

國立交通大學

資訊科學與工程研究所

碩士論文

C 程式中正負整數轉換錯誤之偵測

Detection of Integer Signedness Faults in C Programs



研究生：李泳毅

指導教授：黃世昆 教授

中華民國九十七年四月

C 程式中正負整數轉換錯誤之偵測

研究生：李泳毅

指導教授：黃世昆

國立交通大學資訊科學與工程研究所碩士班

摘 要

每天都有新弱點被發佈，其中有些惡名昭彰的弱點是多數程式員所熟悉的，例如，錯誤使用未限制的複製函數、或接受格式字串的函數。近幾年來，一種稱為整數錯誤的新型態程式弱點被發掘，許多重要的應用程式都有這樣的程式弱點，例如 Microsoft Internet Explorer 以及 PHP 等系統。這種弱點是由於整數溢位後，被運用於配置記憶體。因為整數溢位，配置的記憶體將少於所預期的數量，因而也造成記憶體溢位問題。正負整數轉換是所有整數問題的一部分，我們將在論文中探討、偵測此類問題的方法。

在論文中，我們提出一種偵測技術，以檢驗正負整數轉換相關運算，找出 C 程式可能的錯誤。此方法是基於程式控制流、與實際/符號混合測試技術(Concolic testing)。當測試到潛在問題時，我們使用總體特性檢查 (Universal Property Checking)，更進一步驗證常見、且已知軟體的弱點，判斷是否會引發正負整數轉換問題。

我們提出的方法，已在 linux2.6.17 平台上評估，對幾個有代表性的程式類型進行測試。這些類型分別為：(1)正號整數轉負號整數、(2)負號整數轉正號整數、(3)以及語意錯誤。對於真實案例評估，我們也偵測到 qemu 0.8.2 中的程式錯誤。

關鍵字：隨機測試、軟體驗證、正負整數轉換錯誤

Detection of Integer Signedness Faults in C Programs

Student : Yung-Yi Li

Advisor : Shih-Kun Huang

Institute of Computer Science and Engineering
National Chiao Tung University

Abstract

New vulnerabilities in software come out every day. Some of them are so infamous that most programmers are familiar with, e.g. misuse of unbounded copy functions or format string functions. A new type of vulnerability, called integer errors, emerges in recent years. Many major applications suffer from this kind of vulnerability, for example, Microsoft Internet Explorer and PHP. The vulnerability is caused by integer overflow and the integer is then used as size field to allocate heap memory. Because of the integer overflow, the allocated heap space is far less than what the programmers expect, thereby causing heap overflow then.

We have developed a technique that aims at finding integer signedness bugs in C programs. This technique is based on CONCOLIC-testing (CONCcrete and symbOLIC) and control-path analysis. The control path analysis of the target program will help us identify the program input data which cause a suspicious integer conversion. This suspicious integer conversion may turn to integer signedness bugs by some rare input data. Then we use concolic testing and universal checking to verify whether there is a feasible bug that will be caused by this suspicious integer conversion.

The proposed method, called reflter algorithm, has been evaluated in Linux 2.6.17 with several representative program examples, including signed-to-unsigned and unsigned-to-signed conversions, along with semantic bugs. This method also detects a real bug in qemu 0.8.2.

Keywords: random testing, model checking, integer signedness fault

誌 謝

對於這篇論文的完成，首先要感謝我的指導教授黃世昆教授。這一年多來受到黃教授許多指導。能夠在黃教授經營的研究環境中完成我的碩士論文，我十分感謝。感謝實驗室的昌憲學長對於我在方法、實驗以及論文寫作上的幫助，以及半年多來不斷的互相討論，無論成果如何，都讓我的研究所生涯收穫豐富許多。最後要感謝實驗室的夥伴們：彥佑、揚杰、立文、志晏、友祥、大中、文健、琨翰以及蕙蘭，是你們構成我研究生涯的主體。歡笑、挫折、努力，一切的一切，在實驗室的兩年生活是我永難磨滅的回憶。



Table of Contents

1 Introduction	1
1.1 Numbers in Computer Science and integer error	1
1.2 Integer Conversion	3
1.3 Potentially Dangerous Integer Conversion	4
1.4 Signedness Problems	6
1.5 Input Validation.....	8
1.6 Our Approach	8
2 Software Verification, Testing Coverage and Concolic Testing	10
2.1 Dynamic verification	10
2.2 Test generation	10
2.3 Manually-generated and automatically generated test	11
2.4 Static verification	12
2.5 Concolic testing	12
2.6 ALERT	13
3 Algorithm.....	14
3.1 Refilter Algorithm	14
3.2 The Second Phase	15
3.3 Contribution.....	21
4 Implementation	22
4.1 ALERT Implementation.....	22
4.2 Refilter Algorithm Implementation.....	30
5 Evaluation.....	32
5.1 Signed-to-unsigned Conversion.....	32
5.2 testing of TestAntiSniff.....	34
5.3 Comparison of Calls to Universal Checker	37
5.4 Testing Detail of AntiSniff	38
6 Discussions	42
6.1 False Positive	42
6.2 False Negative	42
7 Related Works	43
8 Conclusions	45
9 References	46
Appendix A: Source Code of Modified AntiSniff	48

List of Figures

FIGURE 1: 4-BIT UNSIGNED NUMBER WHEEL	2
FIGURE 2: 4-BIT SIGNED NUMBER WHEEL	2
FIGURE 3: ALL CATEGORIES OF INTEGER CONVERSION IN C99.....	6
FIGURE 4: EXAMPLE OF INPUT VALIDATION	7
FIGURE 5: SAFE RANGE AND UNSAFE RANGE WHEN SIGNED INTEGER AND UNSIGNED INTEGER CONVERT TO EACH OTHER	8
FIGURE 6: EXAMPLE OF DISADVANTAGE OF RANDOM TESTING.....	11
FIGURE 7: ILLUSTRATION OF REFILTER ALGORITHM.....	14
FIGURE 8: AN EXAMPLE OF SAFE AND UNSAFE RANGE	15
FIGURE 9: SUCCESSFUL INPUT VALIDATION	18
FIGURE 10: UNSUCCESSFUL INPUT VALIDATION	18
FIGURE 11: EXAMPLE OF PATH CONDITION.....	19
FIGURE 12: EXAMPLE OF PATH CONDITION MIX INTEGER CONVERSION CONSTRAINT	20
FIGURE 13: PSEUDO CODE OF REFILTER ALGORITHM	20
FIGURE 14: ARCHITECTURE OF ALERT	23
FIGURE 15: ALERT SIMPLIFICATION.....	24
FIGURE 16: ALERT SIMPLIFICATION.....	25
FIGURE 17: CONSTRAINTS REPRESENT POINTER READ GENERATED BY ALERT.....	28
FIGURE 18: CONSTRAINTS REPRESENT POINTER WRITE GENERATED BY ALERT	29
FIGURE 19: CIL INSTRUMENTATION OF CHECKER.....	31
FIGURE 20: A BUG FOUND IN QEMU 0.8.2 ne2000_receive().....	33
FIGURE 21: SIGNED TO UNSIGNED, UPCAST.....	33
FIGURE 22: UNSIGNED TO SIGNED, THE SAME RANK	34
FIGURE 23: TESTANTI_SNIFF_1.0.C	34
FIGURE 24: DNS PACKET	35
FIGURE 25: TESTANTI_SNIFF_1.1.C	35
FIGURE 26: TESTANTI_SNIFF_1.1.1.C	36
FIGURE 27: TESTANTI_SNIFF_1.1.2.C	37
FIGURE 28: NUMBERS OF CALLS TO UNIVERSAL CHECKER	38
FIGURE 29: WHILE LOOP WILL RUN FOREVER WHEN COUNT BECOMES -1	39
FIGURE 30: A WELL-FORMED INPUT	40
FIGURE 31: A INPUT THAT CAUSE BUFFER OVERFLOW.....	40
FIGURE 32: THE ORIGINAL SOURCE CODE OF ANTI_SNIFF.	41
FIGURE 33: AN EXAMPLE OF FALSE POSITIVE.....	42
FIGURE 34: AN EXAMPLE OF FALSE NEGATIVE	42

1 Introduction

New vulnerabilities in software come out every day. Some of them are so infamous that most programmers are familiar with, e.g. misuse of unbounded copy functions or format string functions. Therefore, these vulnerabilities almost disappear in major applications today[1].

A new class of vulnerability, called integer errors, emerges in recent years. Many major applications suffer to this kind of vulnerability, e.g., Microsoft Internet Explorer[2] and PHP[3].

The vulnerability is caused by integer overflow and the integer is then used to allocate heap memory. Because of the integer overflow, the allocated heap space is far less than what the programmer thinks, thereby causing heap overflow later.

1.1 Numbers in Computer Science and integer error

Numbers are ubiquitous in computer system and mathematics. But numbers in computer systems are different with numbers in mathematics. Numbers in mathematics includes integer, rational number, real number, etc. Numbers in computer system are several bytes in memory annotated with type. Char, short, int, long are some common types of numbers used in computer systems. Usually they are in different size, that is, they occupy different number of bytes in the memory. In mathematics numbers can be as big as you wish in pen and paper, while numbers are limited by the type and its representation in computer system. Therefore, numbers in computer system has some limitation that one in mathematics does not have. For example, if x is in set Z , then x can be 0, 1, -1, 2, -2, etc. But a variable of type unsigned char in computer system usually has max value 255. If programmers do not realize this and perform operation on number in computer system as in mathematics, some errors may occur. For example, assume there is a 4-bit integer in computer system named n . If we store its value in 2's complement format, the value of n is illustrated in Figure 1. We can see that when n is 7, performing of $n+1$ will result in -8 rather than 8. If n is a 4-bit unsigned integer and the value is 15 then $n+1$ will result in 0 rather than 16, as illustrated in Figure 2. When this happens, we say n is "wrap around." Wrap around means when a number in computer system increases beyond its upper bound or decreases below its lower bound, it is

forced to become another number that is different from what it should be in mathematics. Sometimes wrap around causes integer error.

