Mutation-Based Testing of Format String Bugs

Conference Paper · December 2008

DOI: 10.1109/HASE.2008.8 · Source: DBLP

Hossain Shahriar
Kennesaw State University
317 PUBLICATIONS **3,155** CITATIONS

Mutation-based Testing of Format String Bugs

Hossain Shahriar and Mohammad Zulkernine
School of Computing
Queen's University, Kingston, Ontario, Canada
{shahriar, mzulker}@cs.queensu.ca

Abstract

Format String Bugs (FSBs) make an implementation vulnerable to numerous types of malicious attacks. Testing an implementation against FSBs can avoid consequences due to exploits of FSBs such as denial of services, corruption of application states, etc. Obtaining an adequate test data set is essential for testing of FSBs. An adequate test data set contains effective test cases that can reveal FSBs. Unfortunately, traditional techniques do not address the issue of adequate testing of an application for FSB. Moreover, the application of source code mutation has not been applied for testing FSB. In this work, we apply the idea of mutation-based testing technique to generate adequate test data set for testing FSBs. Our work addresses FSBs related to ANSI C libraries. We propose eight mutation operators to force the generation of adequate test data set. A prototype mutation-based testing tool named MUFORMAT is developed to generate mutants automatically and perform mutation analysis. The proposed operators are validated by using four open source programs having FSBs. The results indicate that the proposed operators are effective for testing FSBs.

1. Introduction

Format string bugs (FSBs) imply exploiting format functions (*e.g.*, format functions of ANSI C standard library) through malicious format strings [13]. The number of FSB reports has been increasing over the last few years based on several vulnerability statistics [16, 19, 20]. An application having FSBs might be exposed to several types of attack such as arbitrary reading, writing, and accessing parameters from stacks of format functions. If attack cases are crafted carefully, it is possible to perform malicious activities such as establishing root shell, overwriting global offset table (GOT) table that contains function addresses, etc. [13, 15]. Therefore, testing an application for FSBs is important.

Traditional complementary approaches for detecting and preventing FSBs include source code auditing, static analysis [2, 3, 12, 14], and runtime monitoring [1, 5, 6, 9, 18, 22]. Source code auditing is expensive and time consuming process. Static analysis suffers from source code annotation, recompiling, and numerous false positive warnings. The runtime monitoring approaches augment executable programs to prevent exploitation of FSBs. However, these approaches incur runtime overheads in terms of performance, memory, delay, etc. An effective testing of FSB helps fixing implementations early and decreasing losses incurred by the end users. Obtaining an adequate test data set is an important goal towards an effective testing approach. An adequate test data set implies a collection of test cases that can exploit FSBs. Unfortunately, the existing vulnerability testing approaches [25, 26, 27] do not address the issue of obtaining adequate test data sets for testing FSBs.

In this work, we apply mutation-based testing technique to force adequate testing of FSBs. Mutation is a fault-based testing technique [11], where an implementation is injected with faults to generate mutants. The rule of injecting fault is known as mutation operator. A mutant is said to be killed or distinguished, if at least one test case can produce different output between the mutant and the implementation. Otherwise, the mutant is *live*. If no test case can kill a mutant, then it is either equivalent to original implementation or new test case needs to be generated to kill the *live* mutant. Generating new test cases enhances the test data set. The adequacy of a test data set is measured by mutation score (*MS*), which is the ratio of the number of killed mutants to the total number of non-equivalent mutants. Similarly, we modify the source code to inject FSBs and force the generation of effective test cases that can exploit those bugs.

We apply mutation-based testing of FSBs for the format functions related to ANSI C standard libraries [4] mainly for three reasons. First, these format functions are the primary sources of FSBs [16, 19, 20]. Second, they are still widely used for developing many software such as ftp server (*e.g.*, wu-ftpd), web server (*e.g.*, apache). Third, the existing mutation operators for ANSI C [8, 10, 24] are not designed for testing FSBs of ANSI C format functions.

We propose eight mutation operators to support the testing of FSB along with two distinguishing criteria to kill the mutants. We implement a prototype tool that performs **mu**tation-based testing of **format** functions named MUFORMAT. The tool generates mutants automatically and performs mutation analysis. We demonstrate the effectiveness of the operators with four open source programs containing FSBs. The experiment results indicate that the operators are effective for generating adequate test data set for testing FSBs.

The paper is organized as follows: Section 2 describes the background of FSB and related works. In Section 3, the proposed operators along with the mutant distinguishing criteria are described. Section 4 describes the prototype tool and experimental evaluation of the operators. Section 5 draws the conclusion, and discusses limitations and future work.

2. Background and related work

This section is organized as follows. Section 2.1 describes the details about FSBs and attack types. Section 2.2 describes the related works.

2.1 Format string bug (FSB)

We consider two families of format functions provided by ANSI C library [4]: (i) *printf* family (also includes *fprintf*, *sprintf*, *snprintf*, *syslog*, etc.) and (ii) *vprintf* family (also includes *vfprintf*, *vsprintf*, *vsnprintf*, *vsyslog*, etc.). The *printf* family has the general format, "*int printf (const char *format, …)*". Here, … represent explicit input arguments matching with supplied % directives in *format*. The function returns the number of arguments written in console. The *vprintf* function has the format "*int vprintf (const char *format, va_list ap)*". Here, *ap* is the pointer of variable argument's list. The arguments are accessed by using standard macros provided by ANSI C.

