Mitigating and Monitoring Program Security Vulnerabilities[1]

Technical Report No. 2010-572

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

---

[1] This report should be cited as follows:

H. Shahriar and M. Zulkernine, Mitigating and Monitoring Program Security Vulnerabilities, Technical Report No. 2010-572, School of Computing, Queen's University, Kingston, Canada, June 2010.

ABSTRACT

Today's programs are implemented in a variety of languages and contain serious vulnerabilities which can be exploited to cause security breaches. These vulnerabilities have been exploited in real life and resulted in damages to related stakeholders such as program users. As most vulnerabilities belong to program code, many techniques have been applied to mitigate vulnerabilities before and after program deployment. Unfortunately, there is no comprehensive comparative analysis of different vulnerability mitigation works. As a result, there exists an obscure mapping between the techniques, the addressed vulnerabilities, and the limitations of different approaches. This paper attempts to address these issues. The paper extensively compares and contrasts the existing program security vulnerability mitigation (testing, static analysis, and hybrid analysis) and monitoring techniques. We also discuss other techniques employed to mitigate the most common program security vulnerabilities: secure programming, patching, and program transformation. The survey provides a comprehensive understanding of the current program vulnerability mitigation approaches and challenges as well as their key characteristics and limitations. Moreover, our discussion highlights the open issues and future research directions in the area of program security vulnerability mitigation and monitoring.

Table of Contents

List of Tables

List of Figures

1. Introduction

Today's programs are complex and usually accessible to almost all users. These programs are implemented in a wide variety of languages and run on different execution environments. Programs are developed and tested using a rich set of tools and techniques before actual deployment to ensure that they meet specific requirements in terms of functionality, quality, and performance. Nevertheless, these programs contain *vulnerabilities*[2] that might be exploited intentionally or unintentionally to cause security breaches. Vulnerabilities are flaws in programs that allow attackers to expose, alter, disrupt, or destroy sensitive information [92]. Approximately 50% of all security bugs (or vulnerabilities) occur at program code level [39]. These vulnerabilities are exploited by attackers. Program vulnerabilities (*i.e.*, vulnerabilities arise due to implementation in code) have been addressed in academia and industry for more than twenty years. Still, we observe different security breach (or vulnerability) reports through many publicly available repositories such as Open Source Vulnerability Database (OSVDB) [97] and Common Vulnerabilities and Exposures (CVE) [95]. A number of surveys report significant financial losses by individuals and organizations due to *attacks* exploiting vulnerabilities (*e.g.*, [93]). Therefore, mitigating program security vulnerabilities is extremely important.

If we look at the literature, we notice that many mitigation techniques are applied to the program code before and after their deployment. These techniques are being evolved with novel attack techniques along with program usage (*e.g.*, standalone program vs. client server-based program), implementation languages (*e.g.*, procedural, object oriented, scripting), and processors (*e.g.*, browsers, database engines). Unfortunately, there is no effort to review these techniques in a comparative way. In the past, several empirical studies have attempted to compare tools and techniques for mitigating program security vulnerabilities [17, 29, 39, 56, 76, 80]. However, these studies focus on analyzing works for a single (*e.g.*, buffer overflow) or few vulnerabilities or comparing approaches related to one particular mitigation technique (*e.g.*, static analysis). As a result, we have an obscure mapping between the techniques, the addressed vulnerabilities, and the limitations of different approaches. In this paper, we survey program ("code-level") security vulnerability mitigation approaches to address these issues. We perform a comparative analysis of program security vulnerability mitigation approaches with respect to the three most common mitigation (*testing*, *static analysis*, *hybrid analysis*) and *monitoring* techniques. Moreover, for the sake of completeness, we briefly discuss the *secure programming* guideline related works and the approaches employed in the maintenance stage (*program transformation* and *patching*). We compare and contrast the mitigation approaches based on the features that are commonly discussed in related works

[2] The *italic* letters are used when we define terminologies, introduce any of our proposed classification feature name, and explain program code examples.

(*e.g.*, (31-38]). We also identify open issues for each of the techniques. Our analysis indicates that program vulnerability solutions have been influenced by not only traditional program analysis techniques, but also the diversity of attack mechanisms. Moreover, current approaches have their own limitations with respect to vulnerability coverage, programming languages, and techniques.

The paper is organized as follows. Section 2 provides an overview of the eight vulnerabilities that are most common in today's programs. We analyze the program security vulnerability mitigation related works based on testing, static analysis, and hybrid analysis in Sections 3, 4, and 5, respectively. Section 6 discusses the efforts on secure programming, program transformation, and patching techniques. Section 7 analyzes the efforts on monitoring approaches. Finally, Section 8 draws some conclusions.

2. Program security vulnerabilities

Program security vulnerabilities are specific flaws in program code that result in security breaches such as sensitive information leakage, modification, and destruction. *Attacks* are successful exploitations of vulnerabilities. There are many program security vulnerabilities which can be exploited by attackers. However, we restrict our discussion with respect to eight vulnerabilities based on the taxonomy of Common Weakness Enumeration (CWE) [170]. They are buffer overflow (BOF), format string bug (FSB), SQL injection (SQLI), cross site scripting (XSS), cross site request forgery (CSRF), NULL pointer dereference (NLD), dangling pointer (DAP), and memory leak (MEL). These are the most widely reported and discovered vulnerabilities in program code. Note that several vulnerabilities can be classified as a single source of the same problem. For example, SQLI, XSS, and CSRF can be classified as "insecure interaction between components" based on a high level taxonomy of CWE. Nevertheless, the wide differences in exploitation mechanisms, severity of damages, and existing mitigation efforts have inspired us to discuss these vulnerabilities individually. Moreover, we believe that mitigating these vulnerabilities can stop the exploitations of other vulnerabilities in programs such as buffer underflow [98].

2.1. Buffer overflow (BOF)

A buffer overflow (BOF) vulnerability allows writing data to a program buffer exceeding the allocated size and overwriting the content of the neighboring memory locations [154]. BOF might be present in programs having unsafe library function calls (*e.g.*, ANSI C standard library), lack of null characters at the end of buffers, buffer accesses through pointers and aliases, logic errors (off by one), and insufficient checks before accessing buffers. One of the most subtle BOF vulnerabilities might be present in programs that perform pointer-intensive operations and generate pointer addresses through pointer arithmetic. For example, $p = p + 4$ results in a pointer p to point to a new location four bytes apart from the current location. If a read or write operation is performed through a pointer dereference (*i.e.*, $*p$) and the new memory location does not belong to valid memory regions, a program shows many unexpected behaviors.

Exploitation of BOF depends on memory regions where buffers are located. These include stack, heap, block started by symbol (also known as bss), and data segments. Here, bss and data segments contain static or global data buffers that are uninitialized and initialized, respectively. We provide an example of C code snippet (the function *foo*) in Figure 1 which is vulnerable to BOF. The buffer declared at Line 3 (*buf*) is located in the stack region and has 16 bytes of memories for reading and writing operations. The valid location of this buffer is between *buf[0]* and *buf[15]*. Line 4-5 copies *src* buffer into *buf* using a for loop. However, the code is vulnerable as there is no checking on destination buffer length. As a result, the loop allows copying more than the capacity of *buf*. To understand the effect of an overflow, we consider a snapshot of the stack frame of the function *foo* (Figure 2). The stack stores the argument of *foo* (*src*), the return address (*ret*), the saved frame pointer (*sfp*), and declared variables (*i* and *buf*). Note that the direction of stack growth and buffer copy is opposite to each other. We assume that both *ret* and *sfp* occupy four bytes, whereas *i* occupies two bytes.

```
1.  void foo (char *src) {
2.      int i;
3.      char buf [16];
4.      for (i=0; i<strlen(src); i++)
5.          buf[i] = src[i];
6.      return;
7.  }
```

Figure 1: C code snippet of *foo* function vulnerable to buffer overflow

```
Buffer copy direction ----->
buf [0] …                …  buf [15]   i      sfp    ret     src
[                                    ][     ][     ][     ][       ]
<----- Stack growth direction
```

Figure 2: Stack layout of *foo* function

Storing an input of 17 bytes in length (pointed by *src*) in the *buf* results in one byte overflow. This corrupts the neighboring variable *i* and leads to further unexpected behaviors by the program (assuming no padding performed by a compiler). However, an attacker can corrupt the return address to execute injected code through buffer. This is known as "return address clobbering" or "direct code injection" attack [164]. An attacker inject a payload which can modify the content of *ret* to point to the beginning address of the buffer (*i.e.*, the address of *buf[0]*). In this case, the *src* buffer content must be at least 26 bytes long (16 bytes for the *buf*, two bytes for *i*, four bytes for both *sfp* and *ret*). The injected code might contain a shell code (*e.g.*, "/bin/sh") to launch a remote shell with the root privilege. Note that several conditions need to be met to perform a successful attack such as allowing code execution from stack segment, presence of no null character in the injected payload, and allowing an input of sufficient length to reach the location of *ret*.

A program might not allow copying an input to a buffer which can modify the return address. In this case, an attacker might execute arbitrary code by overwriting the frame pointer (*i.e.*, *sfp*) [155]. The *sfp*

stores the address of stack top of the caller function of *foo*. An attacker modifies the *sfp* of *foo* to point to a location of the buffer. As a result, the first four bytes of the buffer is considered as the stack frame pointer and the next four bytes is considered the return address for the caller of *foo*. The content of the new return address points to the location of injected code. The *foo* function returns to its caller as usual. However, when the caller returns, the injected code is executed.

Many existing countermeasures prevent code injections due to BOF attacks by converting stack segments from executable to non-executable (*e.g.*, [165]). However, attackers still bypass such defenses. For example, a stack overflow modifies return addresses with known system library function call addresses. In this case, the buffer contains the arguments for the function calls. This variation of attack is known as "return-to-libc" or "jump-to-libc" [164]. Note that both saved frame pointer overwriting and return-to-libc are known as "indirect code injection" attacks [164].

BOF attacks can be performed by overflowing buffers located in the heap memory region. A typical attack overflows a neighboring function pointer which stores the address of a function. An attacker might modify the function address with the location of his injected code which results in executing injected code when the function is called. This is a basic form of heap-based BOF attack. Sophisticated heap-based attacks do not even rely on the presence of function pointers in program code. These attacks can indirectly execute injected code by leveraging the known working mechanism of *malloc* and *free* functions as well as memory management information (or meta information) stored in the beginning of allocated and free memory blocks (or chunks) [155]. This information can be obtained from publicly available specifications of memory managers (*e.g.*, dlmalloc [166]).

2.2. Format string bug (FSB)

Format string bug (FSB) vulnerabilities imply invoking format functions (*e.g.*, the format functions of ANSI C standard library [158]) with user supplied format strings that contain arbitrary format specifiers (*e.g.*, %s). As a result, the number of specifiers becomes more than the number of arguments, which allows arbitrary reading and writing in format function stack. For example, a simple *printf("%d", i)* function call prints the value of *i* to the console, where *i* is an integer variable. However, the *printf("%d")* function call results in printing an arbitrary integer value. Moreover, a mismatch between a format specifier and its corresponding argument might result in unexpected behaviors. For example, the function call *printf("%s", i)* writes a string to the console where the string location is considered as the value of *i*. If attack cases are crafted carefully, it is possible to perform malicious activities such as establishing root shells and overwriting global offset tables (GOT) that contain function addresses [136].

2.3. SQL injection (SQLI)

SQLI vulnerabilities are present in programs which generate SQL queries with invalidated user supplied inputs. The inputs might contain arbitrary SQL queries which alter intended queries. These vulnerabilities can be exploited through SQL injection attacks that cause unexpected results such as authentication bypassing and information leakage. We provide an example of an SQLI attack by using the code snippet of a server side program written in JSP as shown in Figure 3. 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). Therefore, an attacker avoids the authentication by executing this query. There are several common SQLI attack types such as tautologies, union queries, illegal/logical incorrect queries, piggybacked queries, stored procedures, inference attacks, and alternate encodings (or Hex encoded queries) [146].

```
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 3: A JSP code snippet for authentication

2.4. Cross site scripting (XSS)

XSS vulnerabilities allow the generation of dynamic Hyper Text Markup Language (HTML) [130] contents (*i.e.*, attributes of tags) with invalidated inputs. These inputs contain HTML tags and JavaScript code that are interpreted by browsers while rendering web pages. As a result, the intended behavior of generated web pages alters through visible (*e.g.*, creation of pop-up windows) and invisible (*e.g.*, cookie

bypassing) symptoms. XSS attacks circumvent traditional security mechanisms employed by browsers such as same origin policy, sandbox, and signed script.