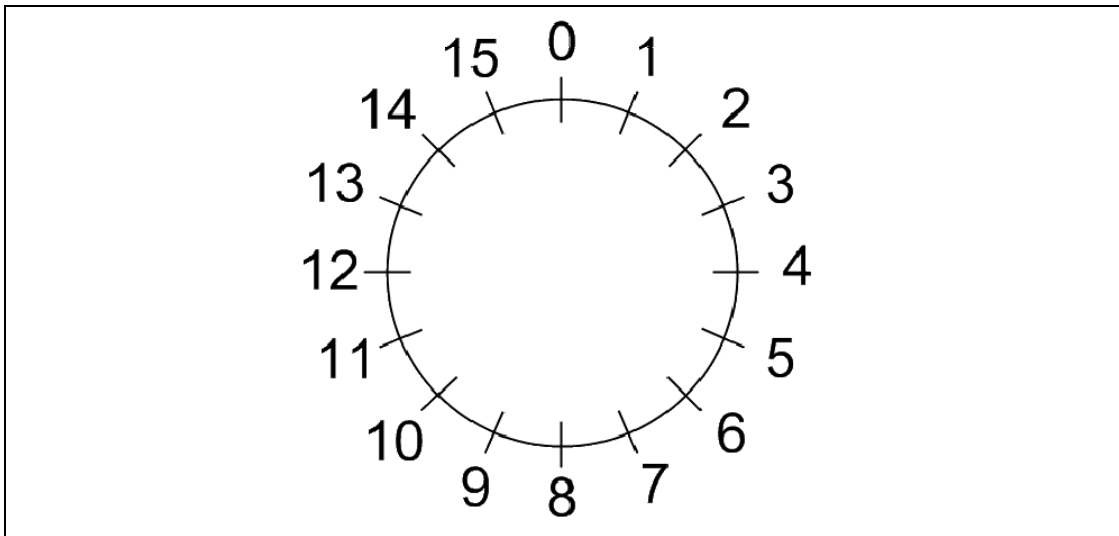


Figure 1: 4-bit unsigned number wheel

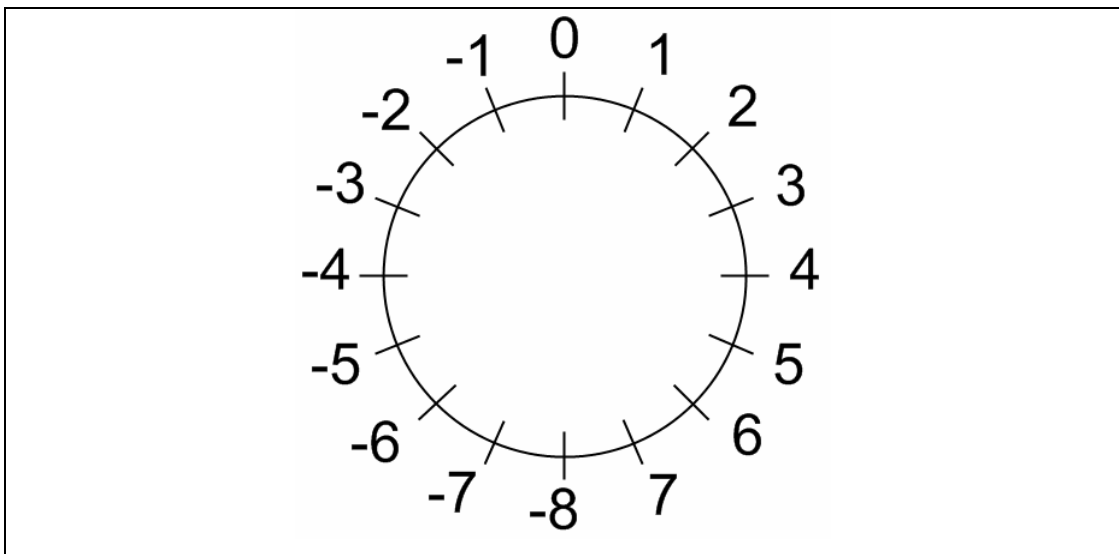


Figure 2: 4-bit signed number wheel

Integer errors happen when programmers use integer operation in computer system but get unexpected result. For example, we increase a 4-bit unsigned integer 15 by 1 and expect to get 16. But we will get 0 instead. If this is what we have expected, we may assume it is 16 and use it in somewhere else and cause some errors. These errors may become exploits in the worst case.

There are four kinds of integer error: integer overflow, integer underflow, truncation problem, and signedness problem. Integer overflow and integer underflow occur when the result of an integer operation exceed its range of representation.

Truncation problem and signedness problem occur when converted number is not in the range of representation of new type. We will describe integer conversion and signedness problem in the following sections.

1.2 Integer Conversion

Integer conversion is an assignment operation from an integer number to another integer number. But the type of the converted number is different with the type of new integer number. Because of different type, converted integer number must be transformed into new type and trying not to lose original information. Not all integer conversions can perfectly reserve the information carried in converted number, sometimes they do lose information. This fact makes some integer conversion potentially dangerous.

Every integer type has an integer conversion rank, which is used to decide the result type of an integer conversion. As a general rule, the larger in size a type is, the higher rank it is in the conversion rank. The detailed definition of conversion rank is defined in C99 standard as follows:

1. No two signed integer types shall have the same rank, even if they have the same representation.
2. The rank of a signed integer type shall be greater than the rank of any signed integer type with less precision.
3. The rank of long long int shall be greater than the rank of long int, which shall be greater than the rank of int, which shall be greater than the rank of short int, which shall be greater than the rank of signed char.
4. The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type, if any.
5. The rank of any standard integer type shall be greater than the rank of any extended integer type with the same width.
6. The rank of char shall equal the rank of signed char and unsigned char.
7. The rank of `_Bool` shall be less than the rank of all other standard integer types.
8. The rank of any enumerated type shall equal the rank of the compatible integer type
9. The rank of any extended signed integer type relative to another extended signed

integer type with the same precision is implementation-defined, but still subject to the other rules for determining the integer conversion rank.

10. For all integer types T1, T2, and T3, if T1 has greater rank than T2 and T2 has greater rank than T3, then T1 has greater rank than T3.

The detail of rules of integer conversion of C language is defined in C99 standard. The rules are as follows:

Signed and unsigned integers

1. When a value with integer type is converted to another integer type other than `_Bool`, if the value can be represented by the new type, it is unchanged.
2. Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.
3. Otherwise, the new type is signed and the value cannot be represented in it; either the result is implementation-defined or an implementation-defined signal is raised.

Usual arithmetic conversions

1. If both operands have the same type, then no further conversion is needed. Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.
2. Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type.
3. Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, then the operand with unsigned integer type is converted to the type of the operand with signed integer type.
4. Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

1.3 Potentially Dangerous Integer Conversion

There are several integer conversions that are potentially dangerous. We will discuss them under C99 standard. We categorize all integer conversions in C99 standard into twelve kinds, as illustrated in Figure 3. We observe that four kinds of integer conversion may cause truncation problem, three kinds of integer conversion may cause signedness problem, and the others are safe. We summarize this table into three rules:

1. Converting any integer to lower rank is dangerous.
2. Converting a signed integer into an unsigned integer that has the same rank is dangerous, and vice versa.
3. Converting a signed integer into an unsigned integer that has higher rank is dangerous.



rank signed	Down Cast	Eqi Cast	Up Cast
Signed to Unsigned	(a)Truncate	(b)Bit pattern preserved	(b)Sign extension
Unsigned to Signed	(a)Truncate	(b)Bit pattern preserved	(d)Zero extension
Signed to Signed	(a)Truncate	(c)Bit pattern preserved	(d)Sign extension
Unsigned to Unsigned	(a)Truncate	(c)Bit pattern preserved	(d)Zero extension

Figure 3: All categories of integer conversion in C99

In these rules, some conversion is dangerous because some values cannot be represented in the new type. There is a lot of vulnerabilities come from this unsafe integer conversion. For each of those unsafe integer conversions, we define two ranges: safe range and unsafe range. Safe range means that any value in this range can be represented in the new type. On the other hand, the unsafe range means that any value in the range that cannot be represented in the new type. For example, a signed integer i is converted into an unsigned integer j , then i of value 0 is in the safe range, but i of value -1 is in the unsafe range.

1.4 Signedness Problems

In C99 standard, a signed integer casts to an unsigned integer following these rules:

1. Convert to a number has lower conversion rank: truncate converted number to match the size of new type
2. Convert to a number has the same conversion rank: preserve the bit pattern of the converted number.
3. Convert to a number has higher rank: sign extension to a signed type whose conversion rank is the same as new type, then do as convert to a number has the same rank as 2 shows.

All these three cases will cause some problems. For example, we often take a user input data buf as well as its length. We save the length in a signed integer, len. And we check it to make sure it is less than the maximum size, so we write a input validation as a one in Figure 4.

```
if (len > max)
    Raise exception
else
    memcpy(buf2, buf, len);
```

Figure 4: Example of input validation

If len of value -1 then it will pass the check. But when len is used in memcpy(), it is converted into an unsigned integer, so it is actually 0xffffffff, nearly 4GB. And this request will definitely denied in 32-bit machines.

Recently, a lot of signedness vulnerability is discovered. This shows that many programmers are not aware of the potential danger brought by careless conversion operation. Even those experienced programmer can make mistake when using conversion operation. And, sanity check added by programmer can act not as programmers have expected. So we focus only on (b) in Figure 3. That is, we focus on dangerous conversion between signed type variable and unsigned type variable. We will provide an efficient method to discover this kind of bug.

When a signed int i is converted into an unsigned int j in a 32-bit system, the safe range is 'i >= 0' and the unsafe range is 'i < 0'. When an unsigned int j is converted into a signed int i, the safe range is 'j < 0x7fffffff' and the unsafe range is 'j >= 0x7fffffff'. As a rule of thumb, the safe/unsafe range can be determined by the MSB: it is in safe range if the MSB is zero. Otherwise, it is in unsafe range as illustrated in Figure 5.

Integer Conversion	Type of i	Type of j	Safe range	Unsafe range
i = j	Unsigned int	Signed int	j >= 0	j < 0
i = j	Signed int	Unsigned int	j <= 0x7fffffff	j > 0x7fffffff

Figure 5: Safe range and unsafe range when signed integer and unsigned integer convert to each other

1.5 Input Validation

The signedness bugs can be avoided when all the integers are declared as unsigned because there is no conversion between signed and unsigned integer in the first place. However, this is usually impossible in the real programs.

Another way to avoid this bug is using accurate input validation. Almost all programs receive input from the outside. Browsers read web pages from web servers via the socket. Editors read documents from the file system. Console programs read options from the command line. All these programs accept the input data and perform their work. However, if they use the input data to perform important operations, they must check it first. The check process is called input validation. Programmers must assume all the input data are malicious. For the integer type of input, the maximum and minimum acceptable values must be checked with. All input values outside of the legal range are rejected.

However input validation is usually absent, because there are many integers are used in program that programmers often forget to check some of them. Sometime, input validation is done but not correctly (e.g., not checking maximum value, not checking minimum value, or both). Thus, we propose a testing mechanism to check all suspect signedness conversion.

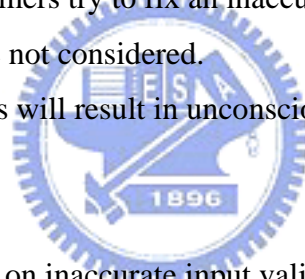
1.6 Our Approach

Our approach is like an automatic input validation method based on concolic testing proposed by K Sen in year 2005.

Concolic testing is a technique that is capable of expanding all execution paths of a program and collects symbolic data along the paths. By this we can accurately trace data flow of input data in which we are interested. We will check data flow of input value to decide whether it will be transformed to another type and lose its information after this operation. Therefore, our method is able to detect unsafe input which is not filtered by programmers' input validation. This idea is inspired by some common mistake that usually made by programmers, such as misuse of signed and unsigned integer type. Programmers may have the following problems:

1. Programmers may not know the standard of C language well.
2. Programmers may assume they have known the standard of C language well but actually they do not.
3. Programmers may forget to add input validation.
4. Programmers may assume they have added accurate input validation.
5. Even when programmers try to fix an inaccurate input validation, there still may be some cases that are not considered.

All of the above problems will result in unconsciously writing of potentially dangerous code.



Our method focuses on inaccurate input validation when input value is transformed between signed type and unsigned type. Usual input validation requires programmers' knowledge of what is valid input. Our method can be regard as a second input validation automatically added to program which requires no program specification. In the future, we hope to expand our method to make it suitable for all kinds of integer conversion.

2 Software Verification, Testing Coverage and Concolic

Testing

Software verification is a set of methods that are used to assure that the all expected requirements in developing software are achieved. There are two approaches to verification: Dynamic verification and static verification.

2.1 Dynamic verification

Dynamic verification is usually called testing. It is proceed during the execution of software to check whether program behavior is in accordance with expected requirement. Testing can be categorized into the follows according to their test scope:

1. unit testing
2. module testing
3. integration testing
4. system testing
5. acceptance testing

Unit testing is a test only on single function. Complete unit testing is the root of a good software testing.



2.2 Test generation

Testing is performed by using test suite as the input of the tested program. The base of testing is generating a large number of effective input. An effective input can increase the code coverage of the testing. Code coverage is usually measured by the following basic metrics:

1. function coverage
2. statement coverage
3. condition coverage
4. path coverage
5. entry/exit coverage.

Function coverage measures the number of functions being executed in the program. Statement coverage measures the line of code been executed in the program. Condition coverage measures the possible execution path been executed in the program. Entry/exit coverage measures the number of call and return of function been executed in

the program. Safe-critical applications are often required to achieve 100% of some metrics in testing.

2.3 Manually-generated and automatically generated test

Test input used in testing can be generated in two different ways: manually generate and automatically generate. Manually generated test input is generated by developer, tester or even user. A developer can write down his own test suite and tester can do the same. Users that come into program failure or annoying weird behavior of program can send error trace back to the software vendor. All these are sources of manually generated test input. Manually generated test input are usually well-formed and have the corresponding expected result of program execution that be checked with. But manually-generated test input has a main disadvantage: the cost per test generation is high.

Automatically input generation overcome the disadvantage of manually-generated input. Inputs are generated in a fast and cheap way. The typical automatically input generation technique is random testing. Random testing is a technique that automatically generates a large number of random test input. But random testing has two main disadvantages: generated input is not well-formed and cannot avoid generating repeated input.