The behavior of format functions depends on format strings. A format function prints all the character supplied in a format string except the % tag, which is known as format specifier tag. When it finds % tag, the next character represents the argument type (except %, which outputs the % character). The type can be string (*%s*), integer (*%d*), float (*%f*), etc. A format function call becomes vulnerable if the number of specifiers exceeds the number of arguments. Moreover, a type mismatch between a specifier and its corresponding argument might corrupt the state of a program or crash a program in the worst case.

For example, let us consider a format function call *printf ("hello %d %s", i, str)*. Here, the format string has two specifiers (*%d, %s*). The arguments *i* and *str* correspond to the two format specifiers, which are integer and string type variable, respectively. The stack organization for *printf* function is shown in Figure 1 (UNIX like operating systems). The return address of the function is saved followed by the address of format string and arguments. Two different pointers are used to keep track of format string (*Fsprt*) and supplied arguments (*Argptr*). The initial position of *Argptr* is immediately after the address of format string. Initially the first six bytes are written to console (*i.e.*, "*hello* "). The *%d* specifier retrieves the value of argument *i* and advances the *Argptr* by four bytes (assuming an integer variable occupies four bytes). The retrieved value is printed in console followed by the space character. The *%s* specifier retrieves the string located at the address *str* and advance the *Argptr* by four bytes again.

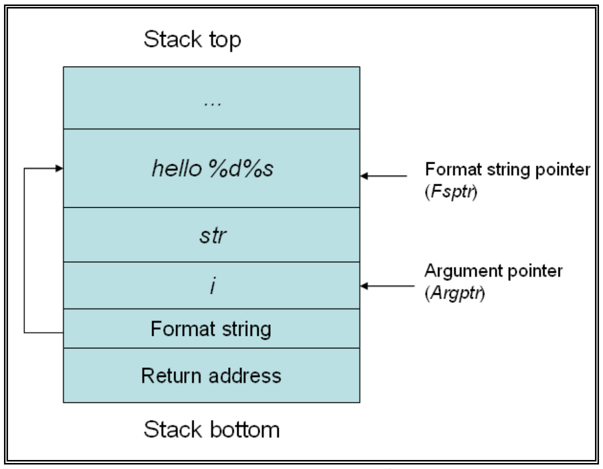


Figure 1: Stack of the *printf* function call

Let us assume that *i* and *str* are not supplied in the format function call. The function call becomes *printf ("hello %d %s").* As before, the *printf* function prints *"hello "* to the console. The *%d* specifier forces the *Argptr* to retrieve four bytes from its current location followed by increasing the *Argptr* by four bytes. Next, the *%s* specifier forces the function to fetch a null terminated string from the address pointed by the next four bytes of *Argptr*. The outcome of this fetch is unpredictable. In the worst case, the address might not belong to valid memory location, and the application crashes due to segmentation violation. The example represents exploitation of FSBs through reading from arbitrary addresses of the stack of format functions.

Teso [13], Silva [15], and Lhee [23] study attacks related to FSBs. We use three types of attack in this work, which are denial of service, arbitrary reading (from stack or by accessing the parameters), and writing in stack (using *%n* or width specifiers).

2.2 Related work

The work related to this paper are summarized in Table 1. The table includes a number of research works that involve format string bugs, and some mutation-based testing tools that motivated our choice of mutation-based testing. Table 2 further summarizes the comparative features covered by different tools for detecting FSBs with respect to our work. Note that none of the existing research work addresses the issue of adequate testing of FSB. The work listed in Table 1 and 2 are discussed in the following paragraphs.

Agrawal *et al.* [10] propose a comprehensive set of mutation operators for ANSI C language, which are applicable for program variables, constants, statements, and operators. However, their proposed operators are not intended for testing FSBs provided by ANSI C libraries. Delamaro *et al.* [8] propose mutation operators for testing integrated C programs. Their proposed operators inject faults (i) inside the called functions and (ii) at the point of function calls (or interfaces). Some operators of the second group are similar to our proposed operators. For example, the ArgDel operator deletes each argument of a function call. In contrast, we remove format string and arguments of format functions. Some format functions have file pointer (*e.g.*, *fprintf*), destination buffer or buffer size arguments (*e.g.*, *snprintf*), which are not removed by our proposed operators.

Csaw is a mutation-based testing tool for C developed by Ellims *et al.* [24]. The tool can distinguish mutants from an implementation based on CPU time usage differences, program crashes due to divide by zero, etc. Csaw implements seven types of operators that include mutating operators and variables, substituting constants (*i.e.*, substituting each text with all the text that are not keywords), mutating variable types (*e.g.*, replacing *unsigned int* with *int*), etc. These operators are not intended for testing FSBs.

ITS4 [2] tool looks for potentially known vulnerable format functions used in an implementation by parsing C source code into a stream of tokens. The resultant tokens are compared against a database of unsafe functions. Similarly, Flawfinder [3] generates a list of potential security flaws by simple text pattern matching of source code. Both of these approaches suffer from false positive warnings.

Table 1: Summary of FSB related work

Work / Tool	Brief summary	Mutation-based testing?	Adequate testing of FSB?
Agrawal *et al.* [10]	Tests of ANSI C program units.	Yes	No
Delamaro *et al.* [8]	Tests integrated ANSI C programs.	Yes	No
CSaw [24]	Tests of ANSI C program units for real time systems.	Yes	No
ITS4 [2]	Scans source code for known vulnerable format functions.	No	No
Flawfinder [3]	Warns about FSB, if format string arguments are not constant.	No	No
Shankar *et al.* [12]	Detects FSB, if format strings are generated from tainted sources.	No	No
Chen et al. [14]	Same as Shankar *et al.* [23] except they demonstrate extended tools support to remove FSB in large applications.	No	No
PScan [9]	Detects FSB, if format strings are not constant and the last argument.	No	No
FormatGuard [18]	Terminates format function calls, if the number of format specifiers does not match with the number of arguments.	No	No
Ringenburg *et al.* [22]	Monitors FSB related attacks due to writing outside valid address.	No	No
Lisbon [1]	Protects applications against FSBs by inserting a canary word at the end of argument list of format functions.	No	No
Libformat [5]	Detects FSB related attacks in, if format strings are in writable memory and contain *%n* specifiers.	No	No
Libsafe [6]	Prevents FSB related attacks, if *%n* specifiers overwrite the return address of format functions.	No	No
Nagano *et al.* [7]	Detects FSB related attacks by using an IDS-based approach.	No	No
Our work	**Generate adequate test data sets for FSBs.**	**Yes**	**Yes**

