MUTATION-BASED TESTING OF BUFFER OVERFLOWS, SQL INJECTIONS, AND FORMAT STRING BUGS

by

Hossain Shahriar

A thesis submitted to the School of Computing

in conformity with the requirements for

the degree of Masters of Science

Queen's University

Kingston, Ontario, Canada

August, 2008

Abstract

Testing is an indispensable mechanism for assuring software quality. One of the key issues in testing is to obtain a test data set that is able to effectively test an implementation. An adequate test data set consists of test cases that can expose faults in a software implementation. Mutation-based testing can be employed to obtain adequate test data sets, and numerous mutation operators have been proposed to date to measure the adequacy of test data sets that reveal functional faults. However, implementations that pass functionality tests are still vulnerable to malicious attacks. Despite the rigorous use of various existing testing techniques, many vulnerabilities are discovered after the deployment of software implementations, such as buffer overflows (BOF), SQL injections, and format string bugs (FSB). Successful exploitations of these vulnerabilities may result in severe consequences such as denial of services, application state corruptions, and information leakage. Many approaches have been proposed to detect these vulnerabilities. Unfortunately, very few approaches address the issue of testing implementations against vulnerabilities. Moreover, these approaches do not provide an indication whether a test data set is adequate for vulnerability testing or not.

We believe that bringing the idea of traditional functional test adequacy to vulnerability testing can help address the issue of test adequacy. In this thesis, we apply the idea of mutation-based adequate testing to perform vulnerability testing of buffer overflows, SQL injections, and format string bugs. We propose mutation operators to force the generation of adequate test data sets for these vulnerabilities. The operators mutate source code to inject the vulnerabilities in the library function calls and unsafe implementation language elements. The mutants generated by the

operators are killed by test cases that expose these vulnerabilities. We propose distinguishing or killing criteria for mutants that consider varying symptoms of exploitations. Three prototype tools are developed to automatically generate mutants and perform mutation analysis with input test cases and the effectiveness of the proposed operators is evaluated on several open source programs containing known vulnerabilities. The results indicate that the proposed operators are effective for testing the vulnerabilities, and the mutation-based vulnerability testing process ensures the quality of the applications against these vulnerabilities.

Acknowledgements

I would like to thank my supervisor, Dr. Mohammad Zulkernine, for all the guidance and support for completing my MSc dissertation. I would like to thank all the members of Queen's Reliable Software Technology group, who supported me through numerous thoughtful discussions. I would like to thank the members of thesis examination committee for their helpful comments and insights. I am grateful to my parents and family members for their love and support during my MSc degree program.

Statement of Originality

I hereby certify that all of the work described within this thesis is the original work of the author. Any published (or unpublished) ideas and/or techniques from the work of others are fully acknowledged in accordance with the standard referencing practices.

Hossain Shahriar
August, 2008

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Testing is an indispensable mechanism for assuring software quality. One important issue that arises during testing is whether or not an obtained test data set is effective in detecting faults [1, 2]. In the literature, this issue has been widely addressed through assessing test data quality or through adequate testing of an implementation. An implementation has been adequately tested if a test data set is obtained that reveals its faults. Here, a test data set is a collection of test cases. Mutation [3, 4, 5] is a fault-based testing technique that is intended to show that an implementation is free from specific faults [6]. Several recent studies [7, 8] suggest that mutation-based testing can reveal real faults introduced by experienced programmers during software implementation. Mutation-based testing has been employed to assess the quality of test data sets [9, 10, 11].

Despite rigorous use of existing testing techniques [1, 2], many vulnerabilities are discovered after the deployment of software implementations, such as buffer overflows, SQL injections, and format string bugs [12, 13, 14, 15]. Vulnerabilities imply "*specific flaws or oversights in a piece of software that allow attackers to do something malicious, expose or alter sensitive information, disrupt or destroy systems or take control of a computer system or program*" [16]. A number of surveys report that buffer overflow, SLQ injection, and format string bug are the most commonly occurring security flaws in software implementations [12, 13, 14, 15]. Successful exploitations of these vulnerabilities may result in severe consequences such as denial of services, application state corruptions, and information leakages. In 2004, the denial of service exploitation alone cost

more than $26 million in financial losses [17] to business organizations. Therefore, testing an implementation against these vulnerabilities is essential.

1.2 Motivation

Traditional complementary approaches to address vulnerabilities include source code auditing [18], static analysis [19, 20, 21, 22, 23, 24], runtime monitoring of implementations [25, 26, 27, 28, 29, 30, 31, 32, 33], a combination of static analysis and runtime monitoring [34, 35], and intrusion detection approaches [36, 37, 38]. Source code auditing is a time consuming and an expensive process. Static analysis tools detect potential vulnerabilities without running implementations. However, they suffer from the requirements for source code annotation and recompiling, as well as numerous false positive warnings. The runtime monitoring approaches augment executable programs to prevent the exploitation of vulnerabilities. However, these approaches incur runtime overheads in terms of performance and memory. Intrusion detection approaches are implemented after software deployment and thus incur additional overhead and cost.

An effective testing approach would detect vulnerabilities before software deployment, preventing the losses incurred by the end users. Obtaining an adequate test data set is an important goal towards effective vulnerability testing. Unfortunately, very few approaches [39, 40, 41, 42, 43] address the issue of testing implementations against vulnerabilities. These approaches do not mutate the source code of implementations to ensure their quality against vulnerabilities. Moreover, the approaches do not provide an indication of whether a test data set is adequate for vulnerability testing. We believe that bringing the idea of traditional mutation-based

test adequacy to vulnerability testing can help address this issue. However, existing mutation-based testing approaches [9, 10, 44, 45, 46, 47, 48] are not intended for testing vulnerabilities.

1.3 Contributions

In this thesis, we propose mutation-based testing of buffer overflows, SQL injections, and format string bugs. Our objective is to force the generation of an adequate test data set that can expose these vulnerabilities. However, the primary challenges of performing adequate testing of vulnerabilities are the lack of (i) mutation operators, which inject vulnerabilities in source code that might lead an application into vulnerable states; and (ii) mutant killing criteria that consider the widespread symptoms of vulnerability exploitations. We address these challenges by contributing the following:

1. We propose mutation operators and mutant killing criteria to support mutation-based testing of vulnerabilities. More precisely, we present the following:
    - *Twelve mutation[1] operators and two mutant killing criteria to perform adequate testing of buffer overflow vulnerabilities [49]*. The operators are based on common vulnerabilities related to the American National Standards Institute (ANSI) C programming language and its standard library functions.
    - *Nine mutation operators and seven mutant killing criteria to conduct adequate testing of SQL injection vulnerabilities [50]*. The operators consider the SQL language syntax and standard library functions for manipulating SQL statements that lead to security vulnerabilities.

[1] The number of operators does not *always* indicate better test coverage.

- *Seven mutation operators and two mutant killing criteria to perform adequate testing of format string bug vulnerabilities [51]*. The operators consider vulnerabilities present in ANSI C format functions, which are often exploited for security violations in real world applications.

2. We develop three prototype testing tools[2] to automatically generate mutants and perform mutation analysis for buffer overflows, SQL injections, and format string bugs. We evaluate the effectiveness of the proposed operators by using 13 open source applications.

There are three major implications of this research. First, a mutation-based vulnerability testing approach helps in engineering the quality of software implementations against vulnerabilities such as buffer overflows, SQL injections, and format string bugs. An implementation can be tested for vulnerabilities, the discovered vulnerabilities can be fixed before the actual deployment of the implementation, and the losses incurred by end users can be prevented. Second, the proposed approach forces a software testing team to generate test cases that can expose the buffer overflow, SQL injection, and format string bug vulnerabilities of an implementation. Finally, the developed tools automate the task of mutation analysis process and reduce the cost of mutation-based vulnerability testing.

[2] While we describe our approach based on three separate tools for the sake of clarity in the descriptions, the three tools can be combined into one single testing tool for buffer overflows, SQL injections, and format string bugs.

1.4 Organization

The rest of the thesis is organized as follows. In Chapter 2, we provide the background information on mutation-based testing and an overview of the three vulnerabilities (*i.e.* buffer overflow, SQL injection, and format string bug) that we address in this thesis. We also describe related works that address the detection and prevention of these vulnerabilities in comparison to our work. Chapter 3 provides the description of our proposed mutation operators and mutant killing criteria for testing buffer overflow vulnerabilities. Moreover, we describe the prototype tool implementation and evaluate the operators. Chapter 4 depicts the proposed mutation operators for testing SQL injection vulnerabilities along with mutant killing criteria. We also provide an overview of the prototype tool for performing mutation-based testing of SQL injection vulnerabilities and discuss the experimental evaluation of the operators. In Chapter 5, we discuss the proposed operators and mutant killing criteria for testing format string bug vulnerabilities, followed by the prototype tool implementation and evaluation of the operators. Finally, Chapter 6 draws the conclusions, limitations, and future work.

Chapter 2
Background and Related Work

This chapter provides an overview of background information and the work related to the thesis. Section 2.1 discusses mutation-based testing. Section 2.2 provides a detailed overview of the three vulnerabilities: buffer overflow (BOF), SQL injection, and format string bug (FSB). Section 2.3 discusses the related work that detect, monitor, and test these vulnerabilities.

2.1 Mutation-based testing

Mutation is a fault-based testing technique [3, 4, 5]. Fault-based testing aims at demonstrating the absence of prespecified faults in a program [6]. Therefore, performing mutation-based testing helps an implementation to be free from specific faults. The core of a mutation-based testing is a set of operators. Each of the operator modifies the source code to inject a fault. The modified program is known as a mutant. A mutant is said to be killed or distinguished relative to a test data set (or a set of test cases), if at least one test case generates different results 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 the original implementation or a new test case needs to be generated to kill the *live* mutant, a method of enhancing a test data set. The adequacy of a test data set is measured by a 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 of an implementation to inject vulnerabilities and force the generation of effective test cases that can expose vulnerabilities.

We show an example in the following about how mutation analysis helps generating an adequate test data set.

Let us consider the program (or implementation) *P* of Table 2.1 (left column) with an initial test data set *T* having one test case {(3, 2)}. Let the mutant *M* be generated by applying the relational operator replacement (ROR) operator [9] at Line 2 (*M* is shown in the right column of Table 2.1). The mutation operator replaces the '>' operator with '≥'. Applying the test case (3, 2) will generate the output 5 for both *P* and *M*. Thus, the mutant is *live*, and the mutation score *MS* is 0 (*i.e.*, 0/1). To kill the mutant *M*, the test data set *T* needs to be enhanced. From observation, it is evident that the test case (4, 4) can kill *M*, as it generates different output between *P* and *M*. Thus, this test case is added to *T* and the enhanced test data set becomes {(3, 2), (4, 4)}. The *MS* of *T* is 1.0. In this way, we have developed an adequate test data set.

Table 2.1: Mutant generation by applying ROR operator

Original program (P)	Mutant (M)

1. int foo (int a, int b){
2. if (a > b)
3. return (a – b);
4. else
5. return (a + b);
}
	1. int foo (int a, int b){
	2. if (a ≥ b) //**ΔROR**
	3. return (a – b);
	4. else
	5. return (a + b);
	}

Mutation-based testing is based on two assumptions: the competent programmer hypothesis (CPH) and the coupling effect (CE). The CPH assumption is based on the empirical experience that a programmer writes nearly correct code during implementation. As a result, only simple faults are injected in an implementation during mutation-based testing. For example, replacing one arithmetic operator with another (*e.g.*, replacing + with *), replacing one variable with another variable of similar type, etc. The CE assumption states that a test data set that can reveal

simple faults is sensitive enough to reveal complex faults. A complex fault consists of multiple simple faults injected in the same line. Therefore, first order mutants are good enough for performing adequate testing of an implementation. We apply first order mutants in this work (*i.e.* each of the mutants contain only one syntactic change or modification).

Depending on the output comparison criteria, there are three types of mutation-based testing: strong mutation [4, 5], weak mutation [52], and firm mutation [53]. In strong mutation-based testing, a mutant is killed or distinguished if the end output between an implementation and a mutant is different. However, Howden [52] proposes the idea of comparing an implementation and its mutant in terms of internal program states or components. This is known as the weak mutation-based testing approach. The comparison of program states between an implementation and a mutant is performed immediately after the components are executed. The components of a program can be variable references (*i.e.*, reading from a variable), variable assignments (*i.e.*, writing to a variable), arithmetic and relational expressions, and boolean values. Woodward *et al.* [53] propose firm mutation-based testing, which lies in between the strong and weak mutation-based testing. In firm mutation analysis, the output between an original program and a mutant can be compared at any point between the mutated line and the end of the program.

In this thesis, we apply the firm mutation-based testing technique for BOF vulnerabilities. However, for SQL injection and FSB vulnerabilities testing, we use the weak mutation-based testing technique.

2.2 Vulnerabilities

In this section, we provide an overview of the three major vulnerabilities namely buffer overflows, SQL injections, and format string bugs in Sections 2.2.1, 2.2.2, and 2.2.3, respectively.

2.2.1 Buffer overflow

A buffer overflow (BOF) consists of writing data to a buffer exceeding the allocated size, and overwriting the contents of the neighboring memory locations. The overwriting might corrupt sensitive neighbor variables of the buffer such as the return address of a function or the stack frame pointer [54]. BOF has many variations which depend on the location of buffer in process memory area [55]. For example, in Figure 2.1, the function *foo* has a buffer named *buf* that is located inside the stack region. The valid location of this buffer is between *buf[0]* and *buf[15]*. The variable *var1* is located immediately after the ending location of the buffer followed by the stack frame pointer (*sfp*) and the return address (*ret*) of the function *foo* as shown in Figure 2.2. The return address indicates the memory location where the next instruction is stored and is read immediately after the function is executed.

```
1.  void foo (int a) {
2.        int var1;
3.        char buf [16];
          …
    }
```

Figure 2.1: C code snippet of *foo* function

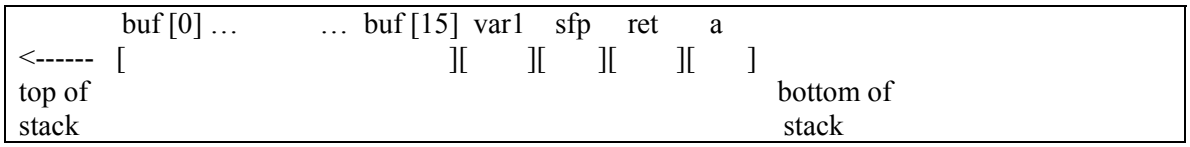

Figure 2.2: Stack layout of *foo* function

A BOF might happen during reading or writing operations. Writing past the *buf* by at least one byte corrupts the value of *var1*. If overwriting spans more than one byte in stack, it might modify the return address (*ret*) of the function *foo*. As a result, when the function tries to retrieve the next

instruction after its execution, the modified location might not fall within the valid address space of the process. A process is comprised of program text, stack, heap, symbol tables, etc. This might result a segmentation fault and the program crashing. Similarly, even reading past the buffer might result in an attempt to access memory locations that fall outside the valid memory range of the process. This also might result in a segmentation fault. Sophisticated techniques are available that exploit BOFs by smashing the stack [54] and the heap [56].

Table 2.2: List of ANSI C library functions vulnerable to buffer overflow

Function name	Brief description
char* strcpy (char* s, const char* ct)	Copies *ct* to *s* including terminating null character and returns *s*.
char* strncpy (char* s, const char* ct, size_t n)	Copies at most *n* characters of *ct* to *s* and returns *s*. Pads with null characters, if *ct* is of length less than *n*.
char* strcat (char* s, const char* ct)	Concatenates *ct* to *s* and return *s*.
char* strncat (char* s, const char* ct, size_t n)	Concatenates at most *n* characters of *ct* to *s* followed by adding null character and returns *s*.
void* memcpy (void* s, const void* ct, size_t n)	Copies *n* characters from *ct* to *s* and returns *s*.
void* memmove (void* s, const void* ct, size_t n)	Copies *n* characters from *ct* to *s* and returns *s*.
void* memset (void* s, int c, size_t n)	Replaces each of the first *n* characters of *s* by *c* and returns *s*.
int sprintf (char* s, const char* fmt, ...)	Writes the formatted (*fmt*) output to string *s*.
int vsprintf (char* s, const char* fmt, va_list arg)	Similar to sprintf, except it has variable argument list *arg*.
int fscanf (FILE* fp, const char* fmt, ...)	Performs formatted (*fmt*) input conversion by reading input from the stream *fp*.
int scanf (const char* fmt, ...)	It is equivalent to fscanf (stdin, fmt, ...).
int sscanf (char* s, const char* fmt, ...)	Similar to fscanf, except the inputs are read from *s*.
char* fgets (char* s, int n, FILE* fp)	Copies at most *n-1* characters from the file *fp* to *s*.
char* gets (char* s)	Similar to fgets, except the file stream is *stdin* and no bounds on copying input into *s*.

The causes of BOF vulnerabilities include logical errors in implementations (*e.g.*, off by one in mod_rewrite.c file of apache-1.3.3 [57]), the usage of library function calls (*e.g.*, ANSI C library functions [58] that do not check the destination buffer size before performing specific operations), absence of a null character at the end of a buffer, etc. Table 2.2 shows a list of ANSI C standard library functions vulnerable to BOF. In this work, we address BOF vulnerabilities due to ANSI C library function calls, lack of null character assignment statements, and incorrect size of buffer variables.

2.2.2 SQL injection

SQL (Structured Query Language) [59] is a standard query language used to manipulate relational databases. An application is said to have SQL injection vulnerabilities (SQLIVs), when SQL queries are generated using an implementation language (*e.g.*, Java Server Pages or JSP) and user supplied inputs become part of the query generation process without proper validation. These vulnerabilities can be exploited through SQL injection attacks (SQLIAs), which might cause unexpected results such as authentication bypassing and information leakage.

