A Combinatorial Approach to Detecting Buffer Overflow Vulnerabilities

Wenhua Wang, Yu Lei, Donggang Liu, David Kung, Christoph Csallner, Dazhi Zhang
Department of Computer Science and Engineering
The University of Texas at Arlington
Arlington, Texas 76019, USA
{wenhuawang,ylei,dliu,kung,csallner,dazhi}@uta.edu

Raghu Kacker, Rick Kuhn
Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, Maryland 20899, USA
{raghu.kacker,kuhn}@nist.gov

Abstract— **Buffer overflow vulnerabilities are program defects that can cause a buffer to overflow at runtime. Many security attacks exploit buffer overflow vulnerabilities to compromise critical data structures. In this paper, we present a black-box testing approach to detecting buffer overflow vulnerabilities. Our approach is motivated by a reflection on how buffer overflow vulnerabilities are exploited in practice. In most cases the attacker can influence the behavior of a target system only by controlling its external parameters. Therefore, launching a successful attack often amounts to a clever way of tweaking the values of external parameters. We simulate the process performed by the attacker, but in a more systematic manner. A novel aspect of our approach is that it adapts a general software testing technique called combinatorial testing to the domain of security testing. In particular, our approach exploits the fact that combinatorial testing often achieves a high level of code coverage. We have implemented our approach in a prototype tool called *Tance*. The results of applying *Tance* to five open-source programs show that our approach can be very effective in detecting buffer overflow vulnerabilities.**

Keywords: Software Security; Security Testing; Buffer Overflow Vulnerability

I. INTRODUCTION

Software security is a priority concern in many security assurance efforts [26]. Existing approaches to software security assurance can be largely classified into two categories. The first category is based on static analysis [6][30], which checks security properties by analyzing the source code of a subject program, without executing the program. The second category is based on dynamic analysis or testing [1][31], which executes the subject program and checks whether the program's runtime behavior satisfies some expected security properties. Static analysis is typically fast and can be used to prove properties about a program. However, static analysis suffers from false positives or false negatives or both. Testing, on the other hand, only reports problems that have been observed at runtime. However, testing requires test input selection and program execution, which can be difficult and time consuming. Furthermore, testing typically cannot be exhaustive.

In this paper, we present a testing approach to detecting buffer overflow vulnerabilities. A buffer overflow occurs when data is written beyond the boundary of an array-like data structure. Buffer overflow vulnerabilities are program defects that can cause a buffer overflow to occur at runtime. Many security attacks exploit buffer overflow vulnerabilities to compromise critical data structures, so that they can influence or even take control over the behavior of a target system [27][32][33].

Our approach is a specification-based or black-box testing approach. That is, we generate test data based on a specification of the subject program, without analyzing the source code of the program. The specification required by our approach is lightweight and does not have to be formal. In contrast, white-box testing approaches [5][13] derive test inputs by analyzing the data and/or control structure of the source code. Black-box testing has the advantage of requiring no access to the source code, and is widely used in practice. However, it often suffers from poor code coverage. Code coverage is considered to be an important indicator of testing effectiveness [2]. A black-box testing approach that has gained significant recognition in practice is called fuzz testing or fuzzing [35], which generates test data randomly. Smart fuzzing [14][15] applies advanced heuristics and/or takes advantage of additional information, e.g., domain knowledge. Fuzzing has been shown effective for detecting software vulnerabilities. However, like other black-box testing techniques, poor code coverage is considered to be a major limitation of fuzzing [35].

A major objective of our approach is to achieve good code coverage while retaining the advantages of black-box testing. Our approach was motivated by a reflection on how buffer overflow vulnerabilities are exploited by the attacker in practice. In most cases the attacker can influence the behavior of a target system only by controlling the values of its external parameters. External parameters are factors that could potentially affect the system behavior. Examples of external parameters include input parameters, configuration options, and environment variables. Therefore, launching a successful attack often amounts to a clever way of exploring the input space, typically by tweaking the values of external parameters. (In this paper, we do not consider interactive systems, i.e. systems that require a sequence of user interactions in the course of a computation, where each interaction may depend on the outcome of the previous interactions.) Security testing essentially needs to do the same thing, but in a more systematic manner and with a good intent.

In a typical exploit attempt, the tweaking of external parameter values consists of the following two major steps. First, the attacker identifies a single external parameter *P* to

978-1-4244-9233-6/11/$26.00 ©2011 IEEE

carry the attack data. We will refer to *P* as the *attack-payload* parameter. *P* is often chosen such that during program execution, its value is likely to be copied into a buffer *B* that is vulnerable to overflow and that is located close to a critical data structure *C*, e.g., a return address on the call stack, which the attacker intends to compromise. The attack data is intended to overwrite *C* in a specific way that allows the attacker to gain control over the program execution. In Section IV.A, we provide empirical evidence that a single external parameter is used to carry the attack data in many buffer overflow attacks. In other words, there exists a single *attack-payload* parameter in many buffer overflow attacks. This observation plays a significant role in the design of our approach, as discussed in Section II.

Second, the attacker tries to assign proper values to the rest of the external parameters. Some of these parameters can take arbitrary values, i.e., the exploit attempt will succeed or fail, regardless of the values of these parameters. However, other parameters may be important for steering the program execution to reach a *vulnerable point*, i.e., a statement that actually copies the value of *attack-payload* parameter *P* into buffer *B* and whose execution may give rise to a buffer overflow. We will refer to these parameters as *attack-control* parameters. Choosing appropriate values for the *attack-control* parameters is as critical as choosing an appropriate value for the *attack-payload* parameter in order to carry out a successful attack.

The two steps described above can be repeated by using a different parameter as the *attack-payload* parameter. The main idea of our approach is trying to *systematize the tweaking of external parameter values in a typical exploit attempt as described above*. Specifically, we identify two conditions that must be met in order to expose a buffer overflow vulnerability. Our approach is centered on how to generate tests such that these two conditions are likely to be satisfied for potentially vulnerable points. We provide guidelines on how to identify *attack-payload* and *attack-control* parameters and a set of values for each of these parameters. Moreover, we adapt a technique called combinatorial testing [4][7] to generate a group of tests for each value identified for each *attack-payload* parameter, such that *one of these tests is likely to steer program execution to reach a vulnerable point* that may be triggered by this value. Combinatorial testing has been shown very effective for general software testing [7][16][20]. In particular, empirical results suggest that there exists a high correlation between combinatorial coverage and code coverage [3][11]. It is this correlation that is exploited in our approach to increase the likelihood for our tests to reach a vulnerable point, and thus the likelihood to detect buffer overflow vulnerabilities.