An input that is not well-formed is unlikely to get pass the input validation of a function. Therefore there is only little chance to increase code coverage whatever the metric is. For example, probability of randomly generated input pass the if statement in the line 2 in Figure 6 is $1/2^{32}$. Redundant input in random testing is another problem although is not as severe as the former.

```
01 Void foo(int i){
02     If(i == 0)
03         //Passed
04     Else
05         Abort();
06 }
```

Figure 6: example of disadvantage of random testing

2.4 Static verification

Static verification is another approach to verify software. It is proceed without actually execution of software. The follows are common method of static verification:

1. Code Inspection
2. Formal Verification
3. Software Model Checking
4. Program analysis

Software model checking is adapting the technique in model checking to check the properties of program. There are two ways to do this:

1. Abstract a real program to a model which a model checker can handle.
2. Make a model checker capable of dealing with real program directly.

Software model checking is able to complete search for program states of a program. Therefore it is able to check whether a property is hold in a program completely. But it has a main disadvantage: state explosion. State explosion occurs when the probable program state grows rapidly during states transition. The speed of growth is so quick that software model checking can hardly used in real program. Recently, two software model checking related tool BLAST[4] and SYNERGY[5]are published. They improve software model checking by dynamically refine abstract model and combine concrete execution information, respectively.

2.5 Concolic testing

Concolic testing is a testing technique that combines symbolic information as in software model checking and concrete information as in random testing. It is first mentioned in DART[6].

Concolic testing is able to expand all execution paths of a real program. Therefore, concolic testing achieves 100% of path coverage naturally. All program properties can be checked along each execution path as long as the information needed for checking is available. And it can be modified to achieve 100% of other metrics, too.

Concolic testing overcomes the disadvantages of random testing and software model checking. High test coverage is easily achieved and there is much less state explosion problem. Concolic testing is inspired by symbolic execution. Symbolic execution is first mentioned in the paper of King[7], and is improved by God. Concolic

testing is a successful method of combining symbolic information and concrete information.

2.6 ALERT

ALERT is a concolic testing tool inspired by CUTE. Like other concolic testing tools, the main feature of ALERT is that it combines both dynamic execution and static execution. ALERT is capable of expanding all distinct control flow path of a program. The ALERT execution model is like a mix of CUTE and EXE[24]. ALERT uses a big loop as a skeleton of execution model. An iteration of this big loop generates a distinct control flow path of a program. This part of ALERT is just like CUTE. EXE uses a different approach. EXE uses fork method to expand different control flow of a program. Each fork call generates a child process to execute toward distinct control flow. If this program contains a never ended loop, ALERT will stop tracking new branching of control flow of a program on a threshold. For example, if an execution path flow through a thousand if-statements, then ALERT will stop tracking the if-statements in the remaining execution.

ALERT uses a symbolic memory model like EXE does. ALERT uses real memory address of a program and its size to track every memory region which is used by a program. All primitive type, array, structure and dynamic allocated memory region will be tracked. We uses a pair of value (start address, offset) to record every memory region. A memory address which is not locates in any range of used memory objects is not legal. CUTE uses a logic address to record all memory a program it used.

ALERT symbolically simulate all machine instruction in concrete execution. This is the reason that ALERT can accurately find a distinct control flow path of a program. ALERT simulate simplified C language. C language is first simplified by a source to source transformation tool. After simplification, a C code will result in a 3-address like code. Then ALERT can easily simulate all C language statements, i.e. all machine instructions

3 Algorithm

In this section we will describe our algorithm to detect signedness conversion and to check whether it is indeed result in software vulnerability. We can see a big picture of this algorithm in Figure 7. First we generate input by concolic testing technique. Then we execute this program and mark all signedness conversion. Then we check if it is dangerous. If it is indeed dangerous, we finally check parameters of memory/string related functions. We will describe in detail in the following paragraphs.

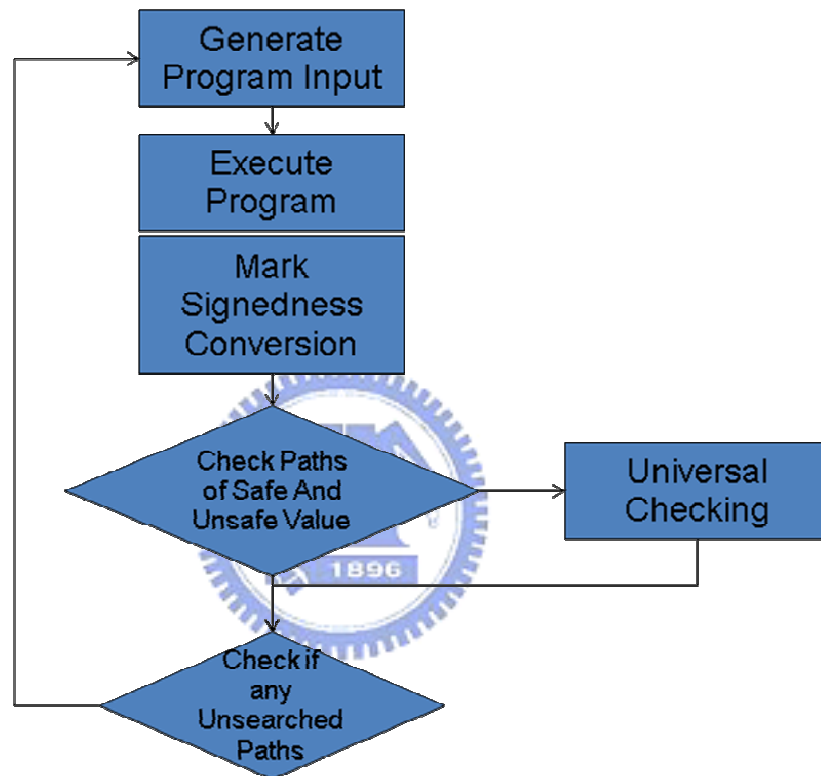


Figure 7: Illustration of Refilter Algorithm

3.1 Refilter Algorithm

Insecure coding style may cause problems. One of these problems becomes vulnerability. There may be several kinds of bug that caused by these habits. There may also be semantic bug that caused by these habits. It is inefficient to check all the bugs by merely universal checking and execution path expanding. And it is also not possible to check an unknown semantic bug by universal checking, because we do not know what to check with the specification.

We propose a 2-phase approach to address this problem:

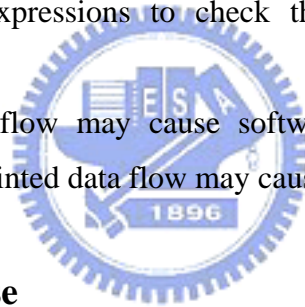
1. Find suspicious execution path that cause by bad coding habits.

2. Check whether there is any memory error along the suspicious execution path.

We focus on a specific kind of bad coding habit here: careless integer conversion and lack of input validation. Careless integer conversion may cause buffer overflow and other kinds of problem. To find all problems without describing them in detail first, we find them by our refilter algorithm. Refilter algorithm is a specialized technique that aimed at finding unfiltered or incorrectly filtered tainted data flow. Once we find an unfiltered tainted data flow, we perform phase 2 of the Refilter Algorithm: universal checking all memory operations along the suspicious execution path. In phase 2 we will check whether there is some specific kind of bug that will easily caused by the unfiltered taint data flow.

Tainted data are those influenced by input directly or indirectly. These must be checked before going into the trusted zone in the program. The task is usually called input validation, which can reject unwanted data or accept desired ones. Programmers can use predicates or regular expressions to check the value of data. These checks are error-prone.

Unfiltered tainted data flow may cause software vulnerability and make software exploitable. Unfiltered tainted data flow may cause logical bug of a program.



3.2 The Second Phase

The key idea is: if programmers do input validation right, an unsafe input value should never trigger the same execution path as its safe counterpart. If not, user may use very large value as input and make program crash. We define a safe range as follows: when an integer conversion operation happens in a program, the value of the converted variable may not be able to be represented by the format of new type. When the logical meaning of value is preserved, it is in “safe range.” Otherwise, it is in “unsafe range.” This idea is illustrated in Figure 8.

```
01 char i;  
02 unsigned char j;  
03 i = -1;  
04 j = i;
```

Figure 8: An example of safe and unsafe range

After execution of line 4 in Figure 8, j becomes 255(0xff) while its original

meaning is -1. So -1 is in the unsafe range of integer conversion in line 4. {So a safe/unsafe range is actually goes with an integer conversion operation, but not a variable alone.}

In dealing with signedness problem, we can expect safe/unsafe ranges of the following form: $a \geq 0$ or $a < 0$. For example, safe range of the integer conversion in line 4 of Figure 8 is “ $i \geq 0$ ” while the unsafe range is “ $i < 0$.”

The goal is to find out whether unsafe input value can trigger the same execution path as safe input value does. There may be a lot of potentially dangerous integer conversions all over the program. To check them efficiently and soundly, we propose a testing method based on ALERT. We call this method “refilter algorithm” because when ALERT perform this algorithm, basically ALERT is doing extra input validation for programmers. If unsafe input value does not filtered out by input validation in program, ALERT will check (filter) it by refilter algorithm.

The Refilter algorithm consists of two steps:

1. Monitor the occurrence of unsafe value data flow.
2. Check whether targeted unsafe value data flow is dangerous.

A unsafe value is a value in unsafe range. An unsafe range is defined by a specific integer conversion. Therefore, to monitor the occurrence of unsafe value, ALERT must identify potentially dangerous integer conversions. To identify potentially dangerous integer conversions, we have to search for all integer conversion in the program we want to check. We use CIL for this task, which builds up an abstract syntax tree (AST) for the program. Then we can traverse this AST and search for integer conversions. Once we find an integer conversion, we insert a checker call into this AST. Then CIL transforms this AST back to source code.

To check targeted unsafe value data flow is dangerous, ALERT must check whether the unsafe value flow through the target integer conversion and flow along the current execution path is dangerous. ALERT must know whether a unsafe value will flow through the same path as a safe counterpart does. ALERT achieve this by checkers and checking function executed at the end of each ALERT iteration. Checkers collects information until the whole execution path is decided. Once the whole current execution

path is decided, we can check the complete unsafe value data flow of each potentially dangerous integer conversion along this execution path. If ALERT checks an unsafe value data flow when the current execution path is not complete, then it may find out a unsafe value flow through partial execution path the same as safe value flow through. But this unsafe value may be filtered out by some input validation in the following execution path.

Just inserting checker call into source code does not fulfill our goal. Those checker need to be triggered. ALERT will systematically search for all execution paths and triggers all checker along the execution paths it finds. Therefore, all checkers will be triggered, that means all integer conversions will be checked.

When checkers are triggered, they check whether a specific integer conversion is really a dangerous one. This task can be performed by CIL, but can also be done during runtime. If the integer conversion checked by ALERT is a dangerous one, ALERT add corresponding constraint to CVCL and these constraints will be solved together with path conditions latter. What constraint is going to be added is depend on types involved in integer conversion and concrete value of converted variable when conversion is performed. The types involved in the integer conversion decide what is safe range and unsafe range of this integer conversion. The concrete value of the converted variable when conversion is performed in runtime decides whether we add a constraint corresponds to safe range or unsafe range. If concrete value is in safe range, we add a constraint corresponding to the unsafe range. Otherwise, we add the one corresponding to the safe range.

At the end of current execution path triggered by ALERT, we enter the main part of this algorithm. ALERT use information collected along current execution path to check whether all integer conversions in the path are safe or unsafe one by one. ALERT keeps track of each dangerous integer conversion and the safe/unsafe range of that. By this, we can ask whether value of each dangerously converted variable can be in another range. If they are in safe range when integer conversion is performed, we want to find out whether they can be in unsafe range and still trigger the same execution path as current path, and vice versa. The unsafe value should be filtered out by input validation of a program. An unsafe value should be handled by exception handling part of a program.

Figure 9 & Figure 10 illustrate this idea. CVCL will solve this and tell ALERT whether it is possible or not. This will generate false positives and false negatives because we do not model all operations that a machine can perform. If CVCL tells ALERT it is possible, ALERT will get input data. Then users can check whether it is feasible by execute the uninstrumented program with input data generated by ALERT.