```
<?
  1. $msg = retrieveComment();
  2. echo ('Comment:' . $msg);
?>
```

Figure 4: A PHP code snipped vulnerable to XSS

There are three types of XSS attacks: stored, reflected, and Document Object Model-based (or DOM-based) [121, 128]. In stored XSS attacks, dynamic HTML contents are generated from unsanitized information that is stored in persistent data storages (*e.g.*, files, databases). A reflected XSS attack occurs, if injected script code (*i.e.*, *<script>alert('xss');</script>)* returns to a browser and gets executed (*e.g.*, a search string supplied in a webpage). JavaScript code that process inputs based on DOM objects [119] (*e.g.*, *document.URL*) are vulnerable to attacks which are denoted as DOM-based XSS attacks [121]. We provide an example PHP code snippet in Figure 4 that is vulnerable to stored XSS attacks. Line 1 retrieves a comment from a persistent storage (*i.e.*, *retrieveComment()*) and saves it to a PHP variable *$msg*. Line 2 writes the comment as HTML output without filtering. If the *$msg* variable contains script code (*e.g.*, *<script>alert('xss')</alert>*) and is sent to a browser without filtering, a user observes an unexpected dialog box with the text "*xss*".

2.5. Cross site request forgery (CSRF)

A CSRF vulnerability occurs, if an HTTP request is sent to a remote server program without the client's knowledge [161]. A CSRF vulnerability may arise in a program, if an input form submission requires no validation, except for a cookie. Cookies are stored in browsers for a long time and added automatically while issuing a request. Moreover, browsers add cookies automatically to HTTP GET requests when loading images and frames, submitting forms, clicking links, or redirecting pages. Thus, a server program cannot differentiate between an HTTP request generated by a legitimate user and a CSRF attack.

There are two types of CSRF attack: reflected and stored [167]. In a reflected CSRF vulnerability, the injected payload resides in a program other than a trusted server program. Thus, a victim is exposed to an attack when he/she logs on to a server program and browse to a different website (or program) simultaneously. In a stored CSRF vulnerability, the malicious code is stored within the trusted server program repositories.

We provide two example code snippets (client and server side) that are vulnerable to CSRF attacks in Table 1. Let us assume that a user is logged on to a site (*www.xyz.com*) that stores his/her profile. The profile includes a contact email address which has the initial value *user@xyz.com*. The client side (the left column of Table 1) provides an HTML interface (*change.html*) to change the email address of a logged on

user legitimately. A new email address provided by a user at Line 5 is updated by the server side script named *editprofile.php* (*i.e.*, the action field of Line 3). Note that the request of the email address change is sent to *editprofile.php* by a hidden field (*action*) and corresponding value (*setemail*) at Line 4. The server side code snippet is shown in the second column of Table 1. It first checks if the request is associated with a valid session (Line 1). If the session is not valid, then the program shows an error message and terminates (Line 2-3). Otherwise, a session is identified as valid (Line 5) and the request is performed by calling the *update_profile* function at Line 6. The function is supplied with the new email address argument (*$POST['email']*).

Table 1: Example program code snippets of an email address change

Client side code (www.xyz.com/change.html)	Server side code (www.xyz.com/editprofile.php)

1. <HTML>
2. <BODY>
3. <FORM action = "editprofile.php" method = "POST">
4. <INPUT type = "hidden" name = "action" value = "setemail">
5. <INPUT type = "text" name = "email" value = "">
6. <INPUT type = "submit" value = "Change Email Address">
7. </FORM>
8. <BODY>
9. </HTML>
	1. if (! session_is_registered($_SESSION['username'])) {
	2. echo "invalid session detected!";
	3. exit;
	4. }
	5. if ($_POST['action'] == 'setemail'){
	6. update_profile($_POST['email']);
	7. }

If a user supplies the new email address as *user2@xyz.com*, the legitimate HTTP request becomes *http://www.xyz.com/editprofile?action=setemail&email= user2@xyz.com*. The browser adds the session information (or cookie) in the request before sending to the server program. Let us assume that the user is logged on to *www.xyz.com* and visiting another website which contains an attacker supplied hyperlink *http://www.xyz.com/editprofile?action=setemail&email=evil@xyz.com*. If the user clicks on the link, then his email address gets changed to *evil@xyz.com*. As a result, the user is a victim of a reflected CSRF attack. In practice, CSRF attacks not only modify profile, but also perform other severe damages such as unauthorized financial transactions and sensitive information deletions. To become a victim of a stored CSRF, the malicious link would be stored in the persistent storage of the website where the user is logged in. It is common to find many such web-based programs such as message boards.

2.6. Other vulnerabilities

Three other vulnerabilities are common in programs namely null pointer dereference (NLD), dangling pointer (DAP), and memory leak (MEL). A NLD vulnerability occurs when a program code retrieves memory contents from a pointer type variable whose value is assigned to *NULL* [159]. A NULL address does not indicate any memory location. As a result, any operation with a NULL pointer results in unexpected behaviors such as program crashes. A DAP (also known as a *wild pointer*) points to a memory location that is no longer a valid memory location for a program. For example, a variable might point to an allocated memory object which might be freed explicitly. The vulnerability occurs, if a program's code

tries to access or store information in these free objects through pointer variables. A memory leak (MEL) is a widespread vulnerability in programs where allocated memories are not freed explicitly or unused objects are not set as NULLs [162]. Unlike other vulnerabilities (*e.g.*, BOF, FSB, SQLI, XSS, and CSRF), it is difficult to exploit NLD, DAP, and MEL directly to execute arbitrary code. Nevertheless, the presence of these vulnerabilities might result in unexpected behaviors such as program crashes (*e.g.*, NLD, DAP) and wastage of memories (MEL) which can eventually lead to an abnormal termination of server programs (or daemons). Modern compilers often generate warning messages so that programmers can fix them while implementing programs. However, these three vulnerabilities are discovered in post release stages.

2.7. Summary

We have discussed eight worst vulnerabilities that can be found in programs which are implemented in a variety of languages. A mapping between programming languages and vulnerabilities is shown in Table 2. The first seven languages are high level, whereas the last two (x86 assembly code and byte code) are intermediate code generated by compilers. From the table, we notice that programs implemented in any of these languages contain a large subset of vulnerabilities (denoted as "Y" in table cells). Note that SQLI, XSS, and CSRF are denoted as *web-based vulnerabilities*. Moreover, BOF, SQLI, and XSS are known as *injection vulnerabilities* as attacks might inject arbitrary code. Programs implemented in C and C++ are not vulnerable to web-based vulnerabilities (*e.g.*, SQLI, XSS, and CSRF) as they are rarely used in implementing web-based programs. Rather, Java, PHP, JSP, and ASP are used to implement web-based programs. JavaScript programs are executed by client side browsers. Due to implementation limitations of browsers, JavaScript code might suffer from XSS and MEL vulnerabilities. Moreover, JavaScript programs contribute to web-based vulnerabilities due to improper validation of inputs. The cells containing "N" indicate that no vulnerability has been identified for the corresponding languages. Intermediate programs generated from source code may inherit vulnerabilities. For example, if a C program contains a BOF vulnerability, the corresponding machine dependant *x86* assembly code also contains the same vulnerability.

Table 2: A mapping between programming languages and vulnerabilities

Language	BOF	FSB	SQLI	XSS	CSRF	NLD	DAP	MEL
C	Y	Y	N	N	N	Y	Y	Y
C++	Y	Y	N	N	N	Y	Y	Y
Java	N	N	Y	Y	Y	Y	N	Y
JSP	N	N	Y	Y	Y	Y	N	Y
PHP	N	N	Y	Y	Y	Y	N	Y
ASP	N	N	Y	Y	Y	Y	N	Y
JavaScript	N	N	Y	Y	Y	Y	N	Y
x86 assembly code	Y	Y	N	N	N	Y	Y	Y
Byte code	N	N	Y	Y	Y	Y	N	Y

Some vulnerabilities (*e.g.*, BOF, FSB, SQLI, NLD) result in observable abnormal behaviors such as segmentation faults and error messages. However, several vulnerabilities (*e.g.*, CSRF, MEL) are difficult to identify through observable behaviors. We discuss the mitigation techniques employed for the vulnerabilities in Sections 3-7. Allocated memory blocks and *memory objects* are used synonymously in this paper.

3. Testing

Testing is one of the most proactive program security vulnerability mitigation techniques before releasing programs. In general, a program under test is provided inputs, executed, and computed outputs are matched with expected outputs. If there is a mismatch between the computed and the expected outputs, then the program implementation does not comply with a desired requirement for a particular input. Testing requirements might be expressed in terms of functionality, performance, and quality. The pioneer works in program testing have focused on developing test case generation based on program code coverage (*e.g.*, branch, loop, data flow, condition), program behavior model coverage (*e.g.*, state and transitions of a finite state machine), and common mistakes performed by programmers (*e.g.*, fault-based testing and mutation analysis) [84]. However, as security of programs is getting an increasing level of attention in recent years, we notice a lot of program security vulnerability testing techniques have emerged. McGraw *et al.* [49] have described program security vulnerability testing as a misunderstood task which is often considered as developing secured firewall rules and port scanning in networks. Program security can be tested based on risk assessment results, requirements, and design errors. However, we restrict our discussion to works that intend to reveal vulnerabilities. These might cause due to the limitations of programming languages, libraries and APIs, environments, and logic errors.

We first show an analogy between software testing and program security vulnerability testing in Section 3.1. In Section 3.2, we compare and contrast the related vulnerability testing works based on the following five features: *test case generation method*, *source of test case*, *test level*, *test case granularity*, and *vulnerability coverage*. Out of these five features, test case generation is the most important steps in security vulnerability testing [9]. Thus, we are motivated to classify existing program security testing approaches based on the test case generation techniques in Sections 3.3-3.9. These include *fault injection*, *attack signature*, *mutation analysis*, *static analysis*, *search*, *program modification*, and *constraint bypassing* techniques. Finally, we discuss open issues in Section 3.10.

3.1. Software testing vs. program security vulnerability testing

Like traditional software testing, we consider program security vulnerability testing as a process of three major steps: identifying testing requirements and coverage, generating test cases, and executing test cases [6]. In the first step, appropriate security requirements are identified based on functional

requirements. Our study considers the requirement in terms of security breaches that occur through the implementation languages (*e.g.*, ANSI C), APIs (ANSI C library, Java library), environment variables (network data unit used in programs), processors (*e.g.*, SQL database engine, HTML parsers, JavaScript interpreter), and malformed inputs used by programs. In traditional testing, test coverage implies whether generated test cases can cover a particular objective related to a program artifact. For example, a program can be tested in a way such that all branches present in the source code are tested, or a finite state machine can be used to generate test cases to cover all transition pairs. Similarly, vulnerability testing approaches often set such goal in advance. For example, a program should be tested for detecting all BOF and SQLI vulnerabilities.

In the second step, test cases are generated by using program artifacts (*e.g.*, source code) and interacting environments (*e.g.*, network protocol data unit or PDU) in a systematic way. A subsequent issue that needs to be addressed by testers is to define the oracle for each test case. Unlike traditional software testing, the end computational results performed by programs rarely play any role for determining oracles (or successful attacks) in program security vulnerability testing. In most of the cases, program states and response messages are used to identify the presence or absence of attacks.

In the final stage, test cases are run against implementations and programs are assessed based on predefined oracles to identify vulnerabilities (or whether a test case exposes the vulnerabilities through a program's response). Overall, program security vulnerability testing process is analogous to traditional software testing process, except each of the stages is handled differently.

3.2. Comparison of program security vulnerability testing approaches

In this section, we perform a comparative analysis of program security vulnerability testing works from the literature based on five criteria [6]: *test case generation method*, *source of test case*, *test level*, *test case granularity*, and *vulnerability coverage*. Table 3 shows a summary of the comparison, while we provide detailed descriptions for each criterion below.

Test case generation method: It implies how a source of test case is converted to a set of test cases. It is interesting to note that most of the traditional software testing techniques have been used or leveraged to conduct test case generation for security testing. We identify seven test case generation techniques [9]. These include *fault injection* (*e.g.*, [15, 52, 64, 65]), *attack signature* (*e.g.*, [86, 91]), *mutation analysis* (*e.g.*, [2, 3]), *static analysis* (*e.g.*, [50, 96]), *search* (*e.g.*, [28]), *program modification* [61], and *constraint bypassing* [57]. We discuss these methods in Sections 3.3-3.9.

Table 3: Comparison summary of program security vulnerability testing works

Work	Test case generation method	Source of test case	Test level	Test case granularity	Vulnerability
Jorgensen *et al.* [15]	Fault injection (inject faults at lexical, syntactic, and semantic level)	Valid data stream	Black box	Input file	BOF
Shahriar *et al.* [1, 2]	Mutation analysis (inject vulnerabilities in code)	Source code (C program)	White box	String or complex data type containing string	BOF
Xu *et al.* [50]	Static analysis (solve path constraints of programs)	Source code and API (C program)	White box	String or complex data type containing string	BOF
Zhang *et al.* [52]	Fault injection (fields of PDU)	Valid PDU	Black box	Sequence of PDU	BOF, FSB
Shahriar *et al.* [3]	Mutation analysis (inject vulnerabilities in code)	Source code (C program)	White box	String or complex data type containing format string	FSB
Haugh *et al.* [60]	Static analysis (interesting function coverage)	Source code (C program)	White box	String or complex data type containing string	BOF
Huang *et al.* [86]	Attack signature (inject attack inputs in HTML forms)	HTML form	Black box	URL	SQLI and XSS
Junjin *et al.* [90]	Attack signature (replace benign test cases with attack inputs)	Executable code (Java byte code)	Black box	URL	SQLI
Kals *et al.* [91]	Attack signature (inject attack inputs in HTML forms)	HTML form	Black box	URL	SQLI, XSS
Adam *et al.* [96]	Static analysis (solve path constraints), attack signature (replace non malicious test cases with attack test case)	Source code (PHP code)	White box	URL	SQLI and XSS
Shahriar *et al.* [4]	Mutation analysis (inject vulnerabilities in code)	Source code (JSP code)	White box	URL	SQLI
Shahriar *et al.* [5]	Mutation analysis (inject vulnerabilities in code)	Source code (PHP code)	White box	URL or sequence of URL	XSS
McAllister *et al.* [134]	Attack signature (replace non malicious test cases with attack test cases)	User session	Black box	Sequence of URL	XSS
Vilela *et al.* [75]	Mutation analysis (inject vulnerabilities in code)	Source code (C program)	White box	String or complex data type containing string	BOF
Tal *et al.* [73]	Fault injection (based on Protocol syntax)	Valid PDU	Black box	PDU	BOF
Allen *et al.* [72]	Fault injection (based on protocol specification)	Valid PDU	Black box	PDU	BOF
Tappenden *et al.* [71]	Attack signature (inject attack inputs in HTML forms)	HTML form	Black box	URL	BOF, SQLI
Kim *et al.* [67]	Fault injection (modify file tags or records)	Input file (HTML, WMF)	Black box	Input file	BOF
Ghosh *et al.* [65]	Fault injection (in program variables)	Program state (variable)	Black box	String or complex data type containing string	BOF
Du *et al.* [64]	Fault injection (in direct and indirect environment variables)	Program environment	Black box	Global variable, network input, file, socket	BOF
Breech *et al.* [61]	Program modification (through compiler)	Source code (C program)	Black box	Modified program	BOF
Offutt *et al.* [57]	Constraint bypassing (inputs in response pages)	HTML form	Black box	URL	SQLI, XSS
Grosso *et al.* [28]	Search technique (genetic algorithm)	Source code (C program)	White box	String or complex data type containing string	BOF
Cadar *et al.* [27]	Static analysis (solve path constraints)	Source code (C program)	White box	String or complex data type containing string	BOF

Source of test case: This criterion identifies what artifacts of programs or environments are used for generating test cases. These include source code of programs (*e.g.*, [2, 60]), vulnerable APIs (*e.g.*, ANSI

C library functions) [50], valid protocol data units (PDUs) (*e.g.*, [52, 73]), valid data streams [15], user sessions [134], executable program code (*e.g.*, [90]), runtime states of programs [65], and program environments [64]. Here, runtime state of a program includes declared variable values of a program. A user session indicates an execution path of a program that is traversed while performing a functionality. Program environment includes a broader range of inputs that might be generated from files, networks, and processors. Attack templates are known attack signatures that result in unintended behaviors in programs.

Test level: It indicates whether security vulnerability testing of a program is performed in a white box or a black box manner. Most of the testing approaches employ black box testing (*e.g.*, [15, 86, 90]). Some approaches explore white box testing mechanisms (*e.g.*, [28, 60]).

Test case granularity: This feature describes what constitutes a test case in program security vulnerability testing. Table 3 shows that test case granularity varies not only on data received by programs and its surrounding environments, but also on vulnerabilities. For example, exploiting BOF vulnerabilities involve generating strings of particular lengths, or complex data types containing strings (*e.g.*, [2, 3]). Similarly, test cases for exposing FSB vulnerabilities require strings (or complex data types) containing format specifiers [3, 58]. However, SQLI and XSS vulnerability exploitations require URLs with appropriate parameters and values (*e.g.*, [4, 5, 86, 90, 91, 96]). Moreover, a sequence of URLs (*e.g.*, [5]) or PDUs (*e.g.*, [72, 73]) might form just one test case since all of them must be applied to programs to exploit vulnerabilities. For example, to perform a stored XSS attack (a variation of XSS), at least two URLs are required to form one test case: one for storing a malicious script and the other to download a page containing that script.

Vulnerability coverage: This feature indicates what particular vulnerability an approach tests. From Table 3, it is obvious that BOF, SQLI, and XSS vulnerabilities have been addressed in most of the approaches. Moreover, very few approaches test multiple vulnerabilities (*e.g.*, [96]).

3.3. Fault injection-based test case generation

Fault injection is one of the most widely used test case generation techniques suitable for performing black box-based testing. The objective is to corrupt input data and variables, execute programs with corrupted data and variables, and observe unexpected responses to conform vulnerabilities. We divide fault injection-based security vulnerability testing works into three types based on the target of corruption: *input data*, *environment*, and *program state*. We describe them in the following three subsections.

3.3.1. Corrupting input data

In this technique, input data processed by programs are modified in ways such that the desired lexical, syntactic, or semantic structures become malformed. The resultant input data are supplied to a program under test to reveal vulnerabilities through abnormal behaviors (*e.g.*, program crashes). For example, Jorgensen *et al.* [15] corrupt valid data stream at lexical (*i.e.*, character level deformation such as replacing

a non printable character with a printable character), syntactic (*i.e.*, lexically correct and syntactically incorrect such as replace a left parenthesis with a space character), and semantic (*e.g.*, changing a date format) levels. Tal *et al.* [73] capture protocol data units (PDUs), modify data fields of these PDUs, then send them back to the server and observe the server application's responses (*i.e.*, whether the protocol daemon running in the server crashes due to segmentation fault or not). Kim *et al.* [67] corrupt the semantic structure of HTML files and Vector image files (WMF) by replacing one tag with another and modifying the fields of records, respectively.

Several works corrupt input data while maintaining the semantic meaning of a sequence of input data. For example, Zhang *et al.* [52] test an FTP (File Transfer Protocol) program by first identifying a valid command packet sequence. They generate malformed packets (*e.g.*, filling packets with large sized strings, special format strings such as *%s* and *%n*) that are valid according to protocol grammar, but might not be processed properly by target programs. Similarly, Allen *et al.* [72] construct a set of valid messages (or packets) into blocks based on a protocol specification. They keep message sequences intact and apply fuzzing in message fields to generate corrupted inputs.

3.3.2. Corrupting environment

In this technique, environment variables of programs are modified. These include environment variables during program initialization (*e.g.*, configuration file) and execution time (*e.g.*, file system inputs, network packets). We note that input data corruption techniques also modify files and network packets. However, these techniques modify the lexical, syntactic, or semantic structures. In contrast, environment variable corruption techniques modify the attributes of inputs [64]. For example, a file can be modified in terms of existence (*e.g.*, file can be deleted), permission (read or write permissions can be toggled), and ownership attributes. A configuration file might point to a list of directories for performing searches. A corruption technique might alter the sequence of directories to test security vulnerabilities.

3.3.3. Corrupting program state

In this technique, an executable program state is modified to check whether program code can handle vulnerabilities or not. Program state might include data variables (*e.g.*, boolean, integer, string) which control program execution as well as sensitive locations where program stores values such as function return addresses [65]. The modified states result in security violations. Thus, a program shows anomalous behaviors, if the implementation does not handle security violations appropriately.

3.4. Attack signature-based test case generation

This is the second most widely used test case generation approach, where test cases are generated by replacing some (or all) parts of a normal input with attack signatures. The attack signatures are developed from the experience and vulnerability reports. We notice that this approach is applied widely to web-based programs. We divide security vulnerability testing related works employing attack signature-based test

case generation method into two categories based on how web pages are traversed. These include *request and response* and *use case*-based test case generation.

3.4.1. Request and response

In this method, a crawler requests a web page and captures the response page. In the response page, it identifies input fields (*e.g.*, HTML forms) which are filled and submitted with malicious inputs. This process enables a tester to reach all pages where attack inputs can be injected and to observe the response of malicious inputs. Vulnerabilities in client or server side programs are observed based on the error messages. For example, Huang [86] and Kals *et al.* [91] apply request and response-based web page crawling to generate test cases (or URLs) to discover SQLI and XSS vulnerabilities. Programs are executed to check attack occurrences by replacing valid test cases with attack inputs (*i.e.*, substituting a URL parameter value with an attack input) [90, 96].

A variation of the approach is to apply a set of programmable APIs to perform crawling. These APIs enable a tester to emulate browsers such as form submission and page redirection. In particular, input form fields can be accessed and modified to inject attack input test cases. The modified from can be submitted and checked for the presence of vulnerabilities in the response pages through customized assertions. Tappenden *et al.* [71] develop a set of programmable APIs named HTTPUnit to detect SQLI vulnerabilities in web-based programs for agile environment (*i.e.*, testing and development occurs simultaneously).

3.4.2. Use case

Traditional request and response-based testing approaches identify vulnerabilities at the interface level of programs. They often fail to reach inner logics of program code which might open the doors for attacks. To alleviate this problem, a use case-based approach first applies interactive user inputs to perform functionalities of programs (*e.g.*, login). This ensures breadth testing of a program's code. Later, the collection of user inputs (a sequence of URLs) required to perform functionalities are replayed back and malicious test cases are injected at injecting points of URLs (instead of previously save user inputs). Fuzzing method (or random fault injection) is used for injecting malicious inputs. This ensures testing the depth of program logics. McAllister *et al.* [134] apply this approach to discover reflected and stored XSS vulnerabilities in web-based programs.