Note that attack data often needs to be carefully crafted in order to carry out a real attack. However, for testing, our goal is to demonstrate the possibility of a buffer overflow, i.e., not to acquire specific control to do any real harm. As discussed in Section II, this greatly simplifies the selection of parameter values, especially for the *attack-payload* parameters.

For the purpose of evaluation, we implemented our approach in a prototype tool called *Tance*. We conducted experiments on five open-source programs: *Ghttpd* [12], *Gzip* [17], *Hypermail* [19], *Nullhttpd* [28], and *Pine* [29]. The results show that our approach can effectively detect buffer overflow vulnerabilities in these programs. In particular, we examined vulnerability reports in three public vulnerability databases. This examination showed that our approach detected all the known vulnerabilities but one for the first four programs. For the last program, i.e., *Pine*, insufficient information was available to determine whether the reported vulnerabilities were the same as the ones we detected. In addition, our approach detected a total of 9 new vulnerabilities that have not been reported in the three databases.

The remainder of this paper is organized as follows: Section II describes our approach and presents an algorithm that implements our approach. Section III describes the design of our prototype tool, namely, *Tance*. Section IV presents the results of our case studies. Section V discusses related work on detecting buffer overflow vulnerabilities. Section VI concludes this paper and discusses future work.

II. THE APPROACH

In this section, we present our approach to detecting buffer overflow vulnerabilities. Section II.A explains combinatorial testing in more detail. Section II.B illustrates the main idea of our approach. Section II.C presents an algorithm that implements our approach. Section II.D provides additional discussion on the practical applications of our approach.

A. Combinatorial Testing

Let *M* be a program with *n* parameters. Combinatorial testing, which is also referred to as *t*-way testing, requires that, for *any t* (out of *n*) parameters of *M*, every combination of values of these *t* parameters be covered at least once. The value of *t* is referred to as the strength of combinatorial testing. Consider a program that has three parameters *p1*, *p2*, and *p3*, each parameter having two values, 0 and 1. Fig. 1 shows a 2-way (or pairwise) test set for these three parameters. Each row represents a test, and each column represents a parameter (in the sense that each entry in a column is a value of the parameter represented by the column). An important property of this test set is that if we pick any two columns, i.e., columns *p1* and *p2*, columns *p1* and *p3*, or columns *p2* and *p3*, they contain all four possible pairs of values of the corresponding parameters, i.e., {00, 01, 10, 11}. An exhaustive test set for these parameters would consist of $2^3 = 8$ tests.

A number of algorithms have been developed for combinatorial test generation [4]. In particular, we reported on a combinatorial testing strategy called *IPOG* [24]. The *IPOG* strategy generates a *t*-way test set to cover the first *t* parameters and then extends this test set to cover the first *t* + 1 parameters. This process is repeated until this test set covers all the parameters. In [24], we reported a tool called *ACTS* (formerly known as *FireEye*) that implements the

IPOG strategy. Our approach uses *ACTS* to generate combinatorial test sets.

Combinatorial testing can significantly reduce the number of tests. For example, a pairwise test set for 10 Boolean parameters only needs as few as 13 tests, whereas an exhaustive test set consists of 1024 tests [7]. Despite this dramatic decrease in the number of tests, combinatorial testing has been shown very effective for general software testing [20]. In particular, empirical results suggest that there exists a high correlation between combinatorial coverage and branch coverage [3][11]. It is this correlation that is exploited in our approach to achieve a high level of code coverage.

Recently, we applied combinatorial testing to several different domains, including web application testing [36], concurrency testing [23], and web navigation graph construction [37]. Combinatorial testing has also been applied to testing configurable systems in the presence of constraints [8]. We believe that our work presented in this paper is the first attempt to apply combinatorial testing to the domain of software security testing.

p1	p2	p3
0	0	0
0	1	1
1	0	1
1	1	0

Figure 1. An example pairwise test set

B. Main Idea

We use an example to illustrate the main idea of our approach. Assume that a statement *L* copies a string variable *S* into a buffer *B*, without checking whether *B* has enough space to hold *S*. Thus, *L* may be vulnerable to buffer overflow. In case that *L* is vulnerable, a test case *T* can detect this vulnerability if *T* satisfies the following two conditions *C1* and *C2*:

- *C1*: *L must be executed during the execution of test T*. In other words, when *T* is executed, the control flow must be able to reach the point where *L* is located.
- *C2*: *When L is executed, B must be overrun*. This typically happens when *S* is a string that is unexpectedly long, or *B* is a buffer that is unexpectedly small, or both. (Section II.D discusses a more subtle case in which *B* may be overrun. The case is not discussed here to avoid distraction from the main idea of our approach.) In other words, either *S*, or *B*, or both, have to take an extreme value.

Note that extreme values are often *syntactically* legal but not *semantically* meaningful. As a result, extreme values are often unexpected and not tested during normal functional testing.

Our approach is centered on how to generate tests such that conditions *C1* and *C2* are likely to be satisfied for potentially vulnerable statements like *L*. Our approach hypothesizes that an internal variable like *S* or *B* in our example derives its extreme value from the value of a single external parameter.

Hypothesis H1: *It is often the case that a buffer is overrun by an extreme value (of an internal variable) that is derived from a single extreme value taken by an external parameter*.

As an effort to validate this hypothesis, we inspected buffer overflow vulnerability reports in three public databases. The results of our inspection, as presented in Section IV, provide strong evidence for the validity of this hypothesis in practice. *Hypothesis H1* is consistent with the typical exploit attempt described in Section I, where the attacker identifies a single external parameter to carry attack data. As discussed later, *Hypothesis H1* allows us to generate tests such that each test only needs to contain a single *attack-payload* parameter.

