Search Based Software Testing for Software Security: Breaking Code to Make it Safer

Giuliano Antoniol
antoniol@ieee.org

SOCCER Laboratory – DGIGL
École Polytechnique de Montréal, Québec, Canada

Abstract

Ensuring security of software and computerized systems is a pervasive problem plaguing companies and institutions and affecting many areas of modern life. Software vulnerability may jeopardize information confidentiality and cause software failure leading to catastrophic threats to humans or severe economic losses.

Size, complexity, extensibility, connectivity and the search for cheap systems make it very hard or even impossible to manually tackle vulnerability detection.

Search based software testing attempts to solve two aspects of the cost - vulnerability problem. First, it's cheaper because it is far less labor intensive when compared to traditional testing techniques. As a result, it can be used to more thoroughly test software and reduce the risk that a vulnerability slips into production code. Also, search based software testing can be specifically tailored to tackle the subset of well known security vulnerabilities responsible for most security threats.

This paper is divided into two parts. It examines promising search based testing approaches to detecting software vulnerabilities, and then presents some of the most interesting open research problems.

Keywords: search based software testing, vulnerability exposure, high dependability software

1 Introduction

A secure computer is one with wiped and melted hard disks, switched off at all times, not connected to a network and buried under tons of rock. Obviously, such a computer would be useless; on the other hand, the absence of any of these conditions may allow an attacker to exploit software vulnerabilities and thus lead to security threats [37].

A vulnerability is a problem that can be exploited by an attacker [37]. A slightly different definition is proposed by the Mitre Corporation[1] as it focuses more on vulnerability consequences: "a vulnerability is a mistake in software that can be directly used by a hacker to gain access to a system or network". The definition of vulnerability does not require that a successful attack be carried out. If a network application is shielded by a firewall it may be impossible to exploit certain types of application vulnerabilities; however, the application remains vulnerable though the system (application plus firewall) is not.

There are countless vital society infrastructures heavily dependent on computerized systems and software applications. From energy or water distribution, to transportation, communication or financial asset management, everyday life depends on correct software behavior. In critical software systems such as health-care, nuclear, or aerospace software applications, a vulnerability may cause severe threats to humans or severe economic losses. If they occur at operating system level, in network or security applications, they can be exploited to gain administrator privileges, perform system attacks, access unauthorized data, or misuse the system.

Software is flexible; it can adapt and evolve; software applications can be connected to each other to perform sophisticated and complex tasks. However, software is also labor intensive and modern applications are often so large and complex that no single developer knows all the details or can predict all eventualities. Even worse, successful software systems operate for decades, and often outlive the hardware and operational environments for which they were originally developed. In addition to size, complexity, connectivity and extensibility, it is also impossible to predict how a software system should function in the future. Simply stated, it is very hard or even impossible to ensure that

[1] See http://cve.mitre.org/about/terminology.html

no vulnerability slips into production code.

Testing has traditionally been one of the main techniques to contribute to a high level of software dependability. Testing activity consumes up to 50% - 70 % of software development resources. Fierce industrial competition, the need to reduce time to market and the search for cheap software often limit testing resources. The problem is exacerbated by the diffusion and complexity of modern web applications where application interfaces may not be easily identified [34]. For example, parsing HTML pages may not accurately and completely identify application interfaces. Consider a simple on line application where the user first authenticates by using a static HTML form to enter user id and password, then a specific user dependent page is generated to collect further data (e.g., actions to be performed, new order details, etc), and finally, this information is processed via a non user accessible and hidden interface (e.g., data base information update). In similar situations, only the first two interfaces will be directly exposed to client code, and testing may have limited effectiveness since large areas of code may remain untested due to lack of information.

Search Based Software Testing (SBST) [48] deals with two aspects of the cost - vulnerability problem. First, the goal is to break the code, which implies that an Oracle is readily available. Thus fully automated testing can be devised. SBST is far less labor intensive than traditional testing techniques and therefore cheaper. Consequently, it can be used to more thoroughly test software to reduce the risk that a vulnerability slips into production code. Furthermore, it can be specifically tailored to tackle sub sets of well known security vulnerabilities such as buffer overflow and SQL injection or attack patterns such as privilege escalation.

The simplest form of SBTS is random testing and random test input data generation. These are appealing since they can free developers from the burden of manually developing test suites. Their major drawback is a lower fault revealing power with respect to other testing strategies [9, 20, 61]. However, when developing mathematical functions where the input space cannot be structured, when there is a lack of knowledge of the input domain, or when testing for vulnerability detection (e.g., to prove communication protocol robustness), developers have to resort to random testing. In general, random testing techniques can be adapted to substantially reduce the manual effort needed to develop test cases. For example, they can create an initial set of test input data to be refined and completed manually.

SBST goes beyond random search and thus it often outperforms random testing; it is easily coupled with coverage criteria or operational profiles, so it significantly increases fault revealing power [25, 26]. Approaches exploiting random testing decrease manual effort, as does SBST, which logically should do even better. This conjecture is supported by empirical evidence [9, 25, 26]; however, no industrial data have been published yet on SBST effort saving.

Traditional testing principles and adequacy criteria still hold when SBST techniques are applied to vulnerability detection. Below we will refer to Search Based SECurity (SBSec) to identify the set of approaches and techniques, possibly derived from SBST or SBST specialization, which are tailored to expose vulnerabilities and improve security. When applying SBSec a good test case exposes defects and system vulnerability. In this context exposing vulnerability implies producing input values that will crash the software, smash protection (e.g., heap, stack, memory), give access to privileged resources or confidential information, alter operating system schedule, increase user privileges and so on. SBSec researchers should focus on fitness functions exploiting programmers' attitude to trust user input, well known vulnerable APIs, poorly designed exception handling, lack of architectural and design skill. Consider the early Microsoft implementation of Crispin Cowan StackGuard [15] in Visual C++.Net and Visual C++ version 7; it was intended to guard against buffer overflow, but it turned out to be vulnerable and easily defeated [37] due to lack of knowledge of already discovered early StackGuard implementation problems.

SBST has already proved successful in many test applications [48]. Search based approaches date back to the mid 1970s [52] and have also been applied to tackle buffer overflow [29], SQL injection [50, 51] and privilege escalation detection [50], protocol testing [25, 46], robustness testing [24], Intrusion Detection System (IDS) design and testing [18, 19, 31].

This paper has a twofold goal; first it aims to provide an overview of recent achievements in SBSE and related techniques to target software vulnerabilities. Second and more importantly it demonstrates that there remain many interesting and challenging research and practical problems in testing for vulnerability detection.

Finally, despite foreseeable SBST/SBSec advancements, it pays to be cautious; paradoxically, the absence of software vulnerability does not reduce security risk. Security is not limited to a program in isolation, a given environment or network configuration. A recent NIST technical guide to information security testing and assessment [57] lists penetration testing [4]. Penetration testing mimics a real-world attack and can be conducted in many different ways. As the guide points out it may include non technical attacks related to social engineering. For example, a tester (or attacker) can call a help desk, and posing as a user, ask for a password reset. An attacker may walk into a room and install a keylogging device or steal a hard disk. Security is a more general concept with profound implications ranging from physical localization and building structure to facility access control, system and system security architecture, network connec-

tivity and server redundancies, personnel screening, and so on.

```
 1: int main (int argc, char **argv){
 2:    char buff[BUFSIZ];
 3:    int how_many = atoi(argv[1]), p=0, w=0;
 4:    strcpy(buff,argv[0]);
 5:    if (argc==3){
 6:      for (p=0; p< how_many; p++)
 7:         strcat(buff,argv[2]);
 8:    } else if (argc>3){
 9:         for (p=0; p< how_many; p++)
10:            for (w=2; w< argc; w++)
11:                strcat(buff,argv[w]);
  }
12:  printf("Buffer: %s\n",buff);
13:  exit(0);
}
```

Figure 1. Example of C code prone to buffer overflow.