3.5. Mutation analysis-based test case generation

Mutation is a fault-based testing technique that is intended to show that an implementation is free from specific faults [14]. Mutation operators are used to generate mutants by injecting faults in an implementation under test. Mutation operators modify program artifacts to inject vulnerabilities and force the generation of effective test cases that can expose the injected vulnerabilities. A mutant is said to be killed or distinguished, if at least one test case can produce different output between the mutant and the

implementation. Otherwise, the mutant is *live*. If no test case can kill a mutant, then it is either equivalent to original implementation or new test case needs to be generated to kill the *live* mutant. Generating new test cases enhances the fault detection ability of the test suite (a set of test cases). The adequacy of a test suite 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. Note that there is a subtle difference between fault injection-based testing (that might employ mutation operator) and mutation-based analysis. In mutation-based analysis, the end objective is to assess test suite quality. However, a fault injection technique is guided by mutation operators with the objective of testing the presence of vulnerabilities in programs by observing anomalous behaviors. Moreover, a mutation-based analysis adds new test cases to increase the *MS*, whereas, injecting a fault results in a new test case (*e.g.*, a modified PDU) in fault-based testing.

The pioneer research of applying mutation-based analysis has been performed by Vilela *et al.* [75] where they assess test suite quality for detecting BOF vulnerabilities in C programs. However, 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), language specific features (*e.g.*, absence of the null character at the end of a buffer). Later, Shahriar *et al.* [1, 2] propose mutation operators to assess test suite qualities for detecting BOF caused by the above issues. Moreover, Shahriar *et al.* [3] propose mutation operators to inject faults in ANSI C format functions (format string and arguments). Furthermore, they propose mutation operators for adequate testing of SQLI [4] and XSS [5] vulnerabilities in web-based programs implemented in JSP and PHP languages, respectively.

3.6. Static analysis-based test case generation

This approach generates test cases by analyzing program source code without executing[3]. The analysis relies on the symbolic execution of program code, where program inputs are assumed to hold arbitrary values represented by symbols [33]. In the context of security testing, the main idea is to extract path constraints and update symbolic values present in path conditions at different statements. The symbolic values are updated with known values based on initialized variables or derived values from inputs [27]. While a path ends or a vulnerable statement (*e.g.*, a buffer access likely to cause a BOF) is reached, a current path constraint is solved with a custom constraint solver to obtain a set of concrete input values (*i.e.*, a test case). Most of the symbolic execution-based test case generation approaches have been applied to detect BOF vulnerabilities in C programs [27, 50]. In these cases, symbolic assignments are preformed for program statements present along feasible paths that include either sensitive memory accesses [27] or

[3] The detailed description of static analysis techniques is provided in Section 4.

potential invalid memory addresses related to pointer variables [50]. Recently, symbolic execution-based analysis has been applied to generate SQLI and XSS vulnerabilities in web-based programs [96].

A variation of static analysis-based test case generation approach employs "interesting function coverage" information to guide test case generation. For example, Haugh *et al.* [60] develop a Systematic Testing of Buffer Overflow (STOBO) tool which instruments a given input program file to identify whether more test cases are required to discover BOF vulnerabilities caused by unsafe *memcpy* and *strcpy* function calls.

3.7. Search-based test case generation

If the test input space for discovering vulnerabilities is huge and identification of vulnerability revealing test cases is time consuming, a practical approach is to apply a search-based technique [21] as a way of generating test cases. For example, discovering BOF with test cases having large sized strings fits well for applying the search-based test case generation technique. In general, a technique applies a suitable search algorithm where a random input is chosen as the initial solution. The solution is evolved over a number of times unless an objective function value remains unchanged.

One of the most widely used search techniques applied for program security testing is genetic algorithm. The major variations in genetic algorithm-based approaches while generating test cases occur in two important stages: fitness functions and mutation operators. The fitness (*i.e.*, objective) function guides the generation of test cases so that vulnerability exploitations are revealed. However, depending on testing objectives, fitness functions vary. For example, Grosso *et al.* [28] define a fitness function to generate BOF test cases which focus on program code coverage (*e.g.*, vulnerable statement with unsafe ANSI C library function calls). Mutation operators (*i.e.*, evolving of solutions) depend on the testing objective. For example, Grosso *et al.* [28] apply a mutation operator, where a numeric value is randomly incremented, and a string is appended with a random string. Such approach allows the generation of test cases to quickly reveal BOF attacks.

3.8. Program modification-based test case generation

In this technique, program instructions are modified by leveraging dynamic compiler techniques which allow accessing intermediate program code before being executed by CPUs. The technique tests programs that have vulnerability exploitation detection mechanisms included. Moreover, it can detect subtle vulnerabilities that otherwise might go undetected due to modification of program environments (*e.g.*, memory layout) by compilers. For example, a compiler applies default padding after an allocated buffer, which prohibits confirming the presence of one byte BOF with test cases. Breech *et al.* [61] develop a testing framework based on a dynamic compiler that can test programs by injecting attack code (*e.g.*, modifying the return address of a function) in basic blocks (*i.e.*, a sequence of code with no branching).

3.9. Constraint bypassing-based test case generation

The main idea of this technique is to generate inputs to bypass input filtering mechanisms employed in a program. Such an approach can be applied to web-based programs, where script code can be written by programmers to filter malicious inputs at browsers. Offutt *et al.* [57] propose three levels of bypass testing for web-based programs to discover client side vulnerabilities such as SQLI and XSS. These include (i) value level (*e.g.*, tests data types, built-in length violations, special input values), (ii) parameter level (*e.g.*, testing violations of underlying relationships among multiple parameters), and (iii) control flow level (*e.g.*, testing under pressing back button, refresh button) testing.

3.10. Open issues

From the comparative study, we notice that current approaches are limited to testing few vulnerability types such as BOF, SQLI, and XSS. Thus, future research work should discover other vulnerabilities that are present in programs such as FSB and CSRF. Moreover, every test case generation method has a narrow perspective of testing program security vulnerabilities. For example, fault-based techniques provide us limited or no information related to vulnerability and code coverage. Therefore, future research direction can combine fault-based technique with other techniques (static or hybrid analysis) to improve the situation. Current fault-based approaches are limited to corrupting input data, environment variables, and program states. We believe that many unknown vulnerabilities can be discovered by injecting faults on program artifacts such as control and data flows. We also notice that attack signature-based test case generation has been improved by combining static analysis. However, their scope is limited to web-based programs and web-based vulnerabilities. Search-based technique has been applied to generate test cases only for testing BOF vulnerabilities. Future research might explore employing the search techniques for generating test cases for other vulnerabilities. Program modification techniques can also be explored for web-based programming paradigms (*e.g.*, modifying client side program state).

4. Static analysis

A common proactive approach to detect security vulnerabilities in program code is to perform static analysis [51]. The approach examines input program code, applies specific rules or algorithms (also known as inference), and derives a list of vulnerable code present in a program that might result in vulnerability exploitations. The greater advantage of performing static analysis is that it does not require executing program code. As a result, the analysis can ignore the issues related to program executions such as the reachability of vulnerable code and the generation of input test cases to traverse the vulnerable code. The pioneer static analysis techniques (*e.g.*, control flow, data flow, inter-procedural analysis) have been developed for compiler generated code optimizations, and they are not intended for detecting security vulnerabilities in program code. As security breaches have become widespread in programming

communities, many tools and techniques have been developed to apply static analysis for discovering vulnerabilities in program code [44, 46, 148].

In the following subsection, we perform a comparative analysis of static analysis works based on the following seven aspects: *inference type, analysis sensitivity, analysis granularity, completeness, soundness, vulnerability coverage,* and *language* supported in the analysis. The effectiveness of any static analysis depends on how accurate the inference is in discovering potential vulnerable code. Thus, we are motivated to classify current static analysis related works based on the underlying inference techniques into four types: *tainted data flow-based*, *constraint-based*, *annotation-based*, and *string pattern matching-based*. We describe these inference techniques in Sections 4.2-4.5. We discuss open issues in Section 4.6.

4.1. Comparative analysis of static analysis approaches

Table 4 provides a comparative summary of the static analysis works related to program security based on seven features: *inference type*, *analysis sensitivity*, *analysis granularity*, *completeness*, *soundness*, *vulnerability*, and *language* [9, 11]. We now discuss these features in the following paragraphs.

Inference type: The core part of any static analysis is to infer potential vulnerabilities by scanning program code. We divide inference types into four categories as shown in Table 4. These are *tainted data flow-based* (*e.g.*, [88, 99, 100, 117]), *constraint-based* (*e.g.*, [20, 46, 47]), *annotation-based* (*e.g.*, [7, 12, 114]), and *string pattern matching-based* (*e.g.*, [42, 44, 140]) approaches. We discuss these inference types in Sections 4.2-4.5.

Analysis sensitivity: A common problem for a static analysis approach is that it might generate false positive warnings to be examined manually. Moreover, an analysis might suffer from false negatives (*i.e.*, vulnerabilities present in program code might be unreported). To reduce the number of false positives or false negatives, the approaches take advantage of pre-computed information based on program code. Such information helps to perform more accurate detection of vulnerabilities. We denote the dependency of such pre-computed information as analysis sensitivity. From the third column of Table 4, we notice that most of the analysis techniques apply some kinds of sensitivity. We identify six types of sensitivity: *control flow* (*e.g.*, [46, 88, 114]), *path* (*e.g.*, [20, 47]), *context* (*e.g.*, [20, 88, 99, 100, 117]), *field* (or instance) [139, 171], *points-to* (*e.g.*, [99, 100]), and *value range* [172].

An analysis is control flow sensitive, if it performs inference technique based on statement execution sequence with respect to a control flow graph. Applying flow sensitivity increases the precision of vulnerability detection (*i.e.*, less false positive warnings). For example, Weber *et al.* [46] improve the approach of Wagner *et al.* [118] by employing a flow sensitive analysis.

Table 4: Comparison summary of static analysis approaches for mitigating security vulnerabilities

Work	Inference type	Analysis sensitivity	Analysis granularity	Completeness	Soundness	Vulnerability coverage	Language
Dor *et al.* [116]	Annotation	Points-to	Intra-procedural	Yes	No	BOF	C
Hackett *et al.* [12]	Annotation	N/A	Inter-procedural	No	No	BOF	C
Le *et al.* [20]	Constraint	Context, path	Inter-procedural	No	No	BOF	C
Tevis *et al.* [42]	String pattern matching	N/A	Statement	No	No	BOF	x86
Viega *et al.* [44]	String pattern matching	N/A	Token	No	No	BOF	C, C++
Weber *et al.* [46]	Constraint	Context, control flow	System dependence graph	No	No	BOF	C
Xie *et al.* [47]	Constraint	Context, path, points-to	Intra-procedural, inter-procedural	Yes[4]	No	BOF	C
Jovanovic *et al.* [88]	Tainted data flow	Context, control flow	Inter-procedural	No	No	SQLI, XSS	PHP
Lam *et al.* [99], Livshits *et al.* [100]	Tainted data flow	Context, points-to	Statement	No	Yes	SQLI, XSS	Java
Wassermann *et al.* [132]	Tainted data flow	N/A	Intra-procedural	No	No	XSS	PHP
Shankar *et al.* [138]	Tainted data flow, annotation	N/A	Intra-procedural, inter-procedural	No	No	FSB	C
Chen *et al.* [139]	Tainted data flow, annotation	Context, field	Statement	No	No	FSB	C
Dekok *et al.* [140]	String pattern matching	N/A	Statement	No	No	FSB	C
Wassermann *et al.* [145]	Tainted data flow	N/A	Intra-procedural	No	Yes	SQLI	PHP
Flawfinder [148]	String pattern matching	Context	Token	No	No	BOF, FSB	C, C++
Wagner *et al.* [118]	Constraint	N/A	Statement	No	No	BOF	C
Xie *et al.* [117]	Tainted data flow	Context, control flow	Block, intra-procedural, inter-procedural	No	No	SQLI, XSS	PHP
Evans *et al.* [114]	Annotation	Control flow	Statement	No	No	BOF, FSB, MEL, NLD	C
Tripp *et al.* [171]	Tainted data flow	Context, field, points-to	Inter-procedural	No	Yes	SQLI, XSS	Java, JSP
Vulncheck [172]	Tainted data flow, annotation	Value range	Statement, data flow, inter-procedural	Yes	No	BOF	C
Ganapathy *et al.* [173]	Constraint	Context, points-to	System dependence graph	No	No	BOF	C
Yang *et al.* [78]	Annotation, tainted data flow	Control flow	Intra-procedural, Inter-procedural	No	N/A	BOF	C
Ashcraft *et al.* [131]	Tainted data flow	N/A	Intra-procedural, Inter-procedural	No	No	BOF	C

Program execution paths that can be derived from a control flow graph might not be feasible. Sometimes, infeasible paths can be determined statically. If an analysis explicitly excludes infeasible paths, we denote it as path sensitive. For example, Le *et al.* [20] detect BOF vulnerabilities that are reachable

[4] The authors report false positive warnings due to the error in their implemented constraint solver. We believe that these errors are correctable and independent of their original approach.

within feasible program paths. Xie *et al.* [47] detect BOF due to pointer dereferences and buffer indexes in feasible program paths based on control flow graphs.

A context sensitive analysis differentiates multiple calls sites of a function with respect to supplied arguments. In contrast, a context insensitive analysis ignores multiple calls of the same function with different arguments. Context insensitivity results in false positive warnings. For example, a program has two vulnerable library function calls of *strcpy (dest, src)*. Here, the function copies the buffer pointed by *src* to the buffer pointed by *dest*. The first function call has *src* string whose value is provided by a user, and the second call contains a constant string whose length can be determined statically to be less than that of the *dest* buffer. A context insensitive analysis reports both of them as vulnerable. However, a context sensitive analysis (*e.g.*, Flawfinder [148]) finds the former call as vulnerable and the later as non vulnerable.

An analysis is said to be field (or instance) sensitive, if different members of instantiate objects are considered as separate variables [139]. Field sensitivity allows reducing false positive warnings. For example, an object data type (*e.g.*, *struct* of C language) might have two member variables of character buffer. One of the buffers contains untrusted data while the other contains trusted data. Without field sensitivity, the entire instance of the structure would be considered untrusted (or tainted) as member variables are not distinguished.

A points-to analysis identifies a set of memory objects that might be pointed by a given pointer variable present in program code. Points-to analysis itself is another direction of research and interested readers can consult the literature related to points-to analysis (*e.g.*, [36]). We restrict our discussion on four concepts that are relevant and used in detecting vulnerabilities: flow sensitive, flow insensitive, context sensitive, and context insensitive points-to analysis. In a flow sensitive points-to analysis, a program's control flow is taken into account. In contrast, in a flow insensitive points-to analysis, statements can be analyzed in any order. Obviously, a flow sensitive points-to analysis is more accurate than that of a flow insensitive analysis. In a context sensitive points-to analysis, function calls accepting pointer type arguments or returning pointers are analyzed separately. In contrast, in a context insensitive points-to analysis, these function calls are considered identical. It is more appropriated to apply context sensitive points-to analysis than that of context insensitive analysis to obtain more precise information.

From Table 4, we note that very few techniques apply *points-to* analysis (*e.g.*, flow insensitive points-to analysis [116]). Most of the BOF detection approaches (as shown in Table 4) do not incorporate points-to analysis explicitly. However, some approaches employ general assumption on pointer data types. For example, Wagner *et al.* [118] assume that a pointer to a structure variable might point to all other similar structure variables present in a program. The assumption results in a huge number of false positive warnings. Xie *et al.* [47] consider a limited number of pointer information such as a pointer pointing to a

buffer and relative distance from the base size of a buffer. However, if a pointer points to an unknown type of memory object, their analysis does not include such information. As a result, some real BOF might be undetected (*i.e.*, increases false negative in the detection). Points-to analysis can be used to reduce false negatives (*i.e.*, detect more vulnerabilities that would otherwise remain undetected). For example, Ganapathy *et al.* [173] apply points-to analysis information while generating constraints on buffers that are being dereferenced by pointer variables. As a result, function calls having pointer to buffer arguments can be analyzed for detecting BOF vulnerabilities.

A value range analysis provides a lower and upper bound of a variable. The information is useful when function calls are supplied with unsanitized arguments that represent buffer sizes. These arguments result in vulnerabilities, if their boundaries exceed expected ranges. For example, the value range analysis might discover that the upper bound of the size argument of *memcpy* function call is MAXINT (maximum value of an integer). An approach might generate a BOF warning in this case [172].

We notice that several works ignore sensitivity in their analyses for the sake of achieving the highest level of scalability (*e.g.*, [44, 118]). The resultant insensitivity allows an approach to analyze large scale programs that contain millions of lines of code. In contrast, adding sensitivity brings not only the benefit of increased precision in an analysis, but also reduced false positive rate in the warnings. Note that several works do not explicitly discuss about sensitivity or insensitivity that are indicated as N/A in Table 4 (*e.g.*, [12, 42, 44, 132]).

Analysis granularity: This feature indicates the granularity level of program code at which static analysis is performed. From Table 4, we notice that static analyses have been performed at different granularity levels. These include program *token* (*e.g.*, [44, 148]), *statement* (*e.g.*, [46, 99, 100]), *block* [117], *intra-procedural* (*e.g.*, [116, 145]), *inter-procedural* (*e.g.*, [12, 20, 88, 171]), and *system dependence graph* (*e.g.*, [173]) levels. Several works combine multiple granularities such as intra-procedural and inter-procedural (*e.g.*, [47, 117, 122]).

A compiler performs lexical analysis which tokenizes program source code to identify keywords, variables, functions, etc. These tokens can be used to detect vulnerabilities.

In a statement level analysis, vulnerabilities are inferred by analyzing program statements. Several works analyze the executable program code such as *x86* [42]. However, such executable code is usually de-assembled first to make partially readable before performing an analysis at the statement level.

In an intra-procedural analysis, a program is analyzed based on either a control flow graph or a data flow graph (a graph which represents data dependencies between a number of operations). Several works apply intra-procedural analysis to form a summary of sensitive variables [116] or specific conditions that are related to vulnerabilities [47].

An inter-procedural analysis examines a function body as well as other function call sites present in the function by accessing the called functions. It is common to avoid analyzing same function multiple times by following a bottom up analysis based on a call graph of a program (*e.g.*, [47]). In a call graph, each node represents a unique function and a directed edge from node *a* to *b* indicates that the function represented as node *a* contains an invocation of the function represented by the node *b*.

In a system dependence graph (SDG), a node represents a program point (*e.g.*, statement), and an edge represents dependency between two program points which can be two types: control flow and data flow. Note that a data flow analysis attempts to compute the values of a data variable at different program points.

Completeness: An approach is said to be complete, if it generates no false positive warnings [139]. Table 4 shows that very few techniques are complete (*e.g.*, [116, 172]). However, the completeness depends on both the underlying assumptions in an analysis and the types of vulnerabilities addressed. For example, Dor *et al.* [116] detect all BOF vulnerabilities present in a program caused by only string variables (*i.e.*, static or dynamic buffers). However, they do not consider BOF vulnerabilities due to pointers and aliases. Vulncheck [172] is complete under the assumption that most BOF vulnerabilities occur through a set of known unsafe library function calls that accept tainted arguments.

In practice, it is challenging to develop an analysis technique that results in no false positive warnings. We identify three common reasons that contribute for an incomplete analysis: analysis sensitivity, result interpretation, and assumption on program code.

Inference approaches based on insensitive analysis of control flow (*e.g.*, [42, 44, 100, 118, 140, 148, 173]), context (*e.g.*, [114, 138]), path [132], points-to (*e.g.*, [12, 46]) result in conservative analyses and generate a high number of false positive warning. Moreover, the imprecision of analysis sensitivity (*e.g.*, flow insensitive points-to analysis) contributes to false positive warnings [99, 100, 171].

The result interpretation of an analysis can be blamed to false positive warnings. For example, the approach of Le *et al.* [20] provides a set of programs paths which do not include safe and infeasible paths. Thus, it is likely that some of the suspected paths in the set might not be vulnerable.

An approach assumes that programmers write specific pattern of code in implementations. The breaking of these assumptions results in false positive warnings. For example, a buffer variable might contain malicious inputs followed by a non-malicious input before using the variable content (*e.g.*, writing to an output console) [139]. The buffer is suspected to be vulnerable in the first usage and it remains suspected for the rest of the program code. A program can be written in an unusual way by storing malicious data instead of terminating programs immediately which might result in a false positive warning [117]. A function might conditionally use a kernel pointer instead of a user supplied pointer [78], where kernel pointer arguments are not vulnerable. Moreover, an approach might rely on correct implementation of functionalities (*e.g.*, validation of inputs stored in buffers [116]). Sometimes, assumptions are made on the

effects of code, which lead to false positive warnings. An approach might not consider the effect of custom sanitization functions [88]. The lack of consideration to the effect of arbitrary type casting of sensitive variables might generate false positive warnings. For example, PHP allows a variable containing malicious string be assigned a boolean value which is non malicious [145]. A program might type cast a signed integer to an unsigned integer. An unsigned integer should be checked only for the upper bound before using the integer as an index of an array. An analysis might ignore type casting effect and generate a false positive warning while using an unsigned buffer index with no lower bound value checking [131].