The main idea of our approach can be described as follows. First, we identify a group of "potential" *attack-payload* parameters and a set of extreme values for each of these parameters. These parameters are "potential" *attack-payload* parameters in the sense that only one of these parameters is used as an *attack-payload* parameter each time a test is constructed. In the remainder of this paper, we will refer to a "potential" *attack-payload* parameter as an *attack-payload* parameter unless otherwise specified. An external parameter *p* is identified to be an *attack-payload* parameter, if *p* taking an extreme value may cause variables like *S* or *B* to take an extreme value.

Second, for each *attack-payload* parameter, we identify a set of *attack-control* parameters, and a set of values for each of these *attack-control* parameters. Let *p* be an *attack-payload* parameter. Intuitively, an external parameter *p'* is identified to be an *attack-control* parameter of *p* if the value of *p'* may affect how the value of *p* is processed. The values of an *attack-control* parameter, which we will refer to as control values, are identified such that they could potentially lead to different application scenarios.

In our approach, the *attack-payload* and *attack-control* parameters and their values are identified manually based on specification and domain knowledge. We provide general guidelines on how to perform this identification in the next section.

After we identify the *attack-payload* and *attack-control* parameters and their values, we are ready to generate actual security tests. Consider the example again. If we knew that *L* was a vulnerable statement, ideally we would want to generate a *single* test to satisfy both *C1* and *C2*. Unfortunately, vulnerable statements like *L* are not known *a priori*. Our approach takes a different perspective. Instead of trying to generate a *single* ideal test for a specific vulnerable statement, we try to generate a *group* of tests for each extreme value (of each *attack-payload* parameter) such that each extreme value can reach as many vulnerable statements as possible.

Specifically, for each extreme value *v* of each *attack-payload* parameter *p*, we generate a combinatorial test set *T* such that (1) *p* takes value *v* in each test in *T*; and (2) *T* covers all the *t*-way combinations of all the *attack-control* parameters of *p*, where *t* is a small integer number that is

expected to be no greater than 6 in practice [20]. *Hypothesis H1* allows us to include only one extreme value (i.e., *v*) in each test. The reason we achieve *t*-way coverage for all the *attack-control* parameters of *p* is to exploit the correlation between combinatorial and code coverage such that, if there is a vulnerable statement like *L* where *v* could cause a variable like *S* or *B* to take an extreme value, then one of the tests in *T* will be likely to reach this statement.

C. Algorithm BOVTest

Fig. 2 presents an algorithm called *BOVTest* (short for Buffer Overflow Vulnerability Test) that implements our approach. This algorithm takes as input a program specification *M* and an integer *t*. *M* is used as the basis for identifying external parameters and values, and *t* is used as the strength for combinatorial test generation. The output of algorithm *BOVTest* is a test set *T* for detecting buffer overflow vulnerabilities in an implementation of *M*.

The algorithm can be largely divided into two parts, *parameter identification* and *security test generation*. Parameter identification includes identification of the set *P* (line 1) of all the external parameters and three particular subsets of *P*, i.e., *Px* (line 2), *Pc* (line 5), and *Pd* (line 6). Note that *Pc* and *Pd* are identified for each attack-payload parameter in *Px* (line 4). Security test generation generates a group of security tests for each extreme value (lines 7 to 13). In the following, we explain each part in detail.

Identification of the set P of all the external parameters (line 1): Generally speaking, an external parameter is any factor of interest that could potentially affect the program behavior. This includes not only the input parameters, but also configuration options and environment variables and other factors that could potentially affect the program behavior.

Identification of the set Px of attack-payload parameters and their values (line 2): To identify *attack-payload* parameters and extreme values, we observe that *attack-payload* parameters often have variable lengths, or indicate the sizes of some other parameters (and thus are likely to be used as the capacity of a buffer). In the former case, an extreme value is often a string value of an excessive length; in the latter case, an extreme value is often an excessively small value. In both cases, the specific values of an *attack-payload* parameter are often not significant for the purpose of testing, i.e., in terms of triggering a buffer overflow (instead of acquiring specific control to do any real harm). This is because buffer overflow vulnerabilities are in essence a mishandling of certain length or size requirements. This observation can also be used to exclude certain parameters from *Px*. For example, string parameters of fixed length are typically not *attack-payload* parameters.

For example, in a network protocol, user payload is likely to be an *attack-payload* parameter, as it is of a variable length. Furthermore, we can identify a payload that is longer than typically expected as one of its extreme values. Note that the specific data in the payload is not important for our purpose.

Identification of the set Pc of attack-control parameters and their values (for each attack-payload parameter in *Px*) (line 5): An external parameter *p'* is an *attack-control* parameter of *attack-payload* parameter *p* if application logic suggests that the value of *p'* could affect how the value of *p* is processed. The term "application logic", instead of "program logic", is used to indicate this identification is based on specification and/or domain knowledge, i.e., not the source code. For each parameter in *Pc*, we identify a set of control values. Control values can be identified using traditional techniques such as domain analysis and equivalence partitioning [25]. Oftentimes, different control values signal different application scenarios, leading to different branches in a program. We point out that security testing is often performed after normal functional testing. Therefore, it is often possible for us to take advantage of knowledge and experience accumulated during functional testing. In particular, we expect that most control values have been identified and tested during functional testing.

Again, consider a network protocol. Assume that we have identified user payload as an *attack-payload* parameter. Then, message type is likely to be a parameter that could affect how user payload is processed in the implementation. This is because the payload often needs to be interpreted differently depending on the type of a message. The control values of this message type parameter would be the different types that are specified in the protocol specification. It is often the case that the different types have already been identified during functional testing.

Identification of the set Pd of non-attack-control parameters and their values (for each *attack-payload* parameter in *Px*) (line 6): *Pd* is the complement set of *Pc*. In other words, *Pd* consists of all the external parameters that are not *attack-control* parameters of *attack-payload* parameter *p*. For each parameter in *Pd*, we simply identify a single default value, which can be any valid value in the domain of the parameter. (A value is valid if it is allowed by the specification. Otherwise, it is invalid.) These default values do not directly contribute to the detection of buffer overflow vulnerabilities in our approach. Instead, these values are only needed to construct complete, thus executable, tests.

The fewer the parameters in *Pc* (and the more the parameters in *Pd*), the fewer the tests generated for each extreme value of *attack-payload* parameter *p*. An imperfect identification of *Pc* and *Pd* may increase the number of tests and/or miss some vulnerabilities, but it does not invalidate our test results. That is, any vulnerability detected by our approach is a real vulnerability. More discussion on this is provided in Section II.D.