2 State of the art

Recently Mitre Corporation[2] and SANS[3](SysAdmin, Audit, Network, Security) Institute, a security research center, published a list of 25 most dangerous programming errors leading to software vulnerability. This list is the result of collaboration between Mitre (Mitre Common Weakness Enumeration - CWE[4]), SANS(Top 20 attack vectors [5]) and many experts in security worldwide. A few examples of common weaknesses are improper user validation (CWE-20), failure to preserve SQL query structure (CWE-89), failure to constrain operations within the bounds of a memory buffer (CWE-119), improper access control (CWE-285) and hard-coded password (CWE-259). Surprisingly, buffer overflow and SQL injection according to Mitre, SANS and other sources [12, 13, 65] are the two vulnerabilities causing most attacks (CWE-20,CWE-89,CWE-119). Like many other types of vulnerability, they are rooted in programmers' bad habit of trusting user provided input and/or not carefully designing exception handling.

2.1 Buffer overflow (CWE-119, CWE-20)

Buffer overflow is essentially an unwanted bad memory usage vulnerability where an application accesses and usually writes critical memory locations. It is important to distinguish between the following related problems [29]: (a) buffer overflow that causes a security failure, (b) buffer overflow that causes a program failure such as a program crash (with no further consequences), and (c) buffer overrun that is a weaker condition simply associated with a buffer index out of range but necessarily associated with more severe forms of buffer overflow.

The term "buffer overflow prone software" indicates software in which some buffer overflow may occur; "vulnerable statements" refers to buffer operations that may access memory locations out of the buffer bounds; and "vulnerable buffers" refers to buffers whose bounds may be overrun at some "vulnerable statements".

The term "buffer overflow revealing data" refers to input data that cause a program execution to reach a buffer overflow condition. Clearly, to obtain buffer overflow, the appropriate conjunction of "buffer overflow revealing data" and "buffer overflow prone software" has to occur.

Consider the simple C program of Figure 1. `strcpy` and `strcat` are POSIX functions that accept two parameters; the `strcpy` copies the content of the second parameter into the first, `strcat` concatenates the second passed parameter to the first.

If the program of Figure 1 is executed passing as parameter an integer (`argc` equal to 2), statements 1, 2, 3, 4, 5, 8, 12 and 13 are executed. This test input data exercises line 4 `strcpy`. When executed passing as parameter an integer followed by a string (`argc` equal to 3) both line 4 and 7 vulnerable statements are exercised. This second set of input data not only forces the execution of more vulnerable statements but also executes several times `strcat` at line 7, potentially generating a buffer overrun.

There are several approaches to protect or identify buffer overrun vulnerabilities which can be divided into two large categories: static approaches and dynamic checks.

Static code analysis tools such as RatScan [6] (a graphical front end to RATS) or the freely available *splint* [22, 42] identify a set of vulnerable statements. *splint* detects both stack and heap-based buffer overflow vulnerability. It produces warnings whenever a vulnerable library function is called; buffer overflow vulnerable functions include function manipulating strings (e.g., strcpy, strcat) and reading inputs into buffers (e.g., gets). These tools tend to be conservative and thus generate a high number of false positives.

CodeSonar[7] from Grammatech is a sophisticated analysis tool for detecting bugs that can pinpoint among other defects buffer overrun and thus help in identifying software vulnerabilities. Interestingly, it allows for a tuning of false positive rates via configuration of different accuracy levels of performed static analyses.

FLF Finder [16] is a tool developed to automate the detection of high-risk functions. It focuses on functions that are likely to be the source of security vulnerabilities. However, the existence of such a tool is predicated on the ability to characterize a function's *security vulnerability likelihood*.

[2] http://cwe.mitre.org/top25/
[3] http://www.sans.org/top25errors/
[4] http://cwe.mitre.org
[5] http://www.sans.org/top20/
[6] http://www.beetlesoft.com
[7] http://www.grammatech.com/products/codesonar/

The main hypothesis is that functions near a source of input are most likely to contain a security vulnerability.

Dynamic boundary checks are native to modern programming languages such as Java. Obviously the appropriate exception handling code must be written to fully take advantage of this feature. Dynamic boundary checks can also be performed via extra code added by the compiler in languages such as C or C++ lacking this functionality. Indeed, compiler based techniques are available to virtually eliminate buffer overflow vulnerabilities with only modest performance penalties. For example, *stackguard* [15] is a patch to *gcc* that enhances the generated executable code. Extra code added by the compiler detects and thwarts buffer-overflow attacks against the stack.

All the above available tools and approaches help either a priori (e.g., RATSCAN, splint, CodeSonar) or at run time (stackguard) to identify buffer overflow. The main drawback is that they do not provide input data leading to the vulnerability exposure and thus the developer is left alone to understand and verify how buffer overflow can happen and to devise test input data leading to the vulnerability exposure.

SBST can be easily tailored to generate test input data exposing buffer overrun [29]. A fitness function to expose buffer overrun vulnerabilities needs to include some key components. It should include some form of *statement coverage* but should favor the coverage of *vulnerable statements* such as *strcpy and strcat*. The *number of executions of vulnerable statements* is other information that can guide the search; the more we execute risky statements the more likely we are to expose a vulnerability.

A further heuristic for improving the fitness function comes from the use of control flow graph *unconstrained nodes* [49]. Unconstrained nodes do not dominate or post-dominate other nodes. Intuitively, test cases covering unconstrained nodes constitute a reduced set of test cases required to meet statement coverage criteria. Unconstrained nodes are often associated with statements *more deeply nested* in the control flow graph. Going back to Figure 1 source code, to execute instruction 11, we need to pass at least three parameters, for example, an integer and two strings. This is the only way to cover the definition - use relation between the passed values of `argv` at line 1 of the `main` and the use of `argv` and its content at instruction 11.

Finally, a fitness function should explicitly use information on buffer boundaries. Overall, as proposed [30] a suitable fitness function, referred to as Basic Boundary Fitness (BBF), can be composed as:

$$\begin{aligned} F(g) &= w_1 \cdot sc_g + w_2 \cdot log(k_g) \cdot vcov_g + \\ w_3 &\cdot nesting_g + w_4 \cdot max_i\{min_j(L(g)_{i,j} - SB_i)\} \end{aligned} \quad (1)$$

where for the test input datum g (i.e., individual g):

- sc_g is the statement coverage ratio;
- $vcov_g$ is the vulnerable statement coverage ratio;
- k_g is the number of executions of vulnerable statements and has to be logarithmically rescaled to avoid its excessive dominance over other factors;
- w_1, w_2, w_3 and w_4 are real, positive weights, indicating the contribution of each factor to the overall fitness function; and
- SB_i is the size of the i-th buffer in the function under test, and $L(g)_{i,j}$ is the upper limit of the buffer accessed at the $j-th$ iteration when executing the current test case.