Table 2: Comparison of FSB detection tools

Tool/ Work	Both families covered?	Stack reading?	Stack writing?	Argument retrieving?	Specifier width?	Specifier mismatch?
ITS4 [2]	Yes	Yes	Yes	No	No	No
Flawfinder [3]	Yes	Yes	Yes	No	No	No
Shankar *et al.* [12]	Yes	Yes	Yes	No	No	No
Chen *et al.* [14]	Yes	Yes	Yes	No	No	No
PScan [9]	No	Yes	Yes	No	No	No
FormatGuard [18]	No	Yes	Yes	No	No	No
Ringenburg *et al.* [22]	Yes	No	Yes	No	No	No
Lisbon [1]	Yes	Yes	Yes	No	No	No
Libformat [5]	Yes	No	Yes	No	No	No
Libsafe [6]	Yes	No	Yes	No	No	No
Nagano *et al.* [7]	Yes	No	Yes	No	No	No
Our work	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**

Shankar *et al.* [12] propose type qualifier inference approach for detecting FSBs, which is similar to taint analysis method. The basic principle of taint analysis is that if an untainted data is derived with a tainted data, it is marked as tainted. A format string is marked as tainted, if it is generated from data coming from the environment. Their approach generates warnings, if tainted format strings are used in format function calls. Chen *et al.* [14] also apply type qualifier inference to remove FSBs from large scale applications with automatic tool support.

Dekok [9] develops the *PScan* tool to detect FSBs in *printf* family functions. The two principles of detecting FSBs are: (i) a format string is not constant and (ii) it is the last argument of a function call. However, the tool does not address format functions that use variable argument list (*e.g.*, *vprintf*). Cowan *et al.* [18] develop *FormatGuard* tool to prevent FSBs during the compilation and linking stages. The tool counts the number of arguments passed during compile time and matches this count with the number of specifier inside format string during runtime. If there is a mismatch, then a warning about FSB is logged and the format function call is aborted. However, the approach cannot prevent several FSBs such as mismatch between format specifiers and corresponding arguments, incorrect width of format specifiers. In contrast, our approach addresses all of the above issues.

Ringenburg *et al.* [22] combine static data flow analysis and generation of runtime white-list to prevent FSBs that can be exploited through malicious *%n* specifier. The white-list implies valid address ranges, where writing operations can be performed during format function calls. The static analysis tracks the caller functions (or wrapper functions) that invoke format functions. These caller functions are registered with their developed APIs to identify the valid address ranges. Any modification outside the valid address ranges during format function calls is detected during runtime. Although the idea is effective to prevent FSB related attacks through writing to arbitrary memory addresses, it does not address other types of attacks such as arbitrary reading from stack.

Li *et al.* [1] propose FSBs related attack preventions during runtime for Win32 binaries. Their developed tool called *Lisbon* converts FSB detection problem into input argument list bound checking problem of variadic functions (*i.e.*, functions that take variable number of arguments). The main idea is to place format function calls inside stub wrapper functions, so that argument lists are identified by stubs. Canary words are placed immediately after argument lists. A canary word should not be accessed during a format function call. After execution of a format function call, it is observed whether the canary word is read or modified.

Robbins develops *Libformat* [5] tool that prevents FSBs during runtime. The tool parses kills an application, if format strings are in writable memory locations and contain *%n* specifiers. However, it cannot prevent attacks related to reading arbitrary memory (*e.g.*, supplying more number of specifiers than the arguments). Tsai *et al.* [6] implement a shared library called *Libsafe* to prevent FSB attacks during runtime. The library intercepts format function calls and check if it can be safely executed. If a function call does not overwrite the return address with *%n* specifiers, then it is considered as safe for execution. Otherwise, a warning message is logged and the process is terminated. The approach is not effective for many types of attack that do not overwrite return addresses (*e.g.*, arbitrary reading from stack).

Nagano *et al.* [7] propose an IDS-based approach to detect attacks related to FSBs. They generate verifier for vulnerable data before their usage. The vulnerable data includes return addresses, function pointers, function arguments, and so on. A verifier contains different attributes such as verification data, verification length, etc. Verifiers are stored at both user memory and kernel area (to keep it free from possible attacks). If there is a mismatch between the attributes of verifier residing in user and kernel area, a signal is send to a user application for possible intrusion. Based on this approach, FSB attacks are detected, if an input overwrites the return address of a format function. However, many attacks do not modify return address (*e.g.*, accessing parameters).

3. Proposed mutation operators

We propose eight mutation operators for adequate testing of FSBs. Table 3 summarizes all the proposed operators, which are divided into two categories: format function call modifications and format string modifications. The first category is applicable for both families (*i.e.*, *printf* and *vprintf*), whereas the second category is applicable for *printf* family having static format string.