Security test generation (lines 7 - 13): The actual generation of a security test set for each extreme value *v* of each *attack-payload* parameter in *Px* proceeds as follows. We first generate a *t*-way test set *T'* for all the parameters in *Pc*, using their control values (line 8). Each test in *T'* is then used as a base test to create a complete test by (1) adding *v* as the value of *Px*, and (2) adding the default value of each parameter in *Pd* (and thus not in *Pc*).

Consider the example shown in Fig. 3. Assume that a system has five parameters, *P1*, *P2*, *P3*, *P4*, and *P5*. Assume that *P4* is an *attack-payload* parameter, and has an extreme

value *LS*, indicating a very long string. Also assume that *P1*, *P2*, and *P3*, with control values 0 and 1, are in *Pc*, and *P5* is in *Pd*, with a default value 0. To generate a pairwise test set for extreme value *LS* of *attack-payload* parameter *P4*, we first generate a pairwise test set for *P1*, *P2*, and *P3*, which is the test set *T* shown in Fig. 1. Next, we add into each test of *T* value *LS* as the value of *P4*, and 0 the value of *P5*. Thus, we obtain the complete test set as shown in Fig. 3.

Algorithm BOVTest

Input: A program specification *M*, and an integer *t*
Output: A test set *T* for detecting buffer overflow vulnerabilities in an implementation of *M*

1. let *P* be the set of all the external parameters of *M*
2. identify a set $Px \subseteq P$ of attack-payload parameters and a set of extreme values for each parameter *p* in *Px*
3. initialize *T* to be an empty test set
4. for each attack-payload parameter p_x in *Px* {
5. identify a set $Pc \subseteq P$ of attack-control parameters for p_x and a set of control values for each parameter *p* in *Pc*
6. let $Pd = P - Pc$, and identify a default value $d(p)$ for each parameter *p* in *Pd*
7. for each extreme value *v* of p_x {
8. build a *t*-way test set *T´* for parameters in *Pc* using their control values
9. for each test τ' in *T´* {
10. create a complete test τ such that for each parameter *p*, $\tau(p) = v$ if $p = p_x$, $\tau(p) = \tau'(p)$ if $p \in Pc$, and $\tau(p) = d(p)$ otherwise, where $\tau(p)$ (or $\tau'(p)$) is the value of parameter *p* in test τ (or τ')
11. $T = T \cup \tau$
12. }
13. }
14. }
15. return *T*

Figure 2. Algorithm BOVTest

p1	p2	p3	p4	p5
0	0	0	LS	0
0	1	1	LS	0
1	0	1	LS	0
1	1	0	LS	0

Figure 3. An example security test set

Assume that the *IPOG* algorithm is used to generate a *t*-way test set (line 8). Let d_c be the maximal number of control values an *attack-control* parameter can take. The size of T' is $O(d_c^t \times \log |P_c|)$ [24]. Let d_x be the maximal number of extreme values an *attack-payload* parameter can take. Let *d* be the maximum of d_c and d_x. The number of tests generated by algorithm *BOVTest* is $O(d^{t+1} \times |P| \times \log |P|)$.

D. Discussion

Recall that buffer overflow occurs when data is written beyond the boundary of an array-like structure. Let *D* be the data to be written. Let *B* be an array-like structure. As discussed earlier, if the size of *D* is larger than the capacity of *B* (either *D* is unexpectedly long, or *B* is unexpectedly small, or both), then *D* will be written beyond the boundary of *B*, i.e., *B* will overflow. There is a more subtle case to consider. In languages like *C*, an array-like structure is often accessed using a pointer, and such a pointer can be moved forward or backward using explicit arithmetic operations. For example, an array variable in *C* is in fact a pointer that points to the beginning of the array. It is possible that the pointer may be moved beyond the upper or even lower boundary of a buffer due to explicit pointer arithmetic operations. If data of any size is written at this point, a buffer overflow will occur.

The above scenario suggests that attention should also be paid to external parameters whose values may be used as an offset in a pointer arithmetic operation, in addition to external parameters of variable length, and external parameters that indicate the length of other parameters. For example, in a record keeping application, the record number may be used as an offset to locate a record. If proper checks are not performed, a negative value, or an unexpectedly large positive value, of the record number could move the base pointer beyond the lower or upper boundary of the structure that keeps all the records.

It is worth noting that extreme values typically matter because of their *extreme* properties, instead of their specific values. For example, what matters for an extreme string is typically its length, instead of the specific characters in the string. This observation greatly simplifies the selection of extreme values.

Our approach is most effective if the *attack-payload* and *attack-control* parameters and values are identified properly. If a vulnerable point can only be reached when an *attack-payload* or *attack-control* parameter takes a particular value, and if this value is not identified, then our approach will miss this vulnerable point. Note that an *attack-payload* parameter may also be involved in a control-flow decision. In this case, some extreme values of this parameter may be able to reach a vulnerable point, whereas other extreme values may not. Nonetheless, our approach ensures *t*-way coverage for each *attack-payload* parameter (with its extreme values) and its control parameters (with their control values). This ensures that no vulnerable point will be missed because a particular *t*-way combination is not tested in our approach.

Fortunately, identification of *attack-payload* and *attack-control* parameters and values does not have to be perfect. In practice, we can exploit this flexibility to scale up or down our test effort. On the one hand, when adequate resources are available, more parameters and values can be identified to acquire more confidence at the cost of creating more tests. For example, if we are unsure about whether an external parameter should be considered to be an *attack-control*

parameter or a particular value should be considered to be an extreme or control value, it is safe to do so. This will likely create more tests, but will also help to detect vulnerabilities that otherwise would not be detected. On the other hand, when the resource is constrained, we can focus our effort only on a subset of parameters and values that we believe are the most important ones to test. Doing so will reduce the number of tests, but may miss some vulnerabilities. Nonetheless, any vulnerability that is detected by our approach is guaranteed to be a real vulnerability.

