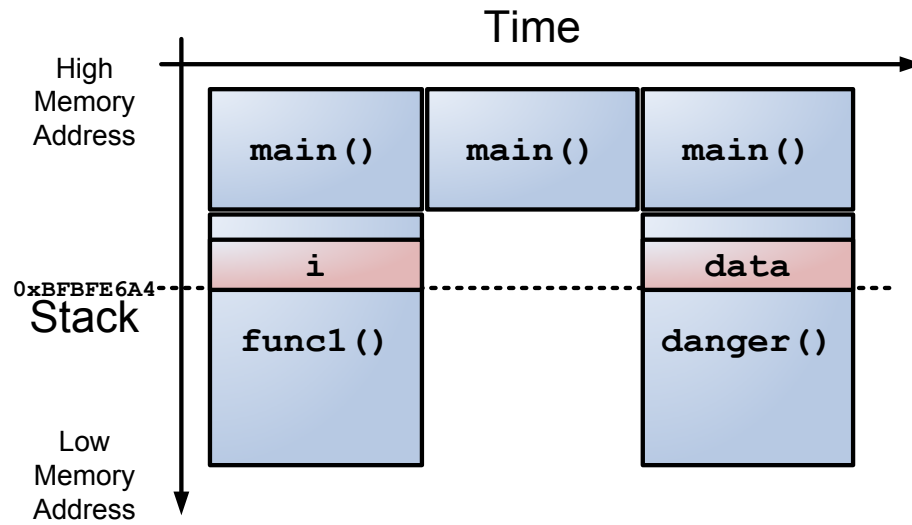


Figure 3: Stack Transition

1.6 Thesis Synopsis

Chapter 2 describes the related work and state of the art. Chapter 3 is the design and implementation of the ALERT concolic software testing platform and Unspecified Software Feature (USF) Checker. Chapter 4 presents the experimental results. Finally, Chapter 5 concludes and discusses about the future work.

2 Related work

We discuss excellent work related to ours in this chapter. In recent years, static and dynamic program analysis have been well developed. Both of them can check properties of the program effectively.

2.1 Static Program Analysis

Static program analysis is to analyze source code without concrete execution.

1. **Lint** [20]:

It is developed by Stephen Johnson in Bell Laboratories. Lint is one of the earliest C program checker that examines C source code and provides feedback to developers about issues not usually caught by a compiler. It uses static analysis and some heuristics to detect common programming errors, such as type errors, abstraction violations, and memory management bugs, but does not detect buffer overflows.

2. **Splint** [21]:

Splint is the short name of “Secure Programming Lint”, formerly called LCLint [13], which is used for statically checking if C programs contains vulnerabilities and coding mistakes. It can statically detect likely buffer overflow vulnerabilities.

3. **UNO** [19]:

As developed by Gerard J. Holzmann in Bell Laboratories, user can define properties for checking, and make the analyzer more precise. The default properties UNO aims to are three most common causes of serious error in C programs: use of uninitialized variables, null-pointer dereferencing, and out-of-bound array indexing. The checking capabilities of UNO can be extended by the user with application-dependent properties, whose syntax is similar to ANSI-C functions.

2.1.1 Software Model Checking

There are two approaches to software model checking. One is “abstraction” and the other is “to improve model checker to accept programs as input.” The trend is starting from program abstraction, since program execution is a kind of asynchronous model, and it is hard to make such a model checker.

Currently software model checking contains three phases: (Figure 4)

1. Abstraction

Program execution will expand the execution tree into infinite states. However, the model checking techniques can only deal with finite states. Thus we must perform abstraction over the program to map its behavior into finite state machine. Abstraction could be one of the three cases:

(a) Under-Abstraction

The abstraction is too specific to fully specify the program behaviors. Usually, we remove the irrelevant property of the program, for example, limiting the integer range or fixing variable into constant.

(b) Over-Abstraction

The abstraction is more general than the original program. There are two popular ways:

i. Type-based Abstraction

For example: map integer into sign abstraction: {negative, positive, zero}

ii. Predicate Abstraction

Replace predicates in the program with boolean variables, and replace each instruction that modifies the predicate with a corresponding instruction that modifies the boolean.

(c) Precise-Abstraction

The behavior of program and the finite state machine are identical.

2. Checking

Using model checker to query the model if it satisfies the property we are interested in.

3. Refinement

If the model checking result is infeasible, we should refine the abstraction model. In most of the cases, false positives are due to rough abstraction. Refining process is guided by these counter-examples. This is the famous CEGAR (Counterexample-guided Abstraction Refinement) loop [12].

There are many advanced research on software model checking:

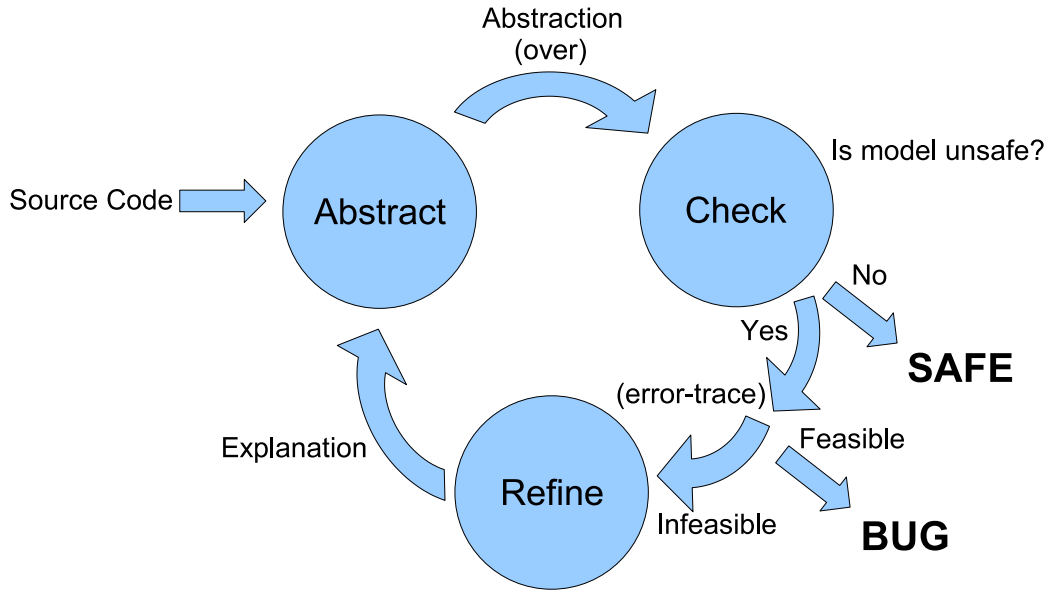


Figure 4: Abstract-Check-Refine Loop of Software Model Checking

1. **SLAM** [8]:

SLAM project is conducted by Microsoft Research, which is widely used in the Windows driver verification. It firstly automatically abstracts C programs to boolean predicates, by using C2BP [5], conducts model checking with BEBOP [6] (a symbolic model checker), and refines the abstraction by Newton tool [7].

2. **BLAST** [17]

BLAST is a software model checker for C programs. BLAST stands for “Berkeley Lazy Abstraction Software Verification Tool.” It also uses concept of CEGAR loop for checking software properties. The “Lazy Abstraction” feature performs abstraction process on-the-fly, only when needed.

3. **Java PathFinder** [28, 22]

Java PathFinder (JPF) is a special Java Virtual Machine (JVM) that can systematically explore program execution paths. It can verify bytecode programs by finding violations of properties from all program execution paths. JPF reports the entire execution path that leads to a defect.

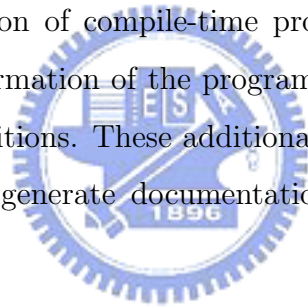
2.1.2 Abstract Interpretation

Abstract interpretation models the effect produced by every statement on the state of an abstract machine. In other words, it “executes” the software based on the mathematical properties of each statement and declaration. The abstract machine specifies an over-approximation on the behaviours of the program, and this makes the program simpler to analyze, at the expense of incompleteness. The abstract interpretation is sound, that is, every true property of the abstract system can be mapped to a true property of the original program.