Soundness: An approach is sound, if it has no false negative [139]. In other words, the approach does not leave any program vulnerability unreported. Table 4 shows that a very small number of works claim to be sound (*e.g.*, [100, 171]). However, it depends both on vulnerability types and assumptions in their analyses. For example, a query-based analysis [99, 100] is sound as long the specified queries accurately represent vulnerability patterns. The approach of Tripp *et al.* [171] is sound provided that a user specified maximum node limit (in a call graph analysis) allows reaching all tainted sinks.

The reasons for being the approaches unsound vary widely. We classify the reasons into four types: language features, analysis sensitivity, result interpretation, and the scope of the problem.

Data types (*i.e.*, features) supported in programming languages might not be considered in the analysis approaches. For example, the approach of Wagner *et al.* [118] is not sound as their analysis does not generate warnings of BOF vulnerabilities caused by pointer arithmetic or complex data structures (*e.g.*, union in C) having buffers as member variables. The approach of Hackett *et al.* [12] does not support global pointers and structure fields having pointer data types. A recursive function call might be analyzed using just a single pass while constructing a function's summary in PHP which might leave potential SQLI and XSS vulnerabilities unreported [117]. The approaches of Shankar *et al.* [138] and Chen *et al.* [139] are not sound as they do not support tainted data flow analysis for integer buffer data type. An integer array containing malicious format string can be converted to a buffer character followed by supplying it to a format function. No FSB warning would be reported in those approaches.

Some approaches might not employ analysis sensitivity which might introduce false negatives. For example, lack of points-to analysis (*e.g.*, [20, 42, 44, 46, 140, 148]).

Some works only generate warnings, if their analysis can infer certainly that vulnerabilities are present in programs. This also allows users to interpret results with high confidence at the cost of missing actual vulnerabilities present in program code. For example, the approach by Xie *et al.* [47] does not generate warnings, if their analysis cannot infer that accessing a buffer is unsafe.

To make an analysis approach manageable, some works explicitly limit their scope of detection for a limited number of vulnerability types present in program code. Thus, they can be considered as not sound with respect to the unaddressed vulnerability types. For example, BOF might be present due to unsafe

ANSI C library function calls, pointer arithmetic, accessing buffers through arbitrary buffer indexes and pointers, lack of null characters at the end of buffers, and user defined functions containing flaws. Dor *et al.* [116] detect all BOF vulnerabilities present in a program caused by only string variables (*i.e.*, static or dynamic buffers) and those accessed after null byte characters. The approach of Evans *et al.* [114] cannot discover BOF caused by arbitrary buffer indexes and pointers. Xie *et al.* [47] assume that most pointer arithmetic and conditional expressions present in branches and loops are linear. Thus, their approach does not report BOF vulnerabilities that might occur through loops or branches containing non linear arithmetic. The approach of Wassermann *et al.* [132] detects only stored and reflected XSS attacks and does not analyze program code generated at browsers to identify DOM-based XSS attacks. Ashcraft *et al.* [131] assume that an integer variable bound checking must be present before accessing a buffer through the integer index variable. However, they allow a variable to compare with any numeric value as opposed to a correct value. Thus, actual BOF present in a program might not be identified.

Vulnerability coverage: This feature indicates what vulnerabilities are detected by an approach. Table 4, it is obvious that BOF, FSB, SQLI, and XSS have been addressed by many works (*e.g.*, [44, 46, 47, 116]). However, very few works detect other vulnerabilities such as MEL [114]. Very few approaches can detect multiple vulnerabilities (*e.g.*, [88, 114]).

Language: This feature highlights the programming languages that are supported by an analysis approach. We notice that most techniques analyze programs implemented in C and Java languages (*e.g.*, [12, 47, 100, 116]). A number of works analyze server side programs written in scripting languages such as PHP *(e.g.*, [88, 145]). Very few approaches analyze executable code (*e.g.*, [42]).

4.2. Tainted data flow-based technique

In this technique, input variables are marked as tainted and their propagations are tracked. Warnings are generated, if tainted inputs or values derived from tainted inputs are used in sensitive operations. We divide tainted data flow-based works into four categories based on the tainting mechanism: *static data type*, *implicit*, *grammar-based*, and *query-based tainting*. They are described in the following subsections.

4.2.1. Static data type tainting

In this technique, tainted information is marked by extending variable type information. This is also known as *type qualifier* approach. The approach takes advantage of statically declared data types supported by programming languages. In particular, the technique requires adding new labels to data variables which might represent vulnerable (“tainted”) or non-vulnerable (“untainted”) input sources. Then the approach checks whether a labeled source or any value derived from a labeled source participates in sensitive operations (*e.g.*, a data buffer containing user supplied inputs is passed to a format function call) or not. If such a case is identified, a vulnerability warning is generated. For example, Shankar *et al.* [138] and Chen *et al.* [139] detect FSB vulnerabilities, if tainted format strings are used in format function calls. Their

approaches label trustworthy function parameters as “untainted” and untrustworthy parameters as “tainted” initially. For example [138], the main function of a program might be marked as follows *int main (int argc, tainted char *argv[])*.

Type checking is a popular approach to infer vulnerabilities [138, 139]. In a traditional type checking system, errors are reported by a compiler, if the type of a variable mismatches with the expected one. However, in a type qualifier system, warnings are raised, if an expected variable type in an expression mismatches according to a qualifier lattice. A qualifier lattice is developed by a programmer before an analysis that represents sub-typing relationships among different variables. For example, a sub-typing relationship can specify “untainted char < tainted char”. It indicates that an expression (*e.g.*, a format function parameter labeled as “tainted”) expecting a tainted variable generates no warning, if an untainted variable is passed. However, for a sub-typing relation “tainted char < untainted char”, a warning is generated if a tainted variable is passed instead of an untainted one.

4.2.2. Implicit tainting

In this technique, program variables are not explicitly labeled as tainted. This approach is suitable for languages where there are no static type declarations in the code (*e.g.*, PHP). The tainted information flow is performed based on pre-computed program information such as data flow and control flow graph. For example, Jovanovic *et al.* [88] mark data as tainted, if they are derived from user inputs (*e.g.*, HTTP requests). They apply data flow analysis to identify locations where suspected tainted data reach to sensitive sinks (*e.g.*, the locations that are vulnerable to XSS attacks).

A variation of the approach is to pass tainted information from lower to higher granular level. For example, Xie *et al.* [117] apply three levels of static analyses based on the control flow graph of a given program. They summarize and pass information from block to intra-procedural and intra-procedural to inter-procedural levels. Basic blocks are simulated by symbolic execution to form summaries such as *error set* (*i.e.*, set of input variables that must be sanitized before entering the block) and *untainted sets* (sanitized locations for each successor of a block). This information is applied to perform intra-procedural analysis which summarizes a function such as *sanitized values* (set of parameters and global variables that are sanitized on function exit). The summaries obtained from intra-procedural analysis are applied to inter-procedural analysis (*i.e.*, function calls at block levels). The inter-procedural analysis generates warnings, if any unsanitized variable is applied to a sensitive operation (*e.g.*, SQL query generation).

Implicit tainting has been applied to programs written in typed languages such as Java and JSP (*e.g.*, taint analysis for Java or TAJ [171]). Tainted data can be tracked using a variation of slicing algorithm known as hybrid thin slicing. The slice captures statements relevant to tainted data flows. It is a forward slicing from a statement *s* which identifies a set of statements that are data-dependant on *s*. The analysis is performed by a set of security rules of the form (*S1*, *S2*, *S3*), where *S1*, *S2*, and *S3* represent a tainted or

untrusted source, a sanitizer, and a sink, respectively. A sink is a pair of a method name and a set of parameters that are vulnerable to injection attacks. The approach checks whether a source is passed to a sink without sanitization or not. Compared to other taint-based approaches, the slicing along with points-to analysis of objects can discover vulnerabilities in programs that have reflection (a technique where a method can be invoked by *Class.forName* and *Method.invoke* properties in Java) and object containers (*e.g.*, *HashMap*).

Ashcraft *et al.* [131] also apply a tainted data flow-based technique to detect BOF vulnerabilities by identifying tainted integer variables (derived from network packets, user data) that are used as array indexes, loop bounds, and length parameters of sensitive functions (*e.g.*, *memcpy*) without sanitization (*e.g.*, performing no upper and lower bound checking for a signed integer variable).

4.2.3. Grammar-based tainting

Tainted data flow can be tracked by grammar production rules where non terminals can be marked as tainted. This approach requires normalizing program source code into static single assignment (SSA) form. In a SSA form, each statement is expressed as an assignment statement where the left hand side sets a variable at most once. For example, a PHP statement "*echo $out*" that outputs the value of variable output (*out*) can have SSA form as "*data1 = out*" [132]. Here, *data1* is introduced as part of SSA which can be further transformed as a production rule like "*data1→ out*". Now, *data1* is a non terminal for the production rule and *out* might be either a terminal or non terminal. The grammar rules obtained from SSA implicitly encode data flow information among non terminals.

While generating grammar production rules, non terminals are marked as tainted if they are obtained from user inputs. The resulted rules form an extended context free grammar. The core part of an approach involves performing string analysis for sanitization routines by modeling their operations through finite state transducers (*i.e.*, finite state machines where transitions result outputs). If tainted inputs are sanitized, the output is expressed as regular expressions which relate the images (*i.e.*, a set of words that can be generated) of transducers. Moreover, tainted markings are removed from the related rules to make a context free grammar. The regular expression generated by a context free grammar and by a sanitization routine is intersected to identify another regular expression that is allowed by a program. If part of attack signatures or malicious scripts (*e.g.*, *' or 1=1* -- is a signature for a tautology attack exploiting SQLI vulnerabilities) can be constructed from an intersected regular expression, then a warning is generated. Wasserman *et al.* apply this approach to detect SQLI [145] and XSS [132] vulnerabilities.

4.2.4. Query-based tainting

All the taint-based techniques discussed so far provide no options to testers or programmers to check vulnerabilities of their choices. Query-based tainting is a step forward to mitigate this limitation. In this technique, a query specifies source objects and rules to transform sources to sink objects. A source object contains supplied inputs. A derived object uses a source object and propagates through a path which might

reach to a sink object. A sink object is considered tainted, if it is derived from zero or more times from a source object. Lam *et al.* [99, 100] develop the Program Query Language (*PQL*) which allows programmers to specify queries related to vulnerable information flow. They apply the technique to discover SQLI and XSS vulnerabilities in web-based programs implemented in Java.

4.3. Constraint-based technique (CBT)

Constraint-based techniques generate safety constraints from program code whose violations imply vulnerabilities. Constraints are propagated and updated while traversing a program. A program might be traversed based on system dependence graphs [46] and control flow graphs [47, 117]. At the end, the analysis identifies whether any solution of a set of obtained constraints exists or not. Obtaining a solution indicates that an exploitation might be possible through input values. Constraint solvers are used to compute input values which violate constraints. Most works leverage integer type constraint solvers for detecting BOF vulnerabilities [46, 47, 118]. Constraint-based approaches are applied to procedural languages that have static data types and rich data structures. Several works have applied CBT techniques to detect BOF vulnerabilities in C programs. We divide constraint-based techniques into three categories for BOF detection: integer range analysis, symbolic value analysis, and demand-driven.

Integer range analysis: The idea of this technique is to formulate constraints by scanning each program statement containing buffer declarations and buffer related operations. Each constraint is expressed in terms of a pair of integer ranges (buffer allocation size and current size of a buffer) for each buffer defined or accessed. For each buffer, a set of constraints are solved to find a range of allocation and current size, which can be denoted as $[a, b]$ and $[c, d]$, respectively. Here, $[a, b]$ implies that allocation size of a buffer can vary from a bytes to b bytes, and $[c, d]$ implies that current size of a buffer can vary from c bytes to d bytes. The two ranges can be analyzed to identify non vulnerable (*i.e.*, $b > c$) and vulnerable (*i.e.*, $b \leq c$) statements.

Wagner *et al.* [118] first apply integer range analysis to detect BOF vulnerabilities. Later, Weber *et al.* [46] improve Wagner's method by applying a flow sensitive analysis of BOF to reduce the number of false positive warnings. In addition to BOF due to string variables, they detect BOF due to global variable usages, function calls, and recursions. Ganapathy *et al.* [173] generate constraints on buffer sizes and allocations using an SDG and solve the constraints using linear programming.

Symbolic value analysis: In this technique, a constraint might contain program variables and whenever possible their values are assigned. Otherwise, they are considered as symbolic values. Xie *et al.* [47] apply symbolic value analysis to detect BOF vulnerabilities caused by invalid buffer indexes, pointer dereferences, and invalid function arguments (buffer addresses and sizes). They traverse a call graph of a program using a bottom-up approach, where a function is analyzed through control flow graph (CFG). During the CFG analysis, at every access to arrays, pointer dereferences, and function calls, safety

constraints (*i.e.*, the negation of valid conditions) are generated. Moreover, constant relations (*e.g.*, $x = 4$) and symbolic constraints between variables (*e.g.*, $x < y$) are captured and propagated. At every potentially dangerous access of arrays, pointer dereferences, or call to routines, a custom constraint solver is used to evaluate the values against safety constraints. A warning is generated, if a solution can be found for a safety constraint.

Demand-driven: Most constraint-based techniques limit their scopes by providing a list of warnings based on built in constraint generation, propagation, and solution mechanisms. However, a recent direction is to provide a programmer the option to formulate queries on vulnerable program locations. This is also denoted as demand-driven analysis. The approach relies on constraint generation, which starts from a location specified by developers. The end output might be a set of prioritized paths that might trigger BOF. Le *et al.* [20] apply this approach to detect different types of paths (*e.g.*, infeasible, safe, user input dependant, vulnerable) vulnerable to BOF. A user specifies queries to know whether (i) buffer accesses at particular program points are safe, and (ii) user inputs can write to buffers. The queries are expressed with constraints in terms of buffer sizes, supplied string lengths, and flag values (to represent constant string values). The queries are propagated along program paths in backward directions (*i.e.*, starting from the point of query to the beginning of a program's main function call) through inter-procedural and context sensitive analyses. If queries can be resolved by checking whether a declared buffer size is less than the supplied input values, a BOF warning is generated.

4.4. Annotation-based technique

In an annotation-based technique, program code is annotated with desired properties in terms of pre and postconditions. Annotations can be specified at both function prototype declaration [12, 114, 116] and statement level [114]. After a piece of code is annotated, an algorithm checks whether data variables can be used safely based on the annotated conditions or not. In this case, a function call site is evaluated to check whether it conforms to specified preconditions or not (*i.e.*, generate error messages). If preconditions are satisfied at the call site, the function body is further examined to ensure that the implementation meets specified postconditions. If a precondition cannot be resolved from a previous statement's postcondition, then a warning message is generated. Warnings generated by an approach might be prioritized based on how well the constraints for accessing buffers are understood. For example, no condition to access a buffer is listed at the top of a warning list, whereas buffer access based on a condition on buffer length is placed at the bottom of a warning list.

Annotation-based approaches generate constraints from the specified pre and postconditions. However, these constraints are evaluated to be true or false to identify whether a program location (*e.g.*, a function call invocation) is free from vulnerability or not. In contrast, a constraint-based approach solves a set of constraints for further analysis. We also note that tainted data flow analysis might rely on annotating

program code such as function prototypes (*e.g.*, [172]). However, these annotations are not involved in generating constraints like the annotation-based approaches. Rather, the annotation facilitates comparing an expected data type (annotated) with an identified data type (through static analysis) to infer vulnerabilities.

Annotations can be specified by expressions supported by an implementation language. For example, Dor *et al.* [116] detect all string manipulation errors that might lead to BOF vulnerabilities by specifying contracts (or annotations) through C expressions. Contracts include preconditions, postconditions, and specific side effects. After adding annotations, a source to source semantic preserving transformation of a given procedure is performed. The converted program code generates errors if contracts are violated. However, the pre and postconditions expressed through implementation language expressions can detect limited types of BOF vulnerabilities such as accesses to buffers after the null-termination bytes.

Several works have detected a wide range of BOF vulnerabilities either by introducing an annotation language or by extending an implementation language. For example, Hackett *et al.* [12] develop an annotation language named *SAL* that allows expressing buffer annotations to describe buffer interfaces through pointers. The annotation of buffer variables might include usage (*e.g.*, a buffer passed to a function and read from), optional (*e.g.*, pointer can be NULL), extent of an initialization (*i.e.*, a lower initialization bound), and capacity (*i.e.*, a lower capacity bound). In contrast, Evans *et al.* [114] annotate function parameters, return values, global variables, and structure fields inside tagged comments in C programs. For example, the */*@notnull@*/* tag before a parameter indicates that any value passed to the parameter should not be NULL. BOF vulnerabilities are detected by adding pre and postconditions to user defined and ANSI C library functions. For example, the library function *strcpy(s_1, s_2)* copies the source buffer s_2 to the destination buffer s_1. The precondition */*@requires maxSet(s_1) ≥ maxRead(s_2) @*/* generates an error message, if it cannot be satisfied at the call site by a checker. Here, the precondition indicates that the maximum index value that can safely access s_1 (*i.e.*, *maxSet(s_1)*) during a write operation must be greater than or equal to the maximum index value that can safely access s_2 (*i.e.*, *maxRead(s_2)*) during a read operation.

An annotation language that provides flexibility and features to annotate large amount of source code can reduce the burden of annotating program code. Yang *et al.* [78] develop an annotation language named MECA in this direction. For example, the statement "*annot tainted annotates ($variable)"* specifies that tainted annotation to be used to bind to any local and global variable, a function's parameter, and the return value of a function. MECA can bind multiple annotations with a variable (*e.g.*, checking both *src* and *dest* pointer parameters with the *len* parameter in a *memcpy(dest, src, len)* function call). Moreover, the language allows expressing conditional annotations (*e.g.*, annotate a variable as tainted based on specific parameters of function calls).

4.5. String pattern matching-based technique

The pioneer static analysis approaches (*e.g.*, [42, 44, 102, 140]) are based on simple string pattern matching technique. These approaches rely on a known set of library function calls that might cause vulnerabilities. A set of rules are developed which represent signature of vulnerable code patterns. In this technique, program code is tokenized to identify vulnerable pattern of strings that represent vulnerable function calls and arguments. For example, ITS4 [44] and Flawfinder [148] analyze the tokenized program code to detect potential vulnerabilities. Dekok [140] detect FSB vulnerabilities in *printf* family functions by examining patterns of vulnerable format function calls which include non constant format strings and format strings as the last argument.

A recent variation of string pattern matching-based technique is to scan executable program code to detect vulnerable function calls. For example, Tevis *et al.* [42] analyze portable executable (PE) files that run on Windows NT/XP to detect BOF vulnerabilities. They detect BOF vulnerabilities by identifying the presence of vulnerable ANSI C library function names in a symbol table and the occurrence of zero filled regions of 50 bytes or more. These regions can be used to load malicious code during BOF attacks.

4.6. Open issues

From the comparative analysis, we notice that introducing analysis sensitivity and performing analysis on fine granular levels of program code result in better detection of vulnerabilities. Most of the current approaches lack completeness, soundness, or both of the properties. Analysis precision and scalability is a tradeoff factor in current analysis approaches. Many approaches suffer from high false positive rates. Therefore, future static analysis approaches can focus in several directions such as detecting less unaddressed vulnerabilities and improving the completeness and soundness properties. From the discussion, we also notice that most of the analysis techniques are geared towards a subset of high level languages such as C, Java, and PHP. Thus, future works should address on developing analysis techniques for programs implemented in other common languages such as ASP and JavaScript.

From the discussions on inference techniques, we note that each inference mechanism is valuable from certain perspectives. For example, annotation-based mechanism is useful to verify that certain vulnerabilities are not present, whereas demand-driven and query-based tainting can help locating prioritized vulnerabilities that must be addressed immediately. Our study indicates that constraint, annotation, and string pattern matching-based inferences currently detect a limited type of vulnerabilities such as BOF, FSB, and SQLI. Thus, future works should improve the current techniques for detecting other vulnerability types such as XSS and CSRF. New approaches can also combine multiple techniques to detect unaddressed vulnerabilities and increase detection accuracies. Another research direction is to apply appropriate granularity and sensitivity for different inference techniques. Moreover, performing static

analysis based on the presence of improper sanitization APIs can be investigated further for different vulnerabilities.

5. Hybrid analysis

Although static analysis techniques detect vulnerable code in programs, they all suffer from a common disadvantage which is numerous false warning reports. As a result, a tester or programmer spends significant amount of time for examining warnings with actual input test cases. Thus, merely depending on the static analysis results might not be practically feasible to detect and fix all suspected vulnerabilities in large scale programs. Many approaches have attempted to improve this situation by automatically examining suspected vulnerable code. These approaches combine static analysis techniques with complementary dynamic analysis techniques [157]. A dynamic analysis is an active analysis technique where program states are observed to confirm vulnerability exploitations [169]. Program states include a wide range of entities during a program execution such as the values of declared variables, structure fields, and contents of memory locations. The combination of static and dynamic analysis is known as *hybrid analysis* technique.