Although this first fitness function performs well, it is not amenable to complete automation. Indeed, test data generation and buffer overrun verification are distinct phases, and both should be automated, including the optimization of weights. When buffer overrun occurs the term $max_i\{min_j(L(g)_{i,j} - SB_i)\}$ is positive; making $max_i\{min_j(L(g)_{i,j} - SB_i)\}$ positive means maximizing $max_i\{min_j(L(g)_{i,j})\}$, in other words, reading or writing as close as possible (and above) the buffer limits. Overall the above fitness function can be rewritten as:

$$\begin{aligned} F(g) &= w_1 \cdot sc_g + w_2 \cdot log(k_g) \cdot vcov_g + \\ w_3 &\cdot nesting_g + w_4 \cdot max_i\{min_j(L(g)_{i,j})\} \end{aligned} \quad (2)$$

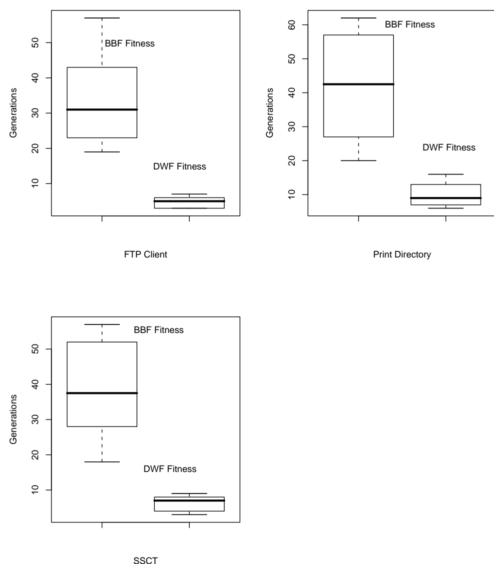
This fitness will be referred to as Dynamic Weight Fitness (DWF); DWF term $max_i\{min_j(L(g)_{i,j})\}$ is non negative; thus weight determination can be restated as a linear optimization problem (see [29]) over a sample of test input data. The main advantage of this formulation is that no manual intervention is required, so the test input data generation process can be fully automated.

The difference in complexity and performance between BBF and DWF is substantial. Figures 2 and 3 report box plots of buffer overrun events for two suites of applications. Reported results correspond to experiments replicated ten times; each time data were collected, in particular the generation at which the buffer overrun was discovered and execution time. The first set of applications, Figures 2, comprises an FTP client[8] (FTP-client) which is representative of network applications. The Print Directory [9] (PD) is a command line file system listing utility which is an alternative to the Unix *ls* shell command. Finally, the Super Spell Checker & Translator [10] (SSCT) is a command-line utility that takes a single word, checks spelling, takes the result(s), and then

[8] http://freshmeat.net/projects/cmdftp/

[9] http://users.foxvalley.net/∼sfehrman/pd

[10] http://www.progsoc.uts.edu.au/∼jedd/ssct



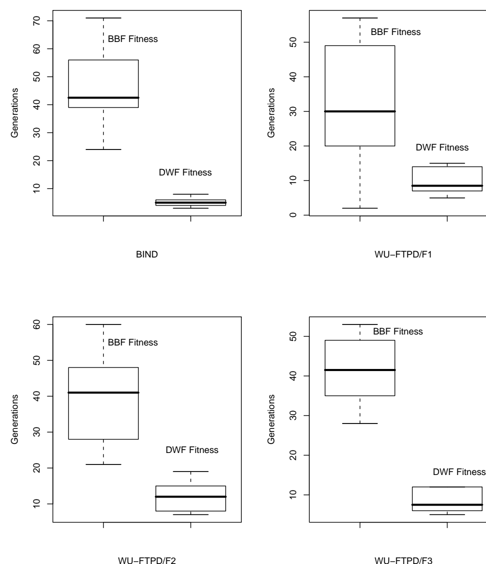


Figure 2. FTP client, PD, SSCT BBF and DWF box plots of buffer overrun events.

Figure 3. WU-FTPD and BIND box plots of buffer overrun events.

translates it. The functions, *readline_tab* (FTP-client), *display_directories* (PD) and *init* (SSCT) are used as unit under test.

BBF and DWF are also compared, Figures 3, on two more programs; the first is BIND, which is probably the most widely used DNS server, while the second, WU-FTPD, is a popular ftp daemon. BIND and WU-FTPD were previously used to compare vulnerability detection tools by R. Lippman at MIT Lincoln Laboratory [70].

Threats identified in the selected functions are reported in Table 1 with a summary description of the unsafe operation and the Common Vulnerability and Exposures (CVE) identifiers.

In both experiments, Figures 2 and 3, DWF outperforms BBF. Despite the effort to manually optimize BBF weights, the fully automated and more complex approach is substantially faster. Similar results are obtained for buffer overflow, see [29] for more details. These results point to an interesting engineering challenge between resources needed to develop a sophisticated tool and the gain in effort and performance achieved later on with respect to simpler approaches. DWF is between four to five time faster on average in terms of number of generations and also in computation time. On the other hand, on the two test sets, BBF never takes longer than ten minutes on a dual core laptop to expose a vulnerability. This is an acceptable time in most situations.

However, DWF offers a further, hidden, benefit; it completely eliminates the subjectivity of tester from the loop since there is no need for manual parameter tuning. Thus, an expert and a novice tester will obtain the same results with very similar effort when testing the same code. This is likely a more general property; in fact, whenever SBSec techniques can be applied to vulnerability detection, they probably reduce the gap of performance between experts and novices.

2.2 SQL injection (CWE-89, CWE-20)

Many web applications accept interactions from users and perform some accesses to databases (DBs) based on assumptions about legitimate input and code that are used to build SQL queries. These applications are possibly vulnerable to SQL-injection attacks [3, 37], which rely on some weak validation of the textual input that is somehow used to compose SQL queries. In some specific contexts, maliciously crafted input that contains SQL instructions or fragments produces queries whose semantics are different from those intended and may threaten the security policies of the underlying databases.

For example, in Figure 4 Java code excerpt, no verification is performed on user supplied fields (i.e., username and userpass) and a userpass string such as $hello\ OR\ '1 = 1';\ drop\ table\ users;$ can possibly destroy valuable information. Incidentally, Figure 4 fragment was extracted by a real Java servlet written for an on-line SOCCER labora-

Program	Reference	Reason
BIND	CA-1999-14	Size of arg of `memcopy` not checked
WU-FTPD/f1	CVE-1999-0878	Several `strcpy` calls without bounds check
WU-FTPD/f2	CA-2003-0466	Wrong size check in one `if`
WU-FTPD/f3	CVE-1999-0368	Several `strcpy, strcat` calls without bounds check

Table 1. BIND and WU-FTPD identified threats

tory application by software engineering students. Clearly, software engineering students must think about the consequences of vulnerabilities and defensive programming. Defensive programming and input validation aim to prevent the insertion of "malicious" strings into SQL queries. Unfortunately they may be sensitive to new patterns of attacks against which defenses were not planned. Furthermore, defensive programming may be better suited to new development and less to protection of legacy systems.

```
public boolean userExists(HttpServletRequest req,
       HttpServletResponse resp)
       throws ServletException, IOException {

 Connection conn = null;
 Statement stmt = null;
 ResultSet rs = null;

 String username = req.getParameter("Username");
 String userpass = req.getParameter("Userpass");

 String query =
    new String("Select * from users where user_name = '"
       + username
       + "' and user_passwd='"
       + userpass + "'" );
    /* data base connection code */
    ...
    /* query execution */
 try {
    ...
  stmt = conn.createStatement();
  rs = stmt.executeQuery(query);
  if (rs.next())
         return true;
   ...
  } catch(Exception e) {
        e.printStackTrace();
  }

}
```

Figure 4. Excerpt of Java code prone to SQL injection.