2.1.3 Assertions and Hoare logic

Hoare logic [18] is the first suggestion to use assertions in programs. There are many tools using this to check program property, and aim on specific program languages.

ESC/Java [14] is a representative work. ESC, standing for Extended Static Checking, is an interactive extension of compile-time program checking. User can add constraints to provide more information of the program via a pre-defined language, such as pre-conditions and post-conditions. These additional information can help static checker probe the source code, and generate documentation. This work affects the following checking tools a lot.



2.2 Dynamic Program Analysis

Dynamic Program analysis is to analyze source code by executing it on a real or virtual processor. In some cases, we may need special external library to collect program status and run-time information.

2.3 Concolic Testing

Last one decade saw a number of attempts to merge static and dynamic program analysis. In recent years, the concept of “directed random testing” has been brought up, and immediately became the major pioneer work.

In the following, we list the leading researches on directed random testing.

1. **DART** [15]:

An automatic component testing tool that guarantees full coverage of path testing.

It uses symbolic evaluation and concrete evaluation to find associations between inputs and conditional control variables, and generates feasible input to cover all execution path of program.

2. **CUTE** [25]:

It is the follow up work of DART. CUTE stands for “Concolic Unit Testing Engine for C”. Concolic is a combined word of concrete and symbolic, and the mechanism of combining these two analyses is admirable.

3. **EGT** [10]:

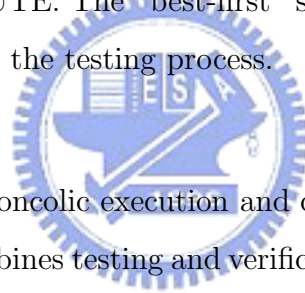
It is very similar to DART and can effectively generate the test case to make program crash.

4. **EXE** [11]:

It is EGT’s following work, which uses the fork model, rather than the iteration model of DART and CUTE. The “best-first” searching method is admirable, which can effectively speed up the testing process.

5. **Synergy**[16]:

Synergy uses DART’s concolic execution and combines the property checking techniques. It perfectly combines testing and verification techniques. With its algorithm, we can check if the bug truly exists, or the software can be verified.



2.4 Ad-hoc Techniques

Traditionally, we can only exploit uninitialized data manually by guessing the program execution path, and use debuggers to measure the variable address in the memory, and then collect variables address overlapping information.

With this work, we can use computer’s computing power and enumeration ability to help people to do this.

2.5 Comparison of Program analysis

Table 1 shows comparisons of main program analysis methods.

From the table, we can see that concolic testing is both source code aware and runs the program concretely. Thus concolic testing will have the advantages from static and

	Program Execution	Source Code Inspection	Language Dependant	Complete	Sound
Static Analysis	No	Yes	Yes	Yes	No
Dynamic Analysis	Yes	No	No	No	Yes
Concolic Testing	Yes	Yes	Yes	No	Yes

Table 1: Comparison of Program Analysis

dynamic program analysis. Concolic testing is sound, since it launches the program in each iteration, and every error run is caused from a real test case input. However, concolic testing cannot guarantee completeness, since the coverage of concolic testing will be bound by the ability of constraint solver.



3 Design and Implementation

In this chapter we discuss the design and implementation of our concolic testing framework, ALERT, and the Unspecified Software Feature Checker.

3.1 ALERT

ALERT stands for “Automatic Logic Evaluation for Random Testing.” ALERT is a software testing framework combining static and dynamic program analysis techniques.

ALERT project started in fall, 2006, by the members of Software Quality Laboratory, College of Computer Science, National Chiao Tung University, under the supervision of Prof. Shih-Kun Huang, ALERT is a concolic software testing platform with various features and many applications.

3.1.1 ALERT System Architecture

The components in ALERT automatic testing framework and how the program is processed to become “self-testing program” are shown in Figure 5.

There are three phases to process the original source code:

1. Simplification

Before inserting our checking code in the original source file, we perform source code simplification for reducing the complexity of analysis procedure. We mainly simplify the program statements, and many sugar forms in the language are eliminated. After simplification, the program still preserves the original semantics.

We use CIL [24] (C Intermediate Language) for source-level transformation. CIL is developed in Berkeley, which can parse C language into syntax tree and express it using tuple type in OCaml [2], and users can adjust the syntax tree for specified purposes. After the preprocessing on syntax tree, the syntax-tree will be interpreted back to simpler C language, which can be accepted by standard C compilers. CIL provides a high-level representation of C programs, and it is distributed with modules for helping C program analysis.

In ALERT, we mainly take advantage of CIL to perform the following simplifications:

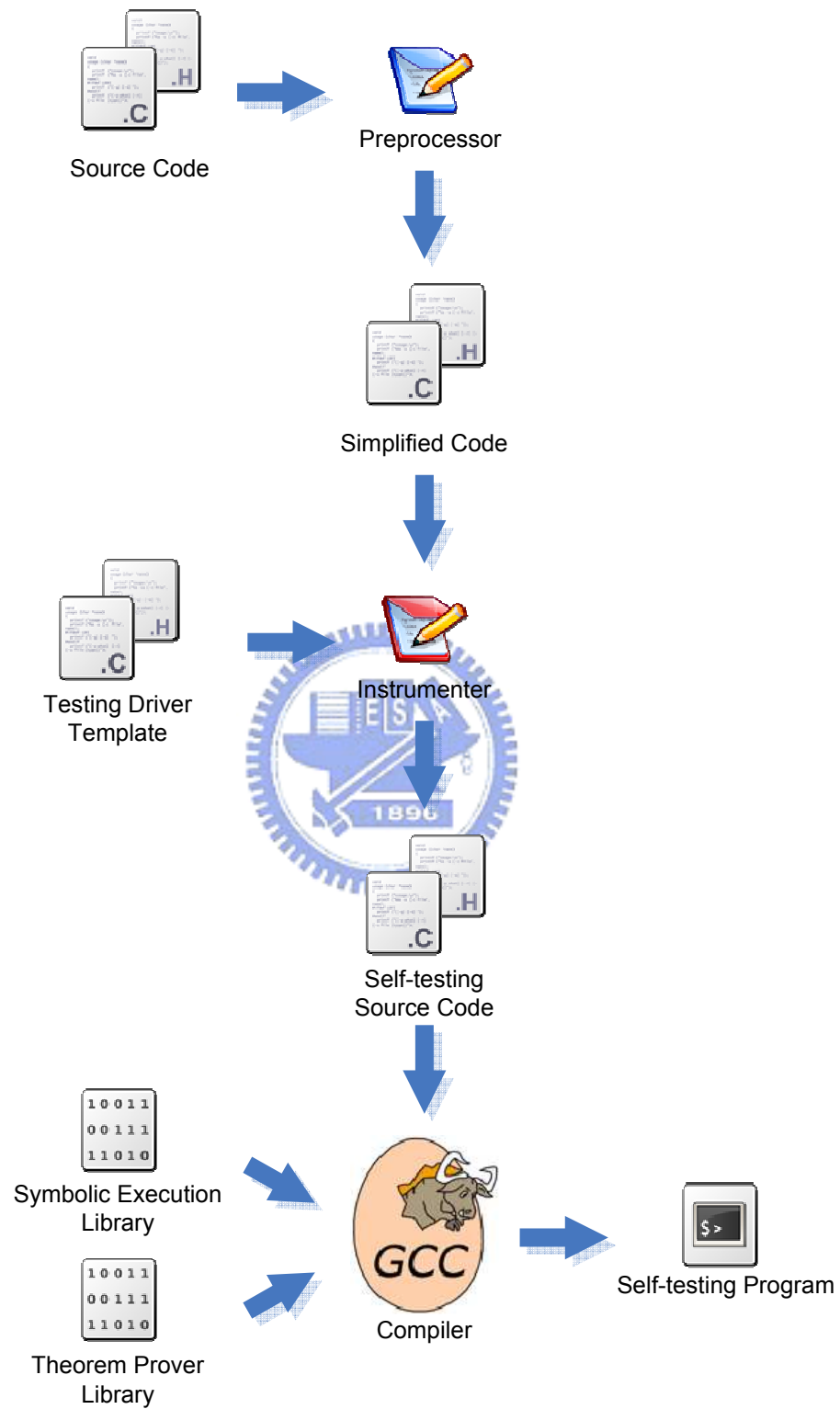


Figure 5: ALERT System Architecture

- (a) Loop statements (`while`, `for` and `do-while`) are changed to a single `while(1)` looping with `goto` statements for `break`.
- (b) All branch statements (`if`, `if-else`, `if-elseif-else` and `switch`) are rewritten into `if-else` statements.
- (c) All predicates in the if-statement are transformed to a binary relation with atomic variables.