Relational databases are manipulated by a data definition language (DDL) and a data manipulation language (DML). The DDL is used to create different objects such as tables, stored procedures, functions, and views, while the DML is used to manipulate database objects. The four common DML statements are SELECT, INSERT, UPDATE, and DELETE, used to retrieve, insert, modify, and delete entities (or rows) from tables respectively. A simplified version of an SQL SELECT query statement is shown in Figure 2.3. The query selects all rows from *table_references* (the list of tables specified in the query). The *where_condition* determines the output generated from the query, specifying an expression that must be evaluated to true to select

each row. If there is no *where_condition*, then by default all the rows are selected from the table. The *where_condition* might include standard functions (*e.g.*, MAX, MIN, COUNT, etc.) and SQL operators. Table 2.3 shows some examples of SQL operators supported in MySQL database [60].

```
SELECT ALL [FROM table_references] [WHERE where_condition]
```

Figure 2.3: A simple SQL SELECT statement

Table 2.3: Example of basic SQL operators used in *where_condition*

SQL Operator	Description
=	Equal operator.
LIKE	Simple pattern matching for varchar (string) type data.
NOT	Negate a value or expression.
BETWEEN ... AND ...	Check whether a value is within a range.
AND, &&	Logical AND.
||, OR	Logical OR.
XOR	Logical XOR.

```
1. String LoginAction (HttpServletRequest request, ...) throws IOException {
2.    String sLogin = getParam (request, "Login");
3.    String sPassword = getParam (request, "Password");
4.    java.sql.ResultSet rs = null;
5.    String qry = "select member_id, member_level from members where ";
6.    qry = qry + "member_login = '" + sLogin + "' and member_password = '" + sPassword + "'";
7.    java.sql.ResultSet rs = stat.executeQuery (qry);
8.    if (rs.next ()) { // Login and password passed
9.          session.setAttribute ("UserID", rs.getString (1));
            …
      }
```

Figure 2.4: JSP code snippet for authentication

We provide an example of an SQLIA by using the code snippet of a server side application written in JSP as shown in Figure 2.4. Lines 2 and 3 extract user-supplied information from the *Login* and *Password* fields into the *sLogin* and *sPassword* variables, respectively. The user input is not filtered and a dynamic SQL query is generated in Lines 5 and 6. Let us assume that a user

provides valid *member_login* and *member_password*, which are "*guest*" and "*secret*", respectively. Then, the query generated at Line 6 appropriately becomes "*select member_id, member_level from members where member_login ='guest' and member_password = 'secret'*". The database engine executes the query at Line 7, and the user is authenticated with a valid *UserID* at Line 9.

A malicious user might supply the input "*' or 1=1 --* " in the first field and leave the second input field blank. The resultant query becomes "*select member_id, member_level from members where member_login ='' or 1=1 --' and member_password ='*'". The query is a tautology as the portion after the symbol "--" is ignored by the database engine ("--" is a comment symbol). The syntax of the altered query is correct. Therefore, the attacker avoids the authentication after the query by executing this query.

Table 2.4: SQL Injection attack examples based on the taxonomy of Orso *et al.* [61]

Attack	Example of attack strings
Tautology	i) ' or 1=1 -- ii) 'greg' LIKE '%gr%' --
Union	iii) ' UNION SELECT 1,1 --
Piggybacked queries	iv) '; show tables; -- v) '; shutdown; --
Inference	vi) '; /*! select concat ('1', '2') */ \g (mySQL) -- vii) '; /*! select '1' + '2' */ \g (MS SQL) --
Hex encoded query	viii) 0x206F7220313D31 Here, 0x206F7220313D31 is hexadecimal representation of the string "' or 1=1')"

Note that tautology is not the only way of performing an SQLIA. Orso *et al.* [61] classify SQLIAs into seven categories. They are tautologies, union queries, illegal/logical incorrect queries, piggybacked queries, stored procedures, inference attacks, and alternate encodings (or Hex encoded queries). This work addresses testing SQLIVs that can be exploited through

SQLIAs of the above types, excepting only stored procedures (more in Section 6.2 of Chapter 6). Table 2.4 shows some of the example attacks.

Tautology is the most common form of SQLIA, however, the form of tautology varies widely [62]. The first two examples of Table 2.4 show a tautology attack by adding either (i)*' or 1=1*, or (ii) '*greg' LIKE '%gr%'* at the end of a query. The first case exploits well known features of disjunctive boolean logic (*i.e.*, *true or X = true*). The second case takes advantage of the regular expression (*%gr%*) matching against a known supplied word ('*greg*'). The union attack adds the UNION keyword along with arbitrary supplied column values (example (iii)), which are always selected in the output result set after the execution of a query. Piggybacked queries are the most harmful type of attacks, where an attacker supplies additional SQL statements at the end of an intended query. These extra statements might reveal table information (example (iv)), shutdown the entire database server (example (v)), create new tables, delete existing tables, etc.

The objective of an inference attack is to gather information, such as server name, version, etc., about the backend database engine through the injection of known functionalities. For example, string concatenation is supported by the *concat* function in MySQL databases (example (vi)), whereas Microsoft SQL server supports the same function with the "+" operator (example (vii)). Maor *et al.* [63] show that SQLIAs can be performed by tracing error messages returned by database engines. Hex-encoded query attacks are preformed by providing attack strings in hexadecimal representation to escape input filters that are often placed on the client side of applications. Example (viii) shows the hexadecimal representation of the tautology attack string shown in example (i). In general, any attack string can be translated to its hexadecimal form.

2.2.3 Format string bug

Format string bug (FSB) vulnerabilities imply invoking format functions (*e.g.*, the format functions of ANSI C standard library) with user supplied format strings without input validation. An application having FSB vulnerabilities might be exposed to several types of attacks 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 access, overwriting global offset table (GOT) that contains function addresses [64, 65].

We consider two families of format functions provided by ANSI C libraries [58]: (i) the *printf* family (also includes *fprintf*, *sprintf*, *snprintf*, *syslog*, etc.) and (ii) the *vprintf* family (also includes *vfprintf*, *vsprintf*, *vsnprintf*, *vsyslog*, etc.). The *printf* family has the general format, "*int printf (const char *format, ...)*". Here, … represents explicit input arguments that should match with supplied format specifiers (*e.g.*, *%s*, *%d*) in *format*. The function returns the number of arguments written to console. The *vprintf* function has the format "*int vprintf (const char *format, va_list ap)*", where *ap* is the pointer of variable argument's list. The arguments are accessed by using standard macros provided by the ANSI C library such as *va_init*, *va_arg*, and *va_end*.

The behavior of a format function depends on the supplied format strings. A format function prints all the characters supplied in a format string except the *%* tag, which is known as the format specifier tag. When it finds a *%* tag, the next character represents the corresponding 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.

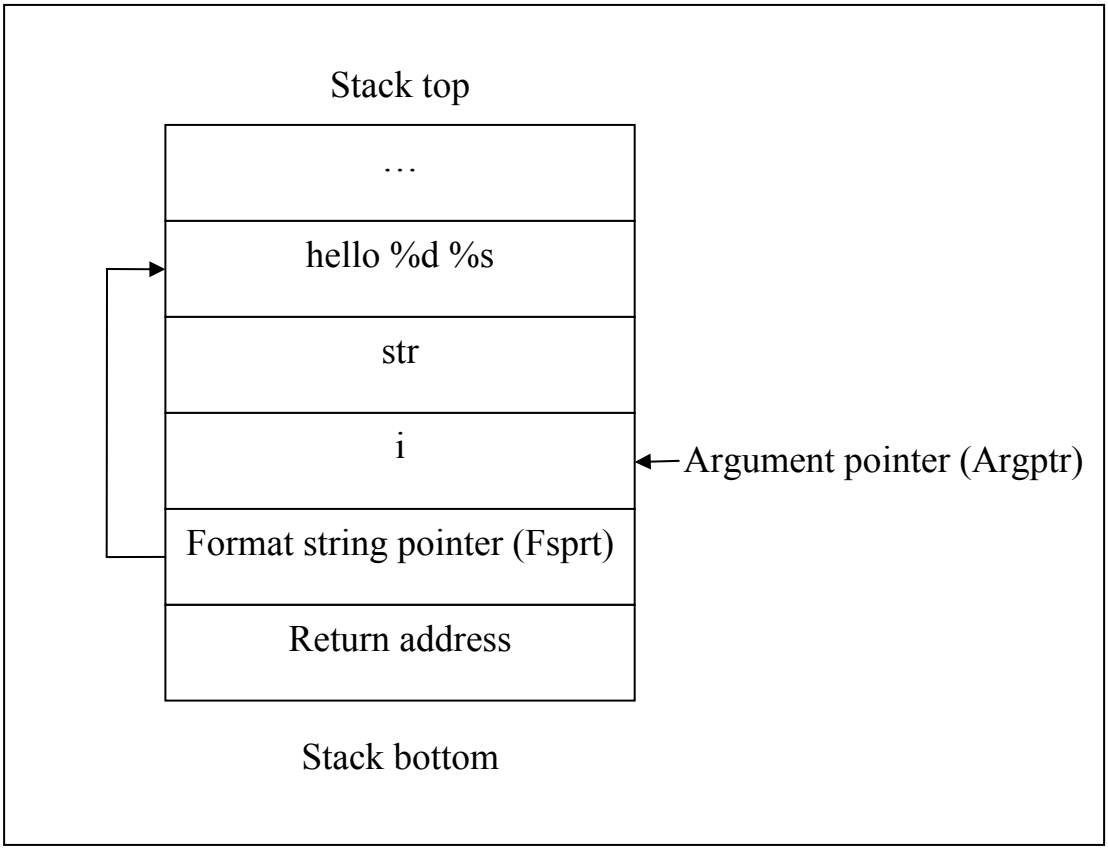

Figure 2.5: Stack of the *printf* function call

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 variables, respectively. The stack organization for *printf* function is shown in Figure 2.5. The return address of the function is saved first, followed by the address of the format string and arguments. Two different pointers are used to keep track of the format string (*Fsprt*) and the supplied arguments (*Argptr*). The initial position of *Argptr* is immediately after

the address of format string, and 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 (we assume that an integer variable occupies four bytes). The retrieved value is printed to 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.

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, reading the string from an invalid memory location might make the application crash. The example represents exploitation of FSB vulnerabilities through reading from arbitrary addresses of the stack of format functions.

Format functions allow writing to the location pointed at by supplied arguments through the *%n* specifier that can be exploited by supplying malicious format strings to an implementation. For example, the format function call *printf ("%n", var1)* results in writing zero (as nothing is written before the *%n* specifier appears in the format string) at the stack location pointed to by the *var1* variable. Format functions also allow retrieving the supplied arguments through the *%n$x* format specifier, where *n* and *x* imply the *n*-th argument and argument type, respectively. For example, the format function call *printf ("%2$d", 2, 4, 6)* results in printing the second arguments (*i.e.,* 4) in a console.

Teso [64], Silva [65], and Lhee [66] study several attacks related to FSB vulnerabilities. We address three types of FSB related attacks in this work, which are denial of service, arbitrary reading and writing in stack, and direct parameter access.

2.3 Related work

We describe the related work for buffer overflow, SQL injection, and format string bug vulnerabilities in Sections 2.3.1, 2.3.2, and 2.3.3, respectively.

2.3.1 Buffer overflow

Table 2.5 shows a summary of the related work in comparison to our work. These work are discussed in the following paragraphs. First, we discuss the mutation-based testing of BOF vulnerabilities. Later, we also discuss some mutation-based testing for C programs that motivated our work.

As can be seen from the table, mutation-based testing has not been utilized to test BOF vulnerabilities through the generation of an adequate test data set except in the work by Vilela *et al.* [67]. They propose two mutation operators which modify static memory allocations (MSMA) and dynamic memory allocations (MDMA). Each of the operators replaces the allocated buffer size *N* with *1*, *N-1*, *N/2,* and *N*2*, respectively (*i.e.*, each operator generates four mutants). Their approach does not consider BOF vulnerabilities due to the limitations of ANSI standard library functions (*e.g.*, *strcpy* function does not check the destination buffer before copying, which might result in BOF vulnerabilities) and other language specific features (*e.g.*, absence of the null character at the end of a buffer might lead to BOF vulnerabilities), which our work addresses. Moreover, each of our proposed operators generates only one mutant.

Table 2.5: Summary of BOF vulnerability related works

Tool name / Work	Brief description	Testing of BOF vulnerabilities?	Adequate testing of BOF vulnerabilities?
Vilela *et al.* [67]	Two mutation operators for testing BOF vulnerabilities.	Yes	Yes
Tal *et al.* [43]	Vulnerability testing of frame-based network protocol implementation by mutating protocol data units.	Yes	No
Allen *et al.* [68]	Finite state model-based vulnerability testing of network protocol implementations through fault injection in valid messages (requests or responses).	Yes	No
FIST [42]	Vulnerability testing of applications through fault injection in variables during runtime.	Yes	No
Du *et al.* [41]	Vulnerability testing of applications through fault injection of direct and indirect environment variables during runtime.	Yes	No
MotOrBAC [69]	Mutation-based testing of security policy specified in OrBAC language.	No	No
Agrawal *et al.* [9]	Mutation operators to test implementation units written in ANSI C language.	No	No
Delamaro *et al.* [10]	Mutation operators to test integrated programs written in ANSI C language.	No	No
Ellims *et al.* [44]	Propose mutation operators to test real-time embedded systems implemented in ANSI C language.	No	No
Our work	**12 mutation operators to test BOF vulnerabilities of ANSI C implementations.**	**Yes**	**Yes**

Tal *et al.* [43] propose the vulnerability testing of a frame-based network protocol implementation where the structure of a protocol data unit (PDU) is specified in a frame. Their approach captures PDUs from client machines, mutates data fields of these PDUs, then sends them back to the server and observes the server application's responses (*i.e.*, whether the protocol daemon running in the server crashes due to segmentation fault or not). Similarly, Allen *et al.* [68] perform vulnerability testing of network protocol implementation by using a fuzz tool and

block-based analysis of messages exchanged between clients and servers in an automated testing framework. The valid messages are separated into relevant blocks supported by protocol specifications and fuzzed to generate corrupted input to discover vulnerabilities in server applications. In contrast, our work addresses BOF vulnerability testing of C programs by mutating source code. We use the limitations of ANSI C language and its associated libraries to inject BOF vulnerabilities.

Ghosh *et al.* [42] mutate the internal states of a program to detect vulnerabilities during runtime, developing the Fault Injection Security Tool (FIST), which injects various types of fault such as corruptions of string variables and overwriting the return address of the stack. In contrast, we do not corrupt the return address directly to emulate BOF attack, but rather force the generation of test cases that expose BOF vulnerabilities.

Du *et al.* [41] perform vulnerability testing of applications by perturbing environment variables during runtime. They consider vulnerabilities from both the indirect environment (*e.g.*, programs using environment variables during initialization process), and the direct environment (*e.g.*, file system inputs, network packets, etc.). They propose fault and interaction coverage-based test adequacy criteria that assess test data sets quality for detecting vulnerabilities due to environment variables. In contrast, we address the issue of assessing test data sets that can reveal BOF vulnerabilities by proposing mutation operators for ANSI C implementations.

Baudry *et al.* [69] propose mutation operators that are designed to reveal implementation faults in security policy specified in the OrBAC language. They propose four categories of mutation operators, which inject faults in types (*e.g.*, permission to prohibition), in rule parameters (similar to mutating the arguments of a function), in hierarchies, and add new rules in policy. Although

testing security policy is important, it does not assure that an implementation is free from security vulnerabilities.

Agrawal *et al.* [9] propose a comprehensive set of mutation operators for the ANSI C language, applicable to program variables, constants, statements, and operators. Some mutation operators instrument source programs in order to achieve functional testing coverage [2] such as statement, branch, loop, and domain coverage. While their operators are not designed to inject BOF vulnerabilities in source code, we nevertheless find that some of their operators are equivalent to our proposed operators. For example, the SSDL operator removes each statement from the original implementation sequentially. Our mutation operator implements a subset of this behaviour, removing only those statements which assign null characters at the end of buffers as a way to inject BOF vulnerabilities. Similarly, the VTWD operator increases and decreases scalar variables, constant numbers, and arithmetic expressions by one, while our proposed operators increase buffer allocation sizes and length arguments by one unit (more details in Chapter 3).

Delamaro *et al.* propose mutation operators for testing integrated C programs (*i.e.*, interface testing) [10], which include two groups of operators that inject faults (i) inside the called functions and (ii) at the point of function calls (or interfaces). The first group is a subset of operators proposed by Agrawal *et al.* [9]. The operators of the second group mutate arguments of the called functions. Some operators of the second group are similar to our proposed operators. For example, we propose mutations of the buffer size arguments of ANSI C standard function calls (*e.g.*, modifying the buffer size argument of *strncpy* function to inject BOF vulnerabilities). However, the ways we mutate function arguments are significantly different from their work. For example, in *strncpy*, the buffer size argument is mutated by setting it to a specific value, whereas

their approach replaces it with all similar types of data variables declared either globally or locally in the same program.

Csaw is the most recent mutation-based testing tool for C, developed by Ellims *et al.* [44]. 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.*, replacing each text iteratively with each text that is not a keyword), increasing and decreasing decimal constants by *+1* and *-1*, respectively, replacing array indexes (*e.g.*, *a[i]* with *a[i+1]* and *a[i-1]*), removing statements and mutating variable types (*e.g.*, replacing *unsigned int* with *int*). However, these operators are not designed to test BOF vulnerabilities. Nevertheless, the constant substitution operator is similar to our approach of replacing safe library function calls (*e.g.*, *strncpy*) with unsafe function calls (*e.g.*, *strcpy*). Our approach is significantly different as we do not simply replace one string with other strings residing in the program.

2.3.2 SQL Injection

Table 2.6 shows a brief summary of prominent related work on the detection and testing of SQL injection attacks (SQLIAs) as well as SQL injection vulnerabilities [22, 27, 34, 35, 36, 37, 39, 40, 70, 71, 72, 73]. These work are discussed in this section. We are also motivated by some of the research [47, 48] that propose mutation-based testing of applications having SQL queries, which we discuss first.

Chan *et al.* [47] apply fault-based testing of database applications. They propose seven mutation operators, which inject faults in the entity relationship model (they call it the conceptual data model) of database-driven applications. The operators modify the cardinality of queries (*e.g.*,

replace *"select count(column$_1$)"* with *"select count(column$_2$)"*), replace attributes with similar types (*e.g.*, change one column name with the others having similar types of data), replace participation constraints (*e.g.*, replace EXIST with NOT EXIST), etc. Their approach is strong mutation-based testing. In contrast, we apply weak mutation-based testing of SQLIVs by injecting faults in both the SQL queries and database API function calls of an application.