Exploitations of FSBs might lead to program crashes, which make the task of distinguishing mutants from their original programs with respect to functional end output difficult. Moreover, exploitations might change internal states of a program (*i.e.*, corruption of program state) without crashing. Therefore, we apply the weak mutation-based testing [17] rather than strong mutation-based testing [11] for killing the mutants. We define two mutant killing criteria as shown in Table 4.

Table 3: Proposed mutation operators

Category	Operator	Description of operator	Killing criteria
Format function call modifications	FSIFS	Insert format strings in format function calls.	C_1
	FSRFS	Remove format strings of format function calls.	C_1
	FSCAO	Change argument's order of format function calls.	C_1
	FSRAG	Remove arguments of format function calls.	$C_1 \| C_2$
Format string modifications	FSRSW	Modify the width of format specifiers.	$C_1 \| C_2$
	FSCFO	Change format specifiers order.	$C_1 \| C_2$
	FSRSN	Replace format specifiers with *%n*.	$C_1 \| C_2$
	FSPSN	Prepend format specifier types with *n$*.	$C_1 \| C_2$

Table 4: Mutant killing criteria

Name	Criteria
C_1	$ES_P \neq ES_M$
C_2	$W_P \neq W_M$
ES_P: Exit code of *P*. ES_M: Exit code of *M*. W_P: # of bytes written by a format function in *P*. W_M: # of bytes written by a format function in *M*.	

Let us assume that *P* is an original program and *M* is a mutant. ES_P and ES_M are the exit status (*i.e.*, the exit code) of *P* and *M*, respectively. The criterion C_1 differentiates a mutant from an original program, when any one of them crashes but not both. We take advantage of the fact that the exit status of a crashed program is different than that of a program having normal termination. Let us assume that W_P and W_M are the number of bytes written by a format function in *P* and *M*, respectively. The criterion C_2 differentiates *P* and *M*, if $W_P \neq W_M$. We discuss the proposed mutation operators from Section 3.1 to 3.8. We also show the relationship between the operators and FSB related attacks in Section 3.9.

3.1. Insert format string (FSIFS)

The FSIFS operator inserts format string in format function calls. It is applicable for both *printf* and *vprintf* family functions. The operator inserts a simple format string containing a string specifier (*i.e.*, *"%s"*). The operator is intended to test whether a format function call has absence of explicit format string that might result in FSBs [13, 15]. The generated mutants are killed with test cases that contain format specifiers and satisfy C_1 criterion.

Table 5: Example application of the FSIFS operator

Test case (src)	Original program (P)	Mutated program (M)	Output (P)	Output (M)	Status
'aaa'	printf (src);	printf ("%s",src); **//ΔFSIFS**	No crash	No crash	Live
'%s%s%s'	As above	As above	Crash	No crash	Killed

Table 5 shows an example application of the operator for two test cases. The first test case *'aaa'* cannot kill the mutant *M* as C_1 criterion is not satisfied. The second test case *'%s%s%s'* forces *P* to read from arbitrary stack location and crash. However, *M* does not crash as it solely prints the test case in console. Thus the mutant is killed and it forces to generate an attack test case that exploits FSBs.

3.2. Remove format string (FSRFS)

The FSRFS operator removes format strings of format functions (both format family functions) to allow FSBs in mutants. Since *vprintf* function must have two parameters (the format string and variable argument pointer), we replace the format string argument with the first argument of the variable list. The generated mutants are killed by test cases, if the first argument contains format specifiers and satisfy C_1 criterion. Table 6 shows an example application of the operator, where the format string *"%s"* is removed from *printf* function. The first row shows that the mutant is not killed by the test case '*aaa*', which is not an attack case. However, the mutant is killed by an attack test case *'%s%s%s'* in the second row as it satisfies the C_1 criterion. If the first argument is not a string variable, then the mutants might not be compiled and need to be removed from analysis.

Table 6: Example application of the FSRFS operator

Test case (src)	Original program (P)	Mutated program (M)	Output (P)	Output (M)	Status
'aaa'	printf ("%s", src);	printf (src); **//ΔFSRFS**	'aaa'	'aaa'	Live
'%s%s%s'	As above	As above	No crash	Crash	Killed

3.3. Change arguments order (FSCAO)

The FSCAO operator changes the argument order of format function calls to allow FSBs. It is applicable for both *printf* and *vprintf* family functions. Here, a format string is considered as an argument. The arguments are shifted to left direction. We do not consider the file pointer (*e.g.*, *fprintf*), destination string variable (*e.g.*, *sprintf*, *vsprintf*), and string length (*e.g.*, *snprintf*, *vsnprintf*) for shifting. For *printf* family, if there are *n* numbers of arguments, then the total number of mutants is *n-1*. However, for *vprintf* family functions, there is no explicit list of arguments. Therefore, only one mutant is generated. The format string and argument pointer are replaced with each other. Some of the generated mutants for *printf* family might not be compilable as the argument that occupy format string might not be string variable. For example, in Table 7, the application of the operator for *printf* function generates two mutants. Here, *src1* and *src2* are both string type variables.

Table 7: Example of FSCAO operator

Original program (P)	Mutated program (M)
printf (“%s %s”, src1, src2);	printf (src1, src2, “%s %s”); //**ΔFSCAO** printf (src2, “%s %s”, src1); //**ΔFSCAO**

Table 8: Example application of the FSCAO operator

Test case (src1, src2)	Original program (P)	Mutated program (M)	Output (P)	Output (M)	Status
‘aaa’, ‘bbb’	printf (“%s %s”, src1, src2);	printf (src1, “%s %s”, src2); //**ΔFSCAO**	No crash	No crash	Live
‘%s%s%s’,’ bbb’	As above	As above	No crash	Crash	Killed

The generated mutants are killed by test cases that represent attacks within the arguments and satisfy the C_1 killing criterion. Table 8 shows an example of mutation analysis for generating effective test cases. The program *P* contains a *printf* function call that prints two of the string variables *src1* and *src2*, respectively. The first row shows that *src1* and *src2* has the value ‘*aaa’* and *‘bbb’*, respectively. The C_1 criterion is not satisfied as neither *P* nor *M* crashes. The second row contains effective test case *‘%s%s%s’* and *‘bbb’*, which forces the C_1 criteria to be true.