Available vulnerability protection and detection strategies can be categorized into dynamic or mixed (static and dynamic) approaches. Dynamic approaches check at run time the legitimacy of user supplied input; they can be further subdivided into two main families: black list and white list [37]. In a black or white list approach, queries are compared with known examples. The black list approach discards inputs leading to queries listed in the black list. On the other hand, only queries appearing in the white list are executed in a white list based approach. Black and white list implementation ranges from a simple table to finite state machines or to grammars. Clearly, creating and maintaining an exhaustive black list is a difficult task and thus white lists are preferable. Both black and white lists are sensitive to obsolescence of checks over time. As the application evolves, new legitimate interactions may be added to an application and lists must be updated.

Mixed static and dynamic approaches try to get the most out of the two strategies. In the static part, source code, data base schema and available information are used to build models of legitimate queries i.e., the white list.

For example, AMNESIA [33] is a tool for SQL-injection protection that uses model-based security by combining static and dynamic analysis of Java applications. AMNESIA builds a static model of legitimate SQL queries that can be generated by Java applications using Java string analysis [11].

The white list strategy has been the focus of several research papers with differences in the way in which white list was implemented, or input analyzed and validated; refer to [8, 14, 28, 47] for a more comprehensive view.

SQL queries that do not include user input or that cannot be reached by user input are intrinsically safe. This observation leads to a different prevention strategy based on the concept of "tainted/untainted" information. For example, *WebSSARI* is a tool [39] that performs flow Sensitive Type Based static analysis on PHP applications. Vulnerable sections of code are instrumented with run time guards against user induced SQL-injections and cross site scripting. WebSSARI uses a Perl-like two-level "tainted/untainted" type system defined for user input; more precisely, it implements a type-aware improvement of tainted/untainted (e.g., *tainted-string, tainted-integer, etc.*).

The tainted - untainted approach was recently extended to positive tainting of legitimate queries by Halfond et al [32, 35]. Positive tainting as opposed to traditional tainting identifies and tracks trusted rather than untrusted information, i.e., it is inspired by white list philosophy. The WASP (Web Application SQL-injection Preventer) tool uses SQL syntax aware evaluation coupled with the concept of positive tainting at character level to identify at run time legitimate queries belonging to a conceptual white list; it is conceptual in that no real white list exists in WASP.

SBSec and SBST have been almost absent in the SQL-injection arena. Test data generation for web applications or data bases has not yet exploited the powerful concept of SBST. Clearly much more can be done by the SBSec/SBST

community; all the necessary tools are available. Static source analysis tools can instrument source code and identify injection points; level and branch distances can be adapted to target the risky nodes; SQL grammars can be used to mutate user input to generate syntactically correct SQL queries. Indeed, it is hard to understand why tools such as WASP, AMNESIA or WebSSARI are not complemented by tool testing robustness to SQL injection.

2.3 Privilege Escalation Testing (CWE-285, CWE-259)

In [50, 51] privilege escalation is modeled as an insider threat. Nodes and transitions in an inter-procedural control flow graph are classified into authorization granting levels and/or tagged with a required level of authorization. Clearly, to perform an operation in a node with an authorization level $aLev$ (or to traverse a transition requiring authorization level $aLev$) an authorization level equal or higher should reach the node (transition). If a path is detected from a lower authorization level to a higher one, this indicates a possible privilege escalation vulnerability. The drawback is that the approach is a static and actually false positives exist. In an unpublished experiment the same authors of [50, 51] used random test data generation to detect privilege escalation in version 2.0.0 of phpBB[11]. SQL queries were captured, randomly mutated and seeded into phpBB. Even this simply search strategy was able to reveal privilege escalation threats. Random performance can probably improve via SBST; for example, a search heuristics can be coupled with the knowledge of statically identified authorization vulnerable nodes to force path execution through those nodes via level and branch distance fitnesses.

2.4 Robustness (Fuzz) and Penetration Testing

As stated in the introduction, penetration testing mimics a real-world attack [57]. There are two main categories: robustness testing and social engineering, non technical attacks. Our focus here is on the technical issues and robustness test to prove that an application is resilient to malformed inputs, incorrect protocol data sequence, or unspecified and ambiguous packet formats (e.g., the old Christmas tree attack) or robust under high load (denial-of-service attacks).

Random testing is one of the traditional weapons used to perform robustness testing [4, 24, 57]. It is often referred to as black box fuzz or fuzz testing. However, the simple uniform sampling on program input space in several experiments was able to detect only between 40% and 90 % of defects [9, 61]. When coupled with operational profiles [61] or with clever generation heuristics [9], fault revealing power of random test input data generation improves and becomes competitive with other testing strategies.

As in traditional testing the same difference between black and white box approaches also exists also in robustness and security testing; see for example a black box approaches [24] versus white box random testing [25, 26]. Random testing coupled with code instrumentation and constraint programming (i.e., white box random testing or white box fuzz) outperforms fuzz testing [25, 26].

Recently, Silvan Marquis et al. [46] took a new perspective. They approached robustness testing of applications via syntax base testing [5] a form of black box testing. Instead of the traditional approach, they observed that application communication protocols are often as complex as programming languages; they extended the Abstract Syntax Notation one (ASN.1) and defined a structure and context sensitive language (SCL) to model application level constraints on the syntax of protocol data unit. SCL can be used to model syntax of application protocols, semantic and constraints and let testers work at higher levels of abstraction, closer to the protocol specification level. Once a SCL abstraction is obtained, it can be used to mutate data of previous sessions captured by sniffing tools. SCL is part of a robustness testing project called Protocol Tester [46].

These concepts are powerful and many SBSec applications are possible. For example, SBST can be plugged into a protocol model creating a hybridization similar to [25, 26].

2.5 Intrusion Detection Systems

Intrusion Detection Systems (IDS) are software applications that identify and prevent possible misuse of a computerized system. Broadly speaking, they can be divided into two groups: signature based IDS, often dedicated to network and network traffic analysis, and anomaly-based IDS, tailored for detecting anomalous, usually process, activities (see for example [17]).

The first group includes IDS such as the open source SNORT [12] or Cisco IDS family [13]. CFengine [14] with the daemon `cfenvd` or RRDTOOL[15] are examples of activity and log analysis tools that can be configured to act as anomaly-based IDS. The main advantage of signature based IDS is that they are fast and often non intrusive, as they can run on a devoted host to sniff and analyze LAN traffic. Their weaknesses is that, being rule based, they fall short in detecting unknown types of attacks. On the other hand, the strength of anomaly-based approaches is their ability to identify unknown attacks. However, since they analyze log files and pattern of system calls, or in general, process and

[11] http://www.phpbb.com/

[12] http://www.snort.org/

[13] http://www.cisco.com/en/US/products/sw/secursw/ps2113

[14] http://www.cfengine.org/

[15] http://oss.oetiker.ch/rrdtool/

user activities, they tend to add extra loads to hosts where they are deployed.