Table 2.6: Summary of works related to SQL Injection vulnerabilities and SQLIAs

Works	Brief summary	Mutation-based testing?	Testing of SQLIVs?
Chan *et al.* [47]	Testing of database applications by proposed mutation operators based on a conceptual data model.	Yes	No
SQLMutation [48]	Mutation testing of SQL SELECT type queries.	Yes	No
SQLUnitGen [39]	Unit testing of web applications against SQLIAs.	No	Yes
Sania [40]	Debugging and testing framework for SQLIAs using static analysis.	Yes	No
SQLGuard [27]	Runtime monitoring of SQLIAs.	No	No
SQLrand [70]	Randomizing SQL keyword to thwart SQLIAs.	No	No
SQLCHECK [22]	Parsing augmented SQL grammar to recognize syntactic deviation due to SQLIAs.	No	No
AMNESIA [34]	SQLIAs prevention by generating valid query model with static analysis and runtime monitoring.	No	No
Muthuprasanna *et al.* [35]	Generating SQL-FSM by static analysis and comparing dynamic queries with the model during runtime.	No	No
Thomas *et al.* [71]	Retrofitting SQL statements with *PreparedStatements* in Java.	No	No
CANDID [72]	Parse tree comparison of intended queries using benign input with the trees generated during runtime.	No	No
Lin *et al.* [73]	Deployment of application gateway to filter SQLIAs.	No	No
ACIR [36]	Intrusion detection and response against web-based attacks for component-based software.	No	No
SQL-IDS [37]	Specification-based intrusion detection system for SQLIAs.	No	No
Our work	**Mutation of SQL queries and database API method calls to generate adequate test data set for SQLIV.**	**Yes**	**Yes**

Tuya *et al.* [48] develop the SQLMutation tool that implements four categories of mutation operators for adequate testing of SQL SELECT queries. These include SQL clause operators (*e.g.*, replacing SELECT with SELECT DISTINCT), operator replacement mutation operators

(*e.g.*, AND is replaced by OR), NULL mutation operators (*e.g.*, replacing NULL with NOT NULL), and identifier replacement (*e.g.*, replacing one column name with other of similar types). In contrast, our proposed operators test SQLIVs and can be applied to SELECT, UPDATE, DELETE, and INSERT type queries. Their approach is based on simple comparison of the end output generated by original and mutated queries. However, we distinguish mutants based on different intermediate database states and result sets returned by queries.

Shin *et al.* [39] combines static analysis with unit testing to detect the effectiveness of SQL injection filters in applications through the SQLUnitGen tool. Static analysis is used to track user input to the point of query generation. Initial test cases generated by the Jcrasher tool are modified so that they contain SQLIAs. Kosuga *et al.* [40] propose an SQLIA testing framework named Sania for the application development and debugging phase. Their approach initially constructs parse trees of intended SQL queries written by developers. Terminal leafs of parse trees typically represent vulnerable spots, which are filled with possible attack strings. The difference between the initial parse tree and the modified parse tree generated from user supplied attack string results in warnings of SQLIAs. Neither of the above approaches inject SQLIVs into the source code like our approach, nor do they address the generation of an adequate test data set for testing for SQLIVs.

Buehrer *et al.* [27] develop the SQLGuard tool that detects SQLIAs during application runtime by comparing the parse tree of an intended SQL query before and after the inclusion of user supplied input. However, the approach is ineffective if the user supplied input does not appear at the leaf of the tree. Our approach does not rely on the generation of an SQL parse tree.

Boyd *et al.* [70] randomize SQL keywords (SQLrand tool) to thwart injection attacks that contain SQL keywords. The main idea is that the injected SQL keywords from attackers have

random numbers appended to them so that they are not interpreted as regular keywords. Their approach can be fooled by performing piggy backed query attacks, where additional statements do not use any SQL keywords, or by any other injection attack where keywords are not necessary; our approach works for attacks having no SQL keywords.

Su *et al.* [22] propose a grammar-based approach to detect and stop queries containing SQLIAs by implementing the SQLCHECK tool. They mark user-supplied portions of queries with a special symbol and augment the standard SQL grammar with a production rule, generating a parser based on the augmented grammar. The parser successfully parses the generated query at runtime if there are no SQLIAs in the generated queries after adding user inputs. In contrast, our proposed approach is based on modifying intended queries and API method calls to inject an SQLIV.

Halfond *et al.* [34] implement the AMNESIA (Analysis for Monitoring and Neutralizing SQL Injection Attack) tool to detect and prevent SQLIAs. The tool first identifies hotspots where SQL queries are issued to database engines. At each hotspot, a query model is developed by using a non-deterministic finite automaton (NDFA). The hotspot is instrumented with monitor code, which matches dynamically generated queries against query models. If a generated query is not consumed by the NDFA query model, then it is considered an attack. In contrast, we address the issue of generating an adequate test data set through mutation-based testing.

Muthuprasanna *et al.* [35] apply both static analysis and runtime monitoring techniques to combat SQLIAs. In the static analysis phase, they perform java string analysis on all the hotspots, where SQL queries are issued to database engines. At each hotspot, string analysis results in Non-Deterministic Finite Automata (NDFA). Each NDFA is converted to a SQL Finite State Machine (SQL-FSM) whose transitions are either SQL keywords or string variables. During runtime, SQL

queries having user inputs are validated in two ways using the SQL-FSMs: first, the start and end node of SQL-FSM and a dynamically generated query should be identical; second, the length of the SQL-FSM chain and the dynamic query is the same. If either of these two conditions are not satisfied, the user inputs is deemed to contain SQLIAs.

Thomas *et al.* [71] demonstrate a technique that automatically fixes Java applications that are vulnerable to SQLIAs. The basic idea is to convert a SQL statement into a *PreparedStatement* in Java that does not allow the semantic modification of queries during runtime. The solution only addresses Java specific implementations and might not be suitable for applications written in other languages that have no support for *PreparedStatements*. In contrast, our proposed operators are applicable for any implementation language that generates SQL queries and calls database APIs.

Sruthi *et al.* [72] prevent SQLIAs by mining intended query structures during runtime. Their approach, called CANDID, makes mirror queries similar to programmer written queries, which are completed with benign input cases. During runtime, the generated parse trees with benign inputs and actual inputs are compared to detect SQLIAs. In contrast, our proposed mutation operators are killed by test cases containing SQLIAs, whereas if test cases are benign, then mutants are not killed.

Lin *et al.* [73] propose an application level security gateway to prevent SQLIAs. The approach has three stages: hybrid analysis, meta-programs, and a redirection mechanism. The hybrid analysis is a combination of black box testing and source code analysis which first identifies all possible entry points for SQLIAs, that are then protected by employing meta-programs that can filter out SQLIAs. The redirection mechanism is used to avoid attack requests propagated to the

web-server. Although their approach is suitable for protecting an existing application, it is a reactive approach, and effective testing can reduce the cost of deploying such an approach.

Uddin *et al.* [36] propose an intrusion detection and response framework named aspect connector for intrusion response (ACIR) aimed at preventing web-based attacks such as SQL injections. The framework addresses the security concerns of component-based software systems. A component's service (or method) requires input validation to prevent attacks. The framework uses a configuration file to describe the name of services, which require input checking. The file also contains necessary methods (or actions) that are invoked if attacks occur through external inputs. The attacks are matched against known signatures or patterns. Although the mechanism is helpful for reducing cross-cutting code for input validation, an effective vulnerability testing strategy can play a complementary role to enhance the security of software.

Kemalis *et al.* [37] construct a specification-based SQL injection attack detection system (SQL-IDS) that uses information from software security specifications. They first define the intended SQL commands using Extended Backus Naur Form (EBNF) specification. After the application is implemented, it is embedded with an IDS architecture that monitors SQLIAs. Queries that arrive from the client side are intercepted and tokenized into SQL keywords, user supplied values, column names, etc. If a sequence of query tokens does not match with any EBNF specification, then an injection attack is assumed present in the intercepted query. Although their approach is proactive, it suffers from the limitation of correctly specifying all desired queries with the specification language in advance (*i.e.*, before the application is implemented).

2.3.3 Format String Bug

The related work for detecting and preventing format string bug (FSB) vulnerabilities are summarized in Table 2.7. The table includes some mutation-based testing tools that motivate our choice of mutation-based testing, and several other research that involve format string bugs. Table 2.8 further summarizes the comparative features covered by different tools with respect to our work. Note that none of the existing research addresses the issue of adequate testing of FSB vulnerabilities. All of these work are briefly discussed in the following paragraphs.

Table 2.7: Summary of FSB vulnerability related works

Works	Brief summary	Adequate testing of FSB?
Agrawal *et al.* [9]	Tests of ANSI C program units.	No
Delamaro *et al.* [10]	Tests integrated programs implemented in ANSI C.	No
CSaw [44]	Tests of ANSI C program units for real time systems.	No
ITS4 [19]	Scans source code for known vulnerable format functions.	No
Flawfinder [20]	Warns about FSB vulnerabilities, if format string arguments are not constant in format function calls.	No
Shankar *et al.* [23]	Detects FSB vulnerabilities, if format strings are generated from tainted sources using type qualifier inference.	No
Chen *et al.* [24]	Same as Shankar *et al.* [23] except they demonstrate extended tools support to remove FSB vulnerabilities in large applications.	No
PScan [31]	Detects FSB vulnerabilities, if format strings are not constant and become last argument of format function calls.	No
FormatGuard [32]	Terminates format function calls, if the number of format specifiers does not match with the number of arguments.	No
Ringenburg *et al.* [33]	Monitors attacks that exploit FSB vulnerabilities due to writing operation outside valid memory address ranges in format function calls.	No
Lisbon [28]	Protects applications against FSB related attacks by inserting a canary word at the end of argument list of format functions.	No
Libformat [29]	Detects FSB related attacks in implementations, if format strings are in writable memory and contain *%n* specifiers.	No
Libsafe [30]	Prevents FSB related attacks during runtime, if *%n* specifier overwrites the return address of format functions.	No
Nagano *et al.* [38]	Detects FSB related attacks during runtime by using an IDS-based approach.	No
Our work	**Generate adequate test data sets for testing FSB vulnerabilities.**	**Yes**

Agrawal *et al.* [9] propose a comprehensive set of mutation operators for the ANSI C language, which are applicable for program variables, constants, statements, and operators. However, their proposed operators are not intended for testing format functions provided by ANSI C libraries.

Delamaro *et al.* [10] propose mutation operators for testing interfaces of C programs which 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 strings 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. Moreover, the ArgStcAli and ArgStcDif operators replace function arguments with similar and dissimilar types, respectively. One of our proposed operators rotates arguments of format family functions to allow FSB vulnerabilities irrespective of argument types.

Table 2.8: Comparison of FSB vulnerability and FSB related attack detection tools

Tool/ Work	Both families covered?	Stack reading?	Stack writing?	Argument retrieving?	Specifier mismatch?
ITS4 [19]	Yes	Yes	Yes	No	No
Flawfinder [20]	Yes	Yes	Yes	No	No
Shankar *et al.* [23]	Yes	Yes	Yes	No	No
Chen *et al.* [24]	Yes	Yes	Yes	No	No
PScan [31]	No	Yes	Yes	No	No
FormatGuard [32]	No	Yes	Yes	No	No
Ringenburg *et al.* [33]	Yes	No	Yes	No	No
Lisbon [28]	Yes	Yes	Yes	No	No
Libformat [29]	Yes	No	Yes	No	No
Libsafe [30]	Yes	No	Yes	No	No
Nagano *et al.* [38]	Yes	No	Yes	No	No
Our work	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**

Csaw is a mutation-based testing tool for C developed by Ellims *et al.* [44]. The tool can distinguish mutants from an implementation based on CPU time usage differences, program crashes due to divide by zero etc; we consider any segmentation fault as a killing criterion for mutants. Moreover, the operators (already described in Section 2.3.1) implemented in the tool are not intended for testing FSB vulnerabilities.

The ITS4 [19] tool looks for 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 [20] generates a list of potential security flaws by simple text pattern-matching in the source code. This risk level is assigned based on the context of function calls (*e.g.*, based on the values of the parameters). Both of these approaches suffer from a high level of false positives amongst the warnings generated.

Shankar *et al.* [23] propose a type qualifier inference approach to detect FSB vulnerabilities, which is similar to the taint analysis method. The basic principle of taint analysis is that if untainted data is derived from tainted data, it is marked as tainted. Therefore, a format string is marked as tainted if it is generated from data coming from the environment. Their type-inference engine generates warnings if tainted format strings are used in format function calls. However, the approach requires annotating trustworthy function parameters as "*untainted*" and untrustworthy parameters as "*tainted*" initially. Recently, Chen *et al.* [24] also apply type qualifier inference (similar approach to Shankar *et al.*) to remove FSB vulnerabilities from large scale applications with automatic tool support. Although there is a demand for developing tools to scan FSB vulnerabilities in large applications, an effective testing method is still required. Mutation-based testing of FSB vulnerabilities is an approach towards reaching that goal.

Dekok [31] develops the *PScan* tool to detect FSB related attacks (*i.e.*, exploitation of FSB vulnerabilities) in the *printf* family of functions. The two principles of detecting FSB related attacks are finding (i) a format string that is not constant and (ii) that is the last argument of a function call. However, in practice many applications generate format strings during runtime [32] and still might not have FSB vulnerabilities. Moreover, the tool does not address format functions that use variable argument lists (*e.g.*, *vprintf*).

Cowan *et al.* [32] develop the *FormatGuard* tool to prevent FSB related attacks 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 specifiers inside the format string during runtime. If the number of format specifiers is greater than the number of arguments, then a warning about FSB vulnerability is logged and the format function call is aborted. However, the tool cannot prevent several FSB vulnerabilities such as mismatches between format specifiers and corresponding arguments. Our approach addresses all of the above issues.

Ringenburg *et al.* [33] combine static data flow analysis and generation of a runtime white-list to prevent FSB related attacks that can be exploited through malicious use of the *%n* specifier. The white-list encodes valid address ranges where writing operations can be performed during format function calls. The static analysis tracks the calling functions (or wrapper functions) that invoke format functions. These calling functions are registered with their developed APIs to identify the valid address ranges. Any modification outside these valid address ranges during format function calls is detected during runtime. Although the idea is effective to prevent FSB related attacks that involve writing to arbitrary memory addresses, it does not address other types of attacks such as arbitrary reading from the stack.

Li *et al.* [28] propose FSB related attack prevention during runtime for Win32 binaries. Their *Lisbon* tool converts the FSB detection problem into the 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, which should not be accessed during a format function call, are placed immediately after argument lists. During the execution of a format function call, it is observed whether the canary word is read or modified. Although this approach effectively prevents most of the attacks related to FSB, our mutation-based testing approach helps to make an application free from FSB vulnerabilities before deploying.

Robbins develops the *Libformat* [29] tool that prevents FSB related attacks during runtime. The tool parses and 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 specifiers than arguments).

Tsai *et al.* [30] implement a shared library called *Libsafe* to prevent FSB related attacks during runtime. The library intercepts format function calls and checks if they can be safely executed. If a function call does not overwrite the return address with *%n* specifiers, then it is considered to be safe for execution; otherwise, a warning message is logged and the process is terminated. The approach is obviously not effective for many types of attack related to FSB that do not overwrite return addresses (*e.g.*, arbitrary reading memory from stack without overwriting).

Nagano *et al.* [38] propose an IDS-based approach to detect FSB related attacks. They generate a verifier for a vulnerable data before its 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, an altered flag, and a control flag. Verifiers are stored in

both user memory and the kernel area (to keep it free from possible attacks). If there is a mismatch between attributes of a verifier residing in user area with respect to a verifier residing in kernel area, a signal is send to a user application about a possible intrusion. The approach can detect FSB related attacks, if inputs overwrite the return addresses of format functions. However, many attacks do not modify return addresses (*e.g.*, accessing parameters, stack reading, etc.).

2.4 Conclusion

This chapter provides basic information about mutation-based testing, including the underlying assumptions and the various types of mutation-based testing. We provide an example to show how mutation-based testing can help in obtaining adequate test data sets.

Brief introductions to the three major vulnerabilities namely BOF, SQL injection, and FSB are provided along with several example attacks that expose those vulnerabilities. We conduct an extensive survey of the related research that addresses these vulnerabilities and several mutation-based testing approaches that motivated us to apply mutation-based testing for security vulnerabilities. The survey clearly shows that very little research applies mutation-based testing for BOF vulnerabilities, and none of the existing work applies mutation-based approach for testing SQL injection or FSB vulnerabilities.

Chapter 3

Mutation-Based Testing of Buffer Overflow Vulnerabilities

In this chapter, we perform mutation-based testing of buffer overflow (BOF) vulnerabilities for the ANSI C language [74] and its standard libraries [58], motivated by three primary factors. First, ANSI C and its libraries are the primary sources of BOF vulnerabilities according to vulnerability databases [12, 13, 14]. Second, even though BOF vulnerabilities related to ANSI C and its libraries have been known for many years, this environment is still widely used for developing many critical software applications such as ftp servers (*e.g.*, wu-ftpd), web servers (*e.g.*, apache), etc. Third, the existing mutation operators for ANSI C [9, 10, 44] are not designed for testing BOF vulnerabilities in particular, so it provides a new domain for mutation-based testing.

The rest of the chapter is organized as follows: Section 3.1 presents the proposed operators along with mutant killing criteria. Section 3.2 discusses the relationship between attacks exploiting BOF and the operators. Section 3.3 describes the prototype tool implementation, and Section 3.4 discusses the evaluation of the operators. Section 3.5 concludes the chapter, summarizing our findings and results.

3.1 Proposed operators and mutant killing criteria

We propose 12 mutation operators divided into five categories: mutating ANSI C standard library function calls, modifying buffer size arguments in ANSI C standard library function calls,

mutating format strings, increasing buffer variable sizes, and removing null character assignment statements. The first three categories consider the inherent vulnerabilities of ANSI C standard library function calls, whereas the remaining two categories consider the limitations of the programming language itself. Table 3.1 summarizes the proposed operators and the corresponding mutant killing criteria. Before describing the operators in detail in Section 3.1.2, we discuss the mutant killing criteria in the following section.

Table 3.1: Proposed operators for testing buffer overflow vulnerabilities and the corresponding killing criteria

Category	Operator	Brief description	Killing criteria
Mutating library function calls	S2UCP S2UCT S2UGT S2USN S2UVS	Replace *strncpy* with *strcpy*. Replace *strncat* with *strcat*. Replace *fgets* with *gets*. Replace *snprintf* with *sprintf*. Replace *vsnprintf* with *vsprintf*.	C_1 or C_2
Mutating buffer size arguments	RSSBO	Replace buffer size with destination buffer size plus one.	C_2
Mutating format strings	RFSNS	Replace "*%ns*" format string with "*%s*".	C_1 or C_2
	RFSBO	Replace "*%ns*" format string with "*%ms*", where, m = size of the destination buffer plus one.	C_2
	RFSBD	Replace "*%ns*" format string with "*%ms*", where, m is the size of the destination buffer plus Δ.	C_1
	RFSIFS	Replace "*%s*" format string with "*%ns*", where n is the size of the destination buffer.	C_1 or C_2
Mutating buffer variable sizes	MBSBO	Increase buffer size by one byte.	C_1
Removing statements	RMNLS	Remove null character assignment statement.	C_1 or C_2

3.1.1 Mutant killing criteria

We observe that writing beyond a buffer might crash a program, which makes it difficult to distinguish mutants from the original program by comparing the final output. It may also change

internal program states without crashing immediately (*e.g.*, when there is a one byte overflow). Moreover, a buffer having no null character at the end might lead a program to read from its neighboring locations, which might cause different output or a program crash. Simply comparing output, then, is insufficient to distinguish mutants from the original program. We apply firm mutation-based testing [53] (described in Section 2.1), where we the states between a program and its mutant are compared anywhere between the mutated statements and the end of the program. We define two criteria that can be used to kill mutants as shown in Table 3.2.