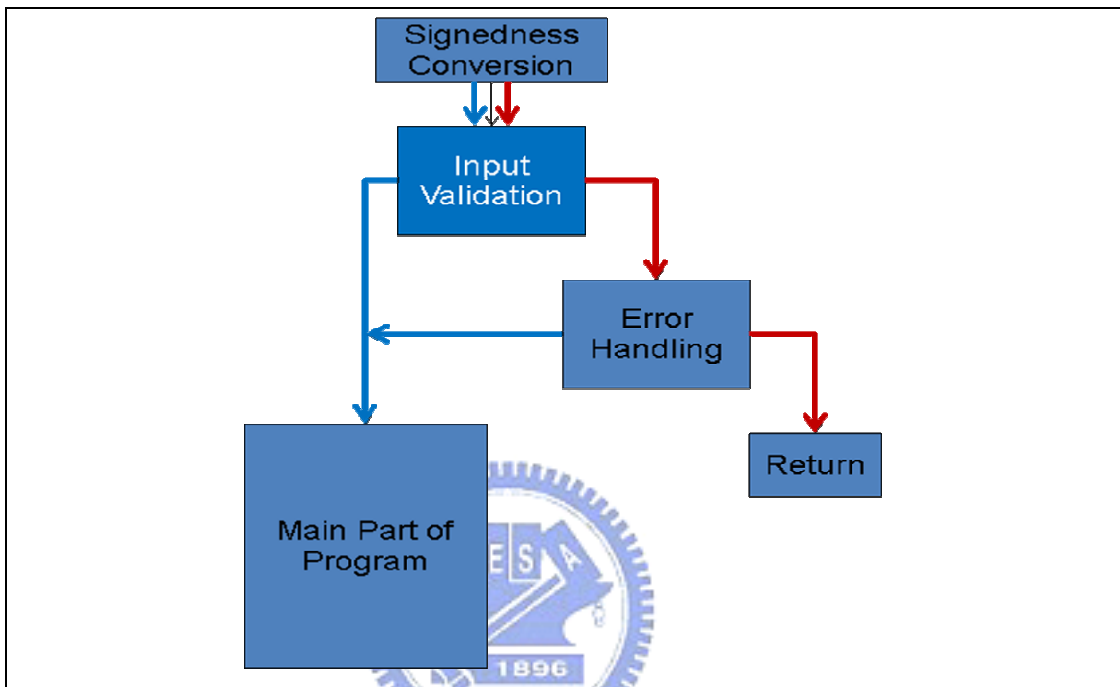


Figure 9: Successful input validation

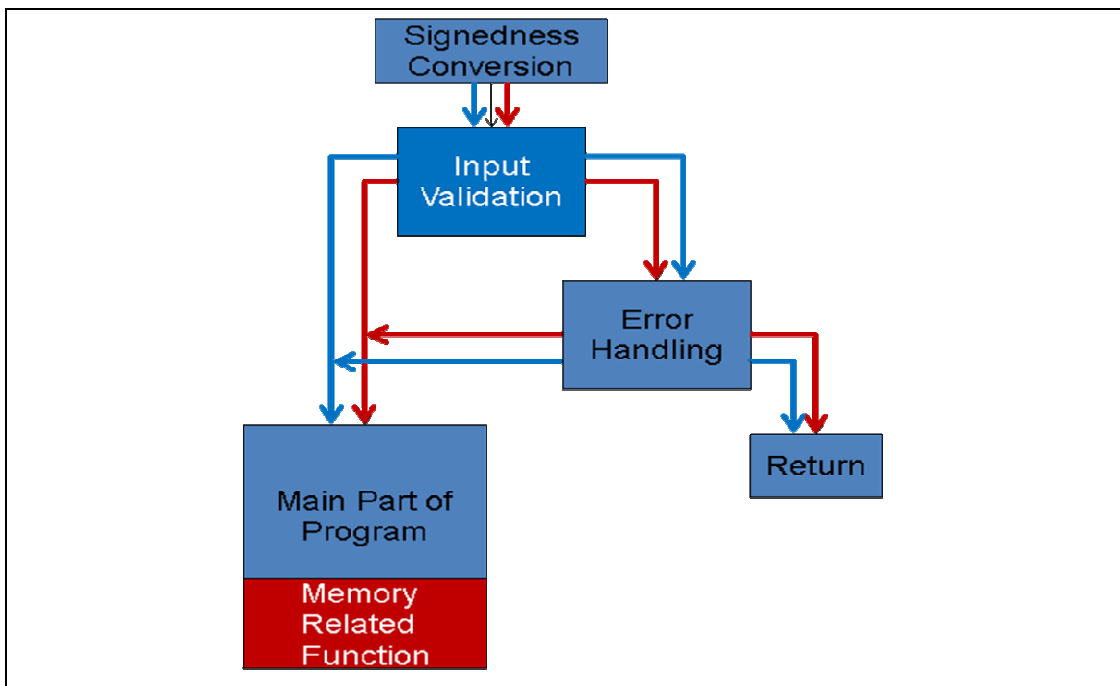
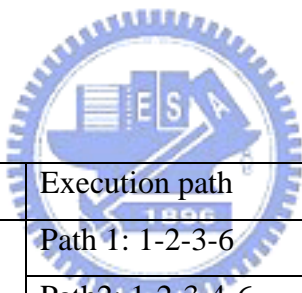


Figure 10: Unsuccessful input validation

When we solve a set of constraint by a solver, we are asking the solver whether these constraints are true in all cases. If it is possible to make the set of constraints false, the solver will give a counter-example that makes the set of constraints evaluates to false. Path conditions are a set of constraints that every transition from initial state to current state of a specific program counter must meet. For example: If the execution path of this program marked by line number in Figure 11 is 1-2-3-6, then variable i must larger than 10 or the execution path will lead to line 4. The path condition of 1-2-3-6 is “ $i \geq 10$.”

Refilter algorithm collects also the constraint of integer conversion itself. When constraints of integer conversion are solved together with path conditions, ALERT are simply checking whether a specific variable can be in a specific range when integer conversion happens while follow current execution path. For example: If the solver report invalid to this set of mixed constraints of Path 2 in Figure 12, then it is impossible to make the variable $i < 0$ on executing line 4 while execution path is 1-2-3-4-5-6.



Source code	Execution path	Path condition
01 int i;	Path 1: 1-2-3-6	$i \geq 10$
02 Int j;	Path2: 1-2-3-4-6	$(i < 10) \& (i + j \leq 5)$
03 If(i < 10)	Path3: 1-2-3-4-5-6	$(i < 10) \& (i + j > 5)$
04 if(i+j > 5)		
05 printf(“foo”);		
06 return;		

Figure 11: example of path condition

Source code	Executoin Path	Path Condition	Integer conversion constraint
01 int i=input();	Path 1: 1-2-3-6	(i>=10)	
02 unsigned int j;	Path 2: 1-2-3-4-5-6	(i<10)	(i@line4 < 0)
03 If(i < 10){			
04 j = i+1;			
05 malloc(j); }			
06 return;			

Figure 12: example of path condition mix integer conversion constraint

Our algorithm is shown in Figure 13.

```

while(there exist some path not searched){
    inputData = getNextInput();
    executeAndMarkUnsafe(inputData);
    for(each of marked signedness conversion){
        safeRange = markedConversion.safeRange;
        unsafeRange = markedConversion.unsafeRange;
        if( solve(PathConstraint, safeRange)&&
           solve(PathConstraint, unsafeRange))
            universalChecking();
    }
    generateNextInput();
}

```

Figure 13: Pseudo code of refilter algorithm

We consider the following four kinds of bug are most related to the unfiltered taint data flow:

1. Memory related library function. The functions with parameter of type `size_t` should be checked, such as `malloc(size_t)`. When calling these functions, the corresponding argument should not be negative.
2. String library function with boundary checking, such `strncpy(char*, const char*, size_t)`.
3. Array index out of boundary

4. Big loop index variable

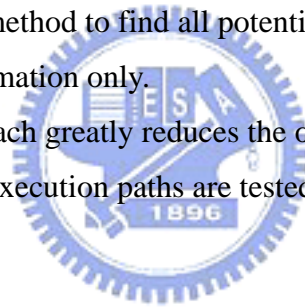
We can instrument a universal check before these instructions to check whether these bugs will occur.

When ALERT executes the suspicious execution path, it will triggers checkers. If any checkers should fail, ALERT will generates the input data that make it fail. Then we can use these input data to check whether it will really cause a problem.

Universal checks are not able to find semantic bugs. If all universal check fails, then programmer should consider this suspicious execution path is cause by a semantic bug.

3.3 Contribution

1. We proposed a new method to find all potential signedness errors. This method rely on control flow information only.
2. The two-phase approach greatly reduces the overhead of the universal checking in that only suspicious execution paths are tested with fully universal checking.



4 Implementation

4.1 ALERT Implementation

ALERT is a concolic software testing tool. The idea of ALERT implementation is inspired by both CUTE[8] and EXE{ref}. Therefore, its implementation is like a mix of them. ALERT learns the CUTE style of concrete execution, which is different from EXE's. CUTE uses depth-first search style in its path searching and finds out single complete execution path at one iteration. EXE implement this in a different way. EXE use depth-first search style, too. But EXE searches for all execution paths simultaneously. When EXE reaches a control branch, it forks child processes to follow each direction. Searching in other directions pauses until searching in one direction is full completed.

There are differences in symbolic storage model of CUTE and EXE, too. CUTE uses logical input map to record symbolic information of bytes in memory, while EXE uses CRED-like memory region recording. CUTE marks memory regions with primitive types or pointer type while all bytes in EXE are non-type. CUTE is not able to handle bitwise operation, which EXE handles well. CUTE does not handle pointer aliasing problem well while EXE do much better than CUTE in solve pointer aliasing.

The solvers they use are different, too. CUTE uses lp-solver[9] as its solver while EXE[24] uses CVCL[10] and STP[11].

ALERT uses CUTE's style of concrete execution and EXE's style of symbolic storage model. Refilter algorithm is implemented as a module of ALERT platform. The architecture of ALERT and refilter algorithm are both illustrated in Figure 14. The refilter checker in Figure 14 is respond for check if the safe range and unsafe range value trigger the same execution path.

The source code in test is first processed by CIL. CIL will do some simplification and instrumentation to the source code. ALERT receives the instrumented source code form CIL and add a test driver. ALERT then compiles the instrumented source code to an executable. The executable will get the input data from a input file during each ALERT iteration. Each input data will trigger an execution path. Finally, ALERT will

stop because of either there is no other path to be explored or the program in test crashes.

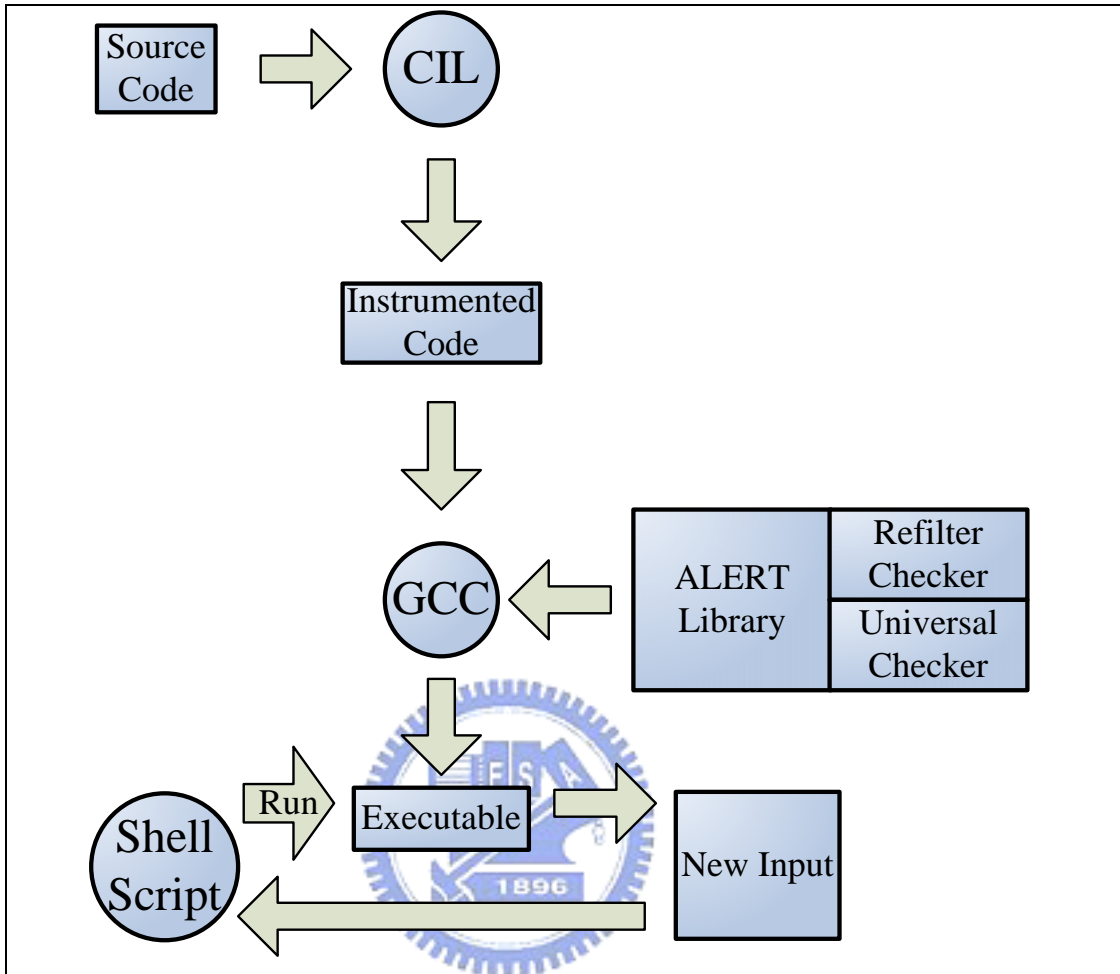


Figure 14: Architecture of ALERT

ALERT uses another solver other than ones used by CUTE and EXE: CVC3. We are preparing to try STP on ALERT recently.