In general, a hybrid analysis technique strives to avail the advantages of both static and dynamic analyses. The static analysis identifies the locations of program code that need to be analyzed during program executions to verify actual exploitations of vulnerabilities. This reduces the number of suspected vulnerabilities reported by a static analysis technique that need to be examined further. Note that the vulnerability mitigation capability of any hybrid analysis technique fairly depends on the inference technique of a static analysis. For example, a static analysis might not identify program code as vulnerable. As a result, the subsequent dynamic analysis cannot find any security breaches in that code. Moreover, obtaining or generating required test cases that can reach the vulnerable locations during a program execution might not be directly addressed by a hybrid analysis technique. Rather, complementary mitigation techniques such as test case generation can address that issue.

In the following section, we compare hybrid analysis works that mitigate program security vulnerabilities based on seven features: *static inference*, *analysis granularity*, *static information*, *dynamic analysis objective*, *program state utilization*, *vulnerability coverage*, and *language*. Note that the lack of discussion on “completeness” and “soundness” of the underlying static analysis techniques by the original authors of the related works prohibit us to include these features in our comparative discussions. The most important step for a hybrid analysis is to actively examine program entities to confirm suspected vulnerabilities based on dynamic analysis objectives. The objectives are related to program states at runtime. Thus, we are motivated to classify hybrid analysis works based on dynamic analysis objectives

into four types: *program operation*, *code structure integrity*, *code execution flow*, and *unwanted value*. They are described in Sections 5.2-5.5. Finally, we discuss open issues in Section 5.6.

5.1. Comparison of hybrid analysis approaches

We compare hybrid analysis works based on seven features: static inference, analysis granularity, static information, monitoring objective, program state utilization, vulnerability coverage, and language addressed. A summary of the comparison is shown in Table 5, while we describe these features in the following paragraphs.

Static inference: This feature indicates the underlying inference mechanism by which a location is identified as vulnerable without executing the code. We reuse the same categorization for inference mechanisms that have been discussed in Sections 4.2-4.5 (*i.e.*, tainted data flow, constraint, annotation, and string pattern matching). Unlike traditional static analysis technique, where inference mechanisms are applied to detect vulnerable code, a hybrid analysis often applies a light weight static analysis, which primarily discovers information flow.

Table 5 shows that three types of inferences are used: *tainted data flow*, *string pattern matching*, and *untainted data flow-based* analysis. Tainted data flow and string pattern matching-based techniques have been discussed in Section 4. Here, we discuss untainted data flow-based inference that is used in several works (*e.g.*, [35, 62]). Untainted data flow represents the extraction of legitimate information that is valid in sensitive program operations (*e.g.*, SQL query execution). For example, Castro *et al.* [35] identify legitimate instruction sets that can define (or assign) the value of a variable. Balzarotti *et al.* [62] define intended workflows by connecting valid views (or program paths). The common characteristic between an untainted and a tainted data flow is that both of them track flows of data. However, an untainted data flow-based technique tracks valid data, whereas a tainted data flow-based technique tracks suspected data. Sometimes, untainted data flow is also mentioned as “positive tainted data flow”.

Analysis granularity: This feature identifies what granularity level of program code is used in a static analysis phase. The granularity levels vary widely. These include statement (*e.g.*, [81]), control flow (*e.g.*, [41, 62]), dataflow (*e.g.*, [35]), and combined granularities such as statement and control flow (*e.g.*, [16]).

Static information: This feature indicates the information gathered during a static analysis which is used in a dynamic analysis. In Table 5, we observe that collected information depend not only on vulnerability types, but also on languages and analysis techniques. For example, BOF can be detected by identifying allocated memory blocks through MAT [19], unsafe pointers used in memory write operations [101], and unsafe library function calls having pointer arguments [16]. However, injection type vulnerabilities (*e.g.*, SQLI and XSS) can be detected by identifying trusted strings [83], hotspots [81], finite state machines of SQL queries [106, 107], and the set of statements involved in generating malicious outputs [41].

Table 5: Comparison summary of hybrid analysis works on program security vulnerabilities

Work	Static inference	Analysis granularity	Static information	Dynamic analysis objective	Program state utilization	Vulnerability coverage	Language
Aggarwal *et al.* [16]	Tainted data flow	Statement and control flow	Function calls having pointer and alias arguments	Program operation (unsafe function calls)	Addition (suspicious scores for pointers and aliases passed as function arguments)	BOF	C
Castro *et al.* [35]	Untainted data flow	Data flow	List of variable that might modify values	Program operation (memory read)	Extraction (allowable definition set)	BOF, FSB	C
Kumar *et al.* [19]	String pattern matching	Statement	Memory allocation table (MAT)	Program operation (memory read and write)	Extraction (base and size of allocated memory blocks)	BOF, DAP, MEL	x86
Monga *et al.* [41]	Tainted data flow	Inter-procedural control flow	Program paths and statements that might modify sensitive sinks	Unwanted input (meta characters)	Addition (store taint labels of untrusted inputs)	SQLI, XSS	PHP
Balzarotti *et al.* [62]	Untainted data flow	Intra and inter-procedural control flow	Module views and intended workflows	Code execution flow	Extraction (valid program paths)	SQLI, XSS	PHP
Halfond *et al.* [81]	String pattern matching	Statement	Valid SQL query models	Code structure (SQL)	Extraction (valid SQL query models)	SQLI	JSP
Halfond *et al*. [83]	String pattern matching	Statement	Trusted strings	Code structure (SQL)	Addition (mark to hard coded strings)	SQLI	Java byte code
Johns *et al.* [87]	String pattern matching	Statement	Programmer written script code	Unwanted value (injected script code)	Addition (replace known code with tokens)	SQLI, XSS	PHP
Wei *et al.* [106]	String pattern matching	Control flow	Identify SQL queries	Code structure (SQL)	Extraction (a FSM of queries before input inclusions)	SQLI	N/A
Muthuprasanna *et al.* [107]	String pattern matching	Inter-procedural control flow	SQL finite state machine (SQL-FSM)	Code structure (SQL)	Extraction (SQL-FSM)	SQLI	Java
Lucca *et al.* [127]	Tainted data flow	Control flow	Tainted sources and sensitive sinks	Unwanted value (meta characters)	Addition (expected output messages to analysis functions)-	XSS	ASP
Ringenburg *et al.* [142]	String pattern matching	Data flow	White listed memory locations	Program operation (format function calls)	Extraction (valid memory addresses)	FSB	C
Yong *et al.* [101]	String pattern matching	Statement	Unsafe pointer dereferences and legitimate memory locations	Program operation (memory read or write)	Addition (tagging memory locations)	BOF, DAP	C

Dynamic analysis objective: This feature describes what attribute of a program is checked at runtime to detect attacks during dynamic analysis phase. As can be seen from Table 5, *program operation* (*e.g.*, [16, 35]), *code structure* (*e.g.*, [81, 106]), *code execution flow* (*e.g.*, [62]), and *unwanted value* (*e.g.*, [87, 127]) are monitored in different hybrid approaches. *Program operation* checks whether memory allocations, releases, and accesses are performed in valid memory regions or not. *Code structure integrity* attribute validates the known structure of program code during runtime. The *code execution flow* checks whether an intended program path can be altered or not without modifying related program states (*e.g.*, session id, cookie). The *unwanted value* attribute checks the presence of unwanted values in program inputs and outputs. We provide details of dynamic analysis objectives in Sections 5.2-5.5. Note that the objectives

are related to program states at runtime and they are a subset of program monitoring objectives (to be discussed in Section 7).

Program state utilization: To detect an attack, a program state needs to be compared with a known program state under attack. These known states are derived through information extraction or addition of program sources. We denote such derivation characteristic as *program state utilization*. Depending on the types of vulnerabilities and dynamic analysis objectives, we divide program state utilization into two categories: *information extraction* (*e.g.*, [35, 62, 81]) and *information addition* (*e.g.*, [16, 41, 87]). The sixth column of Table 5 shows the category of *program state utilization* for each of the work along with particular information.

Program code can be analyzed to extract useful information to detect attacks during a dynamic analysis phase. For example, Ringenburg *et al.* [142] extract whitelists to prevent FSB attacks that write to invalid memories through malicious *%n* specifiers. Castro *et al.* [35] extract reaching definition instruction set for every variable usage. A definition table is updated on every memory write operation (*i.e.*, definition). The table can detect BOF and FSB attacks which define variables not included in the reaching definition instruction set. Balzarotti *et al.* [62] extract all valid paths that a user traverses during a program's execution. During code injection attacks (*e.g.*, SQLI, XSS), these paths are not traversed. Halfond *et al.* [83] mark trusted strings in programs which are mainly hard coded strings written by programmers and include SQL keywords, operators, and literals. These trusted strings help to detect SQLI attacks by checking if certain parts of SQL queries (*i.e.*, keywords and operators) include trusted strings (*i.e.*, no attack) or not (*i.e.*, attack). Extracting valid SQL query models through Finite State Automata (FSA) [106, 107] and Non-Deterministic Finite Automata (NDFA) [81] is widely used to detect SQLI attacks. In these cases, queries are checked to be valid based on the models.

Information can be added in program code which can be retrieved later and compared with a future program state to detect attack. Aggarwal *et al.* [16] assign suspicion score for pointers which are declared locally in functions or passed as arguments to detect BOF. Yong *et al.* [101] mark memory allocation and free operations, global, and static variables with "appropriate" and "inappropriate" to tag legitimate and illegitimate memory locations, respectively in a dynamic analysis. Lucca *et al.* [127] add expected output messages to confirm XSS attacks in suspected files (or pages). These files are supplied with attack input cases and checked for expected responses. Monga *et al.* [41] detect XSS in PHP-based programs by adding tainted labels to untrusted input sources and these labels are propagated to variables derived from these sources during computation. Kumar *et al.* [19] add a memory allocation table (MAT) to keep track of memory blocks and sizes to be used later for detecting BOF vulnerabilities. Moreover, they mark each memory block as active or inactive to identify MEL. Program output locations are checked for the presence of malicious meta characters in tainted variables. Johns *et al.* [87] identify keywords in programmer

written script code and replace them with masks (random tokens). The masked keywords are unmasked before generating responses to prevent code injection exploits.

Vulnerability coverage: This feature indicates what vulnerabilities are covered in a hybrid analysis. Table 5 shows that most approaches address web-based vulnerabilities such as SQLI and XSS (*e.g.*, [41, 81, 87]). Moreover, BOF, FSB, and DAP vulnerabilities have been addressed by several works (*e.g.*, [16, 19, 142]). However, few works have addressed MEL [19] and NLD vulnerabilities.

Language: This feature indicates programming languages (related to implementations) that are supported in hybrid analysis approaches. As can be seen in Table 5, most of the works analyze code written in either server side scripting languages (*e.g.*, PHP, JSP) or in procedural languages (*e.g.*, C). Very few works analyze executable code [19]. A motivation behind analyzing high level scripting languages is the availability of static analysis tools for those languages. For example, Balzarotti *et al.* [62] perform intra and inter-procedural analysis with the help of *Pixy* tool [88] that analyze PHP code. Similarly, procedural languages (*e.g.*, C) have high number of static analysis tools and algorithms (*e.g.*, points-to analysis algorithms).

5.2. Program operation

In this technique, static analysis phase identifies valid memory locations that can be accessed, read, or written during program executions. During a dynamic analysis phase, these locations are checked for any operations performed outside the valid memory locations. For example, Castro *et al.* [2] identify a set of instructions (or statements) that might modify a variable value (*i.e.*, definition) for each variable use (*i.e.*, read) in the dataflow graph of a program during static analysis. During the dynamic analysis, they check whether a value read has been defined by a legitimate set of definition or not. Ringenburg *et al.* [110] perform static data flow analysis to generate white listed addresses (or valid addresses) where writing operations can be performed during format function calls. Any modification outside the registered addresses during format function calls are identified in an active analysis. Yong *et al.* [101] identify dangerous pointer dereferences and legitimate memory locations which pointers can point to during the static analysis. Programs are instrumented to check whether dereferences are pointing to legitimate memory locations or not. If any pointer is used to write (or free) an illegitimate location, then the instrumented program halts further execution.

Hybrid analysis can be applied to executable program code. However, a dynamic analysis is performed first to map executable instructions with virtual addresses and identify program operations related to vulnerability exploitations. Static analysis is performed by decompiling the executable into object code (*e.g.*, assembly). The object code can be analyzed to obtain attributes related to vulnerabilities (*e.g.*, allocated memory sizes). Finally, programs are executed with test cases to check whether vulnerabilities

might be exploited based on the gathered information. Kumar *et al.* [19] apply hybrid analysis of executable C programs to identify BOF vulnerabilities.

5.3. Code structure integrity

In this approach, a static analysis technique is applied to extract valid code structure. During a dynamic analysis, it is checked whether runtime program code conforms to the structure or not. Depending on the types of attacks, the code structure can be modeled with different formal models. For example Halfond *et al.* [81] apply Non Deterministic Finite Automata (NDFA) to model valid queries in each hotspot (*i.e.*, a location where a SQL query is issued to a database engine). In an NDFA, a transition might contain a SQL token, a delimiter, or an input string value. Muthuprasanna *et al.* [107] perform Java string analysis on all the hotspots to construct NDFA, where transitions occur at the character level of a string. Each NDFA is converted to a SQL Finite State Machine (SQL-FSM) where transitions are either SQL keywords or input string variables. In both of the works, if a query is not consumed by the model, a SQLI vulnerability is warned. Wei *et al.* [106] detect SQLI vulnerabilities in stored procedures by identifying queries that might be generated during a program's execution. They develop Finite State Machine (FSM) models of SQL queries, where a transition from one state to another state occurs for a SQL keyword. During a dynamic analysis phase, query statements with user inputs are checked against the FSM. If a query is rejected (or not consumed by an FSM) then an error is flagged. A recent variation of code structure integrity approach is to apply positive tainting on trusted strings (*e.g.*, hard coded strings written by programmers, SQL keywords, operators, and literals) during the static analysis and to check whether runtime generated code are constructed from these trusted strings or not. Halfond *et al.* [83] apply this approach to detect SQLI attacks in Java byte code.

5.4. Code execution flow

In this approach, static analysis is applied to identify valid program execution paths which share a common program state. These paths also represent sequence of operations to perform functionalities. During a dynamic analysis, it is checked whether it is possible to jump from one execution path to another or not. Balzarotti *et al.* [62] apply this approach. They analyze programs to summarize valid execution paths (or *views*) where a path comprises of more than one web page to perform a desired functionality (*e.g.*, authentication). The second stage of the analysis constructs intended workflows. A workflow connects a source view with a target view provided that a hyperlink present in the source view is referenced in the target view and parameters provided through a link is extracted in a target view. The summarized workflow is represented by a graph, where a node contains a page and corresponding view, and an edge represents possible web-based operations (*e.g.*, form submission, redirection). A model checker identifies whether any unintended workflow (*i.e.*, vulnerabilities) is present in the summarized

workflow. A detection algorithm is used to check whether from any view it is possible to jump to another view not included in the intended workflow.

5.5. Unwanted value

The static analysis phase identifies potential locations where inputs or tainted values might reach and perform sensitive operations (sinks). When a program execution reaches a sink, all tainted inputs are checked for suspicious meta characters (*e.g.*, single quotation) that can be used to exploit vulnerabilities such as SQLI and XSS. For example, Monga *et al.* [41] detect XSS in PHP-based programs. Similarly, Lucca *et al.* [127] identify XSS vulnerabilities in Active Server Pages (ASP)-based programs. However, these approaches stop program executions while detecting unwanted values. An alternative approach is to prevent the inclusion of unwanted values in program outputs. This is common for web-based programs that generate HTML outputs and need to avoid unwanted values (*e.g.*, JavaScript code) to avoid attacks (*e.g.*, XSS). For example, Johns *et al.* [87] separate programmer written scripts which are static string constants in program code. A string is analyzed to identify keywords (*e.g.*, HTML attributes, JavaScript words) and replace them with masks (random tokens). These masked keywords are unmasked before generating responses by browsers. As a result, unwanted injected code is replaced with corresponding encoded form.

5.6. Open issues

From the discussion of hybrid analysis approaches, we observe that program operation and code integrity are the two widely used dynamic analysis objectives. Most of the works apply string pattern matching and tainted data flow analysis along with static data or control flow analysis. Very few works perform dynamic analysis followed by static analysis. Moreover, CSRF vulnerabilities have not been addressed by current approaches. The study shows that static analysis influences dynamic analysis stage for most of the works. The precision of static analysis is important and needs to be carefully considered before applying in a hybrid approach. Future works should explore how assumptions behind static analyses influence the vulnerability detection in dynamic analysis stages. We notice that very few approaches employ untained data flow-based static analyses and code execution flow-based dynamic analysis objectives. We believe that employing suitable dynamic analysis objectives and static inference techniques can not only improve the effectiveness of hybrid analysis, but also detect a wide range of vulnerabilities.

6. Other mitigation techniques

In this section, we first describe *secure programming* guidelines. Then we briefly discuss two other approaches that are primarily applied in the maintenance phase: *program transformation* and *patching*.

6.1. Secure programming

Program security breaches can be blamed to programmers who overlook possible vulnerabilities in their implemented code. Moreover, the lack of understanding of an implementation language features (*e.g.*, data types, libraries) contributes in writing code that is vulnerable [160]. Secure programming (or coding) approaches are intended to provide supports for implementing programs in vulnerability free ways and can be considered as the first line of defense to avoid program security breaches. Writing secure code helps reducing subsequent costs of detecting and fixing security vulnerabilities at later stages. Secure programming approaches provide supports in the form of safe APIs, libraries, aspects, and filters.

Table 6: A brief comparison summary of the secure programming approaches

Work	Type of programming support	Vulnerability coverage	Programming language
Speirs *et al.* [38]	API	BOF	C
Tsai *et al.* [43]	Library	BOF	C
Hermosillo *et al.* [85]	Aspect	SQLI, XSS	Java
Juillerat *et al.* [89]	Library	SQLI, XSS	Java
Robbins *et al.* [144]	Library	FSB	C
CSRFGuard [168]	Filter	CSRF	Java

APIs are system calls which allow programmers to perform checks in the code to avoid vulnerabilities. For example, *ptrbounds* [38] is a kernel level API that helps obtaining the writable upper and lower bounds for a given pointer data type to avoid BOF vulnerabilities in C programs. Most of the vulnerabilities can be mitigated by applying safe libraries. For example, the *Libsafe* [43] intercepts all vulnerable library functions that might result BOF attacks. The library code checks stack to identify the maximum number of bytes that can be safely written for each destination buffer. Similarly, *Libformat* library [144] contains improved version of format functions to prevent FSB vulnerabilities by checking whether supplied format strings are in writable memory locations and contain suspicious specifiers or not. Moreover, safe libraries capture the structure of strings that might be used in SQL queries and HTML outputs to prevent code injection attacks. In this case, the library disallows writing of SQL queries and HTML structures in program code directly. Juillerat *et al.* [89] develop such a library named *Stones* to prevent SQLI and XSS vulnerabilities in web programs written in Java. Secure programming approaches are adapted to different programming paradigms. For example, the aspect oriented programming [85] allows weaving special code through *pointcuts* (*i.e.*, a pattern of method or function calls with common signatures) and *advices* (*i.e.*, additional code to be added before or after function calls). Vulnerabilities are caused by invalidated inputs which can be checked through intercepting inputs at pointcuts and detecting (or preventing) the presence of malicious inputs in advices.

To protect server side programs from CSRF attacks, filters can be added [168]. A filter is a mapping between resources (*e.g.*, a server script page that performs a sensitive operation) and corresponding code that intercept HTTP requests to detect possible CSRF attacks. The idea is to verify a request by comparing

a unique request token for an HTTP parameter value with a token stored in a session table. If there is a mismatch, the request is considered as forged and part of a CSRF attack. The filter can redirect a user to an error page. However, if the token matches with the stored value, then a request is forwarded to a server program which generates a response page. The response page is searched for HTML forms and links, and inserted with appropriate unique token parameter values for further prevention of CSRF attacks.

We provide a comparison summary of secure programming approaches in Table 6 which includes three features: *type of programming support*, *vulnerability coverage*, and intended *programming language*. We notice that most secure programming approaches are intended to mitigate a limited type of vulnerabilities namely BOF, FSB, SQLI, and XSS. Moreover, C and Java are the two programming languages having ample supports for secure programming.

6.2. Program transformation

Program security vulnerabilities can be mitigated in a post release stage which is commonly known as maintenance phase [156]. Although program features can be extended, removed, or modified in this stage, we only focus on activities that are intended to fix security vulnerability breaches in the implemented code. Program transformation is one of the most widely used approaches in this direction which removes vulnerabilities by applying structured modification of program source code. In other words, source code of a vulnerable program is transformed to a vulnerability free program. We categorize program transformation related works into two types: *source to source translation* [25, 45, 48, 112, 147] and *code rewriting* [120, 133, 137].