Table 3.2: Mutant killing criteria for BOF vulnerabilities

Name	Killing criteria
C_1	$ES_P \neq ES_M$
C_2	Len(Buf_P) $\leq N$ && Len(Buf_M) $> N$
P: The original implementation. *M*: The mutant version. ES_P: The exit status of *P*. ES_M: The exit status of *M*. *N*: Buffer size. *Len*(Buf_P): Length of *Buf* in *P*. *Len*(Buf_M): Length of *Buf* in *M*.	

We begin with a bit of notation to allow us to formalize the mutant killing criteria. Let us consider that *P* is the original implementation unit, and *M* is a mutant of *P*. ES_P and ES_M indicate the exit status of *P* and *M*, respectively. Buf_P and Buf_M represent the buffer variable (*Buf*) in *P* and *M*, respectively and *N* represents the allocation size of *Buf*, which will be the same in both programs. The valid locations for reading and writing the buffer are thus between *Buf[0]* and *Buf[N-1]*. Therefore, the locations neighboring *Buf* start from *Buf[N]*.

A test case kills *M* based on C_1 when *P* does not crash and *M* does. We take advantage of the fact that the exit status of a crashed program is different than that of a program having normal termination. C_2 distinguishes *P* from *M* if the length of *Buf* in *M* exceeds the allocated size *N* while in the original program *P* the length of *Buf* remains within its declared limits.

3.1.2 Description of the operators

Mutating library function calls

The proposed operators of this category replace safe ANSI library function calls with unsafe function calls. The safe functions check the size of the destination buffers before performing operations such as copy and concatenation, whereas the unsafe functions do not. There are five operators in this group: S2UCP, S2UCT, S2UGT, S2USN, and S2UVS. They replace *strncpy*, *strncat*, *fgets*, *snprintf*, and *vsnprintf* with *strcpy*, *strcat*, *gets*, *sprintf*, and *vsprintf,* respectively. The mutants are killed by test cases that cause a program state which satisfies either of the killing criteria immediately after the execution of function calls.

Table 3.3: Example applications of the S2UCP, S2UCT, S2UGT, S2USN, and S2UVS operators

Original program (P)	Mutated program (M)
char dest [32]; ….. strncpy (dest, src, 32);	char dest [32]; ….. strcpy (dest, src); //**ΔS2UCP**
char dest [32]; ….. strncat (dest, src, 32);	char dest [32]; ….. strcat (dest, src); //**ΔS2UCT**
char dest [32]; ….. fgets (dest, 32, stdin);	char dest [32]; ….. gets (dest); //**ΔS2UGT**
char dest [32]; ….. snprintf (dest, 32, “%s”, src);	char dest [32]; ….. sprintf (dest, “%s”, src); //**ΔS2USN**

char dest [32];
…..
va_list ap;
va_start(ap, fmt);
vsnprintf (dest, 32, fmt, ap);
	char dest [32];
	…..
	va_list ap;
	va_start(ap, fmt);
	vsprintf (dest, fmt, ap); //**ΔS2UVS**

Table 3.3 shows example applications for each of the five mutation operators involving a destination buffer *dest* of size 32 bytes. In the *strncpy, strncat, snprintf,* and *vsnprintf* cases, the safe library function call in *P* is replaced by the unsafe version in *M* and the argument specifying the buffer length is removed. In the third row, *P* contains the *fgets* function call that copies characters from console (*stdin*) to the *dest* buffer. The mutant *M* injects BOF vulnerabilities by replacing the *fgets* with a *gets* call that specifies neither the length of the buffer nor the source of input. Note that the S2UGT operator is applied when the input file source is stdin (*i.e.*, standard input).

Table 3.4 shows an example of mutation analysis for the S2UCP operator with two test cases. The program *P* has a library function call *strncpy*, which is replaced with *strcpy* in the mutant *M*. The first row shows that the mutant remains live with test input (*src*) having length 32 bytes, as it does not trigger any of the mutant killing criteria. The second row shows that the mutant is killed by a test case having input length of 256 bytes that satisfy the C_1 criterion. The input makes *M* to crash. However, *P* does not crash. Similar analysis can be preformed for the other four operators (*i.e.*, S2UCT, S2UGT, S2USN, and S2UVS).

Table 3.4: Mutation analysis example for the S2UCP operator

String length (src)	Original program (P)	Mutated program (M)	Output (P)	Output (M)	Status
[32]	char dest [32]; ….. strncpy (dest, src, 32);	char dest [32]; ….. strcpy (dest, src); //**ΔS2UCP**	No crash	No crash	Live
[256]	As above	As above	No Crash	Crash	Killed

Mutating buffer size arguments

The RSSBO operator modifies the buffer size argument of safe ANSI C library function calls such as *strncpy*, *strncat*, *memcpy*, *memset*, *memmove*, *snprintf*, *fgets*, and *vsnprintf*. The size is set to the length of the destination buffer plus one. The mutants are killed with test cases that satisfy the C_2 killing criterion immediately after execution of the mutated function call. Table 3.5 shows a mutation analysis example for the RSSBO operator with two test cases. The left column of the first row shows that an original program *P* has a local 32-byte buffer named *dest*. *P* contains the *strncpy* library function call that copies at most 32 bytes from the source string (*src*) into the *dest* buffer; the mutant *M* replaces the size argument (whose value is 32) with the value 33 in the function call. A test case (*src*) of 32 bytes having arbitrary characters cannot kill the mutant based on criterion C_2 as the length of *dest* remains same (32 bytes) in both *P* and *M* assuming there is a null character at the end of buffer. However, the second row shows an effective test case having 33 bytes that changes the string length of *dest* to 33 bytes in *M*; however in *P* the string length remains 32 bytes. Thus, the mutant is killed by the second test case, employing the C_2 criterion.

Table 3.5: Mutation analysis example for the RSSBO operator

String length (src)	Original program (P)	Mutated program (M)	Output (P)	Output (M)	Status
[32]	char var1; char dest [32]; … strncpy (dest, src, 32);	char var1; char dest [32]; … strncpy (dest, src, 33); //**ΔRSSBO**	32 bytes	32 bytes	Live
[33]	As above	As above	32 bytes	More than 32 bytes	Killed

Mutating format strings

There are four operators in this category: RFSNS, RFSBO, RFSBD, and RFSIFS. The operators mutate format strings of the *scanf* and *printf* family functions (also known as format functions) to inject BOF vulnerabilities in destination buffer arguments. The RFSNS operator replaces "*%xs*" format specifiers with "*%s*", where *x* represents the maximum number of bytes to be read from a source buffer. The RFSBO and RFSBD operators change the value of *x* to *length(dest)+1* and *length(dest)*+Δ, respectively. Here, *length(dest)* represents the length of the destination buffer. The Δ value is the offset (in bytes) between the buffer starting location and the program's return address in the stack. Here, the format functions are called inside a program. The mutants generated by the RFSNS operators are killed with test cases that satisfy any of the killing criteria immediately after the execution of function calls. The mutants generated by the RFSBO and RFSBD operators are killed by test cases that satisfy criterion C_2 and C_1, respectively.

Table 3.6: Example applications of the RFSNS, RFSBO, and RFSBD operators

Original statement	**Mutated statement**

char dest [32];
sscanf (src, "%32s", dest);
	char dest [32];
	sscanf (src, "%s", dest); //**Δ RFSNS**
	sscanf (src, "%33s", dest); //**ΔRFSBO**
	sscanf (src, "%40s", dest); //**ΔRFSBD**

Table 3.7: Example application of the RFSIFS operator

Original statement	Mutated statement
char dest [32]; sscanf (src, "%s", dest);	char dest [32]; sscanf (src, "%32s", dest); //**Δ RFSIFS**

Table 3.8: Mutation analysis example for the RFSBD operator

String length (src)	Original program (P)	Mutated program (M)	Output (P)	Output (M)	Status
[32]	char dest [32]; ….. sscanf (src, "%32s", dest);	char dest [32]; ….. sscanf (src, "%40s", dest); //**ΔRFSBD**	No crash	No crash	Live
[256]	As above	As above	No Crash	Crash	Killed

Table 3.6 shows example applications of the three operators for *sscanf* function. The RFSNS operator replaces the format string "%32s" with "%s". The RFSBO operator replaces "%32s" with "%33s". The RSSBD operator replaces "%32s" with "%40s" as the offset from the *dest* buffer to the last byte of the return address is 8 (the frame pointer and return address occupy 4 bytes) assuming no other variables are declared before *dest*. Since the size of the *dest* buffer is 32 the value of Δ is 40. The RFSIFS operator replaces string specifiers "%s" with "%xs", where the value of x is *length(dest)*; an example of its application is shown in Table 3.7.

An example of mutation analysis of the RFSBD operator is shown in Table 3.8. The first row shows that a test case (*src*) 32 bytes long is not able to kill the mutants based on criterion C_1 as it does not cause a buffer overflow. However, the second row has a test case of 256 bytes in length that kills the mutant by overwriting the return address of the function on the stack. Similar analysis can be performed with the RFSNS, RFSBO, and RFSIFS operators.

Note that we also propose three mutation operators for testing format string bug vulnerabilities that modify the format strings of format functions. We describe them in Chapter 5 and provide a comparison with the RFSNS, RFSBO, RFSBD, and RFSIFS operators.

Mutating buffer variable sizes

The MBSBO operator increases the buffer variable size by one byte during buffer declarations. The generated mutants may be killed with effective test cases that satisfy the C_1 killing criterion. The state between a mutant and the original is compared after executing the entire program.

Table 3.9: Mutation analysis example for the MBSBO operator

String length (src)	Original program (P)	Mutated program (M)	Output (P)	Output (M)	Status
[34]	2. char dest [32]; … 8. strcpy (dest, src); …	2. char dest [33]; //**ΔMBSBO** … 8. strcpy (dest, src); …	No crash	No crash	Live
[40]	As above.	As above.	Crash	No crash	Killed

Table 3.9 shows a mutation analysis example for the MBSBO operator. The program *P* declares a buffer *dest* of size 32 bytes at Line 2 which is used in Line 8. The mutant increases the buffer size to 33 bytes at Line 2. We notice that both *P* and *M* are vulnerable to BOF as they use the *strcpy* function call that allows copying the *src* string to *dest* buffer without any bound checking. The first row shows that a test case (*src*) 34 bytes long is not able to kill the mutant based on criterion C_1. However, the second row has a test case of 40 bytes that overwrites the return address in *P* causing it to crash. However, the return address in *M* is not overwritten by the test case, so the mutant is killed. We notice that even though a test case having 34 bytes actually overflows the *dest* buffer in both *P* and *M*, the operator helps to reveal the BOF vulnerability by creating one byte difference in the location of the return address.

Removing statements

The RMNLS operator removes statements that assign the null character at the end of a buffer. Since the null character is used to mark the end of a buffer, removing these statements allows contiguous memory locations beyond the buffer size to be easily read. The mutants are killed by test cases that satisfy either the C_1 or C_2 criteria, which are checked at a breakpoint immediately after the removed statement. Table 3.10 shows an example of mutation analysis with an effective test case that kills the mutants based on criterion C_2. Here, the test case (*src*) is 32 bytes long. In *P*, assigning a null character after the call of the safe library function restricts the buffer length of dest to 32 bytes. However, in *M*, removing the null character assignment statement means that the perceived buffer length exceeds the declared *dest* buffer length of 32 bytes.

Table 3.10: Mutation analysis example for the RMNLS operator

String length (src)	Original program (P)	Mutated program (M)	Output (P)	Output (M)	Status
[32]	char dest [32]; … strncpy (dest, src, 32); dest [32] = '\0';	char dest [32]; … strncpy (dest, src, 32); //Δ **RMNLS**	32 bytes	More than 32 bytes	Killed

3.2 Relationship between BOF attacks and the operators

In Table 3.11, we show the relationship between common attacks that expose buffer overflow vulnerabilities and our proposed mutation operators. Several attacks, including buffer overflow by one byte, multiple bytes, and overwriting the return address of a function, have been studied widely [54, 56, 75] and were reviewed in Section 2.2.1. The corruption of return addresses can be detected through test cases that kill mutants generated by the S2UCP, S2UCT, S2UGT, S2USN, S2UVS, RFSNS, RFSBD, RFSIFS, and MBSBO operators. Attacks that expose BOF by one byte

can be detected by test cases that kill mutants generated by the RSSBO and RFSBO operators. The RMNLS operator helps revealing BOF vulnerabilities that cause arbitrary reading of neighboring variables of a buffer in either stack or heap area.

Table 3.11: BOF attacks and the proposed operators

Attack	Operator
Overwriting return addresses.	All except RSSBO and RFSBO.
Overwriting stack and heap.	All the proposed operators except RMNLS.
One byte overflow.	RSSBO and RFSBO.
Arbitrary reading of stack	RMNLS.

3.3 Prototype tool implementation

In this section, we describe the implementation of a prototype tool to perform **mu**tation-based **b**uffer **o**verflow vulnerability **t**esting (**MUBOT**). The tool accepts a C program unit (*e.g.*, function) and automatically generates mutants and also helps in performing mutation analysis for a given test data set. The tool reports the mutation score along with the list of *live* mutants to help a tester generating new test cases. A snapshot of the tool is shown in Figure 3.1. The tool is developed using the TCL 8.1 (Tool Command Language) scripting language. The TCL script is launched with the *wish* program in Cygwin, which is a Linux-like emulator for Windows XP.

The input function is scanned line by line and specific text fragments are replaced based on the chosen operators. The example snapshot of Figure 3.1 shows that all the operators are selected to generate mutants for the input function named edbrowse-bad.c. The tool generates mutants automatically (by clicking on the "Generate Mutants" button). Some of the operators do not generate any mutants (*e.g.*, S2UCP, S2UGT, RSSBO, etc.) for the function as it does not have any library function call on which the operators can be applied to inject BOF vulnerabilities. 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.

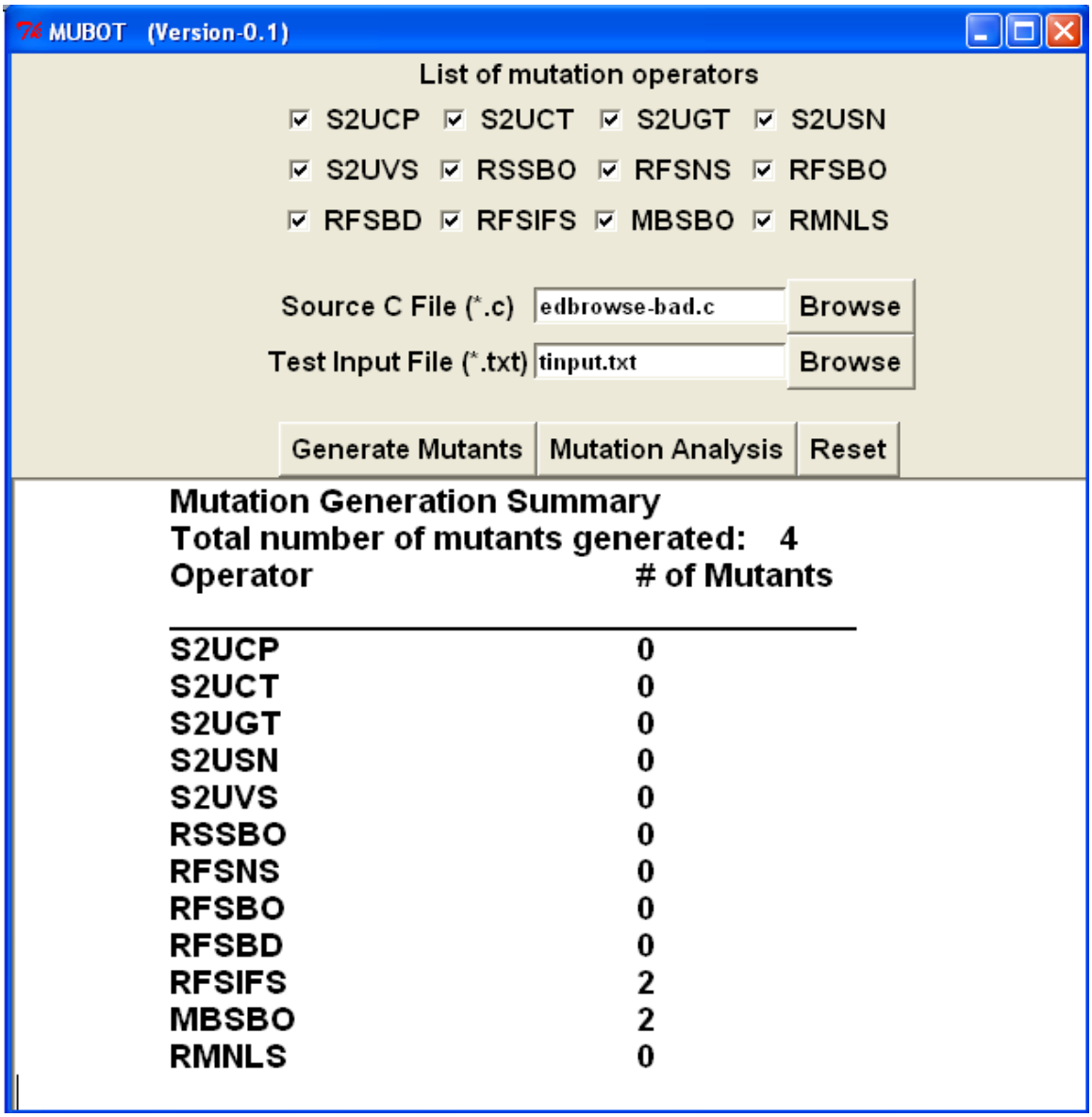

Figure 3.1: Snapshot of the MUBOT tool

An auxiliary text file (*muaux.txt*) saves the name of the mutant file, the line number where the mutation is performed, and the name and size of the buffer variable that needs to be monitored to determine whether the mutants should be killed. In the mutation analysis stage (*i.e.*, by clicking on the "Mutation Analysis" button), the mutants and the original program are compiled and run for each input test case provided by a test data set (*e.g.*, tinput.txt). Depending on the mutation operators, the intermediate or final program state is compared between the mutant and the original implementation. 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.

3.4 Evaluation of the proposed operators

We evaluate the effectiveness of the proposed operators with four open source programs. First, we discuss how the four benchmark programs are chosen, along with their characteristics. We then describe the method by which operator effectiveness is evaluated followed by the results obtained by the evaluation and an analysis of these results. We also show how our proposed operators help to generate effective test cases that can reveal BOF vulnerabilities.