Besides, we also enable the following CIL modules for advanced simplification:

- (a) Simple Three-Address Code

Reduce the complexity of program expressions, give us a form of three-address code, and preserve the semantics of the original program. The unified expression enables us not to deal with various program syntax.

- (b) Simple Memory Operations

Allow variables that contain pointer operation to be simplified via introduction of well-typed temporaries. This promises that each expression will contain only one-level pointer dereference, and reduces the complexity of handling pointer arithmetic.

- (c) One Return

This transformation makes each function with only one return statement. With this assurance, we can safely insert codes which should be executed when returning from a called function at the unique return point of the function.

2. Instrumentation

As the simplification phase, we also use CIL as our instrumentation tool. It is also used by [15, 25, 10, 11]. We use this tool to modify the program to collect the constraint data when the program executes, without modifying the semantic of the original program. In the instrumentation phase, we insert symbolic execution function codes into the test program. The objective and the implementation detail will be discussed in “Symbolic Execution” section (3.1.4).

3. Compiling

We use GNU’s C compiler (`gcc`) to compile the self-testing source code to the self-

testing program. In the compilation process, we link our symbolic execution library and theorem prover library.

3.1.2 ALERT Execution Logic

Execution logic of ALERT is shown in Figure 6.

When the first time testing starts, a set of initial values as the seed input is given. While the “self-testing” program executes, the constraints of execution path are collected by the instrumented code. If any error occurs during the execution, a bug is found. The input of this run is saved for reproducing the fault. While the program executes and exits normally, the last one of the constraints will be negated to make a new constraint set. Then this new constraint set is solved by the constraint solver to generate a input, which directs the program executions along alternative path in the next run.

3.1.3 Program Simplification

Figure 7 is an example of program simplification.

We can see the `for` statement (line 4-5) and the `if-elseif-else` statement (line 6-11) in the original program are transformed into simpler forms. The `for` statement is replaced with `while(1)` and `goto` statements. The `if-elseif-else` statement is rewritten into two-level `if-else` statement.

3.1.4 Symbolic Execution

We usually use values to indicate the program execution state, such as variables values and program counter. Symbolic execution is performed by using logic formulas to represent the state of program execution. In symbolic execution, each executed statement is transformed to a symbolic logical formula. Through symbolic execution of each statement, these symbolic logic formulas will be combined with conjunction operators. When the program terminated, the final conjunction-form formula is the constraints of the execution path we just walked along.

Figure 8 is an example of program with inserted symbolic code. The predicate of different branches are collected by `add_predicate()` function in line 2 and 8. Statements are followed by a respective `symbolic_execution()` function (line 5), which are also used for constraint-collecting.

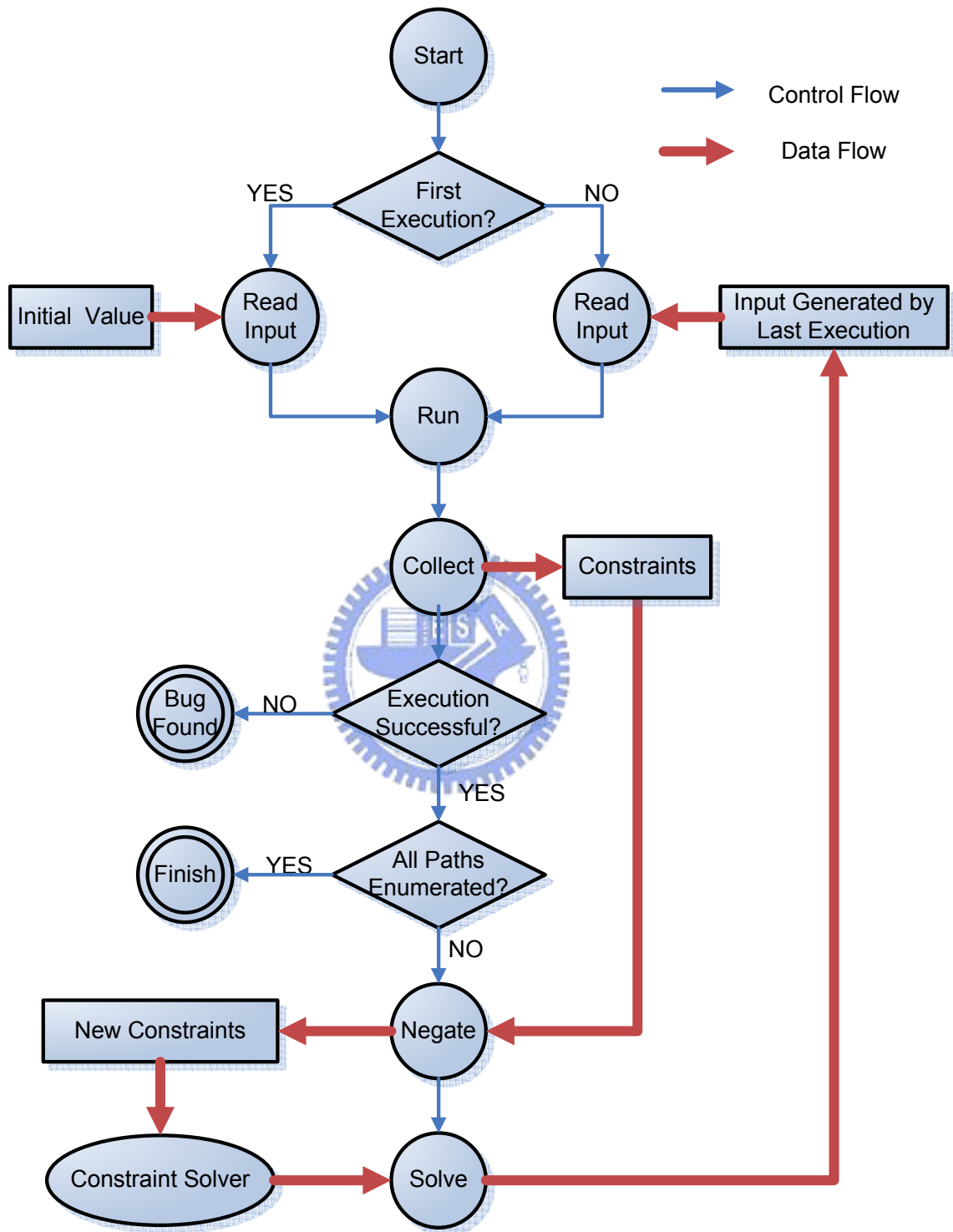


Figure 6: ALERT Execution Logic

Original Program:

```
1 void testme(int i)
2 {
3     int j;
4     for( j = 0; j < 10; ++j )
5         i = 2 * j + i;
6     if (i == 10)
7         printf("if block\n");
8     else if (i == 20)
9         printf("else if block\n");
10    else
11        printf("else block\n");
12 }
```