III. TANCE: A PROTOYPE TOOL

We implemented our approach in a prototype tool called *Tance*. Fig. 4 shows the architecture of *Tance*, which consists of the following major components:

Controller: This is the core component of *Tance*. It is responsible for driving the entire testing process. In a typical scenario, after *Controller* receives from the user the external parameter model of the subject application, it calls *Test Generator* to generate combinatorial tests based on the given parameter model. For each test, *Controller* uses *Test Transformer* to transform it into an executable format, i.e., a format that is accepted by the subject application. Then, *Controller* calls *Test Executor* to execute each test automatically.

Test Generator: This component is responsible for the actual test generation. In other words, this component implements algorithm *BOVTest*. This component first uses a combinatorial test generation tool, called *ACTS* (formerly known as *Fireeye*) [24] to generate a base combinatorial test set, which is used later to derive a set of complete tests as discussed in Section II.

Test Transformer: This component is responsible for transforming each combinatorial test into a format that is accepted by the subject program. This step is necessary because a combinatorial test only consists of parameter values, but programs often require a test to be presented in a particular format. For example, a web server requires each test to be presented as an HTTP request. This component needs to be customized for different programs. *Tance* provides a programming interface that allows the user to hook a third party component into its testing framework.

Test Executor: This component is responsible for carrying out the actual test execution process. For example, this tool will send test requests automatically to HTTP servers. In addition, *Test Executor* will restart HTTP servers before running the next test so that there is no interference between different tests. This component also needs to be customized for different programs. *Tance* also provides a programming interface for integration with an existing test execution environment.

Bounds Checker: This component is used to detect the actual occurrence of a buffer overflow. In our experiments, we used the bounds checking tool reported in [10]. This tool instruments the source code. Source-level information helps analysis of test results for our evaluation purpose. In practice, our tool can be used with bounds checkers that work on binary code, e.g., Chaperon, which require no source code access.

IV. CASE STUDIES

In Section IV.A, we present the results of our inspection on three public vulnerability databases, as an effort to validate *Hypothesis H1*. In Section IV.B, we describe the subject programs as well as the computing environment used in our case studies. In Section IV.C, we present the vulnerability detection results of applying our approach to five open-source programs.

A. Validation of Hypothesis H1

To validate *Hypothesis H1*, we checked three public vulnerability databases: *National Vulnerability Database* (*NVD*) [27], *SecurityFocus* [32], and *SecurityTracker* [33]. For each database, we conducted a search using the keyword "buffer overflow" to retrieve reports on buffer overflow vulnerabilities. For each of the three databases, we inspected the first 100 reports returned by our search. Among the reports we inspected, there are 15 reports cross-referenced between *NVD* and *SecurityTracker*.

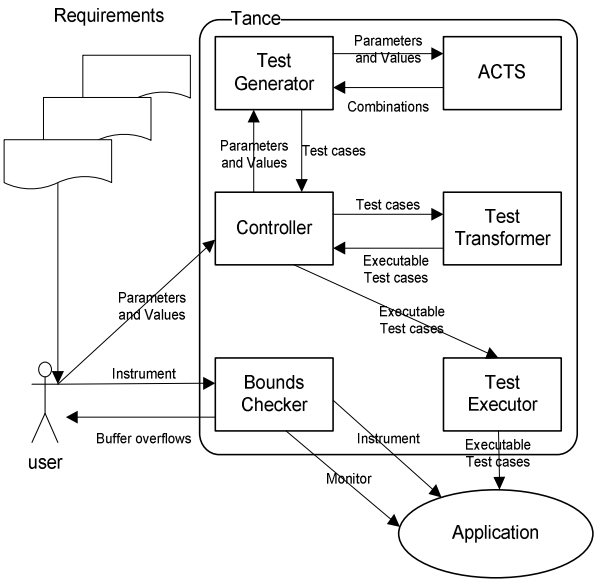

Figure 4. Tance's architecture

Table I shows the results of our inspection. We classify the reports into three categories. The first category (*Explicitly Stated*) includes reports that contain an explicit statement confirming the satisfaction of *Hypothesis H1*. For example, the report *CVE-2010-0361* in *NVD* contains the following statement: "*Stack-based buffer overflow ... via a long URI in HTTP OPTIONS request.*". This statement explicitly states that a single parameter *URI* takes an extreme value. The second category (*Reasonably Inferred*) includes reports that are written in a way that reasonably suggests the satisfaction of *Hypothesis H1*, despite the lack of an explicit statement. For example, the report *CVE-2007-1997* in *NVD* contains the following statement: "*... via a crafted CHM file that contains a negative integer ... leads to a stack-based*

buffer overflow.". This statement does not explicitly identify a parameter, but it is reasonable to believe that one of the fields in a CHM file can be modeled as an *attack-payload* parameter taking the negative value. The third category (*Not Clear*) includes reports that do not fall into the first two categories. For these reports, we do not have adequate information to make a reasonable judgment. We did not find any report that explicitly states that two or more parameters must take an extreme value at the same time to trigger a buffer overflow. In other words, none of the examined reports explicitly disproves *Hypothesis H1*.

In [21], it is reported that NIST examined more than 3000 NVD reports for denial-of-service vulnerabilities and found that 93.1 percent involved only a single *attack-payload* parameter, nearly always an input string that is too long. This result is consistent with our findings.

TABLE I. VALIDATION OF HYPOTHESIS H1

Database	Explicitly stated	Reasonably inferred	Not clear
NVD	49	25	26
SecurityFocus	61	9	30
SecurityTracker	62	29	9

TABLE II. STATISTICS OF SUBJECT APPLICATIONS

Subject	Files	Functions	LOC
Ghttpd	4	16	609
Gzip	34	108	5809
Hypermail	57	401	23057
Nullhttpd	11	38	2245
Pine	449	4883	154301

B. Experimental Setup

Subject Programs: Our case studies use the following five programs:

(1) *Ghttpd* (version 1.4.4) is a fast and efficient web server that supports a reduced set of HTTP requests [12].
(2) *Gzip* (version 1.2.4) is the widely used GNU compression utility tool [17].
(3) *Hypermail* (version 2.1.3) is a tool that facilitates the browsing of an email archive. It compiles an email archive in the UNIX mailbox format to a set of cross-referenced HTML documents [19].
(4) *Nullhttpd* (version 0.5.0) is a web server that handles HTTP requests [28].
(5) *Pine* (version 3.96) is a widely used tool for reading, sending, and managing emails [29].