Benchmark program selection

Table 3.12 shows the four applications that we have selected for evaluating the operators. These applications have BOF vulnerabilities reported in the Common Vulnerabilities and Exposures (CVE) [12] or the Open Source Vulnerabilities Database (OSVBD) [13]. The details of the reported bugs can be found in the table, including where it has been reported, the source file and

location of the vulnerability, and whether the overflow occurs on the stack or the heap. Wu-ftpd, Edbrowse, and Rhapsody suffer from BOF vulnerabilities in their *SockPrintf*, *ftpls*, and *parse_input* functions, respectively, and the vulnerable buffers are located in the stack region. The cmdftp program allocates a dynamic buffer inside its *store_line* function, which affects the heap area.

We call the program versions with vulnerable functions "bad programs" and their corresponding fixed versions "good programs". A fixed version is either a patch or an upgraded version. We obtain the patched version for the Wu-ftpd and the Rhapsody programs. However, for the other two programs, we use the upgraded versions which are Edbrowse-3.3.1 and Cmdftp-0.9.5, respectively.

Table 3.12: Characteristics of four open source programs

Application name	Application type	Bug ID	Source file, function name	Lines of code	Buffer location
Wu-ftpd-2.6.2	Ftp server	CVE-2003-1327	ftpd.c, SockPrintf	8	Stack
Edbrowse-2.2.10	Command line editor browser	CVE- 2006-6909	http.c, ftpls	70	Stack
Rhapsody IRC-0.28b	Text-based IRC client console for Unix	CVE-2007-1502	main.c, parse_input	220	Stack
Cmdftp-0.64	Command line FTP client	OSVDB-2522	cmdftp.c, store_line	7	Heap

It is worth mentioning here that there are several other benchmark programs for testing BOF vulnerabilities, which we do not use for purposes of our evaluation. The benchmark developed by Kratkiewicz [76] consists of 291 sets of C programs that are written based on her proposed taxonomy of buffer overflows. Each set contains one program that is free from BOFs and three programs that have BOFs of various magnitudes defined in terms of minimum (one byte),

medium (10 bytes) and large (more than 4,000 bytes). The benchmark is applied to evaluate the effectiveness of several static analysis tools. However, the provided benchmark programs have no relation with input test cases that can expose these BOF vulnerabilities. In other words, no matter what the test case, the execution of a program always leads to BOF exploits. Mutation-based testing is intended for assessing the quality of the test cases that can expose BOF vulnerabilities, so this benchmark is not a good candidate.

Zister *et al.* [75] develop a benchmark program suite that contains model programs with and without BOF vulnerabilities. It has real world applications which have been reported in the CVE database for having exploitable security bugs. The programs include Bind (a DNS server), Sendmail (an email server), and Wu-ftpd (a ftp server). The benchmark is applied to evaluate the effectiveness of several static analysis tools (similar to [76]). However, most of the model programs have BOFs due to logical errors (*e.g.*, off by one in loop condition). In contrast, we also address BOFs due to ANSI C library function calls, statement removal, etc. Our operators do not address BOFs that are due to implementation logical errors inside loop conditions, if-else blocks, etc.

Newsham *et al.* [77] develop the Analyzer Benchmark (ABM) program suite. The benchmark consists of 91 micro programs and one macro program from a real world application. Several micro programs contain BOFs in stack and heap locations due to ANSI C library method calls (*e.g.*, *strncpy*, *strncat*, etc.) and off by one error in conditional expressions. However, the programs are very small and do not represent real world applications.

Kelvin *et al.* [78] develop a benchmark suite having BOF vulnerabilities from 12 open source programs, which are selected based on vulnerability entries from the CVE list. The suite is applied to test the effectiveness of SATABS, a SAT-based abstract model checker tool that

detects BOF errors in C programs. For each of the applications, several model programs are developed that use stub functions instead of implemented functions. This feature contrasts with our intention of adequate testing of an original implementation for revealing vulnerabilities.

Since none of these existing benchmark suites are suitable for our purposes, we had to carefully choose some open source applications that contain the BOF vulnerabilities reported in the CVE and the OSVBD (see Table 3.12). This is not surprising, because mutation-based testing is new to the field of BOF vulnerability testing and has a very different methodology than the static analysis that most of these benchmark suites are developed for.

Evaluation of operator's effectiveness

To evaluate the effectiveness of our proposed operators, we follow the method used by Delamaro *et al.* [10]. 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.

Construction of the initial test data sets

For each of the programs, we construct 15 initial test data sets. Each test data set consists of 10 test cases having appropriate arguments that can be supplied to a program. The string arguments that are responsible for exposing BOF vulnerabilities are generated using a popular random testing tool named Fuzz [79]. For example, the command "*fuzz* 10 *–p –l* 100 *–o testinput.txt"* generates 10 random strings using a default seed number that each have a length of 100 bytes and writes the result to a file named *testinput.txt*. Since we are not using any specification for generating initial test cases, we believe that using a random testing tool serves the purpose of building input test cases. String lengths are chosen randomly using a uniform random number generator provided by gcc's *rand* function. We scale the return value of the random number generator between 1 and 65,535. We set the return value by observing the benchmark programs developed by Zister *et al.* [75] and Kelvin *et al.* [78].

Results and data analysis

Table 3.13 shows total number of mutants, average *MS*, average test data set sizes, and percentages of test data sets that reveal BOF vulnerabilities. We initially generate mutants for bad programs. The second column of Table 3.13 shows the total number of mutants generated for the

four open source programs. For each of the bad programs, we obtain an adequate test data set *T* that kills all the generated mutants. 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); (ii) if the *MS* of the test data set is not 100%, then analyze the *live* mutants to generate additional test cases to kill them. Steps (i) and (ii) are repeated 15 times (*i.e.*, once for each test data set) to reduce any bias in the analysis. The obtained average *MS* for all the programs are found to be 100% as shown in the third column of the table. We compute the average test data set sizes as shown in the fourth column of Table 3.13. We notice that it is possible to generate adequate test data sets having 100% *MS* for all the bad programs.

Table 3.13: Summary of mutation generation and analysis for BOF vulnerabilities

Name	Total number of mutants	Average *MS* (%) of bad programs	Average test data set size	% of test data sets reveal BOF vulnerabilities
Wu-ftpd	1	100	12.53	100
Edbrowse	4	100	12.86	100
Rhapsody IRC	20	100	20.40	100
Cmdftp	2	100	12.93	100

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 it from its corresponding good program (the last column of Table 3.13) Therefore, our proposed operators help in obtaining adequate test data sets capable of revealing BOF vulnerabilities.

To illustrate how the proposed operators force the generation of adequate test data sets for BOF vulnerabilities, we show the case of Wu-ftpd program. The code snippet of the *SockPrintf* function (from the good version) is shown in Table 3.14 (the left column). In general, the function receives a format string (*format*) and variable arguments (by '...' at Line 1), which are written in

a locally declared buffer named *buf* (Line 3) having a length of 32,768 bytes (Line 4-6). Line 7 places a null character at the end of *buf* to prevent BOF vulnerabilities. The content of the *buf* is written in the file variable *sockfp* at Line 8 through the *SockWrite* function call. The bad (or vulnerable) version of the same program is shown in the right column of Table 3.14. The function call to *vsprintf* at Line 5 results an overflow in *buf*.

Table 3.14: Code snippet of *SockPrintf* function

SockPrintf (good)	SockPrintf (bad)

1. int SockPrintf (FILE *sockfp, char *format, ...){
2. va_list ap;
3. char buf [32768];
4. va_start (ap, format);
5. vsnprintf (buf, sizeof(buf), format, ap);
6. va_end (ap);
7. buf [sizeof(buf)] = '\0';
8. return SockWrite (buf, 1, strlen(buf), sockfp);
}
	1. int SockPrintf (FILE *sockfp, char *format, ...){
	2. va_list ap;
	3. char buf [32768];
	4. va_start (ap, format);
	5. vsprintf (buf, format, ap);
	6. va_end (ap);
	7. return SockWrite (buf, 1, strlen(buf), sockfp);
	}

Table 3.15: Mutants generated from *SockPrintf* function (vulnerable)

Line	Operator	Mutated statement
3	MBSBO	char buf [32769];

Table 3.15 shows the only mutant, generated by the operator MBSBO at Line 3 for the bad program along with the mutated statement. A snapshot of the initial test data set having 10 test cases (*i.e.*, test case number 1-10) is shown in Table 3.16. Each test case has three elements of the form $<t_1, t_2, t_3>$. Here, t_1, t_2, and t_3 represent a file pointer, a format string, and string input, respectively. We set t_1 as standard error console (*stderr*), t_2 as "%s" and keep the value of both t_1 and t_2 same for all test cases. Each value of t_3 is a string generated randomly using the Fuzz tool. We only show the length in bytes in the column for t_3. The resultant test data set does not kill any

of the four mutants, and the *MS* of the initial test data set is zero. The test data set is enhanced with a new test case (*i.e.*, test case number 11) having t_3 of length 32,776, while keeping t_1 and t_2 same as before. The test vector kills the mutant generated by the MBSBO operator. Thus the test data set is adequate as it obtains 100% *MS* and its size is 11 (*i.e.*, 11 test cases). This adequate test data set differentiates between the bad program and the good program by satisfying criterion *C1*. Thus, the adequate test data set helps in revealing the BOF vulnerabilities in the bad program.

Table 3.16: A snapshot of a test data set generated randomly

Test case number	t1	t2	t3
1	stderr	%s	3,500
2	stderr	%s	5,431
3	stderr	%s	64
4	stderr	%s	785
5	stderr	%s	9,243
6	stderr	%s	3,200
7	stderr	%s	100
8	stderr	%s	6,744
9	stderr	%s	16,056
10	stderr	%s	23,122
11	stderr	%s	32,776

3.5 Conclusion

This chapter describes mutation-based testing of BOF vulnerabilities that helps in generating an adequate test data set, which has not been addressed properly in the area of vulnerability testing. We propose 12 mutation operators to support the mutation-based testing process. The operators inject BOF vulnerabilities in ANSI C standard library calls and programming elements such as buffer variables and statements. The mutants generated by the proposed operators can be killed with the proposed distinguishing criteria in the presence of effective test cases. A tool named

MUBOT has been implemented to automatically generate mutants and perform mutation analysis for an initial test data set. The operators are found effective for four open source applications having BOF vulnerabilities.

Chapter 4
Mutation-Based Testing of SQL Injection Vulnerabilities

In this chapter, we apply the concept of mutation-based testing to test SQL injection vulnerabilities (SQLIVs). We propose nine mutation operators capable of injecting SQLIVs in the source code of applications. The mutants generated based on the proposed operators can be killed by effective test cases containing SQL injection attacks (SQLIAs). We implement a prototype **MU**tation-based **S**QL **I**njection vulnerability **C**hecking (testing) tool (**MUSIC**), which automatically generates mutants for applications implemented in Java Server Pages (JSP) and performs mutation analysis. We validate the proposed operators with five open source web-based applications. The proposed operators are found to be effective for detecting SQLIVs and generating an adequate test data set.

The chapter is organized as follows. Section 4.1 describes the proposed operators along with mutant killing criteria to test for SQLIVs. Section 4.2 discusses the relationship between SQLIAs and the operators. Section 4.3 describes the prototype tool implementation and Section 4.4 provides the detail results for the evaluation of the operators. Section 4.5 summarizes the chapter.

4.1 Proposed operators and mutant killing criteria

The nine mutation operators divided into two categories. The first category consists of four operators that inject faults into where conditions (WC) of SQL queries. The second category consists of five operators that inject faults in database API method calls (AMC). A summary of

the proposed mutation operators along with the corresponding killing criteria is provided in Table 4.1. Before discussing the operators in Section 4.1.2, we first discuss the proposed killing criteria for mutants in the following section.

Table 4.1: Proposed operators for testing SQL injection vulnerabilities and the corresponding killing criteria

Category	Operators	Description	Killing criteria
WC	RMWH	Remove WHERE keywords and conditions.	C_1
	NEGC	Negate each of the unit expression inside where conditions.	
	FADP	Prepend "*FALSE AND*" after the WHERE keyword.	C_2
	UNPR	Unbalance parentheses of where condition expressions.	C_3
AMC	MQFT	Set multiple query execution flags to true.	$C_4 \mid\mid C_5 \mid\mid C_6$
	OVCR	Override commit and rollback options.	C_4
	SMRZ	Set the maximum number of record returned by a result set to infinite.	C_6
	SQDZ	Set query execution delay to infinite.	C_7
	OVEP	Override the escape character processing flags.	C_5

4.1.1 Mutant killing criteria

We apply weak mutation-based testing [52] approach (discussed in Section 2.1) for SQLIVs, where database states are compared between an implementation and its mutant immediately after executing a mutated statement. Since we are injecting vulnerabilities in either SQL generation related statements or database API function calls, they can be easily monitored immediately after executing the mutated statement. The strong and firm mutation cannot be used as the result set generated by executing a mutated query might not affect the end output generated by applications.

We propose seven distinguishing or killing criteria for mutants as shown in Table 4.2. Let *I* and *M* be the intended query and its corresponding mutation, respectively. Let us assume that queries

use tables having N total records ($N > 1$). Let n_1 and n_2 be the set of records selected by executing I and M, respectively. Criterion C_1 distinguishes I and M, if either (i) the cardinality of the intersection between n_1 and n_2 is zero or N; or (ii) the cardinality of the union of n_1 and n_2 is greater than N. C_2 distinguishes between I and M, if the cardinality of the intersection of n_1 and n_2 is greater than zero.

Table 4.2: Mutant killing criteria for SQL injection vulnerabilities

Name	Distinguishing criteria
C_1	$(\|n_1 \cap n_2\| = 0) \,\|\|\, (\|n_1 \cap n_2\| = N) \,\|\|\, (\|n_1 \cup n_2\| > N)$
C_2	$\|n_1 \cap n_2\| > 0$
C_3	$s_1 \neq s_2$
C_4	$(i_1 \neq i_2) \,\|\|\, (d_1 \neq d_2) \,\|\|\, (u_1 \neq u_2) \,\|\|\, (o_1 \neq o_2)$
C_5	$p_1 \neq p_2$
C_6	$\|n_2\| > \|n_1\|$
C_7	$(t_1 > T \,\&\&\, t_2 < T) \,\|\|\, (t_1 < T \,\&\&\, t_2 > T)$
I: Intended query. *M:* Mutated query. n_1: The record set selected by *I*. *n2:* The record set selected by *M*. s_1: State where *I* runs successfully and *M* generates error message. s_2: State where *M* runs successfully and *I* generates error message. i_1: # of records inserted by *I*. i_2: # of records inserted by *M*. d_1: # of records deleted by *I*. d_2: # of records deleted by *M*. u_1: # of records updated by *I*. u_2: # of records updated by *M*. o_1: # of database objects created by *I*. o_2: # of database objects created by *M*. p_1: # of external objects created by *I*. p_2: # of external objects created by *M*. t_1: Time elapsed to execute *I*. t_2: Time elapsed to execute *M*. *T:* Default query execution timeout of an application.	

Let s_1 represent an application state where query I runs successfully, and query M results in a syntax error. s_2 represents the opposite state of s_1 (*i.e.*, M runs successfully and I generates a syntax error message). Criterion C_3 is used to distinguish between I and M based on the observation that $s_1 \neq s_2$. Let the execution of I and M results in i_1 and i_2 number of record insertions; d_1 and d_2 number of record deletions; u_1 and u_2 number of records updates; o_1 and o_2 number of new object (*e.g.*, tables, views, etc.) creations. A test case kills a mutant, if the number

of insertions, deletions, updates or creations are different; that is, if any of the four conditions are satisfied: *(i)* $i_1 \neq i_2$ *(ii)* $d_1 \neq d_2$ *(iii)* $u_1 \neq u_2$, and *(iv)* $o_1 \neq o_2$. We denote this killing criterion as C_4.

After execution of *I* and *M*, let the number of external objects created (*e.g.*, files) outside the database be p_1 and p_2, respectively. Criterion C_5 distinguishes between *I* and *M* if $p_1 \neq p_2$. Criterion C_6 distinguishes between *I* and *M* if the number of records selected in *I* is less than that of *M*. Let the time elapsed during execution of *I* and *M* be t_1 and t_2, respectively, and the default timeout set by an application be *T*. Criterion C_7 distinguishes between *I* and *M* if either t_1 or t_2 exceeds *T,* but not both.

4.1.2 Description of the operators

We show a simple database table named *tlogin* in Table 4.3. The table has three columns named *id*, *uid*, and *pwd*, which represent unique identification number of a user, user id, and user password, respectively. Let us assume that an intended query written by a programmer is *"select id from tlogin where uid='" + userid +"'"*. Here, *userid* is a string variable that receives user supplied user id and becomes part of query generation process without filtering. We use the *tlogin* table and the intended query to describe the proposed operators.

Table 4.3: Records of *tlogin* table

id (numeric)	uid (varchar)	pwd (varchar)
1	aaa	aaa
2	bbb	bbb
3	ccc	ccc

Remove SQL where conditions (RMWH)

The RMWH operator removes where conditions of SQL queries, which results in selecting all rows from tables. The generated mutants are killed by test cases that satisfy criterion C_1. Test

cases having tautology or union type attacks kill the generated mutants. This operator is applicable for SELECT, UPDATE, and DELETE type queries.

Table 4.4: Example applications of the RMWH operator

Test case	Intended query (*I*)	Mutated query (*M*)	n_1	n_2	Status
aaa	select id from tlogin where uid='aaa'	select id from tlogin //**ΔRMWH**	{1}	{1, 2, 3}	Live
' or 1=1 --	select id from tlogin where uid='' or 1=1 --	select id from tlogin //**ΔRMWH**	{1, 2, 3}	{1, 2, 3}	Killed
' union select 20 --	select id from tlogin where uid='' union select 20 --	select id from tlogin //**ΔRMWH**	{20}	{1, 2, 3}	Killed

Table 4.4 shows three example applications of the operator. The first row shows the intended query (*I*) and mutated query (*M*) with test case *'aaa'*. We notice that *'aaa'* is a valid *uid* of Table 4.3, and it does not contain an SQLIA. Executions of *I* and *M* result in the record sets n_1 and n_2, which are {1} and {1, 2, 3}, respectively. The intersection and union of thse two sets are {1} and {1, 2, 3}, respectively, making the cardinalities of intersection and union 1 and 3. Therefore, the test case does not satisfy criterion C_1, and the mutant remains *live*. However, the second and third test cases that contain a tautology attack ("*' or 1=1 --*") and a union attack ("*' union select 20 --*") each kill a mutant; the tautology attack leaves the intersection empty, triggering the first clause of criterion C_1, and the union attack results in a larger result set than the entire table, triggering the third clause of the criterion.