There is no perfect solution and both approaches require a compromise between type I and type II errors. A type I error in this context is a false positive, an alarm not corresponding to an attack; a type II error leads to a more risky situation since it corresponds to an undetected attack.

Biologically inspired paradigms such as genetic algorithms (e.g., [10, 60]) and immune systems (e.g., [38, 31]) have been used to design and develop both types of IDS. IDS type I and II errors are usually evaluated on labeled traffic data, typically the DARPA Lincoln Laboratory data set.

The problem of detecting unforeseen intrusion attacks has also been tackled by genetic programming (see for example [44]). Instead of using chromosomes to model packets of traffic data or rules, more complex structures such as trees can be used as in [44] to model IDS rules and generate new rules. These are then evaluated on a subset of unused traffic data containing both legitimate traffic and previously unseen attacks.

Indeed, evaluation data is one of the weaknesses of IDS testing and one area where SBSec can contribute. These data sets are difficult to collect and label, but traffic data can be easily generated. For example, by adapting approaches such as white box fuzz [25, 26] and syntax based testing [46] to protocol description, legitimate as well as incorrect or attack like data can be generated. This data can be used alone or merged with sniffed traffic to create mutated data sets to assess robustness of application or the ability of IDSs to detect unforeseen patterns.

3 Open Challenges

Search based software engineering is such a general, powerful and flexible strategy that it is hard to imagine problems, vulnerabilities or tasks where it cannot be applied.

However, fundamental to any search based problem reformulation is the need to choose the most promising solution in a set. This may help to decide when a problem is very complex or easily tackled via SBSec. To automatically select the better solution, optimization approaches use a fitness function that encodes domain and problem knowledge. Domain and problem knowledge may be hard to model. If we need to test a network application for robustness, we may transform test data generation into syntax testing [46], apply genetic programming to the protocol representation, and promote test data making the network application to consume more resources. The knowledge required, i.e., protocol encoding, is general enough to be reused and easily encoded into a grammar so the development of fitness component (i.e., the protocol grammar) is justified.

On the other extreme, consider the password cracking problem. For example, Unix/Linux passwords are encoded by the DES (the Digital Encryption Standard) algorithm. Actually, a conceptually modified version in which the sequence of characters stored in the operating system files is the public key while the actual user password is the private key. This latter fact makes SBSec approaches to password cracking very hard or non competitive with brute force strategies. Indeed, they likely degenerate into random search since there is no guidance: either the search hits the password or it fails. There is no intermediate state between success (decrypted) and failure; the landscape of a search space is flat with just one success point, the exact character string matching the user password.

The situation may be different if instead of a single password the task is to crack a set of passwords from the same organization and selected by people of the same geographic area and homogeneous culture. Password cracking systems such as John the Ripper [16] use a huge dictionary and apply smart algorithms to search for hits. They also define rules on how input dictionary words should be transformed, concatenated, or altered. People in the same organization and geographical region may use common rules to create the password such as appending a punctuation sign prefixed by a number or date, adding the name of a football player to the date of birth of a relative, and so on. If there is a population of transformations and alteration rules evolving from an initial population of well known alterations, a fitness function can promote rules generating sequences of characters (starting from the dictionary) cracking one or more passwords.

In essence, foreseeable limitations are mostly due to the structure of the problem, the associated search landscape, and the cost of acquiring and encoding problem and domain knowledge.

Overall, there is no single answer when considering what to break, how to do it, and what to do to improve existing approaches and technologies. Although there are some open research questions specific to test input data generation for vulnerability exposure, most of them are of a wider scope and pertain to the widest SBST research community.

3.1 What To Break?

First and foremost, vulnerability is a system wise concept and ultimately it is the system that should be resilient to attacks. Hackers and attacks do not follow guidelines or rules; the only valid rule is that there is no rule. Attacks can be performed at any level and by all means; thus the SBSec community should apply the same strategy. No matter the component level (function, method, class or subsystem) or types of input parameters, SBSec researchers should devise

[16] http://www.openwall.com/john/

approaches to expose vulnerabilities. In hackers' philosophy, binary code is as good as the source code and any approach, knowledge or tool to crash the application, escalate privilege, or access protected resources can be exploited. For example, on Unix machines buffers have a default size, a multiple of some constant (usually defined via the macro BUFSIZ). Also, memory allocation via sbrk C wrapper is a multiple of a fixed quantity (often 4094). To crash the program in Figure 1, this knowledge can easily be incorporated into a test data generation process by initializing a string population with individuals longer than BUFSIZ.

Binary instrumentation technology and tools exist, see for example the Valgrind[17] tool suite, but virtually no work has addressed the problem of generating test input data starting from binaries. This is surprising since from a practical standpoint only the binary may be available. In such a case, SBST researchers could resort to tool sets similar to those used by hackers: binary debuggers, code reverse engineering, dynamic linked library hacking, binary instrumentation, or processor emulation.

3.2 System Wide Test

SBST has been very successful in generating test input data at function method or class level. Techniques and approaches developed at the component level for traditional white and black box criteria can be adapted to vulnerability testing. This is the main idea applied in [29] where SBST statement coverage strategy was adopted for buffer overrun exposure. However, vulnerabilities may depend on the interaction of different components. SBST system wide approaches in testing for vulnerability exposure are limited to [50, 51]. Also, since any SQL injection protection system works at system level, it is hard to understand why tools, approaches and publications exist for both Java and PHP to protect against SQL injection, while virtually none exist on the test input data generation to expose SQL injection. In the same way, only a limited body of work exists [62, 63, 64] specifically targeting exceptions such as numeric overflow or underflow, division by zero, or the more traditional buffer overflow.

3.3 Data Types

When input data are numeric, several mutation, crossover and initialization strategies are available (see for example [23, 27]). The same is no longer true for string data type, pointers, structures or objects and only a few contributions exist [2, 41]. Recently Alshraideh and Botacci elaborated on the search space of strings and operators for string manipulation in SBST [2]. Among the many possible string distances [55], they reported that a modified edit distance using a character ordinal distance performs the best. As noticed in [2], input strings are often derived from English terms and sometimes loosely based on English grammar. Thus, one way to improve search performances could be to incorporate some linguistic structures in the string generation and transformation processes. What types of linguistic knowledge, structure and model work best is another interesting research question. At the lowest level, dictionaries and thesauri could be useful. For example, in password cracking, large multi-language dictionaries are often the key to success. At higher level statistic language models such as n-gram [56, 59] may be used to enforce a tighter linguistic structure in the generated character sequences.

A further challenge is posed by multi-language applications where user input strings can be written in any known language; for example, in the European Community, web applications should be able to handle all the official languages. It is worth noticing that an attacker can use any language but virtually no work exists on generating test input string in languages such as French, Italian, or Spanish.

The dynamic structure, whatever the stored data, is another SBST sub area where interesting research questions remain unanswered. Indeed, only very recently have approaches such as the Concolic testing [25, 40, 45, 58] and hill climbing [41] been used in the presence of dynamic structures. Concolic testing requires symbolic execution and the use of a constraint solver. Scalability and system level testing is unclear; under some specific circumstances data generation via constraint solver degenerates into random search. On the other hand, hill climbing is a local search method prone to local optima that does not necessarily explore the search space efficiently. Challenges range from how a fitness function or a neighborhood should be defined to which are the best operators to apply or how domain and problem knowledge can be incorporated into the search process. On the practical side, scalability, efficiency and robustness are key issues.