All five programs are written in *C*. *Ghttpd* and *Nullhttpd* have been used in other empirical studies for buffer overflow detection methods, e.g., [34]. Table II shows some statistics about the size of these five applications.

Platform Configuration: The five case studies were conducted on a 3GHz machine that has 2GB RAM, running Red Hat Enterprise Linux WS release 4, gcc-3.4.6 and bgcc-3.4.6.

C. Testing Results and Discussion

In our studies, we identified the *attack-payload* and *attack-control* parameters and values in a fairly straightforward manner. In particular, we intentionally avoided the use of any advanced domain knowledge. On the one hand, this illustrates that while advanced domain knowledge helps to make our approach more effective, it is not required. On the other hand, this is part of our effort to reduce the threats to validity as discussed in Section IV.D.

Specifically, all the string parameters of variable length were identified to be *attack-payload* parameters. For each of these parameters, we identified a single extreme value, which is a string typically much longer than normally expected. An integer parameter is identified to be an *attack-payload* parameter only if it obviously indicates the length of another parameter such as *Content-Length*. For each integer *attack-payload* parameter, we identified three extreme values, including a positive number that is smaller than the actual length of the other parameter, zero, and a small negative number. For each *attack-payload* parameter *p*, we identified a parameter *p'* to be an *attack-control* parameter of *p* only if a very strong connection existed between *p* and *p'*. For example, in *Ghttpd* and *Nullhttpd*, a parameter named *Request-Method* indicates whether the request is a *GET* or *POST* request. This parameter was identified to be an *attack-control* parameter of an *attack-payload* parameter *Message-Body*, which represents the payload carried in an HTTP request. A document that explains in detail how these parameters and values were identified is made available online for review [38].

TABLE III. EXTERNAL PARAMETER MODELS

Subject	NP	NXP	ANXV	ANCP	ANCV
Ghttpd	42	5	1.4	2.7	4.8
Gzip	16	3	3.0	13.3	12.3
Hypermail	28	16	2.0	15.5	12.8
Nullhttpd	42	5	1.4	2.7	4.8
Pine (read)	10	10	1.4	2.4	1.9
Pine (write)	7	7	1.3	2.0	1.7

Note: NP = # of parameters, NXP = # of attack-payload parameters, ANXV = average # of extreme values per attack-payload parameter, ANCP = average # of attack-control parameters per attack-payload parameter, ANCV = average # of control values per attack-control parameter.

TABLE IV. STATISTICS ON NUMBER OF TESTS

Subject	Total	Min	Max	Avg
Ghttpd	191	3	36	27.3
Gzip	32	10	12	10.7
Hypermail	200	10	10	10.0
Nullhttpd	191	3	36	27.3
Pine (read)	89	3	8	6.4
Pine (write)	49	3	8	5.4

Note: Total = total # of tests, Min, Max, Avg = minimum, maximum, and average # of tests per extreme value.

TABLE V. VULNERABILITY DETECTION RESULTS

Subject	Detected	Reported	Missed	New
Ghttpd	1	1	0	0
Gzip	2	1	0	1
Hypermail	5	2	1	4
Nullhttpd	5	1	0	4
Pine	7	7	LOI	LOI

Note: LOI = lack of information. Our approach detected 9 new vulnerabilities in total.

Table III shows the number of different types of parameters and values for each subject program. For *Ghttpd* and *Nullhttpd*, we used the same set of *attack-payload* and *attack-control* parameters and values. This is because both programs are HTTP servers and we identified the parameters and values from the same HTTP specification. *Pine* has two operational modes, *read* and *write*. These two modes are tested separately in our experiments, because these two modes have very different interfaces and use different sets of parameters. We would like to emphasize that none of these parameters is private or otherwise internal. To the contrary, each parameter can be set directly be a user or attacker.

Next we present statistics on the number of tests we generated for the five subject programs. Recall that we generate a group of tests for each extreme value of each *attack-payload* parameter. Algorithm *BOVTest* generates the same number of tests for every extreme value of the same *attack-payload* parameter. Each of the generated tests is a system test and therefore represents a scenario that may occur in practice.

Table V presents information about the buffer overflow vulnerabilities detected by our approach. We obtained the number of reported vulnerabilities from two databases, *securityfocus* and *securitytracker*. Our approach detected all the reported vulnerabilities for *Ghttpd*, *Gzip*, and *Nullhttpd*. For *Hypermail*, a buffer overflow was reported but was not detected by our approach. An inspection revealed that this buffer overflow involved a long string returned by a DNS server, which was not modeled as an external parameter in our experiments. For *Pine*, there are 7 reported vulnerabilities, and our approach also detected 7 vulnerabilities. However, no adequate information is available for us to determine whether the 7 reported vulnerabilities are the same as those detected by our approach. Therefore, we put *LOI*, short for Lack of Information, in the table.

In addition, our approach detected a total of 9 new vulnerabilities for the five programs. As an example, Fig. 5 shows a code segment of *Nullhttp* that contains a new vulnerability detected in our experiments. This vulnerability is detected when *in_RequestMethod* is POST, and *in_ContentLength* is a negative number.

D. Threats to Validity

As discussed in Section II, the effectiveness of our approach depends on the proper identification of the *attack-payload* and *attack-control* parameters and their values. Since our experiments use programs that have known vulnerabilities, the validity of our results would be in jeopardy if knowledge of the known vulnerabilities were used to identify these parameters and values in our experiments. To alleviate this potential threat, we tried to only use explicit information that was available in the specification. In addition, each time we identified a particular type of parameter or value, we provided an explicit explanation about how our decision was made in a way that only used information available in the specification, rather than other sources. These explanations were cross-checked by two of the co-authors and are available for review [38].

The validity of our results also depends on the correctness of two tools, namely *ACTS* and the bounds checker, used in our experiments. Both of these two tools have been available for public access for a significant amount of time, and have been used to conduct experiments for other research projects.