Negation of expression in where conditions (NEGC)

The NEGC operator negates unit expressions (*e.g.*, *uid='aaa'* to *uid!='aaa'*) present in where conditions of SQL queries, and can be applied to SELECT, UPDATE, and DELETE type queries. The intersection of the original and mutant record sets (one selected by an arbitrary condition and the other selected by its negation) should be null, provided the semantics of the query do not

change. We take advantage of this observation to force the generation of attack test cases that violate the observation (*i.e.*, satisfy C_1 criterion). Table 4.5 shows three examples of the NEGC operator, where the equal (=) operator in *I* is mutated to not equal (!=) in *M*. The first row shows that *'aaa'* (a non SQLIA test case) cannot kill the mutant. The mutant is killed by tautology (second row) and union attack test cases (third row), where criterion C_1 is satisfied.

Table 4.5: Example applications of the NEGC operator

Test case	Intended query (*I*)	Mutated query (*M*)	n_1	n_2	Status
aaa	select id from tlogin where uid='aaa'	select id from tlogin where uid!='aaa' //**ΔNEGC**	{1}	{2, 3}	Live
' or 1=1 --	select id from tlogin where uid='' or 1=1 --	select id from tlogin where uid!='' or 1=1 -- //**ΔNEGC**	{1, 2, 3}	{1, 2, 3}	Killed
' union select 20 --	select id from tlogin where uid='' union select 20 --	select id from tlogin where uid!='' union select 20 -- //**ΔNEGC**	{20}	{1, 2, 3, 20}	Killed

Prepend '*false and*' after the WHERE keyword (FADP)

The FADP operator prepends '*false and*' after the WHERE keyword. The objective is to nullify any output generated by the execution of an intended query when a test case does not represent an attack. Test cases having SQLIAs kill the generated mutants based on criterion C_2, which checks for a non-null intersection. The operator is applicable to SELECT, UPDATE, and DELETE type queries.

Table 4.6: Example applications of the FADP operator

Test case	Intended query (*I*)	Mutated query (*M*)	n_1	n_2	Status
aaa	select id from tlogin where uid='aaa'	select id from tlogin where false and uid='aaa' //**ΔFADP**	{1}	{}	Live
' or 1=1 --	select id from tlogin where uid='' or 1=1 --	select id from tlogin where false and uid ='' or 1=1 -- //**ΔFADP**	{1,2,3}	{1,2,3}	Killed
' union select 20 --	select id from tlogin where uid='' union select 20 --	select id from tlogin where false and uid ='' union select 20 -- //**ΔFADP**	{20}	{20}	Killed

Table 4.6 shows three example applications of the operator, where the mutant is live with the non attack test case *'aaa'* (in the first row). The mutant is killed by a tautology (second row) and a union attack test case (third row), as the killing criterion C_2 is satisfied by both test cases.

Unbalance parentheses (UNPR)

The UNPR operator makes SQL queries syntactically incorrect to the database engine by appending a left parenthesis at the beginning of the where condition. Test cases having a right parenthesis can make the query syntactically correct and the execution of generated queries successful. However, non malicious test cases generate queries having unbalanced parentheses, and database drivers throw SQL syntax errors. The operator can be applied to SELECT, INSERT, UPDATE, and DELETE type queries. The generated mutants are distinguished by criterion C_3 in the presence of test cases containing SQLIAs which are tautology, union or hexadecimal encoded queries containing an unbalanced right parenthesis. Table 4.7 shows three examples of the UNPR operator where a left parenthesis is added after the WHERE keyword in *M*. The mutant is *live* for the first test case *'aaa'* and killed for the last two attack test cases.

Table 4.7: Example applications of the UNPR operator

Test case	Intended query (*I*)	Mutated query (*M*)	s_1	s_2	Status
aaa	select id from tlogin where uid='aaa'	select id from tlogin where (uid='aaa' //**ΔUNPR**	No error in *I*	Error in *M*	Live
') or 1=1 --	select id from tlogin where uid='') or 1=1 --	select id from tlogin where (uid='') or 1=1 -- //**ΔUNPR**	Error in *I*	No error in *M*	Killed
') union select 20 --	select id from tlogin where uid='') union select 20 --	select id from tlogin where (uid='') union select 20 -- //**ΔUNPR**	Error in *I*	No error in *M*	Killed

Set multiple query execution flags true (MQFT)

The MQFT operator injects SQLIVs by setting the multi query flag value to true. The database API methods are used to connect databases through a URL string specifying the database name

and location, which also allows programmers to specify whether to allow executing multiple queries in a single connection. Allowing multiple queries makes it possible to perform piggybacked query attacks that append arbitrary SQL queries. The operator either adds the flag value by appending it to the connection URL or modifying the existing flag value to true.

Table 4.8: Example applications of the MQFT operator

Test case	Original statement	Mutated statement
' ; show tables; --	String dataCon = "jdbc:mysql://localhost/db1";	String dataCon = "jdbc:mysql://localhost/theDatabase?allowMultiQueries=true"; //**ΔMQFT**
' ; insert into tlogin values ('admin', 'thief'); --	As above	As above

The mutated queries can be killed by test cases that contain piggybacked query attacks. The mutants are killed by any of the C_4, C_5, or C_6 criteria. Table 4.8 shows two example applications of the MQFT operator. In the first row, the original statement contains a database URL in the variable *dataCon*, which does not have the multiple queries execution flag set, thus assuming the default value of *false*. In the mutated statement, the URL is modified by appending "*allowMultiQueries=true*". An example test case that can kill it is "*'; show tables;* --", as it satisfies criterion C_6. Here, the "*show tables*" query provides all the tables of a database in MySQL, so the size of the result set for the mutant is larger than the size for the original program. In the second row, the test case "*'; insert into tlogin values ('admin', 'thief');* --" kills the mutant based on criterion C_4, as the number of rows inserted in the implementation (0) and the mutant (1) are different.

Override commit and rollback options (OVCR)

The commit and rollback options are used to maintain information consistency in database tables. The OVCR operator is used to override the commit and rollback flag to allow SQLIVs, which can be exploited by causing inconsistencies in databases with appropriate SQLIAs. This overriding can be performed either by (i) modifying the commit and rollback flag values or (ii) removing statements that set the flag values of commit and rollback. Since any inconsistent information in tables of a database occurs through INSERT, UPDATE, and DELETE operations, the operator is applicable for these types of queries. The mutants are killed by the test cases that satisfy C_4 criterion.

Table 4.9: Example applications of the OVCR operator

Test case	Original statement	Mutated statement
'; update tlogin set pwd = 'thief' where uid = 'admin' --	conn.setAutocommit (false);	conn.setAutocommit (true); **//ΔOVCR**
';set autocommit=1; update tlogin set pwd = 'thief' where uid = 'admin' --	…. conn.commit();	…. //Remove statement **//ΔOVCR**

Table 4.9 shows two examples of the operator. Here, *conn* is a Java *Connection* class for connecting to databases. The first row shows that the mutated statement sets the flag of *setAutocommit* method to true instead of false. This allows any changes due to INSERT, UPDATE, and DELETE operations to become permanent and create inconsistency through an SQLIA test case (first test case). In the second example, the statement *conn.commit()* is removed, and therefore *autocommit* mode is disabled. However, an SQLIA might set the *autocommit* flag to true and perform further operations to make information inconsistent (second test case).

Set the maximum number of records returned by a result set to infinite (SMRZ)

The database manipulation API often provides developers the ability to specify the maximum number of records that can be returned by the result sets after executing SELECT type SQL queries. The SMRZ operator sets this value to zero if the *setMaxRows* method call occurs, in order to allow a maximum number of records in the case of attack test cases. Therefore, the selected record set is transferred to the result set used by the program without any truncation. If the code is written poorly, all the results might appear in output and information can be leaked. For example, when a user is logged on during authentication, only one record set related to his *userid* should be extracted from database table. Otherwise, it might extract more records and reveal information to an unauthorized user. The generated mutants are killed by the test cases that satisfy criterion C_6.

Table 4.10: Example application of the SMRZ operator

Test case	Original statement	Mutated statement
' or 1=1; --	Statement.setMaxRows (1);	Statement.setMaxRows (0); //**ΔSMRZ**

Table 4.10 shows an example application of the operator. Here, *Statement* is a Java class that supports the execution of queries and the setting of related properties. The mutant is killed by a tautology attack test case, which extracts only one record in the original program and N (N >1) records in the mutated program.

Set query execution delay to infinite (SQDZ)

There are delays between issuing of queries by applications and receiving results from databases. Programmers can specify the maximum delay (or query timeout value) with supported database API method calls. The SQDZ operator sets the delay to infinite as a way of injecting SQLIVs.

This operator forces the generation of test cases that exploit SQLIVs by performing time-based attacks to infer the execution paths of applications [62]. The mutants are killed if a test case satisfies criterion C_7.

Table 4.11: Example application of the SQDZ operator

Test case	Original statement	Mutated statement
';SELECT BENCHMARK (10000000, ENCODE ('abc','123')); ---	Statement.setQueryTimeout (30);	Statement.setQueryTimeout (0); //**ΔSQDZ**

Table 4.11 shows an example application of the operator, where the *setQueryTimeout* method parameter is set to zero to allow infinite execution delay of a query. The attack test case that kills the mutant uses the BENCHMARK function supported by the MySQL database, which repeatedly performs a given task. In this case, the task consists of encrypting the string *'abc'* using password string *'123'* with the function ENCODE 100 million times, which creates a significant delay that might exceed default application query timeout value.

Override the escape character processing flags (OVEP)

The database manipulation API provides developers the option to allow escape character processing. The OVEP operator modifies the flag responsible for processing escape characters. It toggles the flag value and forces the generation of test cases incorporating both escape and non escape characters. The operator generates mutants that can be distinguished in presence of piggybacked query or union attacks using criterion C_5.

Table 4.12 shows two example applications of the operator with attack test cases which kill the mutants. In the first row, *setEscapeProcessing* method's argument is modified from false to true, which means escape character processing is enabled and the "\\" character is treated as "\". In the

second example, the opposite action is performed. Both attack test cases contain the *sp_makewebtask* command that runs the query *"select * from tlogin"* and saves output in the file *users.html.*

Table 4.12: Example applications of the OVEP operator

Test case	Original statement	Mutated statement
*';EXEC master..sp_makewebtask "C:\\\\ users.html", select * from tlogin; --*	Statement.setEscapeProcessing (false);	Statement.setEscapeProcessing (true); //**ΔOVEP**
*' Union EXEC master..sp_makewebtask "C:\\ users.html", select * from tlogin; --*	Statement.setEscapeProcessing (true);	Statement.setEscapeProcessing (false); //**ΔOVEP**

4.2 Relationship between SQL injection attacks and the operators

We show the relationship between the SQLIAs (described earlier in Section 2.2.2) and the proposed operators in Table 4.13. The proposed operators help in generating adequate test data sets that reveal SQLIVs because of tautologies, union queries, piggybacked queries, inference attacks, and hex encoded queries. The attacks are described in Section 2.2.2.

Table 4.13: SQL injection attacks and the proposed operators

Attack	Operator
Tautologies	RMWH, NEGC, FADP, UNPR, SMRZ.
Union queries	RMWH, NEGC, FADP, UNPR.
Piggybacked queries	MQFT, OVCR.
Inference attacks	SQDZ, OVEP.
Hex encoded queries	RMWH, NEGC, FADP, UNPR.

4.3 Prototype tool implementation

We implemented a prototype tool for **MU**tation-based **S**QL **I**njection vulnerabilities **C**hecking (testing) (**MUSIC***)* in Java. An example snapshot of the tool is shown in Figure 4.1. The tool

accepts Java Server Pages (JSP) files and generates mutants based on the chosen operators by clicking on the "Generate" button. Figure 4.1 shows that all the operators that can be chosen. The original and mutated statements are instrumented to monitor both the internal result set returned after query executions and the database state. For each of the JSP files (*e.g.*, Default.jsp), the corresponding Java and Class files are generated (by clicking on the "Gen. class" button) and kept in a real web application container (*e.g.*, an Apache web server) to perform mutation analysis on the input test files (*e.g.*, input.txt file of Figure 4.1). Each test case consists of an appropriate URL having parameters and their corresponding values.

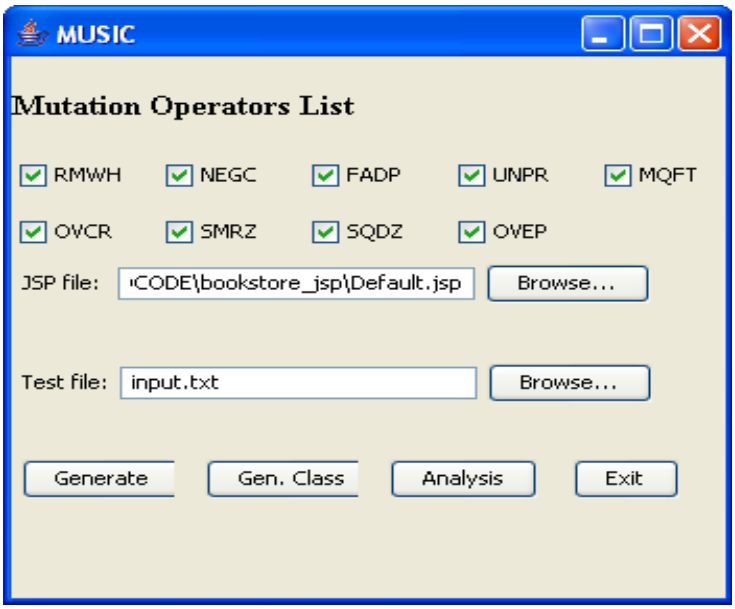

Figure 4.1: Snapshot of MUSIC tool

By clicking on the "Analysis" button, the original program and the mutants are run for each test case. The intermediate result set and database states are compared between a mutant and the original program for each test case to mark it as *killed* or *live*. At the end of mutation analysis, a

report is generated indicating the current mutation score for the supplied test file and the list of *live* mutants. The test set can be enhanced to kill the remaining mutants and increase mutation scores.

4.4 Evaluation of the proposed operators

In this section, we first describe the applications that are used for evaluating the effectiveness of the proposed operators, followed by our evaluation methods. We then provide the results and analysis.

Benchmark program selection

There is no standard benchmark application for testing SQLIVs. However, we use five web-based applications available from an open source web application repository [80] to evaluate the proposed operators. The same applications are used in several related works [22, 34, 39, 40], and that motivated us to use them. The applications are implemented in java server pages (JSP), and they use MySQL databases in the backend.

Table 4.14: Characteristics of five JSP applications

Application	Total files	Lines of code	# of files with SQLIV	SELECT	INSERT	UPDATE	DELETE
Bookstore	10	16,959	7	17	2	2	0
Classifieds	19	10,949	6	7	2	1	1
Events	13	7,242	7	7	3	2	2
Employee directory	10	5,658	7	8	2	2	2
Portal	29	16,453	3	3	1	1	0

The characteristics of the five applications are shown in Table 4.14, which includes the total number of JSP files for each application, total lines of code (LOC), total number of files that might have SQLIVs (*i.e.*, generate SQL queries with user supplied inputs), and the total number

of SELECT, INSERT, UPDATE, and DELETE queries generated by the vulnerable JSP code. Each of the applications has character and numeric filters to prevent SQLIAs.

Method of evaluating operator's effectiveness

We follow the evaluation method of Delamaro *et al.* [10], which has been described in Section 3.5 in detail. We use a benchmark test data set of size 730 containing both attack and non-attack test cases developed by the authors of AMNESIA [34]. Since the chosen applications have both numeric and character input filters, we denote the initial version of the applications as good (*i.e.* non-vulnerable) programs. The bad (*i.e.* vulnerable) programs are obtained by removing the input filters implemented in the applications.

Results and analysis

Table 4.15 shows the mutants generated by WC and AMC categories, average *MS* obtained using the initial test data sets (T_1), enhanced test data sets (T_2), and the percentages of test data sets that reveal SQLIVs. For each of the applications, we generate 15 initial test data sets denoted by T_1. Each test data set contains 25 test cases that are randomly drawn from the benchmark test data set. We take the bad version of the five web applications and use the **MUSIC** tool to generate mutants. The second and third columns of Table 4.15 show the total number of mutants generated for the WC (where conditions) and AMC (API method calls) categories, respectively. We generate Java class files for the mutant JSP files and perform mutation analysis using the T_1 data set for each of the applications.

We find that the mutation score (*MS*) (*i.e.*, ratio of the number of mutants killed to the total number of non-equivalent mutants generated) for each of the applications with the T_1 test data set

is less than 100% (the fifth column of Table 4.15). The lowest *MS* is 64% for the Portal application, whereas the highest *MS* is 78% for the Bookstore application. We analyzed the *live* mutants and find that the T_1 set does not contain test cases to kill the mutants generated by the UNPR operator. The reason is that the initial test data pool does not have any test case containing unbalanced parentheses. We increase the *MS* by enhancing T_1. For each of the applications, we add attack test cases to kill the *live* mutants, and denote the enhanced test data set as T_2. After adding those test cases, the MS increases to 100% for each of the applications (column 6 of Table 4.15). The average test data set size of T_2 for each of the applications is shown in the seventh column of Table 4.15.

Table 4.15: Mutation analysis results for testing of SQLIVs

Application Name	WC	AMC	Total mutants	Average *MS* (%) of bad applications with T_1	Average *MS* (%) of bad applications with T_2	Average test data set size of T_2	% of test data sets revealed SQLIVs
Bookstore	105	73	178	77.53	100	42.5	100
Classifieds	70	41	111	70.27	100	35.2	100
Events	88	49	137	68.61	100	38.4	100
Employee directory	103	50	153	69.28	100	37.9	100
Portal	61	14	75	64.29	100	28.3	100

Now, we want to investigate whether the generated adequate test data sets for each of the bad applications can differentiate the corresponding good applications or not. We notice that at least one test case is able to differentiate a bad program form its corresponding good program for each of the adequate test data sets generated (the last column of Table 4.15). Therefore, the proposed operators are effective in generating adequate test data sets that can reveal SQLIVs in bad applications.

4.5 Conclusion

In this chapter, we apply mutation-based testing approach for testing SQLIVs by proposing nine mutation operators and seven mutant killing criteria for the mutants. Our approach addresses the gap of testing software vulnerabilities and generation of adequate test data sets that can reveal vulnerabilities. The proposed operators inject SQLIVs by mutating both programmer-written queries and database API method calls in implementations. The unique feature of the operators is that the generated mutants can be killed only with test cases containing SQLIAs. Ordinary test cases having no SQLIAs do not kill the mutants. We implement a prototype **MU**tation-based **S**QL **I**njection vulnerabilities **C**hecking (testing) tool named **MUSIC** to automatically generate mutants and perform mutation analysis with input test cases. The operators are found effective for five open source web-based applications written in JSP.