In source to source translation, a program source is taken as input and an enhanced program in the same language is generated automatically. A source to source translation can be implemented in different ways such as using a functional programming language that can replace certain patterns of code with desired patterns (*e.g.*, TXL [150]). However, we restrict our discussion on the enhancement added in program code to mitigate vulnerabilities. We divide program transformation approaches into three categories based on enhancement type: *shifting data to safe region, adding security checks, and enriching program data*. The *shifting data to safe region* approach shifts vulnerable data into safe regions. For example, the *SecureC* translator [25] translates a C program into security-enhanced source code. The translation shifts a buffer memory location into a shadow stack (a read only page, except the location of buffer) to prevent BOF attacks through out of bound writing. Moreover, C programs are enhanced to reposition each stack buffer into heap area [147] to avoid return address corruption through BOF. The *adding security checks* add necessary checks to avoid vulnerabilities. For example, every buffer index and pointer dereference can be preceded by an assertion to prevent BOF attacks [45]. The *enriching program data* approach stores additional information to track valid memory related information. The information is used to prevent invalid memory accesses in program code. For example, to detect BOF and DAP vulnerabilities in C

programs [48, 112], pointer data types are extended to contain additional information such the base and the size of memory objects and the status of memory objects (*i.e.*, allocated or freed).

Table 7: A comparison summary of program transformation related works for mitigating security vulnerabilities

Work	Transformation type	Vulnerability coverage	Programming language
Nishiyama *et al.* [25], Wang *et al.* [45], Dahn *et al.* [147]	Source to source translation	BOF	C
Xu *et al.* [48], Austin *et al.* [112]	Source to source translation	BOF, DAP	C
Reis *et al.* [120]	Code rewriting	BOF	JavaScript
Yu *et al.* [133], Ofunoye *et al.* [137]	Code rewriting	XSS	JavaScript

The *code rewriting* technique is used for rewriting the output of a program as opposed to program source code directly. The output is another high level program that is interpreted or executed. For example, a server script code written in PHP generates HTML code which might contain BOF vulnerabilities due to arbitrary large HTML attribute identifiers [120]. The code can be rewritten at the browser to avoid unexpected results while rendering the page. Similarly, vulnerable JavaScript code can be rewritten to stop XSS attacks [133, 137].

We provide a brief comparison summary of program transformation related works in Table 7 according to *transformation type*, *vulnerability coverage*, and *programming language* used. It is obvious that only BOF, DAP, and XSS have been mitigated using program transformation techniques. Moreover, current approaches have addressed C and JavaScript programs whose sources are transformed to safe equivalents.

6.3. Patching

Patching is a widely used approach in program maintenance phase to fix reported bugs or errors so that modified programs conform to expected functionality, performance, and quality [156]. However, we focus on corrective maintenance which are intended to fix reported security breaches in programs. A patching technique identifies vulnerable code and modifies the program to remove vulnerabilities. Unlike other proactive (*e.g.*, static and hybrid analysis, testing) techniques, patches are generated after the occurrence of attacks. We classify patching works into two types: *source code* and *environment* patching.

The source code patching technique analyzes program source code to identify vulnerable statements that need to be fixed. The common practice is replacing vulnerable code with equivalent safe code. For example, a SQL query statement can be written as a *PreparedStatement* which does not allow the modification of query structure during runtime to prevent SQLI [74, 115]. Moreover, unsafe library function calls can be replaced with their safe equivalents and added vulnerability checks to avoid BOF [37]. A variation of source code patching is to guide patching locations by data flow analysis on the source code of the functions to identify related statements that contribute to vulnerabilities [22]. Patches are generated by determining the size of buffers. Out of buffer reads are redirected within buffers. Out of

bound writings are discarded by replacing unsafe library function calls with safe function calls or skipping through out of bound checking.

Table 8: A brief comparison summary of patching approaches for mitigating security vulnerabilities

Work	Patching type	Vulnerability coverage	Programming language
Gao *et al.* [149], Novark *et al.* [26]	Environment	BOF, DAP	C
Lin *et al.* [22], Smirnov *et al.* [37]	Source code	BOF	C
Lin *et al.* [53]	Environment	BOF	x86
Dysart *et al.* [74]	Source code	SQLI	PHP
Lin *et al.* [103]	Environment	SQLI, XSS	N/A
Thomas *et al.* [115]	Source code	SQLI	Java

In an *environment patching* approach, program environment is modified which might include memory layout, external library addresses, etc. This approach can help in patching programs without stopping their executions. For example, patching to prevent BOF attacks can be performed by redirecting a vulnerable function (*e.g.*, *strcpy*) with an equivalent non-vulnerable function (*e.g.*, *strncpy*) [53] by changing the GOT (Global Offset Table) entries. Many approaches analyze program artifacts that are generated due to vulnerability exploitations. These artifacts help identifying changes to be made in an environment. For example, the crashed program can be analyzed to change environment by adding a canary value at the end of a buffer to prevent BOF [149]. Moreover, the heap image of a crashed program can be dumped to learn the magnitude of the overflowed bytes and memory object de-allocation call sites [26]. The information is used to pad objects and defer object de-allocations to prevent BOF and DAP vulnerabilities, respectively. For web-based programs, patching can be performed in proxies located between a server and a client. In this case, a proxy might be enhanced with input filters (input validation functions) [103] to detect malicious inputs from client side of a program.

We provide a brief comparison summary of patching works that fix program security vulnerabilities in Table 8, where we classify the works based on *patching type*, *vulnerability coverage*, and *programming language*. We note that BOF, DAP, SQLI, and XSS vulnerabilities have been addressed by current approaches. Moreover, most patching works are related to the implementation of C programs. Very few works generate patches for programs whose sources are available in high (*e.g.*, Java and PHP) and intermediate languages (*e.g.*, Java byte code and x86).

7. Monitoring

Vulnerabilities might be exploited at runtime through successful attacks. Given that, it is very important to have a tool which can be used for online monitoring of programs in the operational stage. In a monitoring approach, vulnerability symptoms are checked by comparing the current state of a program with a known state under attack. When there is a match (or mismatch) between the two states, a successful exploitation of a particular vulnerability (or an attack) occurs. The program might be stopped for further

execution. A monitoring tool can help to mitigate the consequences of some vulnerability exploitations. Moreover, it can be utilized in a complementary fashion with other vulnerability prevention techniques such as static analysis (*e.g.*, [12, 20]) and testing (*e.g.*, [52, 86, 91]).

In this section, we compare and contrast program security vulnerability monitoring approaches in order to provide a classification based on the following seven identified characteristics: *monitoring objective, program state utilization, implementation mechanism, environmental change, attack response, vulnerability coverage*, and *language* [7, 8, 10]. The classification is provided in Table 9. We describe these characteristics in Section 7.1. We then classify these works based on '*monitoring objective'* which is a very important characteristic for any monitoring approach. The objectives are *program operation*, *code execution flow and origin, code structure, value integrity*, *unwanted value*, and *invariant*. They are discussed in Sections 7.2-7.7. We discuss open issues in Section 7.8.

7.1. Comparative analysis of monitoring approaches

Monitoring objective: This characteristic indicates program properties during execution which need to be monitored to detect attacks. We classify the works into six categories which are shown in the second column of Table 9. These include program operation (*e.g.*, [31, 163]), code execution flow and origin (*e.g.*, [63, 69, 70]), code structure (*e.g.*, [66, 79]), value integrity (*e.g.*, [24, 30, 68, 129]), unwanted value (*e.g.*, [59, 111, 123, 125]), and invariant (*e.g.*, [13]). Note that some approaches employ multiple monitoring objectives such as program operation and code execution and origin [69, 70]. We discuss these objectives in Sections 7.2-7.7.

Program state utilization: To detect an attack at runtime, a program state needs to be compared with a known program state under attack. These known states might be derived from program states through information extraction, addition, or modification. We denote such derivation characteristic as program state utilization. Depending on the nature of vulnerabilities and monitoring objectives, we divide program state utilization into three categories: *information extraction*, *information addition*, and *information modification*. The third column of Table 9 shows the category of program state utilization for each of the work along with particular information.

Program code can be analyzed to identify (or extract) useful information to detect attacks during runtime. For example, a list of known JavaScript code [125] can be developed to detect XSS attacks at runtime. Dhurjati *et al.* [18] develop a pool allocation table which contains a set of homogenous objects. Each set represents memory objects related to a data type that a pointer might point during a program execution [18]. These sets are used to detect BOF and DAP attacks through pointer dereferences. Cowan *et al.* [141] extract the number of arguments in format function calls to detect FSB attacks.

Table 9: Comparison summary of approaches for monitoring security vulnerability exploitations

Work	Monitoring objective	Program state utilization	Implementation mechanism	Environmental change	Attack response	Vulnerability coverage	Language
Berger *et al.* [58]	Program operation (memory read and write)	Information modification (memory locations)	Spatial rearrangement of memories	Implementation (DLL modification)	Program termination	BOF, DAP	C
Chiueh *et al.* [129]	Value integrity (return address)	Information addition (return address)	Code instrumentation (compiler modification)	Implementation (kernel system call patches)	Program termination	BOF	C
Fetzer *et al.* [163]	Program operation (memory allocation)	Information addition (memory block sizes)	API hook	Implementation (DLL modifications)	Error messages and program termination	BOF	C
Cowan *et al.* [68]	Value integrity (return address)	Information addition (canary value)	Code instrumentation (compiler modification)	Program state utilization (prologue and epilogue)	Attack handler function execution	BOF	C
Gupta *et al.* [126]	Value integrity (return address)	Information addition (return address and stack frame)	Binary rewriting	Program state utilization (parallel stack frame)	Program state recovery	BOF	C
Han *et al.* [13]	Invariant (legitimate API function call sequences)	Information addition and extraction (function names, stack size, and return address)	API hook	Implementation (add DLL functions in programs)	Program termination	BOF	C
StackShield [77]	Value integrity (return addresses)	Information addition (return addresses in global variables)	Code instrumentation (compiler modification)	Program state utilization (modify DATA section)	Program termination	BOF	C
Aggarwal *et al.* [105]	Value integrity (return address, setjmp, longjmp)	Information addition (return addresses in a monitor agent)	Code instrumentation	Implementation (program runs under an agent)	Warning message generation	BOF	C
Kohli *et al.* [108]	Unwanted value (format string specifier)	Information addition (a lightweight hash value of return addresses)	Binary rewriting	Implementation (DLL calls for vulnerable function)	Program termination	FSB	x86
Prasad *et al.* [109]	Value integrity (return address)	Information addition (return addresses in prologues)	Binary rewriting	Program state utilization (return address repository)	Program termination	BOF	x86
Madan *et al.* [24]	Value integrity (return address)	Information addition (encrypted return address)	Code instrumentation (compiler modification)	Program state utilization (prologue and epilogue)	Program termination	BOF	x86
Kiriansky *et al.* [110]	Code execution flow and origin	Information addition (policies that map allowable control transfers with instructions, source and destination)	Dynamic code optimizer extension	Performance (fast lookup of branches by saving addresses in cache memory)	Warning message generation	BOF, FSB	x86
Newsome *et al.* [102]	Program operation (memory write)	Information addition (taint untrusted data source)	Code instrumentation	Performance (save code block in cache)	Invocation of taint analyzer	BOF, FSB	x86
Dhurjati *et al.* [18]	Program operation (memory read and write)	Information extraction (homogenous object sets or pools for each pointer)	Code instrumentation (compiler modification)	Program state utilization (memory pools and free blocks)	Program termination	BOF, DAP	C
Pyo *et al.* [30]	Value integrity (return address)	Information addition (encrypted return address)	Code instrumentation (compiler modification)	Program state utilization (prologue and epilogue)	Attack handler function execution	BOF	C

Work	Monitoring objective	Program state utilization	Implementation mechanism	Environmental change	Attack response	Vulnerability coverage	Language
Rinard *et al.* [31]	Program operation (memory write)	Information addition (hash tables store out of bound data)	Code instrumentation (compiler modification)	Performance (LRU cache to store hash tables)	Program execution continuation	BOF, DAP	C
Salamat *et al.* [32]	Value integrity (similar output in two programs)	Information modification (duplicate program having stack growth in reverse direction)	Code instrumentation	Implementation (system call synchronizations)	Program terminations	BOF	C
Kiciman *et al.* [104]	Program operation **(**JavaScript code execution**)**	Information addition (policies that map instrumentation points and expressions)	Proxy-based tool	Implementation (a proxy with JavaScript parser is added)	Warning message generation	MEL	JavaScript
Suh *et al.* [40]	Code execution flow and origin	Information addition (tag for untrusted input values)	Code instrumentation	Program state utilization (program context switch)	Program termination	BOF, FSB	N/A
Zhou *et al.* [54]	Invariant (instruction sets accessing memories)	Information extraction (instruction sets related to memory accesses)	Code instrumentation (compiler modification)	Performance (*Check Look aside Buffer* to store most recently accessed objects)	Warning message generation	BOF	C
Zhu *et al.* [55]	Value integrity (function pointer)	Information addition (encrypted function pointers in memories)	Code instrumentation (compiler modification)	Implementation (compiler source code modification)	Warning message generation	BOF	C
Alfantookh *et al.* [59]	Unwanted value (SQL code and meta character)	Information extraction (know characters related to SQLI attacks)	Proxy-based tool	Implementation (adding a filter in the IIS server)	Request blockage	SQLI	N/A
Bandhakavi *et al.* [63]	Invariant (parse tree of SQL queries)	Information extraction and addition (save parse trees of intended queries in database)	Code instrumentation	Implementation (code optimization framework)	Warning message generation	SQLI	JSP
Buehrer *et al.* [66]	Code structure (SQL query)	Information addition (pre and postpend SQL queries with random keys)	Code instrumentation	Implementation (static class addition)	Request blockage	SQLI	Java
Clause *et al.* [69]	Program operation, code execution flow and origin	Information addition (taint information of program variables)	Code instrumentation	Program state utilization (bit vector to be saved outside program)	Attack handler function execution	BOF, FSB, SQLI	x86
Dalton *et al.* [70]	Program operation, code execution flow and origin	Information addition (tainted value to program memories and input data)	Code instrumentation	Program state utilization (save register, cache, and memory locations while context switch)	Attack handler function execution	BOF, SQLI, XSS	N/A
Gaurav *et al.* [79]	Code structure (x86 opcode)	Information modification (encrypt opcode with secret key)	Modified processor	Implementation (jump to even addresses)	Runtime exception throwing	BOF	x86
Boyd *et al.* [94]	Code structure (SQL query)	Information addition (random integer after SQL keywords)	Proxy-based tool	Implementation (add a proxy server)	SQL query blockage	SQLI	N/A
Iha *et al.* [122]	Program operation (HTML page generation)	Information modification (inputs are separated from DOM nodes)	Browser extension	Program state utilization (store bind value data into cache)	Malicious script blockage	XSS	N/A

Work	Monitoring objective	Program state utilization	Implementation mechanism	Environmental change	Attack response	Vulnerability coverage	Language
Ismail *et al.* [123]	Unwanted value (special HTML characters)	Information extraction (identify meta characters)	Proxy-based tool	Implementation (proxy server extension)	Attack string encoding and storing	XSS	JavaScript
Jim *et al.* [124]	Unwanted value (JavaScript code)	Information extraction (white listed JavaScript)	Browser extension	Implementation (browser parser modification)	JavaScript code blockage	XSS	JavaScript
Johns *et al.* [125]	Unwanted value (JavaScript code)	Information extraction (known JavaScript used in programs)	Browser extension	Implementation (browser parser modification)	Warning message generation	XSS	JavaScript
Bisht *et al.* [135]	Invariant (DOM of an output page)	Information extraction (parse tree of JavaScript code)	Browser extension	Implementation (browser scanner and tokenizer)	Error message generation	XSS	JSP
Cowan *et al.* [141]	Value integrity (format string)	Information extraction (argument count of functions)	Code instrumentation (compiler modification)	Implementation (compiler patch)	Program termination	FSB	C
Li *et al.* [143]	Program operation (memory read)	Information addition (canary value at the end of argument list)	Code instrumentation	Program state utilization (debug register saving while context switch)	Warning message generation and program termination	FSB	x86
Jones *et al.* [151]	Program operation (memory read and write)	Information addition (data structures containing base and size of objects)	Code instrumentation (compiler modification)	Implementation (DLL modification)	Runtime exception throwing	BOF, DAP	C
Ruwase *et al.* [113]	Program operation (memory read and write)	Information addition (data structures containing base and size of objects)	Code instrumentation (compiler modification)	Implementation (parser generates object and hash table)	Runtime exception throwing	BOF, DAP	C
Etoh *et al.* [152]	Value integrity (canary)	Information modification (pointer variables before buffer variables)	Code instrumentation (compiler modification)	Program state utilization (frame pointer and return address location change)	Program termination	BOF	C
Hastings *et al.* [153]	Program operation (memory read and write)	Information addition (a bit table to track if allocated memory is readable, writable, and both)	Code instrumentation	Program state utilization (bit table is added in object code)	Warning message generation	BOF, DAP, MEL	C
Lhee *et al.* [111]	Unwanted value (large sized inputs)	Information addition (data structure and type table to store buffer variable names and sizes)	Code instrumentation (compiler modification)	Program state utilization (type table is added in object code)	Program termination	BOF	C

Information can be added in a program's state and retrieved later with a future program state to detect attacks. For example, BOF attack detection requires storing return addresses of functions and function pointers adjacent to buffers in safe locations (*e.g.*, [13, 24, 129]). Many approaches add information in executable program code or environment. Such information can be variables (*e.g.*, a canary value before return address of a function [68, 143]), data structures and tables containing allocated buffer size information (*e.g.*, [31, 151]), and taint information of sensitive variables (*e.g.*, [63, 69, 70, 102]). Moreover, policies can be added to monitor allowable control transfers in executables. In this case, a policy might check program instructions with allowable sources and destinations [110]. Kiciman *et al.*

[104] apply policies to identify instrumentation points so that unintended behavior of JavaScript code can be detected and modified. Note that approaches applying invariant-based monitoring objective might employ both information addition and extraction (*e.g.*, [54]).

Some approaches modify current program states (*e.g.*, memory, code) to detect attacks. We identify two ways of performing modifications: *randomization* and *reorganization*. The randomization technique is used in detecting code injection attacks. In this case, non-randomized program code (injected by attackers) becomes different from randomized program code (written by a programmer) [66, 94]. Randomization can be performed on memory locations [58]. Some monitoring techniques reorganize program variables and environments to alter program behaviors during attacks. For example, the growth direction of two stack segments might be set opposite for two versions of a program [32]. This helps to detect BOF attack based on different outputs of two programs. Another example is to reorganize program variables including buffers in such a way so that sensitive variables are placed before stack buffers [152].

Implementation mechanism: This characteristic highlights the way of achieving monitoring objective. We identify eight unique implementation mechanisms as shown in the fourth column of Table 9. These include *spatial rearrangement* of memories [58], *code instrumentation* (*e.g.*, [66, 69, 143]), *API hooking* (*e.g.*, [13, 163]), *proxy-based tool* (*e.g.*, [94, 104, 123]), *browser extension* (*e.g.*, [122, 124]), *modified processor* [79], *binary rewriting* (*e.g.*, [108, 109]), and *dynamic code optimizer extension* (*e.g.*, [110]). We briefly describe them in the following paragraphs.

In a spatial rearrangement technique, memory blocks (or objects) are allocated at random locations and the objects are located apart to reduce BOF. Sometimes, allocated objects are initialized with random values to detect DAP attacks. The mechanism requires developing customized memory managers [58].

Code instrumentation is one of the widely used techniques where program code is enhanced with monitoring and prevention code. Additional code can be injected at the binary level [105]. Binary executable code can be instrumented to add checks (*e.g.*, taint a basic block before passing to a processor [70, 102]). Monitoring code can also be injected in the object code generated by a compiler [153]. Moreover, an operating system kernel can be modified to save registers containing tainted information for program data and control during context switches [69]. Code can be instrumented by modifying (or patching) compilers so that necessary monitoring code is automatically injected into programs. For example, monitoring of BOF requires saving return addresses to safe locations. This can be done by adding necessary code in a function prologue [129].

In API hook technique, vulnerable library function calls are intercepted. The objective of interception depends on vulnerability type. For example, function calls vulnerable to BOF can be replaced with non vulnerable function calls. Sometimes, buffer sizes and function calls are stored to be used at a later stage [13].

The proxy-based tool resides between a client and a server program. The role of the tool depends on monitoring objectives. For example, a proxy may perform *de-randomization* of code (*e.g.*, SQL key words) before submitting SQL queries to a database engine [94]. A proxy can also be used to instrument client side JavaScript code to prevent security breaches [104].

Browser extension enables to monitor client side program execution. In this case, browsers are enhanced with functionalities to support monitoring and prevention of attacks. For example, Firefox can be extended to generate HTML pages in a customized way to prevent script code injection [122].

Processors can execute code provided the supplied instruction sets are supported by CPUs. In a modified processor technique, code randomization is applied with a secret key, where randomized code must be de-randomized before passing to processors [79]. The modification includes adding decryption mechanism before loading to processors.

The binary rewriting technique adds or modifies different section of executable programs such as adding entries to a symbol table and a hash table in a new section to store data [108, 109].

Dynamic optimization frameworks allow runtime analysis of program code at different granular levels (*e.g.*, statement, block). The dynamic code optimizer extension technique leverages such framework to monitor programs to add checks such as allowing or disallowing control flow between two blocks [110].

Environmental change: This characteristic indicates how an approach introduces changes in program execution environment. A program execution environment change might include modification of dynamically linked libraries, compiler, memory, cache, operating system kernel, etc. For web-based programs, environment might include web server, browser, proxy server, etc. The fifth column of Table 9 shows that the environment can be changed due to *implementation mechanism*, *program state utilization*, and *performance*. We describe them in the following paragraphs.

Most of the approaches modify environments due to implementation mechanism. For example, dynamically linked library calls are modified or injected (*e.g.*, memory allocation and free) as part of attack detection [13, 58, 163]. Execution of multiple programs might be controlled through system call synchronization to allow changing of program states due to input and output [32] from the environment. Web-base client program environments are modified by enhancing browser components (*e.g.*, parsers, tokenizers) [124, 125]. Server program environment can be modified by extending proxy servers, adding filters, etc. [59, 123]. Kernel system calls can be modified to detect unintended memory accesses (*e.g.*, read only memory access [129]).