The main external threat to validity is the fact that the five open-source programs used to conduct our experiments may not be representative of true practice. These programs are real-life programs themselves, and are chosen from different application domains. Some programs have also been used in other studies.

```
if (strcmp(conn[sid].dat->in_RequestMethod, "POST")==0) {
   ……
   if (conn[sid].dat->in_ContentLength < MAX_POSTSIZE) {
      ……
      do{
         ……
         rc = recv(conn[sid].socket, pPostData, 1024, 0);
         ……
         x += rc;
         ……
      } while ((rc  ==1024) || (x < conn[sid].dat->in_ContentLength));
      conn[sid].PostData[conn[sid].dat->in_ContentLength]='\0';
      …
   }
   …
}
```

Figure 5. A buffer overflow vulnerability example

V. Related Work

Our work tries to detect buffer overflow vulnerabilities, i.e., whether there exists a static defect that can cause a buffer overflow to occur at runtime. This is different from work on detecting buffer overflows, which tries to detect whether a buffer overflow has actually occurred at runtime [10]. A fundamental difference between the two is that detecting a static defect needs to consider all possible behaviors a program could exercise at runtime, while detecting the occurrence of a runtime phenomenon only needs to deal with the program behavior in a specific runtime scenario. Testing based approaches to detecting buffer overflow vulnerabilities often use techniques for detecting buffer overflows to evaluate a test run. In this respect, these two types of approaches are complementary. As mentioned in Section III, our work uses a bounds checking tool to detect whether a buffer overflow has actually occurred in a test run.

Our work is also different from work on runtime prevention, which tries to prevent buffer overflow attacks from occurring at runtime [39]. For example, StackGuard [9] may terminate a process after it detects that a return address on the stack has been overwritten. Existing approaches to runtime prevention can incur significant runtime overhead. In addition, these approaches are in effect after potentially vulnerable programs are deployed. This is in contrast with our work, which aims to develop and release programs that are free from buffer overflow vulnerabilities prior to deployment.

In the following we focus on approaches to detecting buffer overflow vulnerabilities during the development stage.

In particular, we discuss testing based approaches, i.e., approaches that involve actual program executions. We will not discuss approaches that are based on pure static analysis [18][22][30][40], as they involve quite different techniques. As mentioned in Section I, static analysis suffers from the problem of false positives and/or negatives.

We first discuss black-box testing approaches, which are the most closely related to our work. Fuzzing [35] is among the most widely used black-box testing approaches in security testing. Fuzzing typically starts from one or more legal inputs, and then randomly mutates these inputs to derive new test inputs. Advanced fuzzing techniques [14][15] can also incorporate domain knowledge and/or employ heuristics, e.g., assigning different weights to different components.

As mentioned in Section I, the poor code coverage is a major limitation of fuzzing [35]. In contrast, our approach samples the input space in a systematic manner to achieve a combinatorial coverage. Empirical results suggest that there exists a high correlation between combinatorial coverage and code coverage [3][11]. Our approach is, however, not fully automated. In particular, *attack-payload* and *attack-control* parameters, as well as their values, are identified manually in our approach. Our empirical studies show that this identification can be performed with reasonable effort. Moreover, we believe that this provides an opportunity for us to take advantage of domain knowledge that may be readily available in practice. We point out that many black-box testing techniques, including combinatorial testing in its original form, require individual parameters and values to be identified manually. More discussion on this manual aspect of our approach is provided in the next section.

Recently there has been a growing amount of interest in approaches that combine symbolic execution and testing [5][13][41]. In these approaches, symbolic execution is used to collect path conditions consisting of a sequence of branching decisions. These branching decisions are then negated systematically to derive test inputs that when executed, will explore different paths. In order to detect buffer overflow vulnerabilities, memory safety constraints are formulated and solved together with these path conditions. A potential problem with these approaches is path explosion. Techniques based on functional summaries, generational search and length abstraction have been developed to alleviate this problem. These approaches generate tests in a fully automatic manner. However, symbolic execution often involves extensive instrumentation, either at the source or binary level. Thus, the resulting solutions are usually specific to a particular language, build environment, or platform. Symbolic executions can also be much slower than actual program executions. More importantly, for large and/or complex programs, the number of constraints that have to be solved presents significant challenges to the capacity of existing constraint solvers.

VI. Conclusions and Future Work

In this paper, we presented a black-box testing approach to detecting buffer overflow vulnerabilities. Our approach simulates the process an attacker typically performs to exploit a buffer overflow vulnerability. A novel aspect of our approach is that it adapts a general software testing technique called combinatorial testing to the domain of security testing. In particular, our approach exploits the fact that combinatorial testing often achieves a high level of code coverage. We implemented our approach in a prototype tool called *Tance*. Empirical results of applying *Tance* to five open source programs show that our approach is effective in detecting buffer overflow vulnerabilities in these programs.

In our approach, *attack-payload* and *attack-control* parameters and their values are identified manually based on specification or domain knowledge or both. We provide guidelines for performing such identification. Our empirical studies show that these guidelines are very effective and can be followed with reasonable effort. Security testing is often performed after functional testing. Thus, knowledge and experience obtained from functional testing can be utilized to effectively identify these parameters and values. While our approach is most effective when these parameters and values are identified properly, this identification does not have to be perfect. In practice, we can exploit this flexibility to scale our test effort up or down, depending on the availability of resources. That is, we can intentionally identify more parameters and values to acquire more confidence at the cost of more tests. Or we can intentionally identify fewer parameters and values to reduce test effort at the cost of missing some vulnerabilities.

While fully automated solutions are often desirable, we believe semi-automated solutions like ours also have their merits. In particular, our approach allows us to take advantage of domain knowledge that may be readily available in practice. An effective use of domain knowledge can often make the testing process more efficient, and may discover bugs that cannot be discovered otherwise. Moreover, fully automated and semi-automated solutions can be used in such a way that they complement each other. For example, we can first apply fuzzing and then our approach to achieve higher fault coverage. As mentioned earlier, our approach allows test effort to be scaled up or down, depending on the availability of resources. This flexibility further facilitates the use of our approach in combination with other approaches.

We plan to develop lightweight static analysis techniques to automatically identify *attack-payload* and *attack-control* parameters and their values. These techniques can be applied when source code is available. With these techniques, we will be able to fully automate our test generation process. This will enable a direct comparison between our approach and existing approaches that combine symbolic execution and testing. Such a comparison will help to further evaluate the effectiveness of our approach.

Acknowledgment

This work is partly supported by a grant (Award No. 70NANB10H168) from the Information Technology Lab (ITL) of National Institute of Standards and Technology (NIST).