SQL injection depends on user trusted input and, of course, on the structure of the data base underlying the web application. Testing data bases is a research area; however, here the goal is to integrate knowledge of application (e.g., the fact that it is a multi-tier architecture) and database structure to expose SQL injection. SQL queries are strings, but generated strings seeded into the database, and queries must follow SQL syntax and should also embed SQL fragments potentially leading to vulnerability exposure. This in turn opens another interesting practical question on fitness or distance definition, i.e., how should fitness between generated queries, traversed code, collected information and returned answers be defined so that the search will efficiently generate and expose an SQL injection attack.

Finally, another interesting open problem is how to define the limit between complexity of data types and data

[17] http://valgrind.org/

types manipulation functions and the SBST gain in effort saving (see the need of a compromise between BBF and DWF). Consider the old Acrobat 5 security vulnerability SA8739, released 07-05-2003, which was caused by an error in the JavaScript engine. A malicious user could have exploited this by crafting a special PDF document. The probability of randomly generating such a particular PDF document is virtually zero. Complexity to manipulate PDF documents may not justify the investment in applying SBST, but this is a very general question that applies to any testing automation technology. There is always a trade-off between the cost of adopting a technology and the benefits gained.

3.4 When Enough is Enough

Any white box test adequacy criterion has the indubitable advantage of a clear, well understood and easy measure of how close we are to reaching our testing goal. Once we have covered all branches, we know that all statements have also been covered; if we cover 90 % of branches we know we still have to cover 10 % of edges. White box testing makes it easy to check stopping criteria; unfortunately, there is no one to one relation between achieved coverage and detected defects.

Empirical studies provide evidence on the fault revealing power of different white box coverage criteria [54]. Of course, this and similar studies are done in vitro (i.e., in a controlled environment), but according to the findings [54], we may expect more defect revealing power from data dependency criteria with respect to statement or branch coverage. Nevertheless programmers remain uncertain whether they have run enough test cases to discover enough or all defects.

Statistical testing [53] builds models to estimate software reliability and predict the number of remaining undiscovered defects. Other techniques such as the capture-recapture approaches [6, 21, 66, 67] applied to inspections attempt to find out how many defects are still hidden in the system.

When applying SBST, researchers and practitioners are left wondering if they have generated enough test input data for a specific goal. In such a case even an estimated upper bound can be useful. For example no regression test case selection can rely on Chernoff bound [43] to get an estimate of reliability confidence level. Similar bounds could help programmers to decide when enough data have been generated or if the risk of a vulnerability slipped into code is acceptable.

3.5 Exception Raising and Singularities Search

Ultimately certain families of exploits (e.g., buffer overflow) rely on exceptions causing unplanned behavior. SBST searches the input space for data points leading to singularities in the output and unpredicted and unwanted behaviors. SBST has been applied to the problem of exception raising by John Clark and his colleagues at the University of York [62, 63, 64]. Implicit condition leading to exception (e.g., division by zero, integer or real exceptions) are explicitly modeled via source code transformations (i.e., adding else branches); in this clever approach the problem of exception raising is transformed into a branch coverage problem. Clark applied simulation annealing and genetic algorithm and results have been reported for programs up to about 600 lines. Conceptually, there is no reason why the approach could not be extended to larger chunks of code although the program transformation task can be at the minimum very complex and challenging [1]. For example one of the challenges identified by Harman in his 2008 keynote was: *can we decompose hard exception conditions into a series of easier conditions?*

Once a unit under test has been selected, a technique to reduce the program input domain [36] can be adopted to simplify the program before source code transformation; this also has the advantage of reducing the search space dimension and speeding up the search. However, from a practical standpoint, it remains unclear how large and complex C functions, say with 20 or plus parameters and a cyclomatic complexity above 100, can be transformed and tested. Test data generation to raise exceptions tied to numeric or logical values are reasonably well understood, but certain types of exceptions are due to pointers, memory management, and dynamic structures, and authors have only recently addressed relatively close challenges in [41].

Overall, the problem can be exacerbated by the fact that we almost never have guidance to decide "when enough is enough" and we may be testing for something that is simply impossible to attain due to physical or domain constraints or defensive programming.

A type of attack conceptually related to exception raising is the denial-of-service (DoS) attack. A DoS attack does not necessarily require an exception; it is characterized by an explicit attempt to prevent legitimate users of a service from accessing it. For example, the disk area can be used to store illegal material, the network can be flooded, or the attack target can be the consumption of limited and scarce resources. Temporal testing [7, 68, 69] can be extended to model certain types of DoS. For example, one may want to prove that the CPU, the memory, or other scarce resources will always be available for the most critical processes in the system.

Penetration testing, for example, flooding a LAN to simulate a DoS attack may not be the easiest and best way to apply SBSec to DoS. Incidentally, it may also be a way to run into trouble with colleagues or the network administrator. However, one may be interested in testing critical

component of an operating system under different stress and loading condition. One of the features of Unix is that once a process obtains a chunk of memory, it is never returned to the operating system until the process ends. Thus, for example, SBST can be applied to test if a malicious user can craft process generation or memory allocation strategies leading to a DoS like situation. Clearly, a simulated in vitro environment must be instantiated via virtual machines or on a sub-network isolated from the outside world.

3.6 The Very Best Search Strategy

The optimization community has developed several biology inspired optimization paradigms from genetic to memetic algorithms, ant colonies, particle swarm optimization, artificial immune systems, or bee colonies. Plus, the SBST community can rely on other search strategies such as simple hill climbing, simulation annealing, estimation of distribution algorithm, or tabu search. New heuristics and search strategies are likely to be developed, thus increasing the army of researchers; however, this is also a source of possible confusion by comparing very different things.

If we consider the number of possible optimization strategies and the number of possible applications to detect vulnerabilities, it is unclear what would be the best search strategy for a given testing problem. The testing community mitigates this problem via the software-artifact infrastructure repository (SIR)[18]. Security related issues and vulnerabilities are documented by government agencies (see for example Mitre Corporation or the SANS Institution.

Unfortunately, at the time of writing, the SBSec community lacks a similar resource, a set of vulnerable applications to be used as a benchmark in comparing different techniques and approaches. One way to compare would be to rely, for example, on the set of Unix applications collected by R. Lippman at MIT Lincoln Laboratory[19]. The suite comprises widely used open source programs: BIND, WU-FTPD and SENDMAIL. BIND is probably the most widely used DNS server; WU-FTPD is a popular ftp daemon; SENDMAIL is the most widely used mail transport agent. Several serious vulnerabilities have been identified in these programs. Indeed, these applications have been documented as vulnerable and have been previously used [70] to evaluate the ability of static tools to detect security risks. However, these are C buffer overflow prone applications and the SBSec community needs to cover a larger variety of threats, languages and domains.

Overall, the SBST community should focus on the challenging task of defining a common set of utilities, tasks and problems to compare different approaches in a sound and meaningful way to advance the field and the acceptance of the widest community.