ALERT uses CIL as its source code instrumentation tool and also its source to source transformation tool. ALERT uses the source code simplification feature of CIL to make things simpler to handle. CIL simplified C source code to a 3-address like code while reserving the original semantic. For example, a for loop statement is transformed to an infinite while loop with goto instruction and labels to jump on. This simplification is illustrated in Figure 15. Another example is simplification of struct type. The variables defined in the struct type are renamed. To preserve semantics, each location that uses these variables is transformed too. The address relation between these variables is calculated. This simplification is illustrated in Figure 16.

(Before simplification)

```
1 void testme(int i){
2     char j;
3     j = i;
4     while(j <= 5)
5         printf("j <= 5");
6 }
```

(After simplification)

```
6 #line 5 "test.c"
7 extern int printf() ;
8 #line 1 "test.c"
9 void testme(int i )
10 { char j ;
11     int __cil_tmp3 ;
12     int __cil_tmp4 ;
13     int __cil_tmp5 ;
14
15     {
16 #line 3
17     j = (char)i;
18 #line 4
19     while (1) {
20 #line 4
21         __cil_tmp3 = (int)j;
22 #line 4
23         __cil_tmp4 = __cil_tmp3 <= 5;
24 #line 6
25         __cil_tmp5 = ! __cil_tmp4;
26 #line 4
27         if (__cil_tmp5 != 0) {
28             goto while_0_break;
29         }
30 #line 5
31         printf("j <= 5");
32     }
33     while_0_break: ;
34 #line 1
35     return;
36 }
37 }
```

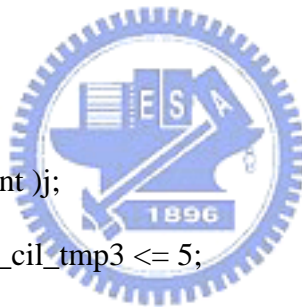


Figure 15: ALERT simplification

(before simplification)

```
1 typedef struct _FOO{
2     int i;
3     char c;
```

```
4 }Foo;
5
6 void testme(void){
7     Foo f;
8     f.i = 1;
9     f.c = 'a';
10 }
(after simplification)
6 #line 6 "test00.c"
7 void testme(void)
8 { char f_c2 ;
9   int f_i3 ;
10
11   {
12 #line 8
13   f_i3 = 1;
14 #line 9
15   f_c2 = (char)'a';
16 #line 6
17   return;
18 }
19 }
```

Figure 16: ALERT simplification

The ALERT implementation includes two main parts: concrete execution and symbolic execution. Of course, the interaction between concrete execution and symbolic execution is also very important and will be described in this chapter.

The skeleton of the concrete execution of ALERT is a big loop. At each iteration of this loop, ALERT will search for a new execution path and solve the input which will trigger new execution path. This big loop contains the following steps:

1. Get input data.
2. Execute test driver.
3. Execute the instrumented tested program.
4. Use SMT-solver to generate new input for next ALERT iteration.

The symbolic execution is blended with concrete execution in this big loop. Symbolic execution of ALERT is basically simulation of all concrete information by SMT-Solver constraints.

ALERT symbolically executes the program by simulating how program are executed in real machine. ALERT achieves this by the follows:

1. ALERT defines a corresponding symbolic instruction for each possible instruction in a program.
2. ALERT symbolically simulates every instruction in the current execution path according the definition in 1.
3. ALERT keeps the symbolic information for every byte of memory used by the program in bit-level precision.

In the following section the detail of the above will be described.

Basically ALERT keeps symbolic information of every byte in memory. But if ALERT keeps a record for every byte in a 4GB system, ALERT will definitely need more than 4GB of memory to do that. In order to avoid this, ALERT uses a CRED like memory map called “object map” for record all memory used in the tested program. ALERT keeps a record for each used variable in object map with its type, start address in memory and size. An object is basically a sequence of bytes with a name attached to it. An object can be a primitive type variable, an array of some type, or variable of user defined type. ALERT also keeps the information of whether a single byte in memory is symbolic. If a byte in memory is symbolic, then ALERT will trace all operations that it involves in. ALERT uses some constraints to record the relation between this symbolic byte and other bytes in memory. These relation are built up by symbolically simulating instructions in a program one by one. ALERT does not keep redundant information concrete variables because they are not influenced by the input. Note that ALERT cannot control execution path only via manipulating the input.

Arithmetic operations are basically simulated by the build-in methods of CVC3 although the division operation is not supported. Therefore almost all arithmetic operations that a program can perform are symbolically simulated by ALERT. Numbers used in arithmetic operation are divided into two categories: integer numbers and floating point numbers. All integer operations are simulated by ALERT. These operations are: addition, subtraction, multiplication, division and modular operation. Other arithmetic operations are transformed into these four basic operations by CIL. The division operation is replaced by our self-assembly division operation by using the CVC3 built-in multiplication operation.

ALERT does not handle floating point number operation. Therefore some false

positives may generate.

Relational, logical and bitwise operation are simulated by the build-ins of CVC3 although some functionally repetitive operations are missing. The missing operations can be implemented with the counterpart operation. Therefore, all these three kinds of operations are fully handled by ALERT.

Assignment statements are essential in any program. They are the main body of data flow in program. ALERT handles assignment with the help from CIL. CIL will simplify the program into 3-address like code. After CIL simplification, all assignment statements in the program will become one of the following forms:

1. $Lvalue = A \text{ op } B$
2. $Lvalue = A$

We do not consider memory related operation like dereference here. The memory related operations are described in the following section.

The main task of simulating assignment statement is the symbolic information linkage between the Lvalue and the right hand side of the assignment operator. But there is a problem to be dealt with: first, variables in a SMT-solver have no time information. For example: “int a; a=1; a=2;” in C program is different from “int a; a=1; a=2; ” in a SMT-solver. The result of the former is variable a has value 2 at the end of execution. The later just gives message “Invalid” if we query the solver with a constraint “a=2”. Our solution is a variable renaming mechanism. For each variable used in SMT-solver, we mark it with timestamp along the execution path. For example, variable i in time 0 will be i_0 in the solver. The time information of a variable gives us the capability to trace the symbolic dataflow of every variable.

To propagate the symbolic information first we check whether the variables involve in assignment statement is symbolic. All cases of are as follows:

1. Both A and B are concrete
2. Otherwise

If both of A and B is concrete, then Lvalue is also concrete after this assignment. In this case, we mark Lvalue as concrete and let the original program do its concrete execution. Otherwise, the Lvalue is symbolic after assignment. In this case, we make a constraint

that Lvalue is equal to the right hand side of the assignment operator. For example: assignment statement “a=b” may becomes “a_1 equals b_0” in SMT-solver.

ALERT handles most pointer operations. But pointer aliasing is not fully handled. Pointer operation includes pointer read and pointer write. After simplification of CIL, all pointer operations in program are as follows:

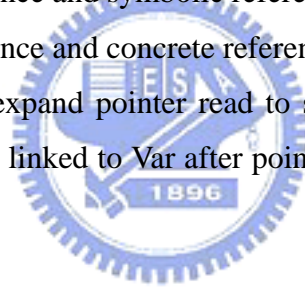
1. Var = *(pointer)
2. *(pointer) = Var

The first is called pointer read, and the second is pointer write.

After simplification of CIL, pointer read operations are reduced to single form: Var = *(pointer). All pointer read operations are divided into four cases:

1. Symbolic reference and symbolic referent.
2. Symbolic reference and concrete referent.
3. Concrete reference and symbolic referent.
4. Concrete reference and concrete referent.

In case 1, ALERT will expand pointer read to several conjunct constraints. Symbolic information of referent is linked to Var after pointer read is expanded. This is illustrated in Figure 17.



Source Code	Constraints
<pre> 1 void testme(int i,int j){ 2 int b[3]; 3 int k; 4 b[0] = i; 5 b[1] = 2*i; 6 b[2] = i+10; 7 k = b[j]; 8 } </pre>	<pre> (((FALSE OR ((mem_16_1_1 = 0bin10111111111011100111001010000100) AND (k_1_1 = (b_97[3]_1 @ b_97[2]_1 @ b_97[1]_1 @ b_97[0]_1)))) OR ((mem_16_1_1 = 0bin10111111111011100111001010001000) AND (k_1_1 = (b_97[7]_1 @ b_97[6]_1 @ b_97[5]_1 @ b_97[4]_1)))) OR ((mem_16_1_1 = 0bin10111111111011100111001010001100) AND (k_1_1 = (b_97[11]_1 @ b_97[10]_1 @ b_97[9]_1 @ b_97[8]_1)))) </pre>

Figure 17: Constraints represent pointer read generated by ALERT

Case 2 is similar to case1 except the reference is concrete. There is nothing to do.

Case 3 and 4 is similar to the cases in assignment statement. The way ALERT handle them is similar, too. ALERT get referent and decide whether it is symbolic by a

way similar to how ALERT decide whether a variable is symbolic. The remains are the same as in assignment statement.

After simplification of CIL, pointer write operations are reduced to one form:

*(pointer) = Var. All pointer write operations are divided into four cases:

1. Symbolic reference and symbolic Var.
2. Symbolic reference and concrete Var.
3. Concrete reference and symbolic Var.
4. Concrete reference and concrete Var.

In case 1, ALERT will expand pointer read to several conjunct constraints. This step is much like the one in pointer read case 1. But the difference is that ALERT will make sure symbolic information of those bytes which are not assigned is keep not changed after pointer write. To do this we have to expand pointer write operation to a large conjunction of constraints. This is illustrated in Figure 18.

Source Code	Constraints
<pre> 1 void testme(int i,int j){ 2 int b[3]; 3 b[i] = j; 4 }</pre>	<pre> (((FALSE OR ((mem_16_1_1 = 0bin10111111110011111001000010010100) AND (k_1_1 = (b_97[3]_1 @ b_97[2]_1 @ b_97[1]_1 @ b_97[0]_1)))) OR ((mem_16_1_1 = 0bin10111111110011111001000010011000) AND (k_1_1 = (b_97[7]_1 @ b_97[6]_1 @ b_97[5]_1 @ b_97[4]_1)))) OR ((mem_16_1_1 = 0bin10111111110011111001000010011100) AND (k_1_1 = (b_97[11]_1 @ b_97[10]_1 @ b_97[9]_1 @ b_97[8]_1))))))</pre>

Figure 18: Constraints represent pointer write generated by ALERT

Case 2 is similar to case1 except the Var is concrete. There is no need to link the symbolic information.

Case 3 and 4 is similar to the cases in assignment statement. The way ALERT handle them is similar, too. ALERT get the Var and decide whether it is symbolic like ALERT decide whether a variable is symbolic. The remains are the same as in assignment statement.

ALERT handle control flow related statement by help from CIL. CIL transform all control related statement to a mixing of if, while, call/return statements.

When finding an if statement, CIL inserts two functions on the “then” block and the “else” block. In the “then” block, the function records the path condition of an execution path goes along the “then” side. The path condition includes:

1. Evaluation of the condition expression, “true” in this case.
2. Condition expression.
3. Other information needed in implementation.