3.4. Remove arguments of format functions (FSRAG)

The FSRAG operator removes arguments of format functions to allow FSBs that can be exploited by arbitrary reading or writing in the stack of format functions. It is intended to test implementations that generate dynamic format strings. This operator makes the number of format specifier higher than that of the arguments. Since *printf* family function calls include explicit argument lists, the number of mutants generated is the total number of arguments supplied. The removal is performed from the left direction. However, the *vprintf* family function calls do not have the explicit argument list. The arguments are specified through a pointer variable (*va_list*), which is supplied to macro function (*va_init*) before retrieving the arguments (*va_arg*). The operator generates only one mutant for *vprintf* function by removing the first argument (assuming at least one argument is present in *va_list*). Mutants are killed by test cases that satisfy any of the two killing criteria.

Table 9: Example application of the FSRAG operator

Test case (fmt, src1)	Original program (P)	Mutated program (M)	Output (P)	Output (M)	Mutant Status
‘%s’, ‘aaa’	printf (fmt, src1);	printf (fmt); //**ΔFSRAG**	3	More than 3 bytes	Killed

Table 9 shows an example application of the operator. Here, the program *P* has a *printf* format function call, which has the format string (*fmt*) “%s” and the argument (*src1*) having value ‘*aaa’*. Applying the FSRAG operator generates the mutant *M* that has the format string only. The mutant is killed based on the C_2 criterion as the number of bytes written to output console is different between *P* and *M*.

3.5. Modify the width of format specifier (FSRSW)

The FSRSW operator replaces each format specifier’s width with a maximum value (*e.g.*, maximum value of two byte integer, 32,767). This modification allows FSBs that can be exploited by overwriting stack inside format functions (*e.g.*, *sprintf*). Moreover, it helps generating very high value for writing to arbitrary memory locations in conjunction with *%n* specifier [13, 15]. It is applicable for *printf* family functions having static format strings.

Table 10: Example of the FSRSW operator

Original program (P)	Mutated program (M)
sprintf (dest, “%s %d”, var1, var2);	sprintf (dest, “%32767s %d”, var1, var2); //**ΔFSRSW** sprintf (dest, “%s %32767d”, var1, var2); //**ΔFSRSW**

Table 10 shows an example application of the operator on *sprintf* format function. Here, *dest* is the destination buffer of size 32 bytes, which is filled up with the variables *var1* (string type) and *var2* (integer type), respectively. The generated mutants overflow the *dest* buffer and modify other sensitive neighbor variables such as the return address. Both of the mutants can be killed by test cases that satisfy any of the two killing criteria.

3.6. Change format specifier’s order (FSCFO)

The FSCFO operator changes the order of format specifiers inside format strings. The operator injects mismatches between format specifiers and corresponding argument types. The wrong order of format specifier might allow arbitrary reading or writing in the stack of format functions. The operator is applicable for static format string and *printf* family functions. The specifiers are rotated in left direction. For dynamically generated format string, it cannot be applied. For *n* number of specifier, the operator generates *n-1* number of mutants. The generated mutants are killed by test cases that satisfy any of the two killing criteria.

Table 11: Example application of the FSCFO operator

Test case (var1, var2)	Original program (P)	Mutated program (M)	Output (P)	Output (M)	Mutant Status
'aaa', 25	printf ("%s %d", var1, var2);	printf ("%d %s", var1, var2); //**ΔFSCFO**	5 bytes	More than 5 bytes or crash	Live

Table 11 shows an example application of the operator, where the original program *P* contains *printf* format function call with a format string "*%s %d*". The corresponding arguments are *var1* and *var2*, which are string and integer type data, respectively. The FSCFO operator generates one mutant, where the format string is modified as "*%d %s*". As a result, the mutated program *M* forces to fetch integer data from *var1* and string from the address pointed by *var2*. This might either corrupt the state of *M* or make it crash.

3.7. Replace format specifiers with *%n* (FSRSN)

The FSRSN operator replaces each format specifier with *%n* to allow arbitrary writing in the stack of format functions. The *%n* specifier forces the modified program to write the total number of bytes outputted so far to the address of corresponding argument considered as integer pointer. In other word, this specifier is used to write outside the format string. The operator is applicable for *printf* family functions having static format strings. The generated mutants are killed by test cases that satisfy any of the two killing criteria.

Table 12: Example application of the FSRSN operator

Test case (var1)	Original program (P)	Mutated program (M)	Output (P)	Output (M)	Mutant Status
25	printf ("%d", var1);	printf ("%n", var1); //**ΔFSRSN**	No crash	Crash	Killed

Table 12 shows an example application of the operator. Here, the *printf* format function has one integer format specifier (*%d*), which is replaced by *%n*. In *M*, zero is written to the address pointed by *var1*, which is outside the valid address range.

3.8. Prepend format specifier types with *n$* (FSPSN)

The FSPSN operator modifies each of the format specifier types (*e.g.*, *d*) with *n$* (*e.g.*, *n$d*) to access the n^{th} parameter from the stack. The value *n* is set as the total number of supplied arguments plus one to allow a mutant program for viewing the content followed by the last explicit argument of format functions. The operator is applicable for *printf* family functions having static format strings. The mutant can be killed with test cases that satisfy any of the two killing criteria.