Simplified Program:

```
1 void testme(int i )
2 {
3     int j ;
4     int __cil_tmp3 ;
5     int __cil_tmp4 ;
6     int __cil_tmp5 ;
7
8     j = 0;
9     while (1) {
10        __cil_tmp3 = j < 10;
11        __cil_tmp5 = ! __cil_tmp3;
12        if (__cil_tmp5 != 0) {
13            goto while_0_break;
14        }
15        __cil_tmp4 = 2 * j;
16        i = __cil_tmp4 + i;
17        j ++;
18    }
19 while_0_break:
20     ;
21     if (i == 10) {
22         printf("if block\n");
23     } else {
24         if (i == 20) {
25             printf("else if block\n");
26         } else {
27             printf("else block\n");
28         }
29     }
30     return;
31 }
```

Figure 7: Program Simplification

3.1.5 Theorem Prover Library

We use CVCL [27, 9], a descendant of *VC Systems from Stanford University, developed under Professor Clark Barrett's leading in New York University. CVCL is an automatic

Original Program:

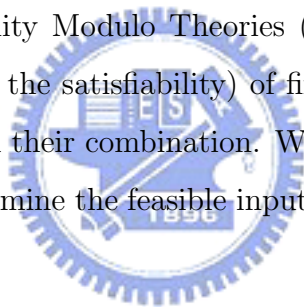
```
1 if (predicate) {
2     a = b + 37;
3     /* ... */
4 } else {
5     /* ... */
6 }
```

Instrumented Program:

```
1 if (predicate) {
2     add_prediecte(branch_id, 1, predicate);
3     /* ... */
4     a = b + 37;
5     symbolic_execution(a, OP_PLUS, b, 37);
6     /* ... */
7 } else {
8     add_prediecte(branch_id, 0, !predicate);
9     /* ... */
10 }
```

Figure 8: Symbolic Execution Code

theorem prover for Satisfiability Modulo Theories (SMT) problems. It can be used to prove the validity (or, dually, the satisfiability) of first-order formulas in a large number of built-in logical theories and their combination. We use its power of bit-level constraint solving to automatically determine the feasible inputs that reach our goal, the given state of program.



3.2 Unspecified Software Feature (USF) Checker

In this section we discuss the design and implementation of USF Checker. USF Checker is based on the ALERT concolic testing framework, combining with other dynamic program analysis tools and integrated with ALERT’s symbolic execution library.

3.2.1 USF Overview

The USF Checker uses directed random testing ability of ALERT to generate the test inputs, and checks the existence of “out-of-specification” inputs of the program under testing.

For the external dynamic program analysis tools, we use GDB, GNU Project Debugger [1], which is a powerful debugger for program instrumentation. We use it to collect the run-time information of the uninstrumented program, which can reflect the real situations

of the program during execution.

3.2.2 USF Execution Logic

Execution Logic of USF Checker is shown in Figure 9.

Following are the execution steps of USF:

1. Initialize input values for the first-run

We should give the same input to uninstrumented program and instrumented one. Therefore, the input values of first run should be generated by instrumented program. We modify the initialized part of ALERT, with an option for producing the initial input values to an external file, which can be used for the input values for the first uninstrumented run.

2. Launch the uninitialized program via GDB script

Use our GDB script to collect the “real” run-time information from the uninstrumented program. The details of GDB script will be discussed in the “Integration with GDB” section.

3. Run the instrumented program

In this step, the instrumented program is launched in the same way as traditional concolic testing. However, the “extra” information collected in the step 2 is mixed into the execution.

4. Convert the output from instrumented program

The outputs from one ALERT’s execution contain too much information, but for the uninstrumented program, only raw inputs are needed. For this reason, we should convert the output generated by instrumented program to the form that original program accepts.

5. Repeat step 2 to 4, until all the paths of programs are enumerated, or a bug is found.

Execution Logic of ALERT with USF Checker is shown in Figure 10.

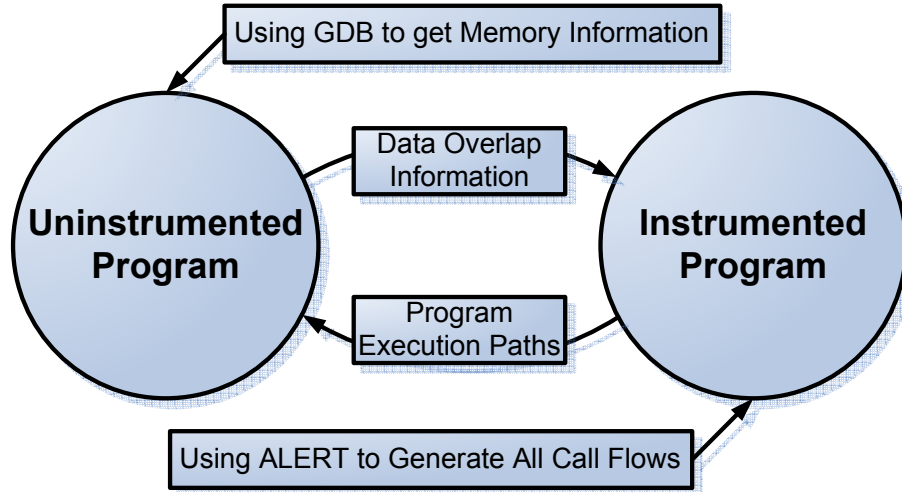


Figure 9: The USF Execution Logic

3.2.3 Symbolic Value Propagation

In an assignment expression, the evaluation result of right-hand-side will be assigned to the variable in the left-hand-side. If the evaluation result of right-hand-side is a symbolic value, then left-hand-side variable will also get the “symbolic” attribute.

The USF Checker extends this concept. In our implementation, we model not only the normal assignment-based symbolic value propagation but also unconventional symbolic value propagations. In order to “initialize” the uninitialized local variables, we should model the symbolic relation between the values left in stack from previous function calls and the local variables in the current function. This relation is lost in the traditional concolic testing, since the frames of function calls in program stack are changed due to the extra local variables and the function calls introduced by instrumentation.

3.2.4 Stack Map

We use storage concept to model the operations of random access memory when the program executes. We use a hash-based map to implement the storage, named StackMap. Each item in the hash map is a structure called StackObject, used to store both byte-level symbolic and concrete value. The definition of StackObject is shown in Figure 11.

Refer to the code in Figure 11. The first three fields in the structure are used to store symbolic values: name, type and cn, which save the symbolic name, type and context number of the variable, respectively. This three-tuple is sufficient to uniquely locate one