Program states are utilized to detect attacks at runtime. However, program states are often enhanced and modified with information. These result in changes of program execution environments. For example, information can be added in registers, cache memories, or extended address space of a program. Program function prologue and epilogue are modified to introduce environment changes [24, 30, 68]. Moreover,

kernel system calls are modified to save program states such as registers, tagging information (or tainted information), and stacks [40, 70, 143]. A duplicate stack frame can be created in an execution environment to save sensitive information [126]. Information can be saved in the environment such as a bit vector table to store the readable and writable status of memory bytes.

Several approaches try to reduce monitoring overhead by storing information in environment. Many approaches use faster memory blocks such as cache, hash, and look aside buffer in this direction as ways of modifying environments. For example, cache memories are used to store the most recent program instructions that access memories [54]. Moreover, hash can be used to store out of bound data during BOF attacks [31]. Furthermore, instrumented code block can be saved in cache memory to avoid instrumenting the same block in future [110].

Attack response: This characteristic describes how monitoring approaches respond to attacks. From the sixth column of Table 9, it is obvious that most of the techniques terminate programs and generate error messages. However, some techniques continue program executions by jumping to attack handler modules that might take corrective actions. For example, recovery of information can be performed based on saved stacks and return addresses [126]. Some approaches silently respond by simply blocking inputs and stopping further processing of inputs [59, 94, 122]. These are widely used approaches for handling web-based attacks. We also notice that several approaches rely on exceptions thrown by processors, as opposed to attack handler functions [18, 79].

Vulnerability coverage: This characteristic indicates what vulnerabilities are addressed by each of the approaches. From the seventh column of Table 9, we note that BOF, DAP, FSB, XSS, and SQLI have been addressed by most of the approaches. Very few approaches have addressed MEL vulnerabilities during runtime [104, 153]. Moreover, very few approaches can detect multiple attacks (*e.g.*, [69, 70]).

Language: This feature indicates the language of implemented programs which are being monitored. Most approaches monitor attacks by modifying programs implemented in C (*e.g.*, [68, 129, 151, 163]), Java [66], and JSP [63, 135] languages. Few approaches modify executable program code to detect attacks (*e.g.*, x86 [109-111, 143]). Some approaches monitor attacks which are not related to any implementation languages (*e.g.*, [40, 70]), where we mention the language feature as N/A. Most of these approaches employ policies which are independent of implementation languages.

7.2. Program operation monitoring

In general, a program takes inputs, processes them, and generates outputs. However, operations related to memory and function call during these phases might indicate vulnerability exploitations or attacks. The *program operation monitoring* objective aims to detect these attacks. We classify *program operation monitoring* approaches into four types: *memory access with strict bound*, *memory access with flexible bound*, *function call*, and *output generation*.

Memory access with strict bound: In this case, an approach does not allow performing read, write, and free operations on memory locations which are not within the valid address space of a program. This monitoring objective facilitates the checking of memory related security vulnerabilities such as BOF, DAP, and MEL. However, the approach requires tracking memory accesses at the fine grained level. For example, the memory bytes can be tagged for readability and writability status [153]. Moreover, memory allocation and free operations can be monitored [163]. Furthermore, computation through pointer type data (*e.g.*, pointer arithmetic) might contribute to memory access related vulnerabilities which can be prevented by tracking the base and size of all memory objects at runtime [18, 113, 151].

Memory access with flexible bounds: In this monitoring objective, accidental or intentional memory operations outside valid address spaces are allowed for the sake of program execution continuation. The related approaches aim to make programs as attack tolerant. However, such approach requires customization of memory managers to deal with invalid memory accesses. For example, Rinard *et al.* [31] prevent BOF as a boundless memory writing approach. They save the out of bound memory values in hash tables during buffer writing operations. Some approaches allow invalid memory operations in such a way that corruptions of variables are not performed. For example, Berger *et al.* [58] develop a runtime memory manager which randomizes the location of memory objects into the heap region and increases the size of allocated objects at least twice. As a result, successive objects are located at a wider gap and chances of BOF attacks are reduced. The DAP vulnerability is prevented by filling random values when allocating memory blocks by a program and executing multiple versions of the same program.

Function call: A monitoring approach might check whether functions are invoked in vulnerable free ways or not. In this case, argument count and argument list of functions might be examined. This objective has been used for detecting FSB attacks during format function calls. For example, Cowan *et al.* [141] count the number of arguments passed in format function calls and match these counts with the number of specifier supplied in format strings during runtime. Li *et al.* [143] check whether arguments are being retrieved beyond an argument list or not to detect FSB attacks.

Output generation: In this objective, it is checked whether vulnerabilities might be exploited while generating outputs with untrusted or tainted data sources. To check attack occurrences, data originating from untrusted sources are marked as tainted, and the propagation of tainted data is tracked. Finally, the sensitive output generation points are monitored for the presence of tainted inputs. For example, Newsome *et al.* [102] check whether format strings are derived from untrusted inputs or not, during format function calls. Sometimes, the output generation process is controlled by approaches to make sure that outputs do not contain injected malicious code. For example, Iha *et al.* [122] propose the generation of an HTML page into two stages: generating a DOM tree with nodes and filling the nodes with literals to

prevent XSS attacks. Kiciman *et al.* [104] intercept JavaScript code before rendering through a browser. They replace vulnerable string constants with safe equivalents to avoid XSS and MEL attacks.

7.3. Code execution flow and origin monitoring

This objective monitors allowable and unallowable control flows in program code. Moreover, it is checked whether program code is loaded from allowable locations or not. The dynamic information flow tracking is a popular approach to check these two properties (*i.e.*, intended execution flow and intended code origin) [40, 69, 70, 110]. Input data sources (*e.g.*, data from file or network) are marked as tainted. If any value generated from a tainted value is used in either code execution flow (*e.g.*, jump location) or as code (*i.e.*, instructions and pointers), programs are halted. Such an approach is useful for detecting vulnerability exploits that change program control flows such as BOF, FSB, and SQLI.

7.4. Code structure monitoring

This objective monitors if an executable code conforms to a desired syntactic structure that is valid and recognized by processors which execute the code. The objective is intended to detect injection code that might be provided through user inputs. A programmer implemented source code is randomized initially. After including user inputs, some parts of the code is de randomized before the code is executed by processors. As a result, attacker supplied code become meaningless to a processor and only the implemented code is executed. For example, Gaurav *et al.* [79] randomize machine code instructions by XORing each opcode with a unique key. Before loading the code by a processor (*i.e.*, interpreter), the code is decoded with the same key. Therefore, any decoded injected code results in invalid opcode and a CPU throws runtime exceptions. The approach can prevent code injection attacks caused by BOF and SQLI. Boyd *et al.* [94] also propose randomization of SQL keywords (*SQLrand* tool) to thwart injection attacks that contain SQL keywords. They add random integer numbers after SQL keywords. A proxy performs de-randomization of queries before sending to database engines. For any query containing injected data, the parse fails to interpret and does not forward it to database engines.

Several approaches compare the code structure before and after including user supplied inputs as ways of detecting code injection attacks. For example, Buehrer *et al.* [66] detect SQLI attacks by comparing the parse trees of SQL queries generated before and after input inclusions.

7.5. Value integrity monitoring

This objective monitors sensitive program or environment values which might be modified during attacks. We categorize related works into four types: *sensitive memory location without modification*, *sensitive memory location with modification*, *injected value*, and *program output*.

Sensitive memory location without modification: In this case, values stored in sensitive memory locations of programs are checked for their corruption. These values are modified by attacks. For

example, the integrity of a function's return address can be checked to detect a BOF attack [77, 105, 109, 126, 129]. In this case, a copy of the value is saved in a safe memory location (*e.g.*, a register or a global variable) before a function call. When a function returns to its caller, the integrity of current return address is checked by comparing the saved return address with the current return address. If the two addresses matches, then the execution of a program continues. Otherwise, an attack is detected.

Sensitive memory location with modification: To avoid guessing of sensitive memory locations by an attacker, several approaches store sensitive values in modified forms. For example, a return address or a function pointer can be encrypted (by XORing with a unique key) and saved in a safe location [24, 30, 55]. When a function finishes execution, the current return address is encrypted with the same key and compared with the saved address. If they do not match, the address is considered modified by an attack.

Injected value: This objective checks the integrity of *injected values* in a program as opposed to sensitive memory locations (*e.g.*, return addresses). When the injected value is modified, an attack is detected. For example, a canary value might be injected before a return address of a function [68, 152]. Before returning from a function to its caller, it is checked whether the canary is intact or not. If it is intact, no attack is detected and a program execution continues by returning a function to its caller. Otherwise, an attack is detected and a program execution might be stopped.

Program output: This objective detects occurrence of attacks by comparing the output of multiple versions of a program. These versions are structurally dissimilar and semantically similar to each other. For example, Salamat *et al.* [32] apply multi-variant code execution approach to detect BOF vulnerabilities. They allow the stack growth of two programs in two directions: downward and upward. In presence of an attack, one version is affected due to overwriting of return address, whereas, another version might remain intact and behave differently. The approach detects the occurrence of an attack, if there is any discrepancy in program outputs.

7.6. Unwanted value monitoring

In this objective, user supplied data is examined and checked for the presence of unwanted values. This objective is common to detect attacks in web-based programs. We divide the related works into two categories based on *input value* and *input attribute*.

Input value: In this case, whitelisted and blacklisted characters are checked before processing inputs by a program. Presence of blacklisted characters might form malicious code through input values. These blacklisted inputs might include SQL meta characters [59], HTML characters [123], and format specifiers [108] which allow SQLI, XSS, and FSB attacks, respectively. Moreover, input values can be examined to confirm whether they contain only known set of inputs. Any unknown input might represent injected code. For example, JavaScript code implemented in a program can be digitally signed to mark as whitelisted. When a page is rendered by a browser, it can be checked whether any JavaScript code present

within the page matches with known whitelisted script code or not. Such approach helps to detect XSS attacks [124, 125].

Input attribute: Program inputs should conform to attributes such as input size. If the input size is large, it might cause BOF attacks. A monitoring objective might check such attribute of inputs to detect attacks [111].

7.7. Invariant monitoring

This objective monitors the violation of constant properties in programs during execution. Extracting invariants requires one to run a program with a set of normal (non attack) input test cases (also known as profiling). A monitor identifies any deviation from the learned invariants (or profiles) during actual program run. The invariant properties depend on the attacks. For example, Han *et al.* [13] apply API invocation fingerprints to detect BOF attacks during runtime. They obtain a set of legitimate API invocation sequences and compare API invocation sequences generated at runtime to identify BOF attack. Program code structure can be applied as invariants. For example, SQLI attacks can be detected by comparing the parse tree generated with normal and actual inputs [63]. Similarly, XSS attacks can be identified by comparing the DOM of a shadow page (containing scripts written by programmers) and an actual generated page [135]. Zhou *et al.* [54] detect memory related vulnerabilities (*e.g.*, BOF) by identifying a set of instructions (AccSet) (*i.e.*, invariants) that access memory objects during program executions. In an actual program run with inputs, it is checked whether any instruction (program counter) accessing memory objects are within an identified set or not.

7.8. Open issues

We observe that monitoring techniques significantly vary according to the above characteristics. Our analysis indicates that BOF attacks have been well addressed through program operation and value integrity-based monitoring objectives. However, few works explore the runtime detection of other attacks (*e.g.*, SQLI and XSS) based on these objectives. Moreover, *unwanted value* and *code structure-based* monitoring have addressed SQLI and XSS attacks. These two objectives can be explored to detect other types of attack (*e.g.*, BOF and FSB). We notice that many approaches monitor very fine grained level of program operations (*e.g.*, accessing a register, memory objects). These approaches may invite high overhead to maintain and process information at fine grained levels (*e.g.*, byte and word). New research should focus on introducing monitoring approaches that can detect attacks using higher granularity levels (*e.g.*, memory block) in order to reduce overhead. For example, randomization can be performed at block level, as opposed to every opcode.

For web-based programs, most of the works address the monitoring issues related to server side programs. Very few works monitor client side programs, while client side programming paradigms are

evolving rapidly. Thus, we need to develop more client side monitoring tools. We also notice that program operation, code execution flow and origin, and code structure monitoring objectives can address multiple attacks. However, many recent attacks do not require injecting code or even corrupting data (*e.g.*, CSRF). Some attacks might not have direct observable symptoms (*e.g.*, DAP, MEL) compared to other attacks such as BOF, FSB, SQLI, and XSS. Therefore, more research is required to develop tools for identifying attacks with unobservable symptoms.

8. Conclusions

In this paper, we perform a comprehensive survey of the works that address detection and prevention of the most commonly occurred and addressed program security vulnerabilities namely buffer overflow (BOF), format string bug (FSB), SQL injection (SQLI), cross site scripting (XSS), cross site request forgery (CSRF), NULL pointer dereference (NLD), dangling pointer (DAP), and memory leak (MEL). We primarily compare and contrast the most widely used vulnerability mitigation (testing, static analysis, and hybrid analysis) and runtime monitoring techniques. Each technique has been explored in detail to perform comparative and qualitative analysis among relevant approaches based on a number of distinguishing criteria. Then we identify the open issues for each of the corresponding techniques. We also briefly discuss the current challenges of some other approaches which provide secure programming guidelines or related program maintenance: program transformation and patching.

Table 10: A mapping between the program security mitigation techniques and the addressed vulnerabilities

Technique	BOF	FSB	SQLI	XSS	CSRF	NLD	DAP	MEL
Testing	Y	Y	Y	Y	N	N	N	N
Static analysis	Y	Y	Y	Y	N	Y	Y	Y
Monitoring	Y	Y	Y	Y	N	N	Y	Y
Hybrid	Y	Y	Y	Y	N	N	Y	Y
Secure programming	Y	Y	Y	Y	Y	N	N	N
Program transformation	Y	N	N	Y	N	N	Y	N
Patching	Y	N	Y	Y	N	N	Y	N

Table 11: A mapping between the program security mitigation techniques and the programming languages

Technique	C	C++	Java	JSP	PHP	ASP	JavaScript	x86	Java byte code
Testing	Y	N	N	Y	Y	N	N	Y	Y
Static analysis	Y	Y	Y	Y	Y	Y	N	Y	N
Monitoring	Y	N	Y	Y	N	Y	Y	Y	N
Hybrid	Y	N	Y	Y	Y	Y	N	Y	Y
Secure programming	Y	N	Y	N	N	N	N	N	N
Program transformation	Y	N	N	N	N	N	Y	N	N
Patching	Y	N	Y	N	Y	N	N	Y	N

Based on our analysis, we provide a summarized mapping between the program security mitigation techniques and the addressed vulnerabilities in Table 10. It relates whether techniques have been applied to mitigate corresponding vulnerabilities (Y) or not (N). It is obvious that existing techniques have devoted considerable effort to partially mitigate a subset of vulnerabilities such as BOF, FSB, SQLI, and XSS. We

also map whether these techniques have been applied to programs written in a particular language (Y) or not (N) in Table 11. We notice a gap between current techniques and underlying programming languages as well. Currently, programs written in C, Java, and PHP are analyzed. However, we should also investigate how vulnerabilities can be detected in the programs written in other programming languages and where source code is not available (*i.e.*, in executable forms). For all these techniques, our common observation is that they can detect only certain types of vulnerabilities at the same time. Some techniques are strictly limited to certain programming languages. Moreover, few works have attempted combined techniques to detect vulnerabilities. We believe that future program security research should explore the detection and prevention of multiple vulnerabilities by applying new and hybrid techniques on the programs written in various programming languages.

Currently, there is no existing work that summarizes and compares current program-based vulnerability mitigation works in detail. This survey will help software security practitioners and researchers to understand pros and cons of these techniques, develop new software security tools, and explore future research avenues. For the sake of the length of the survey and the broadness of this topic, our study is restricted to the techniques that strive to mitigate the vulnerabilities found in the code level only. We also limit our analysis of vulnerabilities for the programs written in procedural, object oriented, and scripting languages. We do not discuss the approaches that primarily develop (*e.g.*, [34, 82]) or evaluate (*e.g.*, [23]) network intrusion detection systems (IDS). Our study also does not include security breaches that can be managed by using formal access control policies (*e.g.*, role-based access control policy or RBAC). More independent surveys are required for the above mentioned topics.

9. References

[1] H. Shahriar and M. Zulkernine, "Test Adequacy of Buffer Overflow Vulnerabilities: A Mutation-Based Approach," *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, Vol. 20, Issue 1, World Scientific, February 2010, pp. 73-101.

[2] H. Shahriar and M. Zulkernine, "Mutation-based Testing of Buffer Overflow Vulnerabilities," *Proc. of the 2nd International Workshop on Security in Software Engineering (IWSSE)*, Finland, July 2008, pp. 979-984.

[3] H. Shahriar and M. Zulkernine, "Mutation-based Testing of Format String Bugs," *Proc. of 11th High Assurance Systems Engineering Symposium (HASE 2008)*, Nanjing, China, December 2008, pp. 229-238.

[4] H. Shahriar and M. Zulkernine, "MUSIC: Mutation-based SQL Injection Vulnerability Checking". *Proc. of the 8th International Conference on Quality Software (QSIC 2008)*, London, UK, August 2008, pp. 77-86.

[5] H. Shahriar and M. Zulkernine, "MUTEC: Mutation-based Testing of Cross Site Scripting," *Proc. of the 5th ICSE Workshop on Software Engineering for Secure Systems*, Vancouver, Canada, May 2009, pp. 47-53.

[6] H. Shahriar and M. Zulkernine, "Automatic Testing of Program Security Vulnerabilities," *Proc. of the 1st International Workshop on Test Automation*, Seattle, USA, July 2009, pp. 550-555.

[7] H. Shahriar and M. Zulkernine, "Classification of Buffer Overflow Vulnerability Monitors," *Proc. of the 4th International Workshop on Secure Software Engineering*, Krakow, Poland, February 2010, pp. 519-524.

[8] H. Shahriar and M. Zulkernine, "Taxonomy and Classification of Automatic Monitors for Program Security Vulnerabilities," *Journal of Systems and Software*, Elsevier (Under 2nd round review).

[9] H. Shahriar and M. Zulkernine, "Mitigating Program Security Vulnerabilities: Challenges and Approaches," *ACM Computing Surveys*, Conditionally accepted in June 2010.

[10] H. Shahriar and M. Zulkernine, "Monitoring Buffer Overflow Attacks: A Perennial Task," To appear in *International Journal of Secure Software Engineering*, IGI Global.

[11] H. Shahriar and M. Zulkernine, "Classification of Static Analysis-based Buffer Overflow Vulnerability Detection," *Proceedings of the 1st International Workshop on Model Checking in Reliability and Security*, Singapore, June 2010, pp. 94-101.

[12] B. Hackett, M. Das, D. Wang, and Z. Yang, "Modular Checking for Buffer Overflows in the Large," *Proceedings of the 28th International Conference on Software Engineering*, Shanghai, China, May 2006, pp. 232-241.

[13] H. Han, X. Lu, L. Ren, B. Chen, and N. Yang, "AIFD: A Runtime Solution to Buffer Overflow Attack," *International Conference on Machine Learning and Cybernetics*, August 2007, Hong Kong, pp. 3189-3194.

[14] 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.

[15] A. Jorgensen, "Testing with hostile data streams," *ACM SIGSOFT Software Engineering Notes*, Volume 28, Issue 2, March 2003, pp. 9

[16] A. Aggarwal and P. Jalote, "Integrating Static and Dynamic Analysis for Detecting Vulnerabilities," *Proceedings of the 30th Annual International Computer Software and Application Conference*, July 2006, pp. 343-350.

[17] K. Kratkiewicz and R. Lippmann, "Using a Diagnostic Corpus of C Programs to Evaluate Buffer Overflow Detection by Static Analysis Tools," *Proc. of the Workshop on the Evaluation of Software Defect Detection Tools*, Chicago, USA, June 2005.

[18] D. Dhurjati and V. Adve, "Detecting all Dangling Pointer Uses in Production Servers", *Proc. of the International Conference on Dependable Systems and Networks*, Philadelphia, USA, June 2006, pp. 269-280.

[19] P. Kumar, A. Nema, and R. Kumar, "Hybrid Analysis of Executables to Detect Security Vulnerabilities," *Proc. of the 2nd Annual Conf. on India Software Engineering Conference*, Pune, India, February 2009, pp. 141-142.

[20] W. Le and M. Soffa, "Marple: A Demand-driven Path-sensitive Buffer Overflow Detector," *Proc. of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Atlanta, Georgia, November 2008, pp. 272-282.

[21] P. McMinn, "Search-based Software Test Data Generation: A Survey," *Software Testing, Verification and Reliability*, Vol. 14, No. 2, pp. 105-156, 2004.

[22] Z. Lin, X. Jiang, D. Xu, B. Mao, and L. Xie, "AutoPaG: Towards Automated Software Patch Generation with Source Code Root Cause Identification and Repair," *Proc. of the 2nd Symposium on Information, Computer and Communications Security*, Singapore, March 2007, pp. 329-340.

[23] G. Vigna, W. Robertson, D. Balzarotti, "Testing Network-based Intrusion Detection Signature Using Mutant Exploits," *Proc. of the Conf. on Computer and Communication Security*, October 2004, Washington DC, pp. 21-30.

[24] B. Madan, S. Phoha, and K. Trivedi, "StackOFFence: A Technique for Defending Against Buffer Overflow Attacks," *Proc. of the Intl. Conference on Information Technology: Coding and Computing*, April 2005, pp. 656- 661

[25] H. Nishiyama, "SecureC: control-flow protection against general buffer overflow attack," *Proceedings of the 29th Annual International Computer Software and Applications Conference*, Edinburgh, Scotland, July 2005, pp. 149-155.

[26] G. Novark, E. Berger, and B. Zorn, "Exterminator: Automatically Correcting Memory Errors with High Probability," *Proc. of the Conf. on Programming Language Design and Implementation,* San Diego, 2007, pp. 1- 11

[27] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler, "EXE: Automatically Generating Inputs of Death," *Proc. of the 13th Conference on Computer and Communications Security*, Alexandria, USA, Nov 2006, pp. 322-335.

[28] C. Grosso, G. Antoniol, E. Merlo, and P. Galinier, "Detecting Buffer Overflow via Automatic Test Input Data Generation," *Computers and Operations Research*, Volume 35, Issue 10, October 2008, pp. 3125-3143.

[29] D. Pozza, R. Sisto, L. Durante, and A. Valenzano, "Comparing Lexical Analysis Tools for Buffer Overflow Detection in Network Software," *Proc. of the 1st International Conference on Communication System Software and Middleware*, January 2006, New Delhi, pp. 1-7.