[18]http://sir.unl.edu/portal/index.html

[19]http://www.ll.mit.edu/IST/corpora.html

3.7 Developing the Very Best Environment

The SBST and SBSec communities should address the difficulties when striving to make software more robust and resilient to attacks by developing more cost-effective vulnerability detection approaches. More specifically, SBSec researchers should develop sophisticated tools for the effective planning, management and implementation of software penetration testing, and vulnerability detection and robustness testing for sizable applications and systems. These tools should be very easy to use; they should pass an equivalent for computer science of the ”mom test”; if my 80-year old mom does not adopt a technology, you lose an important market share, and your technology is likely not to be mature. Similarly, if programmers and managers do not adopt what we develop in our laboratories, then we have a problem. Indeed, one of the key indicators of a mature technology is the ease of use for both non-experts and professionals. Finally, we should strive to make the tools as independent as possible from the skill of the tester so that that both a novice and an expert will obtain identical or very similar results. Tool for stress testing, penetration testing, and buffer overflow detection can help to free valuable resources from the task of identifying most easily detected vulnerabilities and concentrate on more challenging tasks, such as code reviews or IDS alarm auditing where the judgment of the expert is vital and will never be substituted by a software application.

SBSec tool development is no longer an impossible, extremely expensive or very difficult task. Eclipse has given to the software engineering community a tremendous opportunity to set the stage for profound revolution. Technology transfer from research to market depends on several factors such as the domain e.g., automotive versus information technologies, the regulations e.g., food and drug administration, the market opportunity e.g., lacking of competitive technologies. Eclipse integrates software analysis research ideas useful for any test data generation algorithm; its open framework architecture makes it easy to implement and deploy new plugins. By relying on the Eclipse infrastructure to develop our SBSec plugins and make them available over the network, the technology transfer can be substantially shortened and likely measured in years instead of decades.

4 Conclusion

The widespread adoption of computerized systems and the role played by software applications in our society makes it increasingly important to ensure high dependability. Although SBST application to expose software vulner-

abilities is a relatively young field, the potential in term of benefits is enormous.

Every day new applications are developed, old software is modified and a huge amount of software needs to be tested. Network applications and operating systems are exposed to malicious attacks and SBST/SBSec can contribute to exposing vulnerability, thus making information more secure and reducing the risk of financial losses or the threats to vital infrastructures.

5 Acknowledgments

This paper is a brief account of the topics and open problem raised by the author's keynote address at the Second Search Base Testing Workshop in Denver, Colorado, April 2008. However, many other researchers I have worked with contributed to the outlined ideas.

I have worked on search based software engineering with many researchers and I am indebted to them for their help and contributions to the ideas presented in the paper. I have worked on security and vulnerability issues with Tina del Grosso, Massimiliano di Penta, Philip Galinier, Dominic Letarte and Ettore Merlo. In particular, the idea of exposing insider threats came from the work of Ettore Merlo and Dominic Letarte. I am specially indebted to Tina del Grosso and Massimiliano di Penta who helped me to tackle the buffer overflow vulnerability via SBST.

All my research in recent years has been funded by the Natural Sciences and Engineering Research Council of Canada (Research Chair in Software Evolution #950-202658) and available at the SOftware Cost-effective Change and Evolution Research (SOCCER) laboratory web site [20].

I am grateful to my colleagues, collaborators and students. In particular a special thanks to Philippe Galinier, Kamel Ayari and Zeina Awedikian for our many conversations on software testing and meta-heuristic application to software testing.

References

[1] Open problems in testability transformation. In *ICSTW '08: Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 196–209, Washington, DC, USA, 2008. IEEE Computer Society.

[2] M. Alshraideh and L. Bottaci. Search-based software test data generation for string data using program-specific search operators: Research articles. *Softw. Test. Verif. Reliab.*, 16(3):175–203, 2006.

[3] C. Anley. Advanced sql injection. In *Technical report*. NGSSoftware Insight Security Research, 2002.

[20] http://web.soccerlab.polymtl.ca

[4] Arkin, Brad, Stender, Scott, and McGraw. Software penetration testing. *IEEE Security and Privacy*, 3(1):84–87, 2005.

[5] B. Beizer. *Software Testing Techniques 2nd edition*. International Thomson Computer Press, 1990.

[6] L. C. Briand, K. E. Emam, B. G. Freimut, and O. Laitenberger. A comprehensive evaluation of capture-recapture models for estimating software defect content. *IEEE Trans. Softw. Eng.*, 26(6):518–540, 2000.

[7] L. C. Briand, Y. Labiche, and M. Shousha. Stress testing real-time systems with genetic algorithms. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1021–1028, New York, NY, USA, 2005. ACM.

[8] G. T. Buehrer, B. W. Weide, and P. A. G. Sivilotti. Using parse tree validation to prevent sql injection attacks. In *5th international workshop on Software engineering and middleware SIGSOFT: ACM Special Interest Group on Software Engineering*, pages 106 – 113, 2005.

[9] P. Bueno, E. Wong, and M. Jino. Improving random test sets using the diversity oriented test data generation. In *RT '07: Proceedings of the 2nd international workshop on Random testing*, pages 10–17, New York, NY, USA, 2007. ACM.

[10] A. Chittur. *Model Generation for an Intrusion Detection System Using Genetic Algorithms*. Ossining High School - High School Honors Thesis, New York, November 2001.

[11] A. S. Christensen, A. Moller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proc. of the 10th International Static Analysis Symposium SAS*, pages 1–18. Springer-Verlag, June 2003.

[12] S. Christey. Unforgivable vulnerabilities. *Black Hat Briefings 2007 - CVE.MITRE.ORG*, August 2 2007.

[13] S. Christey and R. Martin. Vulnerability type distributions in cve (2001-2006). *White Paper - CVE.MITRE.ORG*, May 22 2007.

[14] W. R. Cook and S. Rai. Safe query objects: Statically typed objects as remotely executable queries. In *Proc. of the 27th International Conference on Software Engineering (ICSE)*. IEEE Computer Society Press, 2005.

[15] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium*, pages 5–5, Berkeley CA USA, 1998. USENIX Association.

[16] D. DaCosta, C. Dahn, S. Mancoridis, and V. Prevelakis. Characterizing the 'security vulnerability likelihood' of software functions. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 266–276, Amsterdam The Netherlands, Oct 2003.

[17] D. Denning. An intrusion-detection model. *IEEE Trans. Softw. Eng.*, 13(2):222–232, 1987.

[18] G. Dozier, D. Brown, H. Hou, and J. Hurley. Vulnerability analysis of immunity-based intrusion detection systems using genetic and evolutionary hackers. *Appl. Soft Comput.*, 7(2):547–553, 2007.

[19] G. Dozier, D. Brown, J. Hurley, and K. Cain. Vulnerability analysis of immunity-based intrusion detection systems using evolutionary hackers. In *GECCO*, pages 263–274, 2004.

[20] J. Duran and S. Ntafos. An evaluation of random testing. *IEEE Trans. Software Eng.*, 10(4):438–444, 1984.

[21] K. E. Emam and O. Laitenberger. Evaluating capture-recapture models with two inspectors. *IEEE Trans. Softw. Eng.*, 27(9):851–864, 2001.

[22] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, pages 42–50, Jan/Feb 2002.

[23] E. Falkenauer. *Genetic Algorithms and Grouping Problems*. Wiley-Inter Science, Wiley - NY, 1998.

[24] J. Forrester and B. Miller. An empirical study of the robustness of windows nt applications using random testing. In *WSS'00: Proceedings of the 4th conference on USENIX Windows Systems Symposium*, pages 6–6, Berkeley, CA, USA, 2000. USENIX Association.