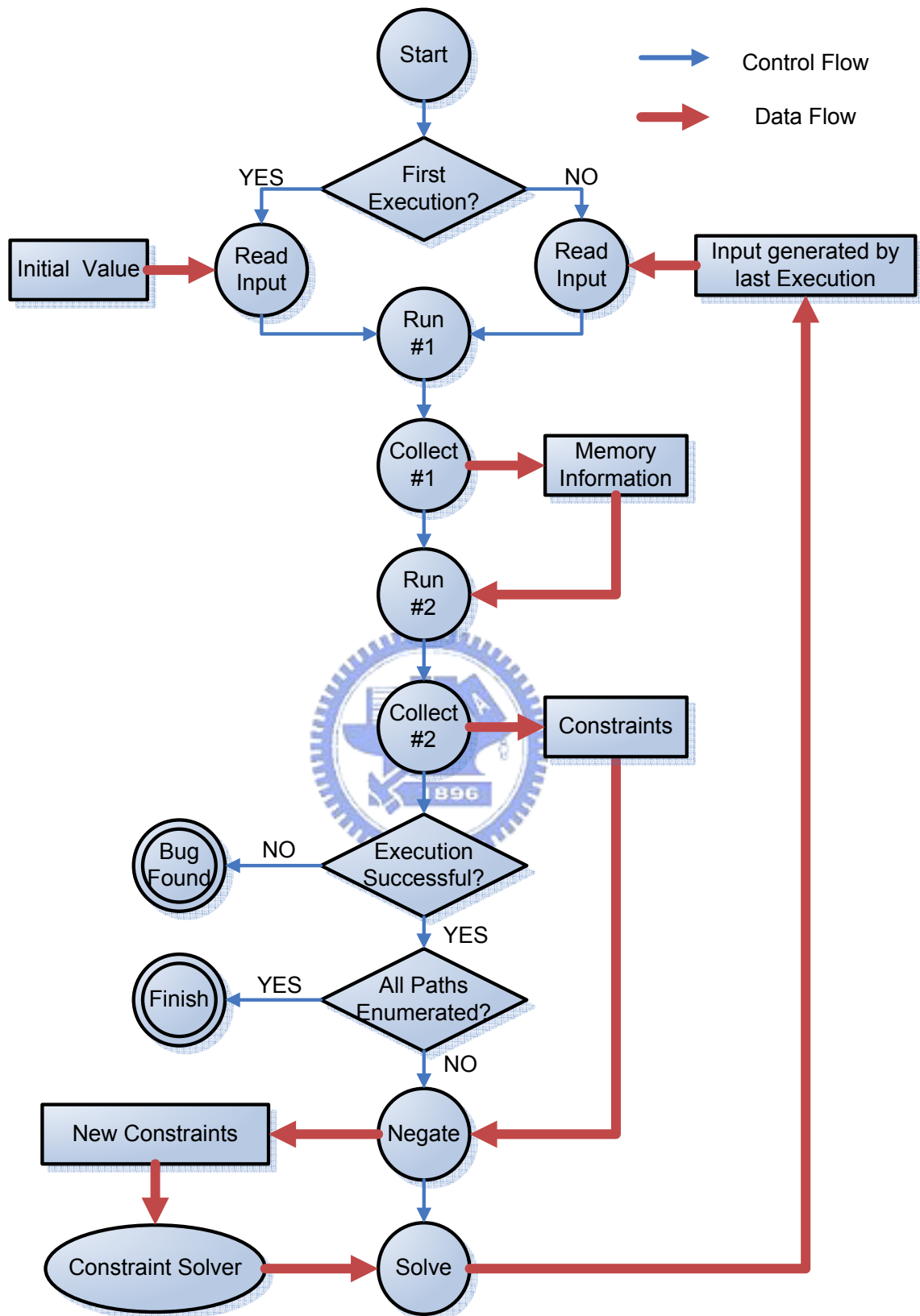


Figure 10: ALERT with USF Checker Execution Logic

```

1 struct StackObject{
2     string      name;
3     TypeName    type;
4     int        cn;
5     int        which_byte;
6     unsigned char byte_value;
7 };

```

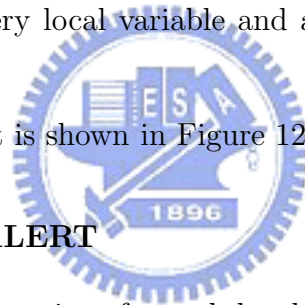
Figure 11: Definition of struct StackObject

variable in the program. The following field, `which_byte`, indicates which byte of the variable for this `StackObject` item. The last field is used to store concrete values.

3.2.5 Integration with GDB

We implement a GDB script to automatically collect the run-time stack transition information. The GDB interrupts program execution at each function’s entry point. The “`info locals`” and “`info args`” commands in GDB list all local variables and arguments in the current stack frame. Every local variable and argument’s addresses in the function are recorded.

Algorithm of GDB script is shown in Figure 12.



3.2.6 Integration with ALERT

We insert the following two functions for each local variable and argument in every functions in the program under test for simulating unconventional data propagation via the values left in stack:

1. `memstore(const char *name, void *ptr, const TypeName type)`
`memstore()` will be called for each local variable and arguments before returning from the function. It is used for storing variable’s symbolic and concrete value to its “real” address of the memory in byte-level precision. After the values are stored, they can be loaded via the `memload()` function.

Algorithm of `memstore` is shown in Figure 13:

2. `memload(const char *name, void *ptr, const TypeName type)`
`memstore()` will be called for each local variable on entering each function. By examining the assembly code of the entry point of a function, we can observe that the stack pointer will be subtracted by a value, the frame size. The newly allocated

```

1:  $P \leftarrow$  the uninstrumented program
2:  $G \leftarrow$  new GDB wrapper
3:  $A \leftarrow \emptyset$  {Address information Map}
4:  $F \leftarrow$  return value of issuing “info functions” command to  $G$ 
5: for all  $f$  in  $F$  do
6:    $G$  set break point at  $f$ 
7: end for
8: issue “run  $P$ ” command to  $G$ 
9: loop
10:   $Name \leftarrow$  function name of the break point
11:  if  $P$  executes over then
12:    break
13:  end if
14:   $L \leftarrow$  return value of issuing “info locals” command to  $G$ 
15:   $R \leftarrow$  return value of issuing “info args” command to  $G$ 
16:  for all  $var$  in  $L \cup R$  do
17:     $X \leftarrow$  return value of issuing “output & $var$ ” command to  $G$ 
18:     $A \leftarrow A \cup \langle i, X \rangle$ 
19:  end for
20: end loop
21: for all  $\langle name, addr \rangle$  in  $A$  do
22:  output  $\langle name, addr \rangle$  to the “meminfo” file
23: end for

```

Figure 12: Algorithm of GDB Script

space is used for storing local variables. After allocation, the values of local variables are initialized. `memstore()` is used to “initialize” each local variables, and simulates the local variables “initialized” by the values left in the previous function frame, and assigns both symbolic and concrete value for the variables.

Algorithm of `memstore` is shown in Figure 14:


Besides, we also add an `update_addrmap()` function in ALERT’s `enter_procedure()`. `update_addrmap()` function is used to load the address information of local variables in current function frame.

```

1:  $Name \leftarrow$  name of the local variable or argument
2:  $Pointer \leftarrow$  pointer to the local variable or argument
3:  $Type \leftarrow$  type of the local variable or argument
4:  $Addr \leftarrow$  “real” address of  $Name$  from “meminfo” file
5:  $S \leftarrow$  size of  $Type$  {size in bytes}
6:  $C \leftarrow$  value dereferenced from  $Pointer$  as  $Type$ 
7: for  $i = 0$  to  $S$  do
8:    $O \leftarrow$  new StackObject
      /* For Symbolic Value */
9:    $O.name \leftarrow Name$ 
10:   $O.cn \leftarrow Current\_Context\_Number$ 
11:   $O.type \leftarrow Type$ 
      /* For Concrete Value */
12:   $O.which\_byte \leftarrow i$ 
13:   $O.byte\_value \leftarrow$   $i$ -th byte of  $C$ 
14:   $StackMap \leftarrow StackMap \cup \langle Addr, O \rangle$ 
15:   $Addr \leftarrow Addr + 1$ 
16: end for

```

Figure 13: Algorithm of memstore()



```

1:  $Name \leftarrow$  name of the local variable
2:  $Pointer \leftarrow$  pointer to the local variable
3:  $Type \leftarrow$  type of the local variable
4:  $Addr \leftarrow$  “real” address of  $Name$  from “meminfo” file
5:  $S \leftarrow$  size of  $Type$  {size in bytes}
6: for  $i = 0$  to  $S$  do
7:   if There is no object at  $Addr$  in  $StackObject$  then
8:     break
9:   end if
10:   $O \leftarrow StackMap[Addr]$ 
      /* For Symbolic Value */
11:   $x \leftarrow$  get CVCL expression of Variable ( $O.name, O.cn$ )
12:   $y \leftarrow$  get CVCL expression of Variable ( $Name, Current\_Context\_Number$ )
13:   $s \leftarrow$  CVCL expression of “EXTRACT  $i$ -th byte from  $x$ ”
14:   $l \leftarrow$  CVCL expression of “EXTRACT  $i$ -th byte from  $y$ ”
15:   $Expr \leftarrow$  CVCL Expression ( $s = l$ )
16:   $push\_constraint(Expr)$ 
      /* For Concrete Value */
17:   $Pointer[i] \leftarrow O.byte\_value$ 
18:   $Addr \leftarrow Addr + 1$ 
19: end for

```

Figure 14: Algorithm of memload()

4 Experimental Results

We conduct experiments of the programs that use uninitialized variables to prove the applicability of our method. Our experimental environment is Intel® Pentium® D CPU 3.40GHz, with FreeBSD 6.2-RELEASE-p7 and gcc in the base system (version 3.4.6 [FreeBSD] 20060305.) We use our framework to check the following three programs:

4.1 Test1: Overlap of Integer Variables

Program Test1 has an integer-type uninitialized local variable in the `danger()` function, while its address overlaps the address of local variables in the previous called function, `func1()`. A permission checking failure will occur if the uninitialized variable matches the specified value. We use this test to demonstrate that our framework can handle symbolic value propagation via address overlap.