When finding a call/return statement, CIL inserts functions that help to propagate the information between the caller and callee. Without side-effect, the most important relation between the caller and callee is the parameter and return value. ALERT will link the relation between function parameters and return value by variable renaming.

4.2 Refilter Algorithm Implementation

Refilter Algorithm is implemented as a module of ALERT. It works independently and has no influence on the original ability of ALERT. Implementation of Refilter algorithm consists of two parts: CIL insertion of refilter checker and refilter checker implementation.

ALERT uses CIL to insert checker before every integer conversion. The instrumentation is illustrated in Figure 19. ALERT can only insert checker before potentially dangerous integer conversion rather than all, but the difference is little. ALERT uses CIL to collect the follows in an integer conversion: variable name, variable type, constant value. The collected information is used in runtime.

(Before instrument)

```
1 void testme(int i){
2     unsigned char c;
3     c = i;
4     if(c = 'a')
5         printf("foo");
6 }
```

(After instrument)

```
21 #line 3
22     c = (unsigned char)i;
```

```

23 #line 3
24  _sqSymExec("c", T_UCHAR, OP_NOP, "i", (unsigned int )i, T_INT,
"SQ_constant", 0,
25          T_INT, T_UCHAR, -1, -1);
26 #line 3
27  _signednessCheck("c", T_UCHAR, OP_NOP, "i", (unsigned int )i, T_INT,
"SQ_constant",
28          0, T_INT, T_UCHAR, -1, -1);
29 #line 4
30  c = (unsigned char )'a';
31 #line 4
32  _sqSymExec("c", T_UCHAR, OP_NOP, "SQ_constant", (unsigned int )'a',
T_INT, "SQ_constant",
33          0, T_INT, T_UCHAR, -1, -1);
34 #line 4
35  _signednessCheck("c", T_UCHAR, OP_NOP, "SQ_constant", (unsigned
int )'a', T_INT, "SQ_constant",
36          0, T_INT, T_UCHAR, -1, -1);

```

Figure 19: CIL instrumentation of checker

The implementation of refilter checker consists of two parts: Add corresponding constraints and solve constraint. These two parts are implemented in different functions.

Refilter checker decides whether it is a potentially dangerous integer conversion at runtime. If so, ALERT records this integer conversion and its corresponding constraint. The corresponding constraint is a constraint to limit the value of converted variable. If the concrete value of the converted variable is in safe range, ALERT adds a constraint which limits the value in unsafe range and vice versa.

Another part of refilter checker is executed at the end of an ALERT iteration. In this part ALERT will ask SMT-solver to solve the path constraints and the integer conversion constraints one by one. If the SMT-solver answers “valid”, then there is unfiltered unsafe input value. ALERT will warn the user if it finds an unfiltered unsafe input value.

5 Evaluation

The evaluation was performed on a machine with Intel Core 2 Duo 1866 MHz CPU and DDR2-667 1GB memory.

The operating system of our machine is linux of kernel version 2.6.17. The version of CIL is 1.3.6. The version of CVC3 is 1.2.1. And the version of GCC is 4.1.2.

Most signedness errors are easy to exploit if the variable is used as the size for memory-related library, such as malloc, memcpy and memset. Our method can find not only this kind of bug, but also semantic bug. In this section, we evaluate our method with four test cases, which are signed-to-unsigned conversion, unsigned-to-signed conversion, signed-to-unsigned upcast, and a semantic bug. The first one is the real bug in qemu and the others are our test cases. We successfully found all the bugs in these test cases.

5.1 Signed-to-unsigned Conversion

Figure 20 shows a bug found in qemu 0.8.2. The original advisory can be found in <http://tavisio.decsystem.org/virtsec.pdf>. The test case is the minimized version of function ne2000_receive and only the core part of this vulnerability, which is a typical example of signed-to-unsigned conversion bug.

The parameters index and s_stop are controlled by the user, which make the user be able to control the variable and make its value negative while parameter **size** is not controllable. In line 13, the sanity check can be bypassed as long as the variable avail is negative. In line 14, memcpy uses the variable len as third parameter, which denote the size. The negative signed integer becomes a large unsigned integer. The size is definitely not expected. Our method finds that the signed variable len is implicitly cast into an unsigned variable, so the variable len is taken into tracking.

```
1 #include <stdlib.h>
2 void
3 testme(int index, unsigned s_stop, int size){
4     int avail, len;
5     unsigned char mem[48*1024];
```

```

6  unsigned char buf[48*1024];
7
8  if(size > 0){
11     avail = s_stop - index;
12     len = size;
13     if (len > avail)
14         len = avail;
15     memcpy(mem + index, buf, len);
16 }
17 }

```

Figure 20: A bug found in qemu 0.8.2 ne2000_receive()

Figure 21 is a test case to demonstrate the signed-to-unsigned upcast operation. In line 8, we assign the unsigned integer *m* with a signed character *n*. If *n* is negative number, e.g., -1, then *m* becomes a very big number (0xffffffff in this case). In line 9, *m* is using as the size parameter of function `malloc`. Function `malloc` can hardly allocate a memory space of this size. If this program does not check the return value for NULL, subsequence use of the pointer *p* will result in memory error.

```

1 #include <stdlib.h>
2
3 void testme(char n){
4     unsigned int m;
6     unsigned int *p;
7
8     m = n;
9     p = malloc(m);
10 }

```



Figure 21: signed to unsigned, upcast

Figure 22 is a test case to demonstrate the unsigned-to-signed conversion. We assign an unsigned character *n* into a signed character *m*, which is used as the index of the array `dat`. If *n* is 255, then *m* is -1. This will result in a buffer underflow in line 10.

```


1 #include <stdlib.h>
2
3 #define size 300
4
5 void testme(unsigned char n){
6     char m;
7     unsigned char buf[size];
8     unsigned char dat[size];
9     m = n;
10    *(buf) = *(dat+m);

```

Figure 22: unsigned to signed, the same rank

5.2 testing of TestAntiSniff

AntiSniff is network card promiscuous mode detector. The following demonstrates how a bug in the DNS packet-parsing code of AntiSniff is wrongly fixed and how the problem of fix can be caught by our method.



```

1 #include "test.h"
2 #include <stdlib.h>
3 #define MAX_LEN 256
4 void testme(char *pkt)
5 {
6     char *indx;
7     int count;
8     char nameStr[MAX_LEN]; //256
9     memset(nameStr, '\0', sizeof(nameStr));
10    indx = (char *) (pkt);
11    count = (char) *indx;
12
13    while (count) {
14        (char *) indx++;
15        strncat(nameStr, (char *) indx, count);
16        indx += count;
17        count = (char) *indx;
18        strncat(nameStr, ".", sizeof(nameStr) - strlen(nameStr));
19    }
20    nameStr[strlen(nameStr) - 1] = '\0';
21
22 }

```

Figure 23: testAntiSniff_1.0.c

This code which in Figure 23 snippet shows a process that extract the domain name from the packet and copy it into the nameStr string. The packet looks like the one in Figure 24.

4	N	c	t	u	3	e	d	u	0
---	---	---	---	---	---	---	---	---	---

Figure 24: DNS packet

The condition expression of the while loop in line13 check whether **count** is equal to 0. When it is not, this loop goes on. If the length of the domain named that copied is larger than **MAX_LEN**, which is 256 in this case, a buffer overflow bug will occur. Refilter algorithm reports nothing about line13 because it is not a signedness bug. But refilter algorithm reports that there is an signedness bug in line16. If the value of **count** is negative and used as the length parameter of the strncpy function in line9, which means **count** is translated to a large unsigned value, a buffer overflow occurs.

```

1 #include "test.h"
2 #include <stdlib.h>
3 #define MAX_LEN 256
4 void testme(char *pkt)
5 {
6     char *indx;
7     int count;
8     char nameStr[MAX_LEN]; //256
9     unsigned int count2;
10    memset(nameStr, '\0', sizeof(nameStr));
11    indx = (char *) (pkt);
12    count = (char) *indx;
13    while (count) {
14        if (strlen(nameStr) + count < (MAX_LEN - 1)) {
15            (char *) indx++;
16            strncpy(nameStr, (char *) indx, count);
17            indx += count;
18            count = (char) *indx;
19            strcat(nameStr, ".", sizeof(nameStr) - strlen(nameStr));
20        } else {
21            printf("Alert! Someone is attempting "
22                "to send LONG DNS packets\n");
23            count = 0;
24        }
25    }
26    nameStr[strlen(nameStr) - 1] = '\0';
27
28 }

```

Figure 25: testAntiSniff_1.1.c

Refilter algorithm reports that a signedness bug is detected in line14(Figure 25). This is the same bug that shows up in line16 in AntiSniff1.0.c. The negative value in count will passed the check in line 15. When **count** is added with the return value of function strlen, it will be casted first to unsigned int, which means **count** will becomes a large unsigned int variable. Then **count** is added with strlen(nameStr), and easily cause it result in wrap around. The result of count plus strlen(nameStr) will be a unsigned integer smaller than **MAX_LEN-1** and get pass the check in line 14.

```

1 #include "test.h"
2 #include <stdlib.h>
3 #define MAX_LEN 256
4 void testme(char *pkt)
5 {
6     char *indx;
7     int count;
8     char nameStr[MAX_LEN]; //256
9     unsigned int count2;
10    memset(nameStr, '\0', sizeof(nameStr));
11    indx = (char *) (pkt);
12    count = (char) *indx;
13    while (count) {
14        if ((unsigned int) strlen(nameStr) + (unsigned int) count <
15            (MAX_LEN - 1)) {
16            (char *) indx++;
17            strncat(nameStr, (char *) indx, count);
18            indx += count;
19            count = (char) *indx;
20            strncat(nameStr, ".", sizeof(nameStr) - strlen(nameStr));
21        } else {
22            printf("Alert! Someone is attempting "
23                "to send LONG DNS packets\n");
24            count = 0;
25        }
26    }
27    nameStr[strlen(nameStr) - 1] = '\0';
28
29 }

```

Figure 26: testAntiSniff_1.1.1.c

In testAntiSniff_1.1.1.c(Figure 26), refilter algorithm reports the same bug that found in AntiSniff1.1.c(Figure 25). This is because that the explicit cast added right before the strlen() is redundant. The return value of strlen is unsigned int type from the

very begining.

```
1 #include "test.h"
2 #include <stdlib.h>
3 #define MAX_LEN 256
4
5 void testme(char *pkt)
6 {
7     unsigned char *indx;
8     unsigned int count;
9     unsigned char nameStr[MAX_LEN]; //256
10    memset(nameStr, '\0', sizeof(nameStr));
11    indx = (char *) pkt;
12    count = (char) *indx;
13
14    while (count) {
15        if (strlen(nameStr) + count < (MAX_LEN - 1)) {
16            indx++;
17            strncat(nameStr, indx, count);
18            indx += count;
19            count = *indx;
20            strncat(nameStr, ".", sizeof(nameStr) - strlen(nameStr));
21        } else {
22            printf("Alert! Someone is attempting "
23                "to send LONG DNS packets\n");
24            count = 0;
25        }
26    }
27    nameStr[strlen(nameStr) - 1] = '\0';
28
29 }
30 }
```

Figure 27: testAntiSniff_1.1.2.c

In the final fix of AntiSniff(Figure 27), the three variable **indx**, **count** and **nameStr** is declared as unsigned type. This solves all problems and Refilter algorithm reports nothing.

5.3 Comparison of Calls to Universal Checker

We try to use refilter algorithm to decrease the original ALERT way of detecting some bug trigger by integer signedness problem like the one in Figure 21. If the program logic is complicated, ALERT may performs many unnecessary universal checks. Refilter algorithm can help ALERT by filter out the execution path without any signedness conversion. We do universal check only when the current execution path fail

to pass refilter algorithm. This means that we may avoid a lot of calls to universal checker.

We use AntiSniff as our target again. The detail of testing is in the next section. We can see the result in Figure 28. The result is promising. We can see that the number of calls to universal checker of refilter algorithm is 1/6 of the counterpart. When the program has no signedness bug like AntiSniff1.1.2, refilter algorithm spends no universal check while original ALERT spends thousands before halt.