Table 13: Example application of the FSPSN operator

Test case (var1)	Original program (P)	Mutated program (M)	Output (P)	Output (M)	Mutant Status
'aaa'	printf ("%s", var1);	printf ("%2$s", var1); //**ΔFSPSN**	3	More than 3 bytes	Killed

Table 13 shows an example application of the operator. Here, the *printf* format function is supplied with a format specifer *%s* and corresponding argument *var1* in *P*. The FSPSN operator prepends the string specifier with *%2$s*, which retrieves the next argument of *var1* in the stack. Here, the mutant is killed by a test case *'aaa'* that satisfy C_2 criterion.

3.9. Relationship between FSB attacks and the operators

Table 14 shows the relationship between the attacks that exploit FSBs and the proposed operators. All the operators can test denial of service attacks. The FSIFS, FSRFS, FSCAO, FSRAG, FSRSW, FSCFO, FSPSN operators test reading from arbitrary memory locations of the stack of format functions. The FSRSW and FSRSN operator tests arbitrary writing to memory locations of stack of format functions.

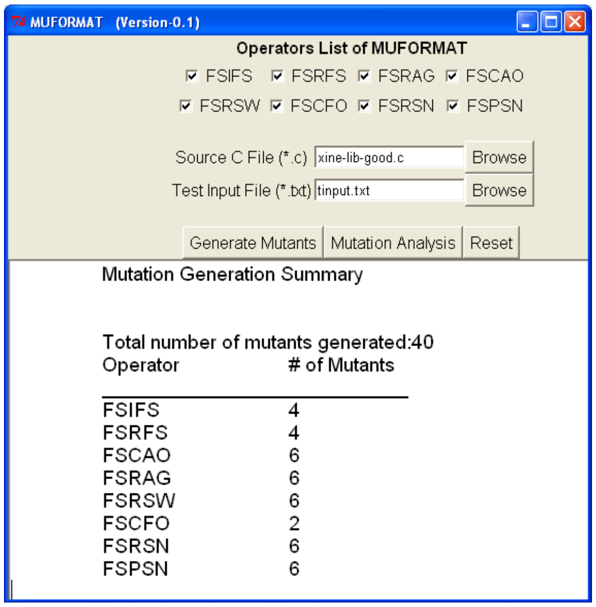
Table 14: Proposed operators and FSB attacks

Attack	Proposed Operators
Program crash (Denial of Service)	All
Reading from arbitrary memory address of stack	FSIFS, FSRFS, FSCAO, FSRAG, FSRSW, FSCFO
Direct Parameter Access	FSPSN
Writing to arbitrary memory address of stack.	FSRSW, FSRSN

4. Implementation and evaluation

4.1. Prototype tool implementation

We implement a prototype tool for performing **mu**tation-based testing of **format** functions named **MUFORMAT**. The tool is developed using the Tool Command Language (TCL 8.1) script that can invoke executable C programs. The TCL script can be launched with the *wish* program of *Cygwin* (a Linux-like emulator that runs in Windows XP). Figure 2 shows an example snapshot of the tool. The tool automatically generates mutants for the format string functions of an implementation (*e.g.*, xine-lib-good.c) by clicking on the "Generate Mutants" button. It scans the code and replaces appropriate texts based on the operators.



Source C File (*.c) xine-lib-good.c

Test Input File (*.txt) tinput.txt

Mutation Generation Summary

Total number of mutants generated:40

Operator	# of Mutants
FSIFS	4
FSRFS	4
FSCAO	6
FSRAG	6
FSRSW	6
FSCFO	2
FSRSN	6
FSPSN	6

Figure 2: A snapshot of the MUFORMAT tool

Each of the mutants generated is named according to the pattern *mux.c*, where *x* is the serial number of the mutant. For example, *mu0.c* is the first mutant. An auxiliary text file (*muaux.txt*) saves the name of the mutant file, the line number where the mutation is performed. In the analysis stage (*i.e.*, by clicking on the "Mutation analysis" button), the input test data set's (supplied in the tinput.txt file in Figure 2) quality is assessed for testing FSB vulnerabilities. The end result of the analysis includes the mutation score (*MS*) and the *live* mutants. A tester can decide on whether the *live* mutants are actually equivalent to the original program and add new test cases to kill the *live* mutants if necessary. The "Reset" button can be used to start a new mutation analysis.

4.2. Evaluation of the proposed operators

The effectiveness of the proposed mutation operators is demonstrated by using four open source applications implemented in ANSI C. The applications have been reported to contain FSBs in Common Vulnerabilities and Exposures (CVE) [19] and Open Source Vulnerabilities Database (OSVBD) [20]. Table 15 shows different characteristics of the four applications. The Xine-lib, sSMTP, Qwik-smtpd, and Xine-ui have FSBs in the functions named *_cdda_save_cached_cddb_infos, die, main,* and *print_formatted*, respectively. We call the vulnerable functions as bad programs and their corresponding fixed versions as good programs. A fixed version is either a patch (Qwik-smtpd-0.3 and Xine-lib) or an upgraded version (Xine-ui-0.99.5 and sSMTP-2.50).