The execution paths of Test1 are shown in Figure 16.

We find this program has three execution paths, and the error path is Path 2, which is triggered when the uninitialized local variable, `data` in function `danger()` is 5.

By using CUTE and ALERT (without USF Checker), we have the following iterations:

Iteration	<code>i</code> (Input)	<code>data</code> (*)	Unwanted Feature Triggered
1	0	0	No
2	2147483647	-2747	No

* Uninitialized Variable

Table 2: CUTE and ALERT (without USF Checker) Iterations of Test1

The corresponding stack transition analyzed by USF is shown in Figure 17.

We can observe that the local variable `data` in `danger()` and local variable `i` in previous called function, `func1()` are with the same address. The USF Checker notices this address overlapping relation, and reflects it into concolic execution.

By using ALERT with USF Checker, we have the following iterations:

From Table 3, because of the introduction of address overlapping information provided by USF, ALERT can fully enumerate three paths in the program, and successfully trigger the unwanted features.

```

1 #include <stdio.h>
2
3 void func1(int arg)
4 {
5     int i = arg;
6     printf ("func1(): i == %d\n", i);
7 }
8
9 void func2(int arg)
10 {
11     printf("func2(): arg == %d\n", arg);
12 }
13
14 void danger(int dummy)
15 {
16     int data;
17     printf ("danger(): data == %d\n", data);
18     if (data == 5)
19         printf("access permit!\n");
20     else
21         printf("access deny!\n");
22 }
23
24 int main(int argc, char *argv[])
25 {
26     int n;
27     FILE *f;
28
29     f = fopen("input", "r");
30     fscanf(f, "%d", &n);
31
32     if (n < 10)
33         func1(n);
34     else
35         func2(n);
36     danger(n);
37
38     fclose(f);
39
40     return 0;
41 }

```

Figure 15: Test1: Overlap of Integer Variables

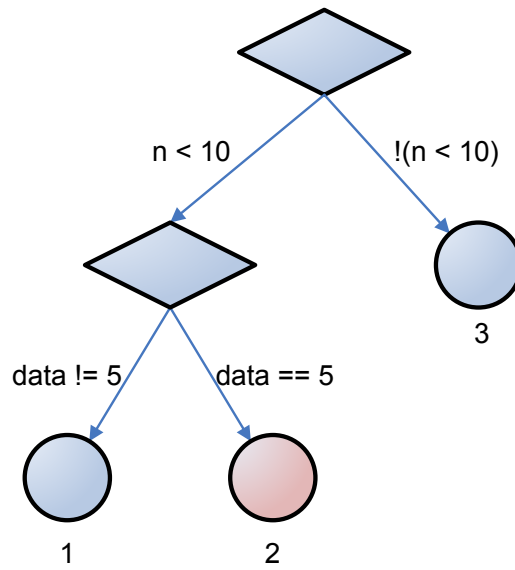


Figure 16: Execution Paths of Test1

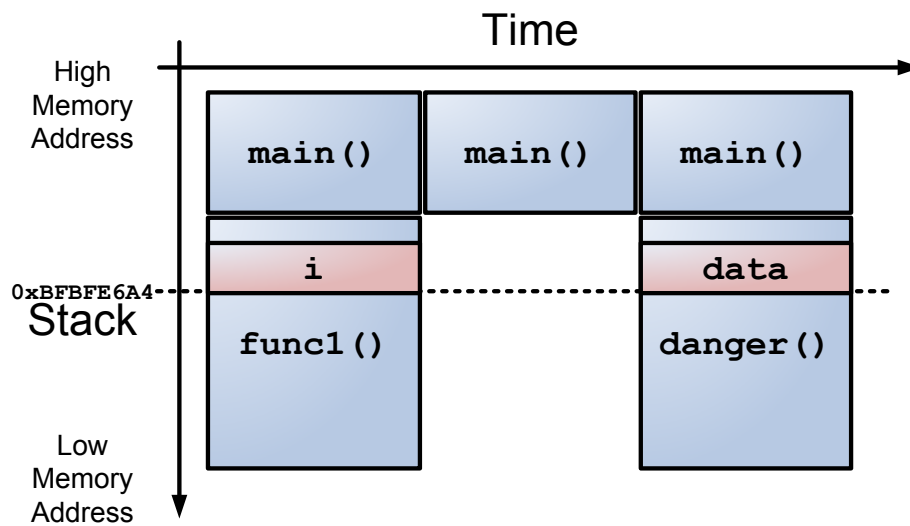


Figure 17: Stack Transition of Test1

Iteration	i (Input)	data (*)	Unwanted Feature Triggered
1	0	0	No
2	5	5	Yes
3	2147483647	2147483647	No

* Uninitialized Variable

Table 3: ALERT with USF Iterations of Test1

4.2 Test2: Two Short Variables Flowing into One Integer Value

Program Test2 has an integer-type uninitialized local variable in the `danger()` function, while its address overlaps the addresses of local variables in the previous called function, `func1()`. A permission checking failure will occur if the uninitialized variable matches the specified value. We use this test to demonstrate our framework handles symbolic value loading in byte-level precision.

The execution paths of Test2 are shown in Figure 19.

We find this program has three execution paths, and the error path is Path 3, which is triggered when the uninitialized local variable, `data` in function `danger()` is 2134843151 (0x7f3f1f0f).

By using CUTE and ALERT (without USF Checker), we have the following iterations:

Iteration	i (Input)	i2 (Input)	data (*)	Unwanted Feature Triggered
1	0	0	483	No
2	11	0	-10755993	No

* Uninitialized Variable

Table 4: CUTE and ALERT (without USF Checker) Iterations of Test2

The corresponding stack transition analyzed by USF is shown in Figure 20.

We can observe that the integer-type local variable `data` in `danger()` and short-type local variable `i` in previous called function, `func1()` are with the same address. Moreover, the address of short-type local variable `i2` in `func1()` is also in the space of `data`. The USF Checker notices the address overlapping relation between these three variables. Since we model the stack memory in the byte-level precision, which reflects these relations into concolic execution, USF can establish the following two symbolic propagations:

1. `i` in `func1()` to the lower two bytes in the `data` in `danger()`

```

1 #include <stdio.h>
2
3 void func1(short arg, short arg2)
4 {
5     short i = arg;
6     short i2 = arg2;
7     printf ("func1(): i == %d\n", i);
8     printf ("func1(): i2 == %d\n", i2);
9 }
10
11 void func2(int arg)
12 {
13     printf("func2(): arg == %d\n", arg);
14 }
15
16 void danger(short dummy, short dummy2)
17 {
18     int data;
19     printf ("danger(): data == %d\n", data);
20     if (data == 2134843151)
21         printf("access permit!\n");
22     else
23         printf("access deny!\n");
24 }
25
26 int main(int argc, char *argv[])
27 {
28     short n, n2;
29     FILE *f;
30
31     f = fopen( "input-short2int", "r" );
32     fscanf( f, "%hd", &n );
33     fscanf( f, "%hd", &n2 );
34
35     if ( n > 10 )
36         func1( n, n2 );
37     else
38         func2( n );
39     danger( n );
40
41     fclose(f);
42
43     return 0;
44 }