	AntiSniff 1.0	AntiSniff 1.1	AntiSniff 1.1.1	AntiSniff 1.1.2
Refilter Algorithm	295	298	298	0
Original ALERT	1881	1868	1868	4289

Figure 28: Numbers of calls to universal checker

5.4 Testing Detail of AntiSniff

This section record the detail of our testing of AntiSniff. Because of the prototype ALERT is unstable, our testing task is full of surprises. The details is listed as following:

1. We have used the original version of AntiSniff (Figure 32) rather than simplified one (Figure 23) in the evaluation of efficiency.
2. If variable **count** in line 37 equals to -1, then the while loop in line37 will run forever. To explain this, we can see while loop will be stop if **count** is equals to 0. If count is set to -1 by the value pointed by **indx** in line35 or line41, it will pass the stop condition of while loop in line37. In line38, pointer **indx** increase by 1. Then in line40 **indx** is added by **count**, which make it point to the original position in line35 or line41. This is illustrated in Figure 29.

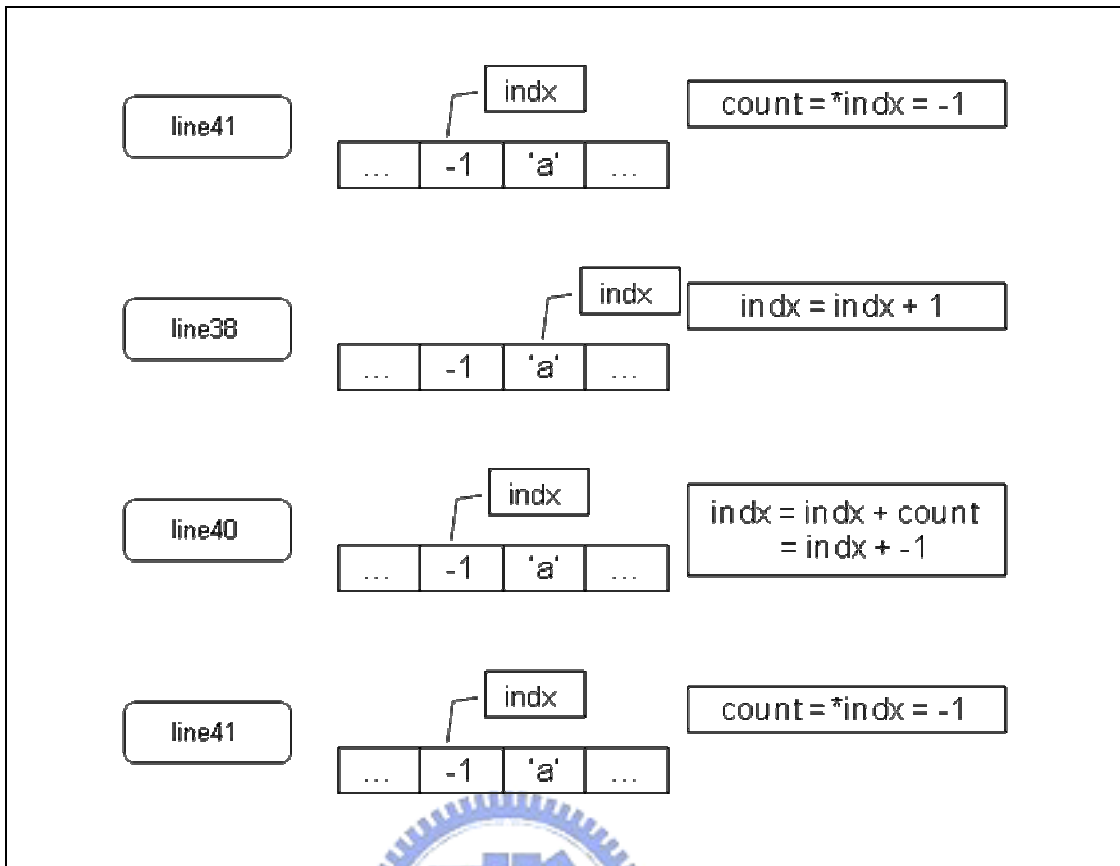


Figure 29: While loop will run forever when count becomes -1

3. During including the standard library source code, we found the `ntohs` library calls is implemented by in-line assembly which cannot be analyzed by CIL. This problem is not solved yet, we simply omit this function call and some code related to it. This omission brings no effect to the absence and presence of the integer signedness bug. Therefore, this omission brings little effect to the evaluation of our algorithm.
4. When adding universal checker to the code, we found the execution time is very long. If we set `MAX_LEN` to 256 as the original code, the SMT-solver will hang. We try to set `MAX_LEN` to 100 or lower, but the execution time is still too long that we do not sure whether it will going to stop. Finally we set `MAX_LEN` to 20. We also make size of function parameter `pkt` to 20. Using this configuration, we have tested AntiSniff successfully.
5. When `MAX_LEN` is 256, the original AntiSniff has a buffer overflow problem as illustrated in Figure 31. Usually a input data of `pkt` will looks like Figure 30. But when a input looks like a one in Figure 31, a buffer overflow problem will occur.

This is due to the lack of checking in the original code. Unfortunately, the solver does not care whether the generated input will cause a buffer overflow problem. So we have to write a checking code that make the input looks like a one in Figure 30. We write some lines of checking code to ensure the input will be well formed.

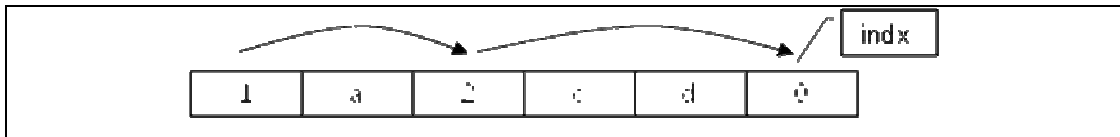


Figure 30: A well-formed input

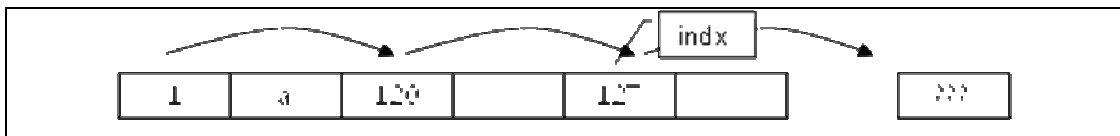
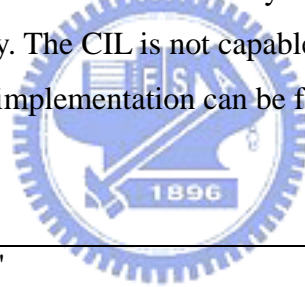


Figure 31: A input that cause buffer overflow

6. We have implemented our own library to replace the standard C library. The reason is that the source code of standard C library is been optimized, and some of them using in-line assembly. The CIL is not capable of handle in-line assembly or gcc extensions. Our own implementation can be found in Appendix A, together with the modified AntiSniff.



```

1 #include "includes.h"
2 #include "anti_sniff.h"
3
4 int watch_dns_ptr(char *pkt, int len, char *ip_match){
5     HEADER dns_h;
6     int dns_offset, rr_offset, rr_size;
7     int count, questionEntries;
8     char *indx;
9     char nameStr[MAX_LEN];
10    char matchPTR[32];
11    int min_str_len;
12
13    memset(nameStr, '\0', sizeof(nameStr));
14
15    if (!make_ptr_str(ip_match, matchPTR)){
16        fprintf(stderr, "error making ptr lookup address\n");
17        return FALSE;
18    }
19
20    dns_offset = SIZE_ETHER_H + SIZE_IP_H + SIZE_UDP_H;
21    rr_offset = dns_offset + SIZE_DNS_H;
22

```

```

23  if (len < SIZE_ETHER_H + SIZE_IP_H + SIZE_UDP_H + SIZE_DNS_H)
24      return FALSE;
25
26  rr_size = len - rr_offset;
27
28  memcpy(&dns_h, (char *) (pkt + dns_offset), sizeof(HEADER));
29
30  questionEntries = ntohs(dns_h.qdcount);
31
32  if (!questionEntries)
33      return FALSE;
34  indx = (char *) (pkt + rr_offset);
35  count = (char *) indx;
36
37  while (count){
38      (char *) indx++;
39      strncat(nameStr, (char *) indx, count);
40      indx += count;
41      count = (char *) indx;
42      strncat(nameStr, ".", sizeof(nameStr) - strlen(nameStr));
43  }
44  nameStr[strlen(nameStr)-1] = '\0';
45
46  min_str_len = (strlen(nameStr) < strlen(matchPTR)) ? strlen(nameStr) :
strlen(matchPTR);
47  if (strncmp(nameStr, matchPTR, min_str_len) == 0){
48      return TRUE;
49  }
50
51  return FALSE;
52 }

```

Figure 32: The original source code of AntiSniff.

6 Discussions

6.1 False Positive

Our algorithm has false positive when the unfiltered unsafe value is not used. For example:

```
1 void testme(int i){
2   unsigned int j;
3   i = i;
4 }
```

Figure 33: An example of false positive

The signedness conversion in line3 in Figure 33 makes value -1 of integer i unsafe. But the converted value in j is used nowhere. In this kind of situation, the signedness conversion does no harm.

6.2 False Negative

Our method has false negative when the unsafe value is used before it is separated from safe value. For example:

```
1 void testme(int i){
2   unsigned int j;
3   void * p;
4   j = i;
5   p = malloc(j);
6   if(i < 0)
7     return;
8   *p = 1;
9 }
```

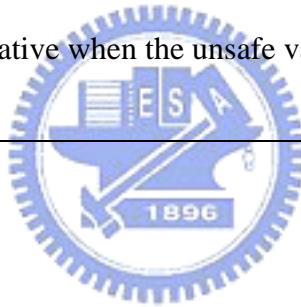


Figure 34: An example of false negative

The signedness conversion in line3 in Figure 34 makes value -1 of integer i unsafe. It is latter used in function malloc in line4 and generates a null pointer p. But unsafe value like -1 will be filtered out in the if statement in line6. Therefore, our method report no bug but the null pointer p will cause null pointer dereference in line8.

7 Related Works

RICH[12] has the same motivation as us. This work covers more integer problems than us, including integer overflow, underflow and truncation. The authors of the paper conduct a detailed survey about integer security and try to detect them at run time. Every integer operation is instrumented with a piece of detection code. The detection is instrumented with either CIL[13] or a GCC extension. RICH formally model a dangerous integer related operation in C program and detects its happening in runtime. RICH does not check parameters of memory/string related functions. RICH has high false positive and false negative rate but good performance.

Because of integer problems are easy to exploit with memory allocation. Catchconv[14] uses the Valgrind framework[15] to collect symbolic constraints, then intercept function malloc() and related functions to check whether the high bit of the parameter about size is set. In addition, it checks for program crash. We can find more kinds of bugs than memory errors. Moreover, the constraints they collected is too much to be solved by current solver. This greatly limits its usability.

The Big Loop Integer Protection (BLIP)[16] compiler extension transforms programs to detect overly large counters in loops. BLIP does not detect whether a dangerous integer operation happens. BLIP does not check the parameters of memory/string related functions, neither. BLIP uses a fixed threshold of loop index, resulting in many false negatives and false positives.

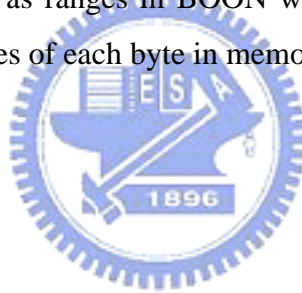
The constraint solver, CVC3, plays an important role in our testing framework because it does refilter checking and generates inputs for full path coverage. CVC3 is one of the SMT solvers. EXE and Catchconv use STP as their solver. There is an annual SMT-LIB competition which uses the benchmark[17]. Yices[18] and UCLID[19] performs well in this competition. We are currently conducting a survey about these solvers and their capabilities in use with symbolic execution.

Synergy[5] takes a formal approach to property checking by either proving the property as valid or disproving the property as invalid. , SLAM[20], BLAST[4]

Autodafe[21] use the well-form input and randomly change some of the input data. In this approach, it is supposed to explore more paths than pure random testing.

Active property checking that implemented on SAGE system[22] that is aim at bug finding. SAGE system processes a program in machine instruction level, therefore, SAGE add checkers that checks for low level program properties. Active property checking uses universal checkers in finding specific bug while Refilter algorithm add checker as a helper that point out the most common vulnerabilities that a bad programming style can lead to.

BOON[23] is a static analysis tool which aim at finding buffer overflow bugs. BOON uses pure static approach and is control flow insensitive. BOON has higher false positive rate because of lacking of control flow information. ALERT is able to take advantage of control flow information thus reduce the false positive rate. Possible value of a variable is modeled as ranges in BOON while ALERT uses constraint strictly to describe all possible values of each byte in memory.