Table 15: Characteristics of the four programs

Name	Application type	Vulnerability ID	Source file, function name	LOC
Xine-Lib-1.0.1	Multimedia player library	CVE-2005-2967	input_cdda.c, _cdda_save_cached_cddb_infos	20
sSMTP-2.48	Simple mail transport agent	CVE-2004-0156	log.c, die	23
Qwik-smtpd-0.3	SMTP server	OSVDB- 34241	qwik-smtpd.c, main	336
Xine-ui-0.99.4	Multimedia player	CVE- 2006-1905	main.c, print_formatted	23

To evaluate the effectiveness of our proposed operators, we follow the method used by Delamaro *et al.* [8]. They also employ this method to evaluate the effectiveness of their proposed mutation operators for C programs. The evaluation approach consists of the following two stages. In the first stage, a good (*i.e.* non-vulnerable) and corresponding bad (*i.e.*, vulnerable) programs are obtained, and the proposed operators are applied to generate mutants for a bad program. An initial test data set is generated randomly and the adequacy of the test data set is determined (*i.e.*, whether *MS* of the initial test data set is 100%). Here, *MS* is the mutation score, which is the ratio of the total number of mutants killed to the total number of non-equivalent mutants. More test cases are added to make it adequate (*i.e.*, bringing *MS* close to 100%). To avoid any bias in the result, several adequate test data sets (*e.g.*, 15) are constructed having an *MS* close to 100%.

In the second stage, for each of the adequate test data sets previously constructed, it is checked whether at least one test case can generate different output between a bad program and its corresponding good program. In our case, rather than comparing output, we check to see that at least one test case distinguishes the bad program from the good program based on any of the killing criterion. An adequate test data set is said to be effective if it is able to reveal vulnerabilities in this way. The percentage of the total number of test data sets that are able to reveal vulnerabilities are computed along with their average test data set sizes. We discuss the construction of the initial test data set before providing the experimental results.

We generate 15 initial test data sets for each application. Each of the test data set consists of 10 test cases. Each of the test cases has the necessary arguments to supply as program input and contains a format string that is generated randomly by using the Fuzz tool [21]. The length of format strings is set between 1 and 512 bytes by using a normally distributed random number generator. Random strings generated by the Fuzz tool are refined by retaining (i) the characters (or format specifiers) supported by ANSI format functions specification [4] symbols such as '*s*', '*%*', '*d*', '*n*', and (ii) numeric characters.

Table 16: Mutation generation summary

Operator	Xine-lib	sSMTP	Qwik-smtpd	Xine-ui
FSIFS	5	3	14	8
FSRFS	4	2	12	3
FSCAO	5	2	11	6
FSRAG	6	3	20	6
FSRSW	5	2	21	6
FSCFO	1	0	7	0
FSRSN	5	2	7	6
FSPSN	5	2	7	6
Total	36	16	99	41

Table 17: Summary of mutation analysis

Name	**Avg. *MS* (%) of bad programs**	**Avg. test data set size**	**% of test data set that revel FSBs**
Xine-lib	100	16.5	100
sSMTP	100	16.7	100
Qwik-smtpd	100	59.6	100
Xine-ui	100	17.9	100

Table 16 shows the number of mutants generated for all the operators for each of the bad programs. We obtain an adequate test data set *T* that kills all the generated mutants for each bad program. The *T* set is generated by the following two steps: (i) generate an initial test data set containing 10 test cases (discussed in the previous paragraph), and (ii) if the *MS* of the test data set does not reach 100%, then analyze the *live* mutants to generate additional test cases to kill them. Here, *MS* is the mutation score, which is the ratio of the total number of mutants killed to the total number of non-equivalent mutants generated. Steps (i) and (ii) are repeated 15 times (*i.e.*, for all the test data sets) to reduce any bias in the analysis.

Table 17 shows the average *MS*, average test data set sizes, and percentage of test data sets that reveal FSB vulnerabilities. We kill the generated mutants and develop 15 adequate test data sets for each of the bad programs. The second and third columns of Table 17 show the average *MS* and test data set sizes for each of the bad programs. We notice that each of the adequate test data sets obtain 100% *MS*. We then investigate whether at least one test case of each of the adequate test data sets can distinguish between a bad program and its corresponding good program. We find that all the adequate test data sets for a bad program can distinguish its corresponding good program (the fourth column of Table 17) Therefore, our proposed operators help in obtaining adequate test data set capable of revealing FSB vulnerabilities.

5. Conclusions

This work proposes mutation-based testing of format string bugs (FSBs) that force the generation of adequate test data sets. The test adequacy problem has not been addressed in the area of testing FSBs as we have observed in the extensive survey of the related work provided in this paper. By applying the proposed approach, an implementation can be tested for FSBs. The discovered bugs can be fixed before the actual deployment, and the losses incurred by end users can be prevented. We propose eight mutation operators along with two killing criteria to support the mutation-based testing of FSBs. The operators inject FSBs in the source code and force the generation of test cases to detect FSBs that may lead to various types of exploitations such as program crash, arbitrary reading, writing, and direct parameter access. The operators test FSBs for both *printf* and *vprintf* format family functions. Moreover, several proposed operators test FSBs due to format specifer widths and mismatches, which have not been addressed in the traditional approaches for detecting FSBs. A prototype testing tool named MUFORMAT is implemented to automatically generate mutants and perform mutation analysis. The tool provides the list of *live* mutants, which helps the tester to generate additional test cases for making a given test data set adequate for testing FSBs. The operators are found effective for four open source applications having FSBs.

Our approach has several limitations. Both the equivalent-mutant detection (generated by FSCFO operator) and test case addition for killing live mutants are manual. Several mutation operators (*e.g.*, FSRFS, FSRAG) generate non-compilable programs, which need to be removed manually. Some of the proposed operators (*e.g.*, FSRFS, FSRAG) instrument the code as a way of achieving mutants for *vprintf* family functions as there are no library functions in ANSI C to retrieve the supplied arguments with types in the absence of format strings. We plan to address complex form of attacks exploiting FSBs such as writing to destructors section (dtors), global offset table (GOT) through mutation-based testing.