```

Figure 18: Test2: Two Short Variables Flowing into One Integer Value

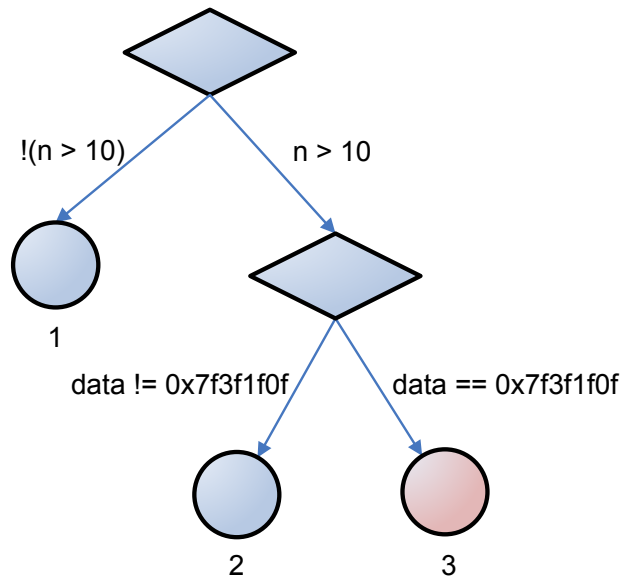


Figure 19: Execution Paths of Test2

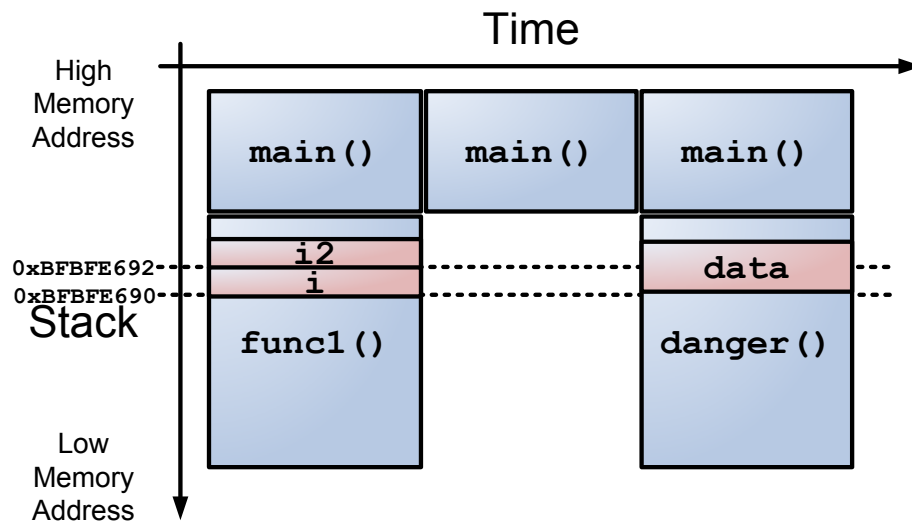


Figure 20: Stack Transition of Test2

2. `i2` in `func1()` to the higher two bytes in the `data` in `danger()`

By using ALERT with USF Checker, we have the following iterations:

Iteration	<code>i</code> (Input)	<code>i2</code> (Input)	<code>data</code> (*)	Unwanted Feature Triggered
1	0	0	0xbfbfef774	No
2	0x000b	0xffff (-1)	0x000bffff	No
3	0x7f3f	0x1f0f	0x7f3f1f0f	Yes

* Uninitialized Variable

Table 5: ALERT with USF Iterations of Test2

From Table 5, because of the introduction of address overlapping and variables combining information provided by USF, ALERT can fully enumerate three paths in the program, and successfully trigger the unwanted feature.

4.3 Test3: One Integer Variable Flowing into Two Short Variables

The following program has two short-type uninitialized local variables in the `danger()` function, while their addresses overlap the address of integer local variable in the previous called function, `func1()`. A permission checking failure will occur if the two uninitialized variables match the specified value. We use this test to demonstrate that our framework handles symbolic value storing in byte-level precision.

The execution paths of Test3 are shown in Figure 22.

We find this program has four execution paths, and the error path is Path 4, which is triggered when both the uninitialized local variable, `data` and `data2` in function `danger` equal to 3871 (0x0f1f) and 16255 (0x3f7f) respectively.

By using CUTE and ALERT (without USF Checker), we have the following iterations:

Iteration	<code>i</code> (Input)	<code>data</code> (*)	<code>data2</code> (*)	Unwanted Feature Triggered
1	0	6213	-861	No
2	11	2	214	No

* Uninitialized Variable

Table 6: CUTE and ALERT (without USF Checker) Iterations of Test3

The corresponding stack transition analyzed by USF is shown in Figure 23.

```

1 #include <stdio.h>
2
3 void func1(int arg)
4 {
5     int i = arg;
6     printf ("func1(): i == %d\n", i);
7 }
8
9 void func2(int arg)
10 {
11     printf("func2(): arg == %d\n", arg);
12 }
13
14 void danger(int dummy)
15 {
16     short data, data2;
17     printf ("danger(): data == %hd\n", data);
18     printf ("danger(): data2 == %hd\n", data2);
19     if (data == 3871) {
20         if (data2 == 16255)
21             printf("access permit!\n");
22         else
23             printf("access deny!\n");
24     } else
25         printf("access deny!\n");
26 }
27
28 int main(int argc, char *argv[])
29 {
30     int n;
31     FILE *f;
32
33     f = fopen( "input-int2short", "r" );
34     fscanf( f, "%d", &n );
35
36     if ( n > 10 )
37         func1(n);
38     else
39         func2(n);
40     danger(n);
41
42     fclose(f);
43
44     return 0;
45 }