REFERENCES

[1] D. Aitel, "The Advantages of Block-based Protocol Analysis for Security Testing", Immunity Inc, 2002. DOI= http://www.net-security.org/article.php?id=378.

[2] J. H. Andrews, L.C. Briand, Y. Labiche and A.S. Namin, "Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria", IEEE Transactions on Software Engineering, 32(8): 608-624, 2006.

[3] K. Burr, and W. Young, "Combinatorial Test Techniques: Table-based Automation, Test Generation and Code Coverage", Proceedings of the International Conference on Software Testing Analysis and Review, pp. 503-513, 1998.

[4] R. Bryce, C. J. Colbourn, M.B. Cohen, "A framework of greedy methods for constructing interaction tests," Proceedngs of the 27th International Conference on Software Engineering (ICSE), pp. 146-155, 2005.

[5] C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill, and D.R. Engler, "EXE: Automatically Generating Inputs of Death", Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS), pp. 322-335, 2006.

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

[7] M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG System: An Approach to Testing Based on Combinatorial Design", IEEE Transactions on Software Engineering, 23(7): 437-444, 1997.

[8] M. B. Cohen, M. B. Dwyer, and J. Shi, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: a greedy approach", IEEE Transactions on Software Engineering, 34(5), pp. 633-650, 2008.

[9] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, and Q. Zhang, "StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks", Proceedings of the 7th conference on USENIX Security Symposium, pp. 5-5, 1998.

[10] R. Dhurjati and V. Adve, "Backwards-compatible Array Bounds Checking for C with Very Low Overhead", Proceedings of the 28th IEEE International Conference on Software Engineering, pp. 162-171, 2006.

[11] S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino, "Applying design of experiments to software testing", Proceedings of IEEE International Conference on Software Engineering, pp. 205–215, 1997.

[12] Ghttpd-1.4.4. DOI= http://gaztek.sourceforge.net/ghttpd/.

[13] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing", Proceedings of the 2005 ACM SIGPAN Conference on Programming Language Design and Implementation (PLDI), pp. 213-233, 2005.

[14] P. Godefroid, A. Kiezun, and M.Y. Levin, "Grammar-based Whitebox Fuzzing", Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI), pp. 206-215, 2008.

[15] P. Godefroid, M. Levin, and D. Monlnar, "Automated Whitebox Fuzz Testing", Proceedings of the Network and Distributed Security Symposium, 2008.

[16] M. Grindal, J. Offutt, and S.F. Andler, "Combination Testing Strategies: A Survey", Software Testing, Verification and Reliability, 15(3): 167-199, 2005.

[17] Gzip-1.2.4. DOI= http://www.gzip.org/.

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

[19] Hypermail-2.1.3. DOI= http://www.hypermail.org/.

[20] R. Kuhn, D.R. Wallace, and A.M. Gallo Jr, "Software Fault Interactions and Implications for Software Testing", IEEE Transactions on Software Engineering, 30(6):418-421, 2004.

[21] R. Kunh and C. Johnson, "Vulnerability Trends: Measuring progress", IEEE IT Professional, 12(4):51-53, 2010.

[22] W. Le, M. L. Soffa, "Marple: a Demand-Driven Path-Sensitive Buffer Overflow Detector", Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 272-283, 2008.

[23] Y. Lei, R. Carver, R. Kacker, D. Kung, "A Combinatorial Strategy for Testing Concurrent Programs", Journal of Software Testing, Verification, and Reliability, 17(4):207-225, 2007.

[24] Y. Lei, R. Kacker, R.D. Kuhn, V. Okun, and J. Lawrence, "IPOG/IPOD: Efficient Test Generation for Multi-way Combinatorial Testing", Software Testing, Verification and Reliability, 18(3):287-297, 2007.

[25] A. Mathur, "Foundations of Software Testing", Addison-Wesley Professional, 2008.

[26] G. McGraw, "Software Security", IEEE Security & Privacy, 2(2): 80-83, 2004.

[27] National Vulnerability Database. DOI= http://nvd.nist.gov/.

[28] Nullhttpd-0.5.0. DOI= http://www.nulllogic.ca/httpd/.

[29] Pine-3.96. DOI= http://www.washington.edu/pine/.

[30] M. Pistoia, S. Chandra, S.J. Fink, and E. Yahav, "A Survey of Static Analysis Methods for Identifying Security Vulnerabilities in Software Systems", IBM Systems Journal, 46(2):265-288, 2007.

[31] J. Roning, M. Laakso, A. Takanen and R. Kaksonen, "PROTOS – Systematic Approach to Eliminate Software Vulnerabilities". DOI= http://www.ee.oulu.fi/research/ouspg/.

[32] SecurityFocus. DOI= http://www.securityfocus.com/.

[33] SecurityTracker. DOI= http://www.securitytracker.com/.

[34] E. C. Sezer, P. Ning, C. Kil and J. Xu, "Memsherlock: An Automated Debugger for Unknown Memory Corruption Vulnerabilities", Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS), pp. 562-572, 2007.

[35] M. Sutton, A. Greene, and P. Amini, "Fuzzing: Brute Force Vulnerability Discovery", Addison-Wesley, 2007.

[36] W. Wang, S. Sampath, Y. Lei, and R. Kacker, "An Interaction-Based Test Sequence Generation Approach for Testing Web Applications", Proceedings of the 11th IEEE High Assurance Systems Engineering Symposium, pp. 209-218, 2008.

[37] W. Wang, Y. Lei, S. Sampath, R. Kacker, D. Kuhn, J. Lawrence, "A Combinatorial Approach to Building Navigation Graphs for Dynamic Web Applications", Proceedings of 25th IEEE International Conference on Software Maintenance, pp. 211-220, 2009.

[38] W. Wang, and D. Zhang, "External Parameter Identification Report". University of Texas at Arlington. DOI= https://wiki.uta.edu/pages/viewpageattachments.action?pageId=35291531.

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

[40] Y. Xie, A. Chou, and D. Engler, "ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors", Proceedings of 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 327-336, 2003.

[41] R. Xu, P. Godefroid, R. Majumdar, "Testing for Buffer overflows with length Abstraction", Proceedings of the 2008 International Symposium on Software Testing and Analysis, pp. 27-38, 2008.