Acknowledgement
This work is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

6. References

[1] Li, W., Chiueh, T., "Automated Format String Attack Prevention for Win32/X86 Binaries", In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, Miami, December 2007, pp. 398-409.
[2] ITS4: Software Security Tool, Available: http://www.cigital.com/its4/
[3] FlawFinder, http://www.dwheeler.com/flawfinder/ (Accessed January 2008)
[4] C Standard Library, http://www.utas.edu.au/infosys/ info/documentation/C/CStdLib.html (Accessed January 2008)
[5] Robbins, T., Libformat, http://archives.neohapsis.com/ archives/linux/lsap/2000-q3/0444.html (Accessed January 2008)
[6] Tsai, T., and Singh, N., "Libsafe 2.0: Detection of format string vulnerability exploits", Technical report, Avaya Labs, February 2001.
[7] Nagano, F., Tatara, K., Sakuri, K., and Tabata, T., "An Intrusion Detection System using Alteration of Data", In *Proceedings of the 20th International Conference on Advanced Information Networking and Applications (AINA'06)*, Vienna, April 2006, pp. 243-248.
[8] Delamaro, M., Maldonado, J., and Mathur, A., "Integration testing using interface mutations", In *Proceedings of the Seventh International Symposium on Software Reliability Engineering*, IEEE Computer Society Press, White Plains, New York, pp 112-121, October, 1996.
[9] DeKok, A., "Pscan (1.2-8) Format string security checker for C files", http://packages.debian.org/etch/pscan (Accessed January 2008)
[10] Agrawal, H., DeMillo, R., Hataway, R., Hsu, W., Hsu, W., Krauser, E., Martin, R., Mathur, A., and Spafford, E., "Design of Mutant Operators for C Programming Language," Technical Report SERC-TR41-P, Software Engineering Research Center, Purdue University, April 2006.
[11] DeMillo, R. A., Lipton, R. J., and Sayward, F. G., "Hints on test data selection: Help for the practicing programmer", *IEEE Computer Magazine*, Vol.11, 1978, pp. 34-41.
[12]. Shankar, U., Talwar, K., Foster, J., and Wagner, D., "Detecting Format String Vulnerabilities with Type Qualifiers", In *Proceedings of 10th USENIX Security Symposium*, August 2001, Washington, D.C., pp. 201–218.
[13] Scut/team teso, "Exploiting Format String Vulnerabilities", 2001, Accessed from http://doc.bughunter.net/format-string/exploit-fs.html
[14]. Chen, K., Wagner, D., "Large-Scale Analysis of Format String Vulnerabilities in Debian Linux", In *Proceedings of the Workshop on Programming Languages and Analysis for Security (PLAS' 07)*, San Diego, June 2007, pp. 75-84.
[15] Silva, A., "Format Strings", Gotfault Security Community, Version 2.5, Nov 2005, Accessed from http://www.milw0rm.com/papers/5 (April 2008).
[16] Common Weakness Enumeration (CWE), Vulnerability Type Distributions in CVE, Version 1.1, May 2007, http://cwe.mitre.org/documents/vuln-trends/index.html (Accessed Jan 2008)
[17] Howden, W. E., "Weak mutation testing and completeness of test sets," *IEEE Transaction on Software Engineering*, Volume 8, Number 4, July 1982, pp. 371-379.
[18] Cowan, C., Barringer, M., Beattie, S., Hartman, G., Frantzen, M., and Lokier, J., "FormatGuard: Automatic Protection From printf Format String Vulnerabilities", In *Proceedings of the 10th USENIX Security Symposium*, August 2001, Washington, D.C., pp. 191- 200.
[19] Common Vulnerabilities and Exposures (CVE), http://cve.mitre.org/
[20] Open Source Vulnerability Database (OSVDB), http://osvdb.org/
[21] Miller, B.P., Fredriksen, L., and So, B., "An Empirical Study of the Reliability of UNIX Utilities", *Communications of the ACM,* Vol. 33, Issue 12, December 1990, pp. 32-44.
[22] M. Ringenburg and D. Grossman, "Preventing format-string attacks via automatic and efficient dynamic checking", In *Proceedings of the 12th ACM conference on Computer and communications security (CCS)*, November 2005, Alexandria, VA, USA. pp. 354-363.
[23] Lhee, K., and Chapin, S., "Buffer Overflow and Format String Overflow Vulnerabilities", *Journal of Software-Practice and Experience*, Volume 33, Issue 5, 2003, pp. 423-460.
[24] Ellims, M., Ince, D.C., and Petre, M., "The Csaw C Mutation Tool: Initial Results", In *Proceedings of 3rd Workshop on Mutation Analysis (Mutation 2007)*, Cumberland Lodge, Windsor, UK, September 2007, pp. 185 - 192.
[25] Tal, O., Knight, S., and Dean, T.R., "Syntax-based Vulnerability Testing of Frame-based Network Protocols", In *Proceedings of the 2nd Annual Conference on Privacy, Security and Trust*, Fredericton, Canada, October 2004, pp 155-160.
[26] Du, W. and Mathur, A., "Testing for software vulnerability using environment perturbation", In *Proceedings of International conference on Dependable Systems and Networks (DSN 2000)*, New York, June 2000, pp. 603-612.
[27] Ghosh, A. K., O'Connor, T., and McGraw, G., "An automated approach for identifying potential vulnerabilities in software", *IEEE Symposium on Security and Privacy*, Los Alamitos, CA, USA, May 1998, pp. 104-114.