```

Figure 21: Test3: One Integer Variable Flowing into Two Short Variables

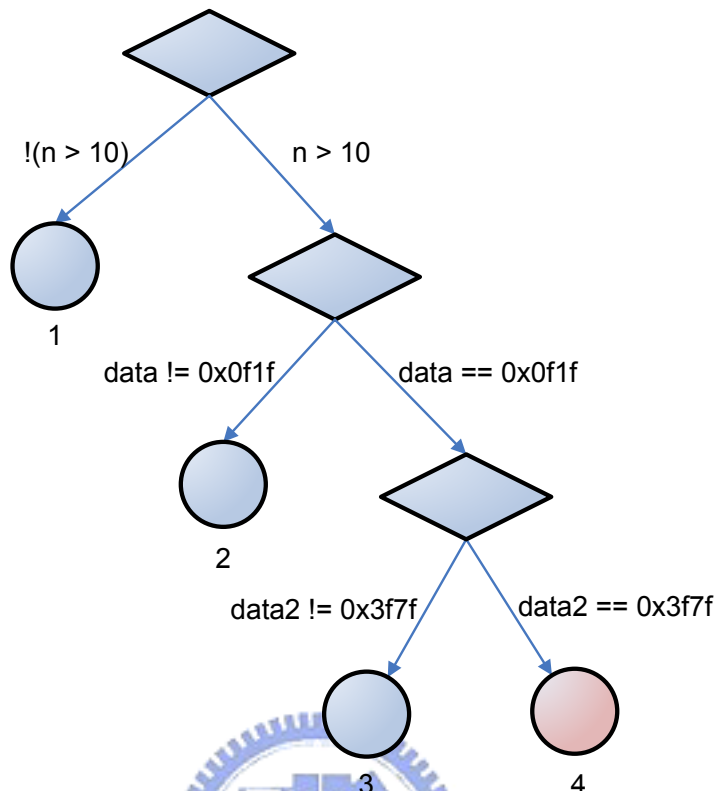


Figure 22: Execution Paths of Test3

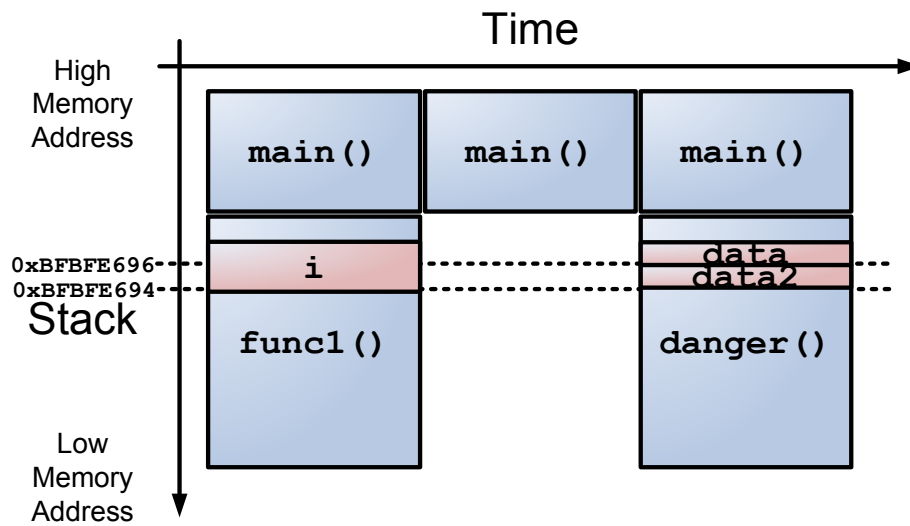


Figure 23: Stack Transition of Test3

We can observe that the short-type local variables `data` and `data2` in `danger()`, and integer-type local variable `i` in previous called function, `func1()` are with the same address. Moreover, the address of local variable `data2` in `danger()` is also in the space of `i`. The USF Checker notices the address overlapping relation between these three variables, and reflects this relations into concolic execution. USF can establish the following two symbolic propagations:

1. Higher two bytes of `i` in `func1()` to `data` in `danger()`
2. Lower two bytes of `i` in `func1()` to `data2` in `dnager()`

By using ALERT with USF Checker, we have the following iterations:

Iteration	i (Input)	data (*)	data2 (*)	Unwanted Feature Triggered
1	0	0	0	No
2	11	0	11	No
3	0x0f1f0000	0x0f1f	0x0000	No
4	0x0f1f3f7f	0x0f1f	0x3f7f	Yes

* Uninitialized Variable

Table 7: ALERT with USF Iterations of Test3

From Table 7, because of the introduction of address overlapping and variables combining information provided by USF, ALERT can fully enumerate four paths in the program, and successfully trigger the unwanted feature.

4.4 Summary

Table 8 shows comparison of UFS checker and traditional concolic tester (CUTE).

Generated Paths	UFS	CUTE	Unwanted Feature Founded	UFS	CUTE
Test1	3	2	Test1	Yes	No
Test2	3	2	Test2	Yes	No
Test3	4	2	Test3	Yes	No

Table 8: Comparison of UFS Checker and CUTE

We find that our ALERT with UFS checker can traverse the program with more paths. And since we model the unconventional symbolic data propagation, we can effectively trigger the unwanted features of a program.

5 Conclusion

Our work presents a tool which can effectively find the “acceptable wrong inputs” which cause the program to act abnormally. These inputs are called the “out-of-specification” inputs, and the presence of this kind of input means the program implementation does not exactly meet its design.

We do not simulate all local variables operations in stack when symbolic execution, such as stack space allocation, since the addresses of local variables depend on the compiler, operating system and computer architecture. Simulating all combinations of above three items is tedious, and cannot be fully enumerated. Instead, once we discover a memory overlap phenomenon that can cause a fault, we will calculate if we can repeat it in other combination of compiler, operating system and computer architecture by modifying the arguments when conducting symbolic execution.

Our USF Checker extends the ability of concolic testing. The two phase execution and simulation of real memory operations push the concolic testing closer to the concrete side of the program testing. With our work, the information lost in the instrumenting process is recovered, thus we can get more accurate program testing result.

5.1 The Uncertainty Principle

Currently, concolic testing schemes are all done by an instrumented program, and collects information only from it. This may lose important information of the original (uninstrumented) program. In our work, we collect the real information from the original program first, then mix them into concolic execution. So concrete part of our testing is closer to the program’s real behavior. That is why we can capture and resolve the “real” unspecified software features.

5.2 Future Work

ALERT platform with the USF Checker is a promising tool. After resolving unspecified software features, we can further manipulate the program behavior. In other words, we can evaluate the program’s vulnerability.

1. More uninitialized parts

(a) Union data type

Union data type is a special structure in C/C++ language, which can store more types with different size and share with the same space. If we use larger type before smaller type, the bytes used by larger type will not be cleared, and the left-over values can be used for other way.

(b) Unused padding bytes

When conducting experiments, we found that the padding for alignment between a function's two arguments may change from the options given to the compiler. These unused padding bytes may also be used for storing information and shell codes for malicious usage.

2. Integration with other dynamic analysis tools

(a) Valgrind [3]

In this thesis, we choose GDB as the tool for run-time information collection. However, GDB's capability is limited. For example, we cannot retrieve the information such as external library calls and `malloc()/free()` operations in the heap section. Seward and Nethercote's work [26] presented a method to shadow memory by a piece of metadata, called a definedness bit, which can globally locate used but not defined error. We would integrate Valgrind into our framework to improve the accuracy in testing.

3. Exploit Generator

With the tool like Metasploit [23], whose ability of customizing attacking, and the ability to find the attacking point for our tool, we can develop a powerful exploit generator. The symbolic propagation chain in the program found by our testing framework can be used for advanced attacks. That is, we can inject exploit code through the symbolic propagation chain

References

- [1] GDB: The GNU Project Debugger, <http://www.gnu.org/software/gdb/>.
- [2] Objective Caml, <http://caml.inria.fr/ocaml/>.
- [3] Valgrind, <http://valgrind.org/>.
- [4] IEEE Standard Glossary of Software Engineering Terminology, IEEE std 610.12-1990, 1990.
- [5] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 203–213, New York, NY, USA, 2001. ACM Press.
- [6] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 113–130, London, UK, 2000. Springer-Verlag.
- [7] Thomas Ball and Sriram K. Rajamani. Generating abstract explanations of spurious counterexamples in C programs. Technical report, Microsoft Research, Redmond, 09 2002.
- [8] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, New York, NY, USA, 2002. ACM Press.
- [9] Clark Barrett and Sergey Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker. In Rajeev Alur and Doron A. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification (CAV '04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer-Verlag, July 2004. Boston, Massachusetts.
- [10] Cristian Cadar and Dawson R. Engler. Execution generated test cases: How to make systems code crash itself. In *Proceedings of the 12th SPIN Workshop on Model Checking Software*, San Francisco, USA, 2005.
- [11] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 322–335, New York, NY, USA, 2006. ACM Press.
- [12] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.
- [13] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: a tool for using specifications to check code. In *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 87–96, New York, NY, USA, 1994. ACM Press.

- [14] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM Press.
- [15] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, New York, NY, USA, 2005. ACM Press.
- [16] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. SYNERGY: a new algorithm for property checking. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 117–127, New York, NY, USA, 2006. ACM Press.
- [17] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software verification with Blast. In *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN)*, volume 2648 of Lecture Notes in Computer Science, pages 235–239. Springer-Verlag, 2003.
- [18] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
- [19] Gerard J. Holzmann. Static source code checking for user-defined properties. In *IDPT '02: Proceedings of The 6th World Conference on Integrated Design & Process Technology*, Pasadena, CA, USA, 2002.
- [20] Stephen Johnson. Lint, a C Program Checker. Technical Report 65, Bell Laboratories, 1978.
- [21] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 14–14, Berkeley, CA, USA, 2001. USENIX Association.
- [22] Flavio Lerda and Willem Visser. Addressing dynamic issues of program model checking. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 80–102, New York, NY, USA, 2001. Springer-Verlag New York, Inc.
- [23] Metasploit LLC. The Metasploit Project: An Equal Opportunity Exploiter, <http://www.metasploit.com/>.
- [24] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag.
- [25] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 263–272, New York, NY, USA, 2005. ACM Press.

- [26] Julian Seward and Nicholas Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference*, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.
- [27] Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A Cooperating Validity Checker. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceedings of the 14th International Conference on Computer Aided Verification (CAV '02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 500–504. Springer-Verlag, July 2002. Copenhagen, Denmark.
- [28] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model Checking Programs. In *ASE '00: Proceedings of the 15th IEEE international conference on Automated software engineering*, page 3, Washington, DC, USA, 2000. IEEE Computer Society.