Chapter 5
Mutation-Based Testing of Format String Bug Vulnerabilities

In this chapter, we apply the mutation-based testing technique to perform adequate testing of format string bug (FSB) vulnerabilities in the ANSI C compliant format functions [58]. These format functions are the primary sources of FSB vulnerabilities [12, 13, 14], and they are still widely used for developing many software applications such as ftp servers (*e.g.*, wu-ftpd) and web servers (*e.g.*, apache). However, the existing mutation operators for ANSI C [9, 10, 44] are not designed for testing FSB vulnerabilities.

We propose seven 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 FSB vulnerabilities. The experiment results indicate that the operators are effective for generating adequate test data sets for testing FSB vulnerabilities.

The chapter is organized as follows: Section 5.1 describes the proposed operators along with mutant killing criteria for adequate testing of FSB vulnerabilities. Section 5.2 discusses the relationship between attacks due to FSB vulnerabilities and the operators. Section 5.3 describes the prototype tool implementation, and Section 5.4 discusses our evaluation of the operators. Section 5.5 summarizes the chapter.

5.1 Proposed mutation operators and mutant killing criteria

We propose seven mutation operators for adequate testing of FSB vulnerabilities, which are divided into two categories: format function call modifications and format string modifications. The first category is applicable to both families (*i.e.*, *printf* and *vprintf*) of format functions, whereas the second category is applicable only to *printf* family functions having static format strings. Table 5.1 summarizes all the proposed operators along with the corresponding killing criteria. Before discussing the operators in Section 5.1.2, we first discuss the proposed killing criteria for mutants in Section 5.1.1.

Table 5.1: Proposed operators for FSB vulnerabilities and the corresponding killing criteria

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 from format function calls.	
	FSCAO	Change the argument order in format function calls.	
	FSRAG	Remove arguments from format function calls.	$C_1 \mid\mid C_2$
Format String modifications	FSCFO	Change the order of format specifiers.	
	FSRSN	Replace format specifiers with *%n*.	
	FSPSP	Prepend format specifier types with *n$*.	

5.1.1 Mutant killing criteria

Exploitation of FSB vulnerabilities might lead to program crashes, which make the task of distinguishing a mutant from an original program solely with respect to final output difficult. Moreover, exploitations might corrupt the internal program state without crashing. Therefore, we

apply weak mutation-based testing [52] rather than strong mutation-based testing [4, 5] for killing the mutants. We define two mutant killing criteria as shown in Table 5.2.

Let us assume that P is an original program, and M is a mutant of P. ES_P and ES_M are the exit status (*i.e.*, the exit code) of P and M, respectively. Criterion C_1 differentiates M from P when either 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 corresponding format functions in P and M, respectively. Criterion C_2 differentiates P and M if $W_P \neq W_M$.

Table 5.2: Mutant killing criteria for FSB vulnerabilities

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.	

5.1.2 Description of the operators

Insert format string (FSIFS)

The FSIFS operator inserts format strings in format function calls and is applicable to both the *printf* and *vprintf* family of 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 is lacking an explicit format string, which might result in FSB vulnerabilities [64, 65]. The generated mutants are killed with test cases that satisfy criterion C_1 and contain format specifiers. Table 5.3 shows an example application of the operator for two test cases. The first test case *'aaa'* cannot kill the mutant M as criterion C_1 is not satisfied. The second test case *'%s%s%s'*

forces *P* to read from arbitrary stack locations and crash. However, *M* does not crash as it prints the test case in console. Thus, the mutant is killed, and it forces the tester to generate attack test cases that exploit FSB vulnerabilities.

Table 5.3: Mutation analysis example for 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

Remove format string (FSRFS)

The FSRFS operator removes format strings from both families of format functions to create FSB vulnerabilities in mutants. Since the *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 to the function contains format specifiers and the exit status of the execution of the mutant and original program satisfy criterion C_1. Table 5.4 shows an example application of the operator, where the format string *"%s"* is removed from *printf* function. The first row shows that the test case '*aaa*' cannot kill the mutant *M*, as it does not satisfy criterion C_1. However, the mutant is killed by the attack test case *'%s%s%s'* in the second row as *M* crashes and *P* does not. If the first parameter of the mutated function is not a string variable, then the mutants might not be compiled and need to be removed from analysis.

Table 5.4: Mutation analysis example for the FSRFS operator

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

Change arguments order (FSCAO)

The FSCAO operator changes the parameter order of format function calls to generate FSB vulnerabilities; it is applicable to both *printf* and *vprintf* family functions. Here, the format string is considered to be a movable parameter, but 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. When the FSCAO operator applies, we shift all movable parameters to the left, replacing the last parameter with the first. For the *printf* family, if there are *n* parameters then this parameter shifting is applied several times in sequence, generating a total of *n-1* mutants. However, for the *vprintf* family of functions, there is no explicit list of arguments, so only one mutant is generated. In this case, the format string and argument pointer are replaced with each other. Some of the generated mutants for the *printf* family might not be compilable as the parameters that occupy the format string position might not be string variables.

Table 5.5: Mutation analysis example for 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, src2, “%s %s”); //**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 5.5 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* have the value ‘*aaa’* and *‘bbb’*, respectively. Criterion C_1 is not satisfied as neither *P* nor *M* crashes. The second row

contains effective test case *'%s%s%s'* and *'bbb'*, which crashes *M* and causes criterion C_1 to be true.

Remove arguments of format functions (FSRAG)

The FSRAG operator removes arguments of format functions to create FSB vulnerabilities that can be exploited by arbitrary reading or writing of the stack of format functions. The operator is intended to test implementations that generate dynamic format strings. After applying this operator, the number of format specifiers is higher than the number of parameters. Since the *printf* family function calls include explicit argument lists, the number of mutants generated is the total number of arguments supplied. The removal is performed starting from the leftmost argument, each application of the operator removing the next argument in the list. The *vprintf* family function calls do not have an explicit argument list. The arguments are specified through a pointer variable (*va_list*), which is initialized by a macro (*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 either criterion C_1 or C_2.

Table 5.6: Mutation analysis example for the FSRAG operator

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

Table 5.6 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*.

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 *printf* family functions having static format strings, and the specifiers are rotated to the left. For a dynamically generated format string, it cannot be applied because the specifiers aren't know at compile time. For *n* specifiers, the operator generates *n-1* mutants. The generated mutants are killed by test cases that satisfy either C_1 or C_2.

Table 5.7: Mutation analysis example for the FSCFO operator

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

Table 5.7 shows an example application of the operator, where the original program *P* contains a *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 to "*%d %s*". As a result, the mutated program *M* is forced to fetch integer data from *var1* and string data from the address pointed to by *var2*. This might either corrupt the state of *M* or make it crash.

Replace format specifiers with *%n* (FSRSN)

The FSRSN operator replaces each format specifier with *%n* to allow arbitrary writing to the stack of format functions. The *%n* specifier forces the modified program to write the total number of bytes written so far to the address of the corresponding argument considered as integer pointer. In other words, this specifier is used to write outside the format string. The operator is applicable to *printf* family functions having static format strings. The generated mutants are killed by test cases that satisfy either C_1 or C_2 criteria. Table 5.8 shows an example application of the operator. Here, the *printf* format function has one integer format specifier (*%d*) which is replaced with *%n*. In *M*, the value zero is written to the address pointed by *var1*, which is outside the valid address range. As a result, *M* crashes. However, *P* does not crash so the mutant is killed. Thus the test case kills *M* based on the C_1 criterion.

Table 5.8: Mutation analysis example for the FSRSN operator

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

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 to be the total number of supplied arguments plus one to allow an attacker of a mutant program to view the content located immediately after the last explicit argument of the format function. The operator is applicable to *printf* family functions having static format strings. The mutant can be distinguished with test cases that satisfy either C_1 or C_2.

Table 5.9: Mutation analysis example for 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	5	Killed

Table 5.9 shows an example application of the operator. Here, the *printf* format function is supplied with a format specifer *%s* and a corresponding argument *var1* in *P*. The FSPSN operator prepends the string specifier with *%2$s*, which retrieves the argument after *var1* in the stack. Here, the mutant is killed by a test case *'aaa'* that satisfies criterion C_2.

5.1.3 Buffer overflow vs. Format String Operators

Note that the four operators proposed for injecting buffer overflow vulnerabilities in Chapter 3, namely RFSNS, RFSBO, RFSBD, and RFSIFS, also modify the format string similar to the FSCFO, FSRSN, and FSPSN operators described here. The primary differences between these two sets of operators are the following:

1. Operators that inject buffer overflow vulnerabilities modify only the string specifiers inside format strings. However, operators related to FSB vulnerabilities may modify all format specifiers (including the string specifier).

2. The operators that test FSB vulnerabilities force the generation of test cases that crash format function calls because of arbitrary reading and writing to the stack of format functions. However, the operators related to buffer overflow (RFSBO, RFSBD, RFSNS, and RFSIFS) inject vulnerabilities by allowing one or more bytes of overflow. They help in generating test cases that crash programs by overwriting their return addresses due to format function calls.

3. Buffer overflow related operators can be applied to format functions that both scan (*e.g.*, *scanf*, *sscanf*, etc.) and print (*e.g.*, *printf*, *sprintf*, *vsprintf*, etc.). However, FSB related operators are applicable to print related functions only.

5.2 Relationship between FSB related attacks and the operators

Table 5.10 shows the relationship between the attacks that exploit FSB vulnerabilities (already described in Section 2.2.3) and our proposed operators, which all force the generation of test cases that can expose FSB vulnerabilities. The FSIFS, FSRFS, FSCAO, FSRAG, and FSCFO operators generate attack test cases that read from arbitrary memory locations on the stack of format functions. The FSPSP operator tests applications for direct parameter access from the stack of format functions. The FSRSN operator tests arbitrary writing to memory locations of stack of format functions.

Table 5.10: FSB attacks and the proposed operators

Attack	Proposed Operators
Program crash (Denial of Service).	All the proposed operators.
Reading from arbitrary memory address of stack.	FSIFS, FSRFS, FSCAO, FSRAG, FSCFO.
Direct parameter access.	FSPSP.
Writing to arbitrary memory address of stack.	FSRSN.

5.3 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 5.1 shows a snapshot of the tool.

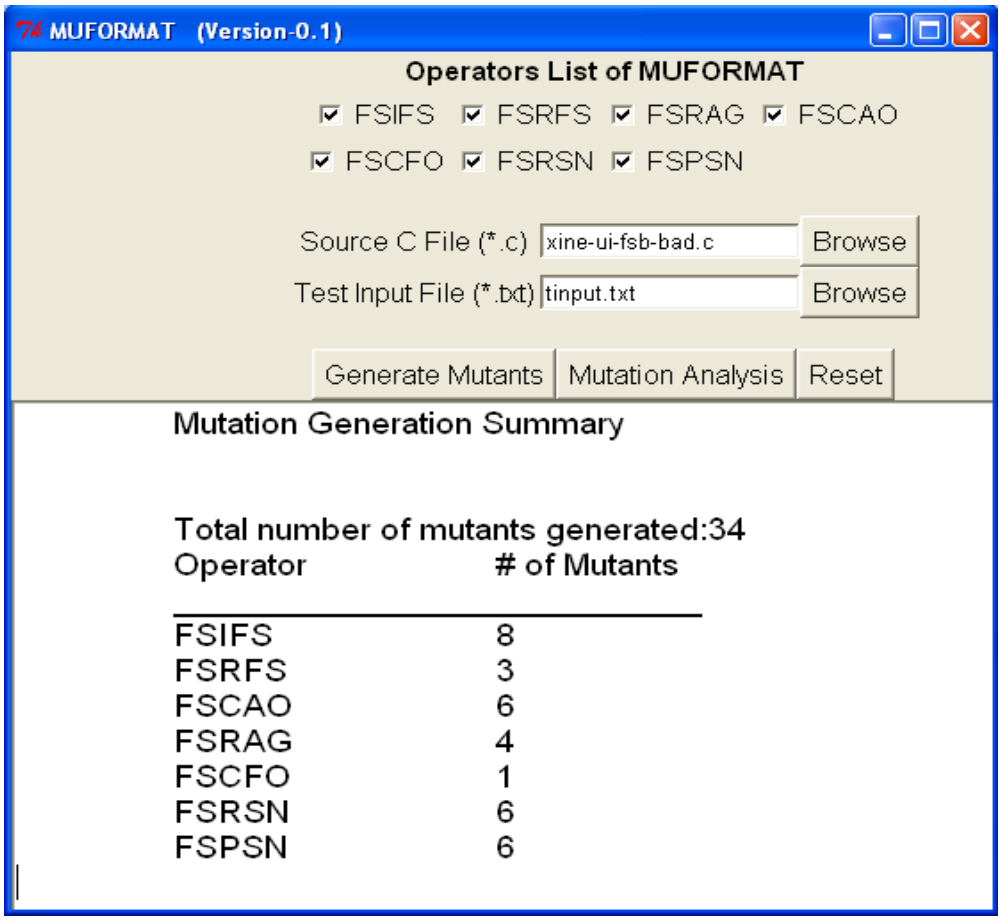

Figure 5.1: Snapshot of MUFORMAT tool

The tool automatically generates mutants for the format string functions of an implementation (*e.g.*, xine-ui-good.c) by clicking on the "Generate Mutants" button. 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 5.1) 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.

5.4 Evaluation of the proposed operators

In this section, we first describe the benchmark applications that are used for evaluating the effectiveness of the proposed operators. We discuss the evaluation methods followed by the results obtained from the experimental analysis.

Benchmark program selection

There is no standard benchmark program for testing FSB vulnerabilities. We choose four open source applications implemented in ANSI C to show the effectiveness of the proposed operators. These applications have been reported to contain FSB vulnerabilities in the Common Vulnerabilities and Exposures (CVE) [12] or the Open Source Vulnerabilities Database (OSVBD) [13]. Table 5.11 shows various characteristics of the four applications. Xine-lib, sSMTP, Qwik-smtpd, and Xine-ui have FSB vulnerabilities in the functions named *_cdda_save_cached_cddb_infos, die, main,* and *print_formatted* respectively. We call the vulnerable functions “bad programs” and their corresponding fixed versions “good programs”. A fixed version is either a patch or an upgraded version. We obtain the patched version for Qwik-smtpd-0.3 and Xine-lib programs. However, for the other two programs, we use the upgraded versions which are Xine-ui-0.99.5 and sSMTP-2.50, respectively.

Table 5.11: Characteristics of four open source programs

Name	Application type	Vulnerability ID	Source file, function name	Lines of code
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

Evaluation of the proposed operators

We follow the evaluation method of Delamaro *et al.* [10], which have been described in Section 3.5 in detail. We describe the construction of the initial test data sets before describing 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 [79]. 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 [58] symbols such as '*s*', '*%*', '*d*', '*n*', and (ii) numeric characters. If a string does not contain any of the format specifiers related characters or numbers, then it is discarded, and a new format string is generated.

Results and analysis

Table 5.12 shows the number of mutants generated for the proposed operators. For each of the bad programs, we obtain an adequate test data set *T* that kills all the generated mutants. 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 5.12: Mutants generated for the four bad programs

Operator	Xine-lib	sSMTP	Qwik-smtpd	Xine-ui
FSIFS	4	3	14	8
FSRFS	3	2	11	3
FSCAO	5	3	21	6
FSRAG	5	3	21	6
FSCFO	1	0	11	0
FSRSN	5	2	21	6
FSPSP	5	2	21	6
Total	28	15	120	35

Table 5.13: Mutation analysis results for testing of FSB vulnerabilities

Name	**Average *MS* (%) of bad programs**	**Average test data set size**	**% of test data sets that reveal FSB vulnerabilities**
Xine-lib	100	16.5	100
sSMTP	100	16.7	100
Qwik-smtpd	100	59.6	100
Xine-ui	100	17.9	100

Table 5.13 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 5.13 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 5.13) Therefore, our proposed operators help in obtaining adequate test data set capable of revealing FSB vulnerabilities.

5.5 Conclusion

This chapter proposes mutation-based testing of format string bug (FSB) vulnerabilities that forces the generation of adequate test data sets. The test adequacy problem has not been addressed in the area of testing FSB vulnerabilities as we have observed in the extensive survey of the related work provided in Chapter 2. By applying our proposed approach, an implementation can be tested for FSB vulnerabilities. The vulnerabilities discovered can be fixed before the actual deployment, and potential losses incurred by end users can be prevented.

We propose seven mutation operators along with two killing criteria for mutants to support the mutation-based testing of FSB vulnerabilities. The operators inject FSB vulnerabilities in the source code and force the generation of test cases to detect FSB vulnerabilities that may lead to various types of exploitation such as program crash, arbitrary reading, writing, and direct parameter access. The operators test FSB vulnerabilities for both the *printf* and *vprintf* family of format functions. Moreover, several proposed operators test FSB vulnerabilities due to format specifer mismatches, which has not been addressed in the traditional approaches for detecting FSB vulnerabilities. A prototype testing tool named MUFORMAT is implemented to automatically generate mutants and perform mutation analysis. The tool produces a list of *live* mutants, which helps the tester generate additional test cases, making a given test data set

adequate for testing FSB vulnerabilities. The operators are found effective for four open source applications having FSB vulnerabilities.

Chapter 6
Conclusion, Limitations, and Future work

6.1 Conclusions

This thesis applies the idea of mutation-based testing to ensure adequate testing of vulnerabilities in software implementations. By applying our proposed approach, many real world exploits can be prevented before a software implementation is deployed. From our extensive survey, we find that traditional techniques that address some of the common vulnerabilities such as buffer overflows, SQL injections, and format string bugs do not focus on the issue of adequate testing of vulnerabilities. We propose mutation operators to support mutation-based testing of these vulnerabilities. Each of the proposed operators injects vulnerabilities in implementations based on commonly used library function calls (*e.g.*, ANSI C libraries, Java library classes for SQL statement manipulation) and implementation language limitations. The mutants generated by the operators can be killed by effective test cases that expose vulnerabilities. We also propose necessary mutant killing criteria that consider symptoms of vulnerability exploitations. We implement three prototype testing tools to automatically generate mutants and perform mutation analysis with input test data: MUBOT, MUSIC, and MUFORMAT. Our proposed operators are evaluated using these tools to modify several open source applications that have been reported to contain vulnerabilities. The results indicate that our mutation-based testing technique is an effective and feasible approach to conduct vulnerability testing.

6.2 Limitations and Future Work

We plan to extend the prototype tool's capability for handling complex forms of program statements. Moreover, the proposed mutation operators in this work do not cover all attacks due to buffer overflow, SQL injection, and format string bug vulnerabilities. We plan to propose additional operators in future in this regard and perform more experiments with large scale applications. Some of the specific limitations of this thesis and future work are elaborated in the following paragraphs.