[30] C. Pyo, B. Bae, T. Kim, and G. Lee, "Run-time Detection of Buffer Overflow Attacks without Explicit Sensor Data Objects," *Proc. of the Intl. Conf. on Information Technology: Coding and Computing*, Las Vegas, April 2004, pp. 50

[31] M. Rinard, C. Cadar, D. Dumitran, D. Roy, and T. Leu, "A Dynamic Technique for Eliminating Buffer Overflow Vulnerabilities (and Other Memory Errors)," *Proceedings of the 20th Annual Computer Security Applications Conference*, Tucson, USA, December 2004, pp. 82-90

[32] B. Salamat, A. Gal, T. Jackson, K. Manivannan, G. Wagner, and M. Franz, "Multi-variant Program Execution: Using Multi-core Systems to Defuse Buffer-Overflow Vulnerabilities," *Proc. of the International Conference on Complex, Intelligent and Software Intensive Systems*, Spain, March 2008, pp. 843-848.

[33] J. King, "Symbolic Execution and Program Testing," *Communications of the ACM*, Volume 19, Issue 7, July 1976, pp. 385-394.

[34] E. Bertino, A. Kamra, and J. Early, "Profiling Database Application to Detect SQL Injection Attacks," *Proc. of the Intl. Performance, Computing, and Communications Conference*, New Orleans, April 2007, pp. 449-458.

[35] M. Castro, M. Costa, and T. Harris, "Securing Software by Enforcing Data-flow Integrity," *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA, 2006, pp. 11-11.

[36] M. Hind, "Pointer Analysis: Haven't We Solve This Problem Yet?," *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, Snowbird, Utah, June 2001, pp. 54-61.

[37] A. Smirnov and T. Chiueh, "Automatic Patch Generation for Buffer Overflow Attacks," *Proc. of the 3rd Intl. Symposium on Information Assurance and Security*, Manchester, UK, August 2007, pp. 165-170, ISBN: 0-7695-2876-7

[38] W. Speirs, "Making the Kernel Responsible: A New Approach to Detecting & Preventing Buffer Overflows," *Proc. of the 3rd IEEE International Workshop on Information Assurance*, Washington, March 2005, pp. 21-32

[39] V. Okun, William Guthrie, Romain Gaucher, and Paul Black, "Effect of Static Analysis Tools on Software Security: Preliminary Investigation," *Proc. of the 3rd Workshop on Quality of Protection QoP*, October 2007, Alexandira, Virginia, pp. 1-5.

[40] G. Edward Suh, J. Lee, D. Zhang, and S. Devadas, "Secure Program Execution via Dynamic Information Flow Tracking," *Proceedings of the 11th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, California, October 2006, pp. 85-96.

[41] M. Monga, R. Paleari, and E. Passerini, "A Hybrid Analysis Framework for Detecting Web Application Vulnerabilities," *Proc. of the 5th ICSE Workshop on Software Engineering for Secured Systems*, Vancouver, Canada, May 2009, pp. 87-96.

[42] J. Tevis and J. Hamilton, "Static Analysis of Anomalies and Security Vulnerabilities in Executable Files," *Proceedings of the 44th Annual Southeast Regional Conference*, Melbourne, Florida, 2006, pp. 560-565.

[43] T. Tsai and N. Singh, "Libsafe: Transparent System-wide Protection Against Buffer Overflow Attacks," *Proc. of the International Conference on Dependable Systems and Networks*, Bethesda, USA, June 2002, pp. 541

[44] J. Viega, J. Bloch, T. Kohno, and G. McGraw, "Token-based scanning of source code for security problems," *ACM Transactions on Information and System Security (TISSEC)*, Volume 5, Issue 3, August 2002, pp. 238-261

[45] L. Wang, J. Cordy, and T. Dean, "Enhancing Security Using Legality Assertions," *Proceedings of the 12th Working Conference on Reverse Engineering*, Stuttgart, German, November 2005, pp. 35-44.

[46] M. Weber, V. Shah, and C. Ren, "A Case Study in Detecting Software Security Vulnerabilities Using Constraint Optimization," *Proc. of the Workshop on Source Code Analysis and Manipulation*, Italy, November 2001, pp. 3-13.

[47] Y. Xie, A. Chou, and D. Engler, "ARCHER: Using Symbolic, "Path-sensitive Analysis to Detect Memory Access Errors," *Proceedings of the 9th European Software Engineering Conference*, Helsinki, Finland, 2003, pp. 327-336.

[48] W. Xu, D. DuVarney, and R. Sekar, "An Efficient and Backwards-Compatible Transformation to Ensure Memory Safety of C Programs," *Proc. of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, California, 2004, pp. 117-126.

[49] G. McGraw and B. Potter, "Software Security Testing," *IEEE Security and Privacy* 2(5), 2004, pp. 81-85.

[50 R. Xu, P. Godefroid, and R. Majumdar, "Testing for Buffer Overflows with Length Abstraction," *Proceedings of the International Symposium on Software Testing and Analysis*, July 2008, Seattle, USA, pp. 27-38.

[51] B. Chess and G. McGraw, "Static Analysis for Security," *IEEE Security and Privacy* 2(6), 2004, pp. 76-79.

[52] X. Zhang, L. Shao, and J. Zheng, "A Novel Method of Software Vulnerability Detection based on Fuzzing Technique," *Intl. Conf. on Apperceiving Computing and Intelligence Analysis*, December 2008, pp. 270-273.

[53] Z. Lin, B. Mao, and L. Xie, "A Practical Framework for Dynamically Immunizing Software Security Vulnerabilities," *Proc. of the 1st Intl. Conference on Availability, Reliability and Security*, April 2006, pp. 348-357.

[54] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas, "AccMon: Automatically Detecting Memory-related Bugs via Program Counter-based Invariants," *Proc. of 37th Intl. Symposium on Microarchitecture (MICRO)*, Portland, Oregon, 2004, pp. 269-280.

[55] G. Zhu and A. Tyagi, "Protection against Indirect Overflow Attacks on Pointers," *Proceedings of the 2nd International Information Assurance Workshop (IWIA'04)*, Charlotte, North Carolina, April 2004, pp. 97-106.

[56] M. Zitser, R. Lippmann, and T. Leek, "Testing Static Analysis Tools using Exploitable Buffer Overflows from Open Source Code," *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Newport Beach, California, November 2004, pp. 97-106.

[57] J. Offutt, Ye Wu, X. Du, and H. Huang, "Bypass Testing of Web Applications," *Proc. of the 15th International Symposium on Software Reliability Engineering (ISSRE 2004)*, Saint-Malo, France, November 2004, pp. 187-197.

[58] E. Berger and B. Zorn, "DieHard: Probabilistic Memory Safety for Unsafe Languages," *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Ottawa, Canada, June 2006, pp. 158-168.

[59] A. Alfantookh, "An Automated Universal Server Level Solution for SQL Injection Security Flaw," *Proc. of the International Conference on Electrical, Electronic and Computer Engineering*, September 2004, pp. 131-135.

[60] E. Haugh and M. Bishop, "Testing C Programs for Buffer Overflow Vulnerabilities," *Proc. of the Network and Distributed System Security Symposium (NDSS)*, San Diego, California, February 2003.

[61] B. Breech and L. Pollock, "A Framework for Testing Security Mechanisms for Program-based Attacks," *Proc. of the 2005 ICSE Workshop on Software Engineering for Secure Systems*, Missouri, May 2005, pp. 1-7.

[62] D. Balzarotti, M. Cova, V. Felmetsger, and G. Vigna, "Multi-Module Vulnerability Analysis of Web-based Applications," *Proc. of the 14th ACM Conference on Computer and Communications Security*, Alexandria, October 2007, pp. 25-35.

[63] S. Bandhakavi, P. Bisth, P. Madhusudan, and V. Venkatakrishnan, "CANDID: Preventing SQL Injection Attacks using Dynamic Candidate Evaluations," *Proceedings of the 14th ACM Conference on Computer and communications security*, Alexandria, Virginia, Oct 2007, pp. 12-24.

[64] 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.

[65] A. Ghosh, T. O'Connor, and G. McGraw, "An Automated Approach for Identifying Potential Vulnerabilities in Software," *IEEE Symposium on Security and Privacy*, California, 1998, pp. 104-14.

[66] G. Buehrer, B. Weide, and P. Sivilotti, "Using Parse Tree Validation to Prevent SQL Injection Attacks," *Proc. of the 5th Intl. Workshop on Software Engineering and Middleware*, Lisbon, Portugal, 2005,pp.106-113.

[67] H. Kim, Y. Choi, D. Lee, and D. Lee, "Practical Security Testing using File Fuzzing," *Proc. of International Conference on Advanced Computing Technologies (ICACT)*, Hyderabad, India, February 2008, pp. 1304-1307.

[68] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Automatic Adaptive Detection and Prevention of Buffer Overflow Attacks," *Proc. of the 7th USENIX Security Conference*, San Antonio, Texas, January 1998.

[69] J. Clause, W. Li, and A. Orso, "Dytan: A Generic Dynamic Taint Analysis Framework," *Proc. of the International Symposium on Software Testing and Analysis*, London, United Kingdom, 2007, pp. 196-206.

[70] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: A Flexible Information Flow Architecture for Software Security," *Proc. of the 34th Intl. Symposium on Computer Architecture*, San Diego, California, 2007, pp. 482-493.

[71] A. Tappenden, P. Beatty, J. Miller, A. Geras, and M. Smith, "Agile Security Testing of Web-based Systems via HTTPUnit," *Proc. of Agile Development Conference (ADC)*, Denver, Colorado, July 2005, pp. 29- 38.

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

[73] O. Tal, S. Knight, and T. Dean, "Syntax-based Vulnerabilities Testing of Frame-based Network Protocols," *Proc. of the 2nd Annual Conference on Privacy, Security and Trust*, Fredericton, Canada, October 2004, pp. 155-160.

[74] F. Dysart and M. Sherriff, "Automated Fix Generator for SQL Injection Attacks," *Proceedings of the 19th International Symposium on Software Reliability Engineering*, Seattle, Washington, November 2008, pp. 311-312.

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

[76] J. Fonseca, M. Vieira, and H. Madeira, "Testing and Comparing Web Vulnerability Scanning Tools for SQL Injection and XSS Attacks," *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*, Melbourne, Australia, December 2007, pp. 365-372.

[77] Stack Shield, A "stack smashing" technique protection tool for Linux, Vendicator, January 2001, Accessed from www.angelfire.com/sk/stackshield

[78] J. Yang, T. Kremenek, Y. Xie, and D. Engler, "MECA: An Extensible, Expressive System and Language for Statically Checking Security Properties," *Proceedings of 10th ACM Conference on Computer and Communications Security*, Washington DC, USA, October 2003, pp. 321-334.

[79] Gaurav Kc, A. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," *Proc. of the 10th ACM Conf. on Computer and Communications Security*, Washington, October 2003, pp. 272-280.

[80] J. Wilander and M. Kamkar, "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention," *Proc. of the 10th Network and Distributed System Security Symposium*, February 2003, California, pp. 149-162.

[81] W. Halfond and A. Orso, "Combining static analysis and runtime monitoring to counter SQL-injection attacks," *Proc. of the 3rd International Workshop on Dynamic Analysis*, Missouri, May 2005, pp. 1-7.

[82] K. Kemalis and T. Tzouramanis, "SQL-IDS: A Specification-based Approach for SQL-Injection Detection," *Proc. of 23rd ACM Symposium on Applied Computing (SAC'08)*, March 2008, Fortaleza, Brazil, pp. 2153- 2158.

[83] W. Halfond, A. Orso, and P. Manolios, "Using positive tainting and syntax-aware evaluation to counter SQL injection attacks," *Proc. of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Portland, Oregon, November 2006, pp. 175-185.

[84] 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.

[85] G. Hermosillo, R. Gomez, L. Seinturier, and L. Duchien, "AProSec: an Aspect for Programming Secure Web Applications," *Proceedings of the 2nd International Conference on Availability, Reliability and Security*, Vienna, Austria, April 2007, pp. 1026-1033.

[86] Y. Huang, S. Huang, T. Lin, and C. Tsai, "Web Application Security Assessment by Fault Injection and Behavior Monitoring," *Proceedings of the 12th International Conference on World Wide Web*, Budapest, May 2003, pp. 148-159.

[87] M. Johns and C. Beyerlein, "SMask: Preventing Injection Attacks in Web Applications by Approximating Automatic Data/Code Separation," *Proceedings of the ACM Symposium on Applied computing*, Seoul, Korea, March 2007, pp. 284-291.

[88] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities," *Proc. of the IEEE Symposium on Security and Privacy*, Oakland, May 2006, pp. 258-263.

[89] N. Juillerat, "Enforcing Code Security in Database Web Applications using Libraries and Object Models," *Proc. of the 2007 Symposium on Library-Centric Software Design*, Montreal, Canada, 2007, pp. 31-41.

[90] M. Junjin, "An Approach for SQL Injection Vulnerability Detection," *Proc. of the 6th International Conference on Information Technology: New Generations*, Las Vegas, Nevada, April 2009, pp. 1411-1414.

[91] S. Kals, E. Krida, C. Kruegel, and N. Jovanovic, "SecuBat: A Web Vulnerability Scanner," *Proc. of the 15th International Conference on World Wide Web*, Edinburgh, Scotland, May 2006, pp. 247-256.

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

[93] Symantec Internet Security Threat Report, Trends for July–December 07, Volume XII, April 2008, Accessed from http://eval.symantec.com/mktginfo/enterprise/white_papers

[94] S. Boyd and A. Keromytis, "SQLrand: Preventing SQL Injection Attacks," *Proc. of 2nd International Conference on Applied Cryptography and Network Security*, Springer, 2004, pp. 292–302

[95] Common Vulnerabilities and Exposures (CVE), http://cve.mitre.org

[96] A. Kieżun, P. Guo, K. Jayaraman, and M. Ernst, "Automatic Creation of SQL Injection and Cross-site Scripting Attacks," *Proceedings of the 31st Intl. Conf. on Software Engineering*, Vancouver, Canada, May 2009, pp. 199-209.

[97] Open Source Vulnerability Database (OSVDB), http://osvdb.org

[98] Range and Type Error Vulnerability, Accessed from http://www.owasp.org/index.php/ Category:Range_and_Type_Error_Vulnerability

[99] M. Lam, M. Martin, B. Livshits, and J. Whaley, "Securing Web Applications with Static and Dynamic Information Flow Tracking," *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, San Francisco, California, January 2008, pp. 3-12.

[100] V. Livshits and M. Lam, "Finding Security Vulnerabilities in Java Applications with Static Analysis," *Proceedings of the 14th conference on USENIX Security Symposium*, Baltimore, USA, July 2005, pp.18.

[101] S. Yong and S. Horwitz, "Protecting C Programs from Attacks via Invalid Pointer Dereferences," In *ACM SIGSOFT Software Engineering Notes*, Volume 28, Issue 5, September 2003, pp. 307-316.

[102] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," *Proceedings of the Network and Distributed System Security Symposium*, San Diego, Feb 2005.

[103] J. Lin and J. Chen, "The Automatic Defense Mechanism for Malicious Injection Attack," *Proc. of 7th Intl. Conference on Computer and Information Technology*, Fukushima, Japan, Oct 2007, pp. 709-714.

[104] E. Kiciman and B. Livshits, "AjaxScope: A Platform for Remotely Monitoring the Client-Side Behavior of Web 2.0 Applications," *Proc. of 21st Symposium on Operating Systems Principles, Stevenson*, Washington, 2007, pp. 17-30.

[105] A. Aggarwal and P. Jalote, "Monitoring the Security Health of Software Systems," *Proceedings of the 17th International Symposium on Software Reliability Engineering*, North Carolina, November 2006, pp. 146-158.

[106] K. Wei, M. Muthuprasanna, and S. Kothari, "Preventing SQL Injection Attacks in Stored Procedures," *Proceedings of the Australian Software Engineering Conference*, Sydney, Australia, April 2006, pp. 191-198.

[107] M. Muthuprasanna, K. Wei, and S. Kothari, "Eliminating SQL Injection Attacks- A transparent Defense Mechanism," *Proc. of the 8th International Symposium on Web Site Evolution*, Philadelphia, Sept 2006, pp 22-32.

[108] P. Kohli and B. Bruhadeshwar, "FormatShield: A Binary Rewriting Defense against Format String Attacks," *Proc. of the 13th Australasian Conf. on Information Security and Privacy*, Australia, July 2008, pp. 376-390.

[109] M. Prasad and T. Chiueh, "A Binary Rewriting Defense against Stack based Buffer Overflow Attacks," *Proceedings of USENIX 2003 Annual Technical Conference*, San Antonio, Texas, June 2003, pp. 211-224.

[110] V. Kiriansky, D. Bruening, and S. Amarasinghe, "Secure Execution via Program Shepherding," *Proceedings of the 11th USENIX Security Symposium*, San Francisco, August 2002, pp. 191-206.

[111] K. Lhee and S. Chapin, "Type-Assisted Dynamic Buffer Overflow Detection," *Proceedings of the 11th USENIX Security Symposium*, San Francisco, August 2002, pp. 81-88.

[112] T. Austin, S. Breach, and G. Sohi, "Efficient Detection of All Pointer and Array Access Errors," *Proc. of the Conference on Programming Language Design and Implementation*, Orlando, June 1994, pp. 290-301.

[113] O. Ruwase and M. Lam, "A Practical Dynamic Buffer Overflow Detector," *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2004, pp. 159-169.

[114] D. Evans and D. Larochelle, "Improving Security Using Extensible Lightweight Static Analysis," *IEEE Software*, January 2002, pp. 42-51.

[115] S. Thomas and L. Williams, "Using Automated Fix Generation to Secure SQL Statements," *Proceedings of the 3rd International Workshop on Software Engineering for Secure Systems*, Minneapolis, May 2007, pp 9-14.

[116] N. Dor, M. Rodeh, and M. Sagiv, "CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C," *Proc. of the Conf. on Programming Language Design and Implementation*, California, June 2003, pp. 155-167.

[117] Y. Xie and A. Aiken, "Static Detection of Security Vulnerabilities in Scripting Languages," *Proceedings of the 15th Conference on USENIX Security Symposium*, Vancouver, Canada, July 2006, pp. 179-192.

[118] D. Wagner, J. Foster, E. Brewer, and A. Aiken, "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities," *Proc. of Network and Distributed System Security Symposium*, San Diego, February 2000, pp. 3-17.

[119] Document Object Model (DOM) Level 1 Specification, Version 1.0, October 1998, http://www.w3.org

[120] C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir, "BrowserShield: Vulnerability-driven filtering of dynamic HTML," *ACM Transactions on the Web*, Volume 1, Issue 3, September 2007, Article No. 11.

[121] A. Klein, "DOM-based Cross Site Scripting or XSS of the Third Kind," July 2005.

[122] G. Iha and H. Doi, "An Implementation of the Binding Mechanism in the Web Browser for Preventing XSS Attacks: Introducing the Bind-Value Headers," *Proc. of the Intl. Conf. on Availability, Reliability and Security*, 2009, pp. 966-971.

[123] O. Ismail, M. Etoh, Y. Kadobayashi, and S. Yamaguchi, "A Proposal and Implementation of Automatic Detection/Collection System for Cross-site Scripting Vulnerability," *Proceedings of the 18th International Conference on Advanced Information Networking and Applications*, 2004, pp. 145-151.

[124] T. Jim, N. Swamy, and M. Hicks, "Defeating Script Injection Attacks with Browser-Enforced Embedded Policies," *Proceedings of the 16th International Conference on World Wide Web*, Alberta, May 2007, pp. 601-610.

[125] M. Johns, B. Engelmann, and J. Posegga, "XSSDS: Server-Side Detection of Cross-Site Scripting Attacks," *Proceedings of the Annual Computer Security Applications Conference*, Anaheim, CA, December 2008, pp. 335-344.

[126] S. Gupta, P. Pratap, H. Saran, and A. Kumar, "Dynamic Code Instrumentation to Detect and Recover from Return Address Corruption," *Proc. of the Workshop on Dynamic Systems Analysis*, China, 2006, pp. 65-72.

[127] G. Lucca, A. Fasolino, M. Mastoianni, and P. Tramontana, "Identifying Cross Site Scripting Vulnerabilities in Web Applications," *Proc. of the 6th Intl. Workshop on Web Site Evolution*, Chicago, September 2004, pp. 71-80.

[128] G. Zuchlinski, The Anatomy of Cross Site Scripting, November 2003.

[129] T. Chiueh and F. Hsu, "RAD: A Compile-Time Solution to Buffer Overflow Attacks," *Proc. of the 21st International Conference on Distributed Computing Systems*, Arizona, USA, April 2001, pp. 409-417.

[130] HTML 4.01 Specification, http://www.w3.org/TR/REC-html40, December 1999.

[131] K. Ashcraft and D. Engler, "Using Programmer-Written Compiler Extensions to Catch Security Holes," *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, USA, May 2002, pp. 143.

[132] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008)*, Leipzig, Germany, May 2008, pp. 171-180.

[133] D. Yu, A. Chander, N. Islam, and I. Serikov, "JavaScript Instrumentation for Browser Security," *Proc. of the 34th Symposium on Principles of Programming Languages (POPL'07)*, Nice, France, January 2007, pp. 237-249.

[134] S. McAllister, E. Kirda, and C. Kruegel, "Leveraging User Interactions for In-Depth Testing of Web Applications," *Proc. of the 11th International Symposium on Recent Advances in Intrusion Detection*, Cambridge, Massachusetts, USA., pp. 191-210.

[135] P. Bisht and V. Venkatakrishnan, "XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks," *Proc. of the 5th Intl. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, Paris, July, 2008, pp.23-43.

[136] Scut/team teso, "Exploiting Format String Vulnerabilities," 2001, Accessed from http://doc.bughunter.net/format-string/exploit-fs.html (January 2009).

[137] E. Ofuonye and J. Miller, "Resolving JavaScript Vulnerabilities in the Browser Runtime," *Proceedings of the 19th International Symposium on Software Reliability Engineering*, Washington DC, November 2008, pp. 57-66.

[138] U. Shankar, K. Talwar, J. Foster, and D. Wagner, "Detecting Format String Vulnerabilities with Type Qualifiers," *Proceedings of the 10th USENIX Security Symposium*, 2001, pp. 201-220.

[139] K. Chen and D. Wagner, "Large-Scale Analysis of Format String Vulnerabilities in Debian Linux," *Proc. of the Workshop on Programming Languages and Analysis for Security (PLAS' 07)*, San Diego, June 2007, pp. 75-84.

[140] A. Dekok, "Pscan (1.2-8) Format String Security Checker for C Files," http://packages.debian.org/etch/pscan

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

[142] M. Ringenburg and D. Grossman, "Preventing Format-string Attacks via Automatic and Efficient Dynamic Checking," *Proc. of 12th Conference on Computer and Communications Security*, Nov 2005, Alexandria, pp. 354-363.

[143] W. Li and T. Chiueh, "Automated Format String Attack Prevention for Win32/X86 Binaries," *Proceedings of 23rd Annual Computer Security Applications Conference*, Miami, Dec 2007, pp. 398-409.

[144] T. Robbins. Libformat, http://archives.neohapsis.com/ archives/linux/lsap/2000-q3/0444.html

[145] G. Wassermann and Z. Su, "Sound and Precise Analysis of Web Applications for Injection Vulnerabilities," *Proc. of the 2007 PLDI Conference*, pp. 32-41.

[146] W. Halfond, J. Viegas, and A. Orso, "A Classification of SQL-Injection Attacks and Countermeasures," *Proceedings of IEEE International Symposium on Secure Software Engineering*, Arlington, Virginia, March 2006.

[147] C. Dahn and S. Mancoridis, "Using Program Transformation to Secure C Programs Against Buffer Overflows," *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003)*, November 2003, pp. 323-332.

[148] FlawFinder, Available at http://www.dwheeler.com/flawfinder

[149] Q. Gao, W. Zhang, Y. Tang, and F. Qin, "First-aid: surviving and preventing memory management bugs during production runs," *Proceedings of the 4th European Conference on Computer System*, Germany, 2009, pp. 159-172.

[150] J. Cordy, "The TXL Source Transformation Language," *Science of Computer Programming*, Volume 61, Issue 3, pp. 190-210, Elsevier North-Holland.

[151] R. Jones and P. Kelly, "Backwards-compatible Bounds Checking for Arrays and Pointers in C Programs," In *Proc. of Automated and Algorithmic Debugging*, Sweden, 1997, pp. 13-26.

[152] H. Etoh, GCC Extension for Protecting Applications from Stack-smashing Attacks, http://www.trl.ibm.com/projects/security/ssp

[153] R. Hastings and B. Joyce, "Purify: Fast Detection of Memory Leaks and Access Errors," *Proceedings of the USENIX Winter Conference*, San Francisco, CA, January 1992, pp. 125-138.

[154] Aleph One, "Smashing the Stack for Fun and Profit," *Phrack Magazine,* Volume 7, Issue 49, November 1996, Accessed from http://insecure.org/stf/smashstack.html

[155] 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 University Leuven, Belgium, October 2006, Accessed from http://www.fort-knox.be/files/CW463.pdf.

[156] S. Mancoridis, "Software Analysis for Security," *Proc. of Frontiers of Software Maintenance*, Beijing, October 2008, pp. 109-118.

[157] M. Ernst, "Static and Dynamic Analysis: Synergy and Duality," *Proceedings of ICSE Workshop on Dynamic Analysis*, Portland, May 2003, pp. 24-27.

[158] C Standard Library in http://www.utas.edu.au/infosys/info/documentation/C/ CStdLib.html

[159] NULL Pointer Dereference, Accessed from http://cwe.mitre.org/data/definitions/476.html

[160] R. Seacord, "Secure coding in C and C++ of Strings and Integers," *IEEE Security & Privacy*, Volume 4, Issue 1, Feb 2006, pp 74-76.

[161] CWE-352: Cross Site Request Forgery, http://cwe.mitre.org/data/definitions/352.html

[162] C. Erickson, "Memory Leak Detection in Embedded Systems," September 2002, Accessed from http://www.linuxjournal.com/article/6059

[163] C. Fetzer and Z. Xiao, "Detecting Heap Smashing Attacks through Fault Containment Wrappers," *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems*, New Orleans, USA, October 2001, pp. 80-89.

[164] U. Erlingsson, "Low-level Software Security: Attacks and Defenses," Microsoft Research Technical Report MSR-TR-07-153, November 2007.

[165] PaX Project, Accessed from http://pax.grsecurity.net/docs/pax.txt

[166] D. Leah, A Memory Allocator, April 2000, Accessed from http://g.oswego.edu/dl/html/malloc.html

[167] J. Burns, Cross Site Request Forgery: An Introduction to A Common Web Application Weakness, White paper, Information Security Partners LLC., 2007.

[168] OWASP CSRFGuard Project, Accessed in March 2010 from http://www.owasp.org/index.php/CSRFGuard_2.2_Configuration_Manual

[169] P. Guo, *A Scalable Mixed-Level Approach to Dynamic Analysis of C and C++ Programs*, Master of Engineering thesis, Massachusetts Institute of Technology, USA, May 2006.

[170] Common Weakness Enumeration, 2009 CWE/SANS Top 25 Most Dangerous Programming Errors, Accessed from http://cwe.mitre.org/top25

[171] O. Tripp, M. Pistoia, S. Fink, M. Sridharan, and O. Weisman, "TAJ: Effective Taint Analysis of Web Applications," *Proceedings of the Programming Language Design and Implementation*, Dublin, June 2009, pp. 87-97.

[172] A. Sotirov, *Automatic Vulnerability Detection Using Static Analysis*, MSc Thesis, The University of Alabama, 2005, Accessed from http://gcc.vulncheck.org/sotirov05automatic.pdf

[173] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek, "Buffer Overrun Detection using Linear Programming and Static Analysis," *Proceedings of the 10th ACM conference on Computer and Communications Security*, Washington D.C., USA, October 2003, pp. 345-354.