8 Conclusions

We propose a new algorithm to check for signedness faults in C programs. Our approach is based on concolic testing. We use concolic testing to expand computation tree of a C program. For each execution path of this tree, we check if safe value and unsafe value of a signedness conversion can both trigger this path. If they do, we consider this as a signedness fault and uses universal checker to check parameters of memory/string related functions. We call this algorithm “refilter algorithm.” We have implemented it as a module of ALERT software testing platform. We use a qemu , AntiSniff and other manually crafted integer conversion testcases to evaluate the algorithm. The result shows we can find all integer signedness faults in a C program. Moreover, we show that our approach is more efficient than original concolic testing technique.



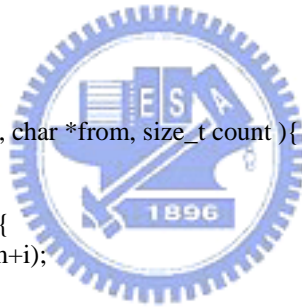
9 References

- [1] J. Koziol, D. Litchfield and D. A. and, *The Shellcoder's Handbook : Discovering and Exploiting Security Holes*. John Wiley Sons, 2004.
- [2] SecurityTracker, "Microsoft Internet Explorer Integer Overflow in Processing Bitmap Files Lets Remote Users Execute Arbitrary Code," feb. 2004.
- [3] SecurityTracker, "PHP emalloc() Integer Overflow May Let Remote Users Execute Arbitrary Code," apr. 2004.
- [4] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala and R. Majumdar, "The blast query language for software verification," in *SAS*, 2004, pp. 2-18.
- [5] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori and S. 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*, 2006, pp. 117-127.
- [6] P. Godefroid, N. Klarlund and K. Sen, "DART: Directed automated random testing," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 213-223.
- [7] J. C. King, "Symbolic execution and program testing," *Commun ACM*, vol. 19, pp. 385-394, 1976.
- [8] K. Sen, D. Marinov and G. 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*, 2005, pp. 263-272.
- [9] lp_solve. web page: http://tech.groups.yahoo.com/group/lp_solve/.
- [10] C. Barrett and S. Berezin, "CVC lite: A new implementation of the cooperating validity checker," in *Proceedings of the 16Th International Conference on Computer Aided Verification (CAV '04); Lecture Notes in Computer Science*, 2004, pp. 515-518.
- [11] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Computer Aided Verification (CAV '07)*, 2007,
- [12] D. Brumley, T. Chiueh and R. J. and, "Efficient and accurate detection of integer-based attacks," in *Proceedings of the Annual Network and*, 2007,
- [13] G. C. Necula, S. McPeak, S. P. Rahul and W. 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*, 2002, pp. 213-228.
- [14] D. Molnar and D. Wagner, "Catchconv: Symbolic execution and run-time type inference for integer conversion errors," feb 2007.

- [15] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007, pp. 89-100.
- [16] O. Horovitz, "Big loop integer protection," *Phrack*, dec. 2002.
- [17] S. Ranise and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," 2006.
- [18] B. Dutertre and L. d. Moura, "A fast linear-arithmetic solver for DPLL(T)," in *Proceedings of the 18th Computer-Aided Verification Conference; LNCS*, 2006, pp. 81-94.
- [19] R. E. Bryant, S. K. Lahiri and S. A. Seshia, "Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions," in *Computer Aided Verification; Lecture Notes in Computer Science*, 2002, pp. 78-92.
- [20] T. Ball and S. K. Rajamani, "Automatically validating temporal safety properties of interfaces," in *SPIN '01: Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, 2001, pp. 103-122.
- [21] M. Vuagnoux, "AUTODAFE an Act of Software Torture," 2005.
- [22] P. Godefroid, M. Levin and D. Molnar, "Automated Whitebox Fuzz Testing," 2007.
- [23] D. Wagner, J. S. Foster, E. A. Brewer and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities," in *NDSS*, 2000,
- [24] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill and D. R. Engler, "EXE: Automatically generating inputs of death." in *ACM Conference on Computer and Communications Security*, 2006, pp. 322-335

Appendix A: Source Code of Modified AntiSniff

```
1 void * mymemcpy( void *to, void *from, size_t count ){
2     size_t i;
3     char * myFrom = (char *)from;
4     char * myTo = (char *)to;
5
6     for(i=0;i<count;i++)
7         *(myTo+i) = *(myFrom+i);
8     return to;
9 }
10
11 void* mymemset( void* buffer, int ch, size_t count ){
12     size_t i;
13     char * mybuffer;
14     mybuffer = (char *)buffer;
15     for(i=0;i<count;i++)
16         *(mybuffer+i) = (char)ch;
17     return buffer;
18 }
19 size_t mystrlen( char *str ){
20     size_t count;
21     count = 0;
22     while(*(str+count) != '\0')
23         count++;
24     return count;
25 }
26
27 char *mystrncpy( char *to, char *from, size_t count ){
28     size_t i;
29     size_t n;
30     for(i=0;i<count;i++){
31         *(to+i) = *(from+i);
32     }
33     n = mystrlen(from);
34     if(count < n)
35         for(i=count;i<n;i++)
36             *(to+i) = '\0';
37     return to;
38 }
39
40 char *mystrncat( char *str1, char *str2, size_t count ){
41     size_t i;
42     size_t n;
43     n = mystrlen(str1);
44     for(i=0;i<count;i++){
45         *(str1+n+i) = *(str2+i);
46         if( *(str2+i) == '\0' )
47             break;
48     }
49     return str1;
50 }
51 char * mystrchr( char *s, int ch ){
52     unsigned char *char_ptr;
53     unsigned char c;
54     c = (unsigned char)ch;
55     for(char_ptr= (unsigned char *) s;; ++char_ptr){
56         if(*char_ptr == c)
57             return (void *)char_ptr;
```



```

58
59     if(*char_ptr == '\0')
60         return NULL;
61     }
62     return NULL;
63 }
64
65 size_t
66 mystrspn (s, accept)
67     char *s;
68     char *accept;
69 {
70     char *p;
71     char *a;
72     size_t count = 0;
73
74     for (p = s; *p != '\0'; ++p)
75     {
76         for (a = accept; *a != '\0'; ++a)
77             if (*p == *a)
78                 break;
79             if (*a == '\0')
80                 return count;
81             else
82                 ++count;
83     }
84
85     return count;
86 }
87
88 char *
89 mystrprbrk (char * s, char * accept){
90     char * a;
91     while( *s != '\0')
92     {
93         *a = accept;
94         while (*a != '\0')
95             if (*a++ == *s)
96                 return (char *) s;
97         ++s;
98     }
99
100     return NULL;
101 }
102
103 static char *olds;
104
105 char *
106 mystrtok (s, delim)
107     char *s;
108     char *delim;
109 {
110     char *token;
111
112     if (s == NULL)
113         s = olds;
114
115     s += mystrspn (s, delim);
116     if (*s == '\0')
117     {

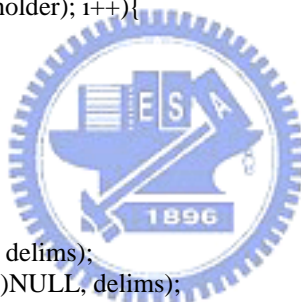
```



```

118     olds = s;
119     return NULL;
120 }
121
122 token = s;
123 s = mystrpbrk (token, delim);
124 if (s == NULL)
125     olds = mystrchr (token, '\0');
126 else
127     {
128         *s = '\0';
129         olds = s + 1;
130     }
131 return token;
132 }
133 int make_ptr_str(char *address, char *returnHolder){
134     char *ptr1, *ptr2, *ptr3, *ptr4;
135     char holder[MAX_LEN];
136     int dot_cnt=0, i;
137     char delims[2];
138     delims[0] = '.';
139     delims[1] = '\0';
140
141     mystrncpy(holder, address, MAX_LEN);
142
143     for (i=0 ; i < mystrlen(holder); i++){
144         if (holder[i] == '.')
145             dot_cnt++;
146     }
147
148     if (dot_cnt != 3)
149         return FALSE;
150
151     ptr1 = mystrtok(holder, delims);
152     ptr2 = mystrtok((char *)NULL, delims);
153     ptr3 = mystrtok((char *)NULL, delims);
154     ptr4 = mystrtok((char *)NULL, delims);
155
156     sprintf(returnHolder, MAX_LEN, "%s.%s.%s.%s.in-addr.arpa", ptr4,
157             ptr3, ptr2, ptr1);
158
159     return TRUE;
160 }
161
162 /* watch_dns_ptr examines DNS packets for Query types of PTR (has an IP
163     address and is attempting to look up a name. It returns true if the IP
164     address in the DNS packet matches the one handed to it.
165
166     A couple of caveat's... we only check one Query though you could be seeing
167     a variable number of queries in one packet. This is not seen too often
168     in the wild and hell... this is beta code. .mudge */
169
170 int testme(char *pkt, int len, char *ip_match){
171     HEADER dns_h;
172     int dns_offset, rr_offset, rr_size;
173     int count, questionEntries;
174     char *indx;
175     char nameStr[MAX_LEN];
176     char matchPTR[128];
177     int min_str_len;

```




```

178
179  memset(nameStr, '\0', sizeof(nameStr));
180  if (!make_ptr_str(ip_match, matchPTR)){
181      fprintf(stderr, "[DEBUG][TESTED] error making ptr lookup address\n");
182      return FALSE;
183  }
184
185  dns_offset = SIZE_ETHER_H + SIZE_IP_H + SIZE_UDP_H;
186  rr_offset = dns_offset + SIZE_DNS_H;
187
188  if (len < SIZE_ETHER_H + SIZE_IP_H + SIZE_UDP_H + SIZE_DNS_H)
189      return FALSE;
190
191  rr_size = len - rr_offset;
192
193  //memcpy(&dns_h, (char *) (pkt + dns_offset), sizeof(HEADER));
194  memcpy(&dns_h, (char *) (pkt), sizeof(HEADER));
195
196  questionEntries = ntohs(dns_h.qdcount);
197
198  // XXX temporarily removed code
199  /* if (!questionEntries)
200      return FALSE; */
201
202  rr_offset = 0;
203  indx = (char *) (pkt + rr_offset);
204  count = (char *) indx;
204  count = (char *) indx;
205  if(count == -1){
206      return FALSE;
207  }
208  if(count > 5){
209      return FALSE;
210  }
211  (char *)indx++;
212  indx += count;
213  count = (char *)indx;
214  if(count == -1){
215      return FALSE;
216  }
217  if(count > 5){
218      return FALSE;
219  }
220  (char *)indx++;
221  indx += count;
222  count = (char *)indx;
223  if(count != 0){
224      return FALSE;
225  }
226
227  rr_offset = 0;
228  indx = (char *) (pkt + rr_offset);
229  count = (char *) indx;
230  fprintf(stderr, "[DEBUG][TESTED] check point 2\n");
231
232  while (count){
233      fprintf(stderr, "[DEBUG][TESTED] count=%d
234      mystrlen(nameStr)=%d\n", count, mystrlen(nameStr));
234      fprintf(stderr, "[DEBUG][TESTED] count=%d\n", count);
235      printf("[DEBUG][TESTED] indx=%x\n", indx);

```



```

236     printf("[DEBUG][TESTED] strlen(nameStr)=%d\n",strlen(nameStr));
237     (char *)indx++;
238     mystrncat(nameStr, (char *)indx, count);
239     indx += count;
240     count = (char)*indx;
241
242     // if count = -1, this loop will run forever
243     if(count == -1){
244         printf("[DEBUG][TESTED] count == -1, fall into endless loop\n");
245         break;
246     }
247     //mystrncat(nameStr, ".", sizeof(nameStr) - strlen(nameStr));
248     mystrncat(nameStr, "...", sizeof(nameStr) - strlen(nameStr)); // [8]
249 }
250 printf("[DEBUG][TESTED] out of loop\n");
251 fprintf(stderr, "[DEBUG][TESTED] check point 3\n");
252
253 // XXX temporarily removed code
254 /* min_str_len = (strlen(nameStr) < strlen(matchPTR)) ? strlen(nameStr) : strlen(matchPTR);
255 if (strncmp(nameStr, matchPTR, min_str_len) == 0){
256     return TRUE;
257 }*/
258
259 return FALSE;
260 }
261

```