The equivalent mutant detection procedure is manual. The generation of input test data is semi-automatic. Although we use an automatic random test generation tool for creating initial test data sets, building test cases requires careful attention by the tester. This is due to the fact that a test case must reach the mutated line during mutation analysis. Otherwise, no useful comparisons between a mutant and the original program can be made. Moreover, additional test case generation for killing the *live* mutants is manual in our approach. In terms of effectiveness, mutation-based testing is promising [81]. However, mutation-based testing is more expensive in terms of time and computing than many other testing techniques such as data flow-based testing [81]. We believe that it is up to software stakeholders to decide whether to opt for mutation-based vulnerability testing after considering all the pros and cons.

The four programs used in evaluating the proposed operators for testing buffer overflow (BOF) vulnerabilities were chosen in such a way so that at least one argument is responsible for BOF attacks. However, some BOF vulnerabilities could be more complicated than those included in the chosen programs. Our approach does not address BOFs related to pointers and aliases due to lack of runtime memory information during the mutation generation process. Moreover, we consider the vulnerabilities inherent in ANSI C standard library functions as a fault model, which

might be contradictory to other mutation-based testing works that use language syntaxes. However, these libraries are an inevitable part of many critical applications, and they are extensively used by programmers even though their limitations are known to the software engineering community. Another current issue is that the mutant generation tools are simple. Therefore, before testing, programs need to be preprocessed manually to remove comments and simplify nested function calls (*e.g.*, decomposing "*strcpy (strcat (dest, src), src2);"* into "*strcat (dest, src);"* and "*strcpy (dest, src2);"*). Our future plan also includes testing of other vulnerabilities related to ANSI C memory leaks, double free, and null pointer dereferences.

For SQL injection vulnerability testing, we do not address the SQL injection vulnerabilities (SQLIVs) of stored procedures, as attacks on stored procedures are very rare in practice. The proposed operators are effective for testing SQL queries having simple forms of where conditions. However, we plan to propose operators for complex conditions (*e.g.*, conditions having UNION, INNER JOIN, etc.) in the future. We plan to extend the tool's capability by generating mutants for other popular implementation languages (*e.g.*, PHP), and enhance the operators to address other web-based attacks related to injections (*e.g.*, cross site scripting).

Several mutation operators that we propose for testing format string bug vulnerabilities (*e.g.*, FSRFS, FSRAG) might 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. Moreover, the FSCAO operator generates only one mutant for *vprintf* family functions. This is because ANSI C does not provide any flexibility to reconstruct variable argument lists through the *va_list* data structure. We also

plan to use mutation-based testing to address more complex forms of attack exploiting FSB vulnerabilities such as writing to the destructors section (dtors) or the global offset table (GOT).

References

[1] A. Mathur, *Foundations of Software Testing,* First edition, Pearson Education, 2008.

[2] H. Zhu, P. Hall, and J. May, "Software Unit Test Coverage and Adequacy", *ACM Computing Surveys (CSUR)*, Volume 29, Issue 4, December 1997, pp. 366-427.

[3] R. Hamlet, "Testing Programs with the Aid of a Compiler," *IEEE Transactions on Software Engineering*, Volume 3, Issue 4, July 1977, pp. 279-290.

[4] R. DeMillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer", *IEEE Computer Magazine*, Volume 11, Issue 4, 1978, pp. 34-41.

[5] T. Budd, R. Lipton, R. DeMillo, and F. Sayward, "Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs", In *Proceedings of the Seventh Conference on Principles of Programming Languages*, ACM Press, Las Vegas, January 1980, pp. 220-233.

[6] L. Morell, "A Theory of Fault-based Testing", *IEEE Transactions on Software Engineering*, Volume 16, Issue 8, 1990, pp. 844–857.

[7] J. Andrews, L. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?", In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, ACM Press, May 15-21, 2005, St. Louis, MO, pp.402-411.

[8] J. Andrews, L. Briand, Y. Labiche, and A. Namin, "Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria", *IEEE Transactions on Software Engineering*, Volume 32, Issue 8, Aug. 2006, pp. 608-624.

[9] H. Agrawal, R. DeMillo, R. Hataway, W. Hsu, W. Hsu, E. Krauser, R. Martin, A. Mathur, and E. Spafford, "Design of Mutant Operators for C Programming Language", *Technical Report SERC-TR41-P*, Revision 1.04, Software Engineering Research Center, Purdue University, Indiana, USA.

[10] M. Delamaro, J. Maldonado, and A. Mathur, "Integration Testing Using Interface Mutations", In *Proceedings of the Seventh International Symposium on Software Reliability Engineering*, IEEE Computer Society Press, White Plains, New York, October, 1996, pp. 112-121.

[11] Y. Ma , J. Offutt , and Y. Kwon, "MuJava: An Automated Class Mutation System", *Journal of Software Testing, Verification & Reliability*, Volume 15, Issue 2, June 2005, pp.97-133.

[12] Common Vulnerabilities and Exposures (CVE), http://cve.mitre.org (Accessed in January 2008).

[13] Open Source Vulnerability Database (OSVDB), http://osvdb.org (Accessed in January 2008).

[14] CERT/CC Advisories, http://www.cert.org/advisories (Accessed in January 2008).

[15] Common Weakness Enumeration (CWE), Vulnerabilities Type Distributions in CVE, Version 1.1, http://cwe.mitre.org/documents/vuln-trends/index.html (Accessed in January 2008).

[16] M. Dowd, J. McDonald, and J. Schuh, *The Art of Software Security Assessment*, Addision-Wesley publications, 2007.

[17] L. Gordon, M. Loeb, W. Lucyshyn, and R. Richardson., "Ninth CSI/FBI computer crime and security survey", Technical Report RL32331, C.S.I. Computer Security Institute, 2004. Accessed from www.theiia.org/iia/download.cfm?file=9732 (January 2008).

[18] Source Code Analysis Tools, Open Web Application Source Project (OWASP), Accessed from http://www.owasp.org/index.php/Source_Code_Audit_Tools (January 2008).

[19] ITS4: Software Security Tool, Accessed from http://www.cigital.com/its4 (November 2007).

[20] FlawFinder, Accessed from http://www.dwheeler.com/flawfinder (November 2007).

[21] D. Evans and D. Larochelle, "Improving Security Using Extensible Lightweight Static Analysis", *IEEE Software*, Volume 19, Issue 1, 2002, pp. 42-51.

[22] Z. Su and G. Wasserman, "The Essence of Command Injection Attacks in Web Applications", In *Proceedings of Symposium on Principles of Programming Languages POPL'06*, January 2006, South Carolina, USA, pp. 372-382.

[23]. U. Shankar, K. Talwar, J. Foster, and D. Wagner, "Detecting Format String Vulnerabilities with Type Qualifiers", In *Proceedings of the 10th USENIX Security Symposium*, August 2001, Washington, D.C., pp. 201-218.

[24]. K. Chen and D. Wagner, "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.

[25] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks", In *Proceedings of the 7th USENIX Security Conference*, San Antonio, Texas, January 1998, pp. 63-78.

[26] R. Hastings and B. Joyce, “Purify: Fast Detection of Memory Leaks and Access Errors”, In *Proceedings of USENIX Winter Conference*, San Francisco, January 1992, pp. 125-138.

[27] G. Buehrer, B. W. Weide, and P. Sivilotti, "Using Parse Tree Validation to Prevent SQL Injection Attacks", In *Proceedings of the 5th International Workshop on Software Engineering and Middleware (SEM '05)*, Lisbon, Portugal, September 2005, pp.106-113.

[28] W. Li and T. Chiueh, “Automated Format String Attack Prevention for Win32/X86 Binaries”, In *Proceedings of 23rd Annual Computer Security Applications Conference (ACSAC)*, Miami, Florida, December 2007, pp. 398-409.

[29] T. Robbins. Libformat, Accessed from http://archives.neohapsis.com/archives/linux/lsap/2000-q3/0444.html (January 2008).

[30] T. Tsai and N. Singh, “Libsafe 2.0: Detection of format string vulnerability exploits”, Technical report, Avaya Labs, February 2001.

[31] A. DeKok, “Pscan: Format String Security Checker for C Files”, http://packages.debian.org/etch/pscan (Accessed Jan 2008).

[32] C. Cowan, M. Barringer, S. Beattie, G. Hartman, M. Frantzen, and J. Lokier, “FormatGuard: Automatic Protection From printf Format String Vulnerabilities”, In *Proceedings of the 10th USENIX Security Symposium*, August 2001, Washington, D.C., pp. 191- 200.

[33] 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.

[34] W. Halfond and A. Orso, "AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks", In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, November 2005, Long Beach, CA, USA, pp.174-183.

[35] M. Muthuprasanna, K. Wei, and S. Kothari, "Eliminating SQL Injection Attacks - A Transparent Defense Mechanism", In *Proceedings of the Eighth IEEE International Symposium on Web Site Evolution (WSE'06)*, Philadelphia, September 2006, pp. 22-32.

[36] M. Uddin, H. Shahriar, and M. Zulkernine, "ACIR: An Aspect-Connector for Intrusion Response", In the *Proceedings of the First International Workshop on Security in Software Engineering (IWSSE 2007)*, IEEE CS Press, Beijing, China, July 2007, pp. 249-254.

[37] G. Kapfhammer and M. Soffa, "A Family of Test Adequacy Criteria for Database-Driven Applications*"*, *SIGSOFT Software Engineering Notes*, Volume 28, Number 5, 2003, ACM, New York, NY, USA, pp. 98-107.

[38] F. Nagano, K. Tatara, K. Sakuri, and T. Tabata, "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.

[39] Y. Shin, L. Williams, and T. Xie, "SQLUnitGen: SQL Injection Testing Using Static and Dynamic Analysis", In supplemental *Proceedings of the 17th IEEE International Conference on Software Reliability Engineering (ISSRE 2006),* November 2006, Raleigh, NC, USA, ISBN 978-0-9671473-3-3-8.

[40] Y. Kosuga, K. Kono, M. Hanaoka, M. Hishiyama, Y. Takahama, "Sania: Syntactic and Semantic Analysis for Automated Testing against SQL Injection", In *Proceedings of the 23rd Annual Computer Security Applications Conference, 2007 (ACSAC 2007)*, Miami, December 2007, pp. 107-117.

[41] W. Du and A. Mathur, "Testing for software vulnerabilities using environment perturbation", *International conference on Dependable Systems and Networks (DSN 2000)*, New York, NY, June 2000, pp. 603-612.

[42] A. Ghosh, T. O'Connor, and G. McGraw, "An automated approach for identifying potential vulnerabilities in software", In *Proceedings of IEEE Symposium on Security and Privacy*, Los Alamitos, CA, USA, May 1998, pp. 104-114.

[43] O. Tal, S. Knight, and T. Dean, "Syntax-based Vulnerabilities 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.

[44] M. Ellims, D. Ince, M. Petre, "The Csaw C Mutation Tool: Initial Results", In *Proceedings of the Third Workshop on Mutation Analysis (Mutation 2007)*, Windsor, UK, September 2007, pp. 185-192.

[45] M. Delamaro, J. Maldonado, "Proteum - A Tool for the Assessment of Test Adequacy for C Programs", In *Proceedings of the Conference on Performability in Computing Systems (PCS 96)*, July 1996, pp. 79-95.

[46] B. Marick, A tutorial introduction to GCT, http://www.testingcraft.com/gct-tutorial.pdf (Accessed April 2007).

[47] W. Chan, S. Cheung, and T. Tse, "Fault-based Testing of Database Application Programs with Conceptual Data Model", In *Proceedings of the Fifth International Conference on Quality Software (QSIC 2005)*, Los Alamitos, California, September 2005, pp. 187-196.

[48] J. Tuya, M. Suárez-Cabal, and C. Riva, "Mutating Database Queries", *Journal of Information and Software Technology*, Volume 49, Issue 4, April 2007, pp. 398-417.

[49] H. Shahriar and M. Zulkernine, "Mutation-based Testing of Buffer Overflow Vulnerabilities", To appear in the *Proceedings of the Second International Workshop on Security in Software Engineering (IWSSE 2008)*, IEEE CS Press, Turku, Finland, July 2008, pp. 979-984.

[50] H. Shahriar and M. Zulkernine, "MUSIC: Mutation-based SQL Injection Vulnerability Checking", To appear in the *Proceedings of the Eighth International Conference on Quality Software (QSIC 2008)*, IEEE CS Press, London, August 2008. (10 pages)

[51] H. Shahriar and M. Zulkernine, "Mutation-based Testing of Format String Bugs", Submitted for publication, July 2008.

[52] W. Howden, "Weak Mutation Testing and Completeness of Test Sets", *IEEE Transaction on Software Engineering*, Volume 8, Number 4, July 1982, pp. 371-379.

[53] M. Woodward and K. Halewood, "From Weak to Strong: Dead or Alive? An Analysis of Some Mutation Testing Issues", In *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, Banff, Alberta, Canada, July 1988, pp. 152-158.

[54] Aleph One, "Smashing the Stack for Fun and Profit", *Phrack Magazine,* Volume 7, Issue 49, November 1996, Accessed from http://www.phrack.org/archives/49/P49-14.

[55] Y. Younan, F. Piessens, and W. Joosen, "Protecting Global and Static Variables from Buffer Overflow Attacks without Overhead", Report CW463, Department of Computer

Science, Katholieke Universiteit Leuven, Belgium, October 2006, Accessed from http://www.fort-knox.be/files/CW463.pdf.

[56] M. Kaempf, “Smashing the Heap for Fun and Profit”, Accessed from http://doc.bughunter.net/buffer-overflow/heap-corruption.html, 2002 (September 2007).

[57] CVE-2006-3747, Common Vulnerabilities and Exposures, Accessed from http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3747 (September 2007).

[58] C Standard Library in http://www.utas.edu.au/infosys/info/documentation/C/CStdLib.html (Accessed in January 2008)

[59] International Standards Organization, Information technology – Database languages – SQL, ISO/IEC 9075:1992, Third edition.

[60] MySQL 5.0 Reference Manual, Accessed from http://dev.mysql.com/doc/refman/5.0/en/tutorial.html (January 2008).

[61] W. G. Halfond, J. Viegas, and A. Orso, “A Classification of SQL-Injection Attacks and Countermeasures”, In *Proceedings of IEEE International Symposium on Secure Software Engineering (ISSSE 2006)*, Arlington, Virginia, March 2006.

[62] SQL Injection Walkthrough, Accessed from http://www.securiteam.com/securityreviews/5DP0N1P76E.html (February 2008).

[63] O. Maor and A. Shulman, “Blindfolded SQL Injection”, iMPERVA, 2002, http://www.imperva.com/resources/ adc/blind_sql_server_injection.html (Accessed February 2008).

[64] Scut/team teso, "Exploiting Format String Vulnerabilities", 2001, Accessed from http://doc.bughunter.net/format-string/exploit-fs.html (January 2008).

[65] A. Silva, “Format Strings”, Gotfault Security Community, Version 2.5, November 2005, Accessed from http://www.milw0rm.com/papers/5 (April 2008).

[66] K. Lhee and S. Chapin, “Buffer Overflow and Format String Overflow Vulnerabilities”, *Journal of Software-Practice and Experience*, Volume 33, Issue 5, 2003, pp. 423-460.

[67] P. Vilela, M. Machado, and E. Wong, “Testing for Security Vulnerabilities in Software”, *Proceeding Software Engineering and Applications (SEA 2002)*, Cambridge, USA, November 2002.

[68] W. Allen, D. Chin, and G. Marin, “A Model-based Approach to the Security Testing of Network Protocol Implementations”, In *Proceedings of the 31st IEEE Conference on Local Computer Networks*, November 2006, pp. 1008 – 1015.

[69] T. Mouelhi, Y. Le Traon, and B. Baudry, “Testing security policies: going beyond functional testing”, In *Proceedings of International Symposium on Software Reliability Engineering (ISSRE'07),* Trollhättan, Sweden, November 2007, pp. 93-102.

[70] S. Boyd and A. Keromytis, “SQLrand: Preventing SQL injection attacks”, In *Proceedings of the Second Applied Cryptography and Network Security Conference (ACNS 2004)*, Volume 3089 of Lecture Notes in Computer Science, Springer-Verlag, 2004, pp. 292-304.

[71] S. Thomas and L. Williams, “Using Automated Fix Generation to Secure SQL Statements”, In *Proceedings of the Third International Workshop on Software Engineering for Secure Systems (SESS’07)*, Minneapolis, MN, USA, May 2007, pp. 9-14.

[72] S. Bandhakavi, P. Bisth, P. Madhusudan, and V. Venkatakrishnan, “CANDID: Preventing SQL Injection Attacks Using Dynamic Candidate Evaluations”, In

Proceedings of the 14th ACM conference on Computer and communications security (CCS'07), Alexandria, Virginia, October 2007, pp. 12-24.

[73] J. Lin and J. Chen, "The Automatic Defense Mechanism for Malicious Injection Attack", In *Proceedings of the Seventh International Conference on Computer and Information Technology(CIT2007)*, Fukushima, Japan, October 2007, pp 709-714.

[74] American National Standard for Information Systems Programming Language C, *Technical Report ANSI X3.159-1989*, ANSI Inc., New York, USA 1990.

[75] M. Zister, *Securing Software: An Evaluation of Static Source code Analyzers*, MSc Thesis, Massachussets Institute of Technology, Cambridge, MA, USA, August 2003.

[76] Kendra Kratkiewicz, *Evaluating Static Analysis Tools for Detecting Buffer Overflows in C Code*, MSc Thesis, Harvard University, March 2005, Accessed from: www.ll.mit.edu/IST/pubs/KratkiewiczThesis.pdf (November 2007).

[77] T. Newsham and B. Chess, "ABM: A Prototype for Benchmarking Source Code Analyzers", In *Proceedings of the Workshop on Software Security Assurance Tools, Techniques, and Metrics (SSATTM '05)*, Long Beach, California, 2005.

[78] K. Ku, T. Hart , M. Chechik, and D. Lie, "A Buffer Overflow Benchmark for Software Model Checkers", In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering Automated Software Engineering (ASE 2007)*, Atlanta, Georgia, USA, November 2007, pp. 389-392.

[79] B. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities", *Communications of the ACM,* Volume 33, Number 12, December 1990, pp. 32-44.

[80] Open Source Web Applications with Source Code in ASP, JSP, PHP, Perl, ColdFusion, ASP.NET/C#, http://gotocode.com (Accessed February 2008).

[81] E. Wong and A. Mathur, "Fault Detection Effectiveness of Mutation and Data flow Testing", *Software Quality Journal*, Volume 4, Number 1, March 1995, pp. 69-83.