[25] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, New York, NY, USA, 2005. ACM.

[26] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. Technical Report MS-TR-2007-58, Microsoft Research, May 2007.

[27] D. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning*. Addison-Wesley Pub Co, Jan 1989.

[28] C. Gould, Z. Su, and P. Devanbu. Jdbc checker: A static analysis tool for sql/jdbc applications. In *Proc. of the 26th International Conference on Software Engineering (ICSE) - Formal Demos*, pages 697–698. IEEE Computer Society Press, 2004.

[29] C. D. Grosso, G. Antoniol, E. Merlo, and P. Galinier. Detecting buffer overflow via automatic test input data generation. *Comput. Oper. Res.*, 35(10):3125–3143, 2008.

[30] C. D. Grosso, M. D. Penta, G. Antoniol, E. Merlo, and P. Galinier. Improving network applications security: a new heuristic to generate stress testing data. In *Proc. of the Genetic and Evolutionary Computation Conference*, pages 1037–1043, Washington DC USA, June 25-29 2005.

[31] J. Gupta and S. Sharma. *Handbook of Research on Information Assurance and Security*. IGI Global, New York, November 2009.

[32] W. Halfond, A. Orso, and P. Manolios. Wasp: Protecting web applications using positive tainting and syntax-aware evaluation. *IEEE Transactions on Software Engineering (TSE)*, 34(1):65–81, 2008.

[33] W. G. J. Halfond and A. Orso. Combining static analysis and runtime monitoring to counter sql-injection attacks. In *Proc. of the 3rd International ICSE Workshop on Dynamic Analysis (WODA)*. IEEE Computer Society Press, May 2005.

[34] W. G. J. Halfond and A. Orso. Improving test case generation for web applications using automated interface discovery. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 145–154, New York NY USA, 2007. ACM.

[35] W. G. J. Halfond, A. Orso, and P. Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 175–185, New York NY USA, 2006. ACM.

[36] M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, and J. Wegener. The impact of input domain reduction on search-based test data generation. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 155–164, New York, NY, USA, 2007. ACM.

[37] G. Hoglund and G. McGraw. *Exploiting Software How to Break Code*. Addison-Wesley Publishing Company, AWDD, 2004.

[38] H. Hou and G. Dozier. Immunity-based intrusion detection system design, vulnerability analysis, and genertia's genetic arms race. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 952–956, New York, NY, USA, 2005. ACM.

[39] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proc. of the 12th International World Wide Web Conference (WWW)*, May 2004.

[40] K. Inkumsah and T. Xie. Evacon: a framework for integrating evolutionary and concolic testing for object-oriented programs. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 425–428, New York, NY, USA, 2007. ACM.

[41] K. Lakhotia, M. Harman, and P. McMinn. Handling dynamic data structures in search based testing. In *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1759–1766, New York, NY, USA, 2008. ACM.

[42] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, pages 177–190, Washington D. C, August 13-17 2001.

[43] W. P. Li and M. J. Harrold. Using random test selection to gain confidence in modified software. In *International Conference on Software Maintenance (ICSM 2008)*, pages 267–276, Beijing, China, October 2008.

[44] W. Lu and I. Traore. Detecting new forms of network intrusion using genetic programming. *Computational Intelligence*, 20(3):475–494, 2004.

[45] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 416–426, Washington, DC, USA, 2007. IEEE Computer Society.

[46] S. Marquis, T. Dean, and S. Knight. Scl: a language for security testing of network applications. In *CASCON '05: Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, pages 155–164. IBM Press, 2005.

[47] R. McClure and I. Kruger. Sql dom: Compile time checking of dynamic sql statements. In *Proc. of the 27th International Conference on Software Engineering (ICSE)*, pages 88–96. IEEE Computer Society Press, 2005.

[48] P. McMinn. Search-based software test data generation: a survey. *Softw. Test Verif. Reliab*, 14(2):105–156, 2004.

[49] E. Merlo and G. Antoniol. A static measure of a subset of intra-procedural data flow testing coverage based on node coverage. In *Proceedings of CASCON-99 - ponsored by IBM Canada and the National Reasearch Council of Canada*, pages 173–186, Mississauga (Ontario), November 8-11 1999.

[50] E. Merlo, D. Letarte, and G. Antoniol. Insider and ousider threat-sensitive sql injection vulnerability analysis in php. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 147–156, Washington DC USA, 2006. IEEE Computer Society.

[51] E. Merlo, D. Letarte, and G. Antoniol. Automated protection of php applications against sql-injection attacks. In *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 191–202, Washington DC USA, 2007. IEEE Computer Society.

[52] W. Miller and D. L. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, 2(3):223–226, sep 1976.

[53] J. D. Musa, A. Iannino, and K. Okumoto. *Software Reliability Measurement Prediction Application Professional Edition*. McGraw-Hill, NY, 1990.

[54] A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Trans. Softw. Eng.*, 32(8):608–624, 2006. James H. Andrews and Lionel C. Briand and Yvan Labiche.

[55] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[56] D. M. R. *Spoken dialogues with computers*. Academic Press, Inc., Orlando, Florida 32887, 1998.

[57] K. Scarfone, M. Souppaya, A. Cody, and A. Orebaugh. Technical guide to information security testing and assessment. Technical Report Spec. Publ. 800-11, U.S. DEPARTMENT OF COMMERCE - National Institute f Standards and Technology, September 2008.

[58] K. Sen. Concolic testing. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 571–572, New York, NY, USA, 2007. ACM.

[59] S. M. Shieber. *An Introduction to Unification-Based Approaches to Grammars*, volume 1. Center for the Study of Language and Information, Leland Stanford Junior University, 1986.

[60] T. Shon and J. Moon. A hybrid machine learning approach to network anomaly detection. *Inf. Sci.*, 177(18):3799–3821, 2007.

[61] P. Thévenod-Fosse, H. Waeselynck, and Y. Crouzet. An experimental study on software structural testing: Deterministic versus random input generation. In *FTCS*, pages 410–417, 1991.

[62] N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. In *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, pages 73–81, New York, NY, USA, 1998. ACM.

[63] N. Tracey, J. Clark, K. Mander, and J. McDermid. Automated test-data generation for exception conditions. *Softw. Pract. Exper.*, 30(1):61–79, 2000.

[64] N. Tracey, J. Clark, J. McDermid, and K. Mander. A search-based automated test-data generation framework for safety-critical systems. Systems engineering for business process change new directions:174–213, 2002.

[65] M. Vernon. Top five vulnerabilities. *IT management - Risk Management*, (201840):http://www.computerweekly.com/Articles/2004/04/16/201840/Top+five+threats.htm, 2004.

[66] G. S. Walia and J. C. Carver. Evaluation of capture-recapture models for estimating the abundance of naturally-occurring defects. In *ESEM '08: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 158–167, New York, NY, USA, 2008. ACM.

[67] G. S. Walia, J. C. Carver, and N. Nagappan. The effect of the number of inspectors on the defect estimates produced by capture-recapture models. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 331–340, New York, NY, USA, 2008. ACM.

[68] J. Wegener, M. Grochtmann, and B. Jones. Testing temporal correctness of real-time systems by means of genetic algorithms. In *Software Quality Week*, 1997.

[69] J. Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Syst.*, 21(3):241–268, 2001.

[70] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 97–106, New York, NY, USA, 2004. ACM Press.