

國立交通大學

資訊科學與工程研究所

碩士論文

以搜尋式測試方法偵測程式溢位弱點

Detecting Buffer Overflow Vulnerabilities via Search-based Testing

研究生：黃琨翰

指導教授：黃世昆 教授

中華民國九十八年六月

以 搜 尋 式 方 法 偵 測 程 式 溢 位 弱 點
Detecting Buffer Overflow Vulnerabilities via Search-based Testing


研 究 生：黃琨翰

Student : Kuen-Han Huang

指 導 教 授：黃世昆

Advisor : Shih-Kun Huang

國 立 交 通 大 學
資 訊 科 學 與 工 程 研 究 所
碩 士 論 文



A Thesis
Submitted to Institute of Computer Science and Engineering
College of Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in
Computer Science

June 2009

Hsinchu, Taiwan, Republic of China

中 華 民 國 九 十 八 年 六 月

以搜尋式測試方法偵測程式溢位弱點

學生：黃琨翰

指導教授：黃世昆 老師

國立交通大學資訊科學與工程學系（研究所）碩士班

摘要

緩衝區溢位攻擊是一種最惡名昭彰的軟體安全問題。有些工具已經被發展作為緩衝區溢位弱點偵測之用。儘管有偵測的能力，大部分的現有工具無法產生能夠觸發溢位的測試案例。我們提出一個新的方法來解決針對溢位偵測的測試案例產生問題。這個方法使用搜尋式結構測試，能夠找到測試輸入使得程式執行走到目標點，也就是溢位產生的地方。搜尋式測試方法的概念是將產生測試資料以公式化轉換為搜尋的問題。在搜尋式測試中，一個被稱作鏈結方法的資料相依分析技巧可以幫助處理因為資料相依引起的搜尋失敗。鏈結方法被應用在找出影響緩衝區存取是否越界的程式敘述，接著產生抽象路徑引導程式執行滿足緩衝區溢位的條件。論文中展示的兩個最佳化技巧可以減少鏈結方法中在不必要路徑上的花費。在結果評估中顯示，與原有的搜尋式方法相比，我們的方法可以以較有效率的方式來偵測緩衝區溢位。

Detecting Buffer Overflow Vulnerabilities via Search-based Testing

Student : Kuen-Han Huang

Advisors : Dr. Shih-Kun Huang

Department of Computer Science and Engineering
National Chiao Tung University

ABSTRACT

Buffer overflow attacks are one of the most notorious software security problems. A few tools have been developed to detect buffer overflow vulnerabilities. In spite of the detection capability, most of the existing tools can not generate test cases to trigger an overflow. We propose a new approach that addresses the issue of test case generation for buffer overflow detection. The approach uses search-based structural testing to find test inputs that drive program execution to reach the target node where a buffer overflow could occur. The idea of search-based testing is to formulate the test data generation for a program under test as a search problem. In search-based testing, a data dependence analysis technique called the Chaining Approach can help to handle the search failure due to data dependencies. The Chaining Approach is applied to identify the program statements that have influence on whether a buffer access is out of bound or not, then abstract paths are derived to lead the program execution to satisfy a buffer overflow condition. Two optimization techniques are presented to reduce the cost of exercising unnecessary paths in the Chaining Approach. The evaluation results show that our approach can find test data for buffer overflow detection in a more efficient way than using the original approach in search-based testing.

誌 謝

我要感謝我的指導教授，黃世昆老師，在兩年的研究過程中給予指導及建議，而且不論在方法或實作上都會與學生互相討論，讓我們知道如何把研究做好。也要謝謝昌憲學長，給我研究方向的意見，在實驗室的時候也會一起研究有趣的問題。還想向在實驗室曾經一起努力打拼的同伴表達謝意：友祥、彥廷、文健、士瑜、泳毅、慧蘭。最後要謝謝我的爸媽在求學生涯上的支持。



Table of Contents

摘要.....	i
Abstract.....	ii
誌謝.....	iii
Table of Contents.....	iv
List of Tables.....	v
List of Figures.....	vi
1. Introduction.....	1
1.1 Motivation.....	1
1.1.1 Static Analysis.....	3
1.1.2 Dynamic Analysis.....	3
1.2 Problem Description and Objective.....	4
2. Background.....	6
2.1 Search-based Test Data Generation.....	6
2.2 Objective Function.....	8
2.3 The Chaining Approach.....	10
2.3.1 Example for the Chaining Approach.....	12
2.3.2 Formal Description.....	13
3. Related Work.....	14
4. Method.....	16
4.1 Objective Function for Buffer Overflows.....	16
4.2 Applying the Chaining Approach for Buffer Overflow Detection.....	17
4.2.1 Extra Out-of-bound Checking.....	17
4.2.2 Optimization Strategies for the Chaining Approach.....	19
4.2.2.1 Look-Ahead Strategy.....	19
4.2.2.2 Tainted-First Strategy.....	24
5. Implementation.....	29
5.1 Static Frontend.....	29
5.2 Dynamic Component: Search-based Testing.....	29
6. Results.....	31
6.1 Test Objects.....	31
6.2 Experimental Setup.....	32
6.3 Evaluation Results.....	33
7. Conclusion.....	35
References.....	37

List of Tables

Table 1. Summary of recent CERT advisories (Last updated: 2004)	2
Table 2. Summary of US-CERT technical cyber security alerts	2
Table 3. Objective function calculation.	9
Table 4. Average objective evaluations for each test object against different strategy configurations.....	34
Table 5. Comparison of objective evaluations of strategies to the original approach without any optimization strategies.	34



List of Figures

Figure 1. Example function for objective value calculation.	9
Figure 2. Example for the Chaining Approach.	11
Figure 3. Bound check instrumentation	18
Figure 4. Procedure of Look-Ahead strategy	21
Figure 5. Example function for Look-Ahead strategy	23
Figure 6. Event sequence tree derived from the example of Look-Ahead strategy.	23
Figure 7. Example for Tainted-First strategy	26
Figure 8. Data dependency graph of the example function for Tainted-First strategy. ...	27
Figure 9. Event sequence tree of the example function for Tainted-First strategy.	28
Figure 10. Test input generation process	29



1. Introduction

1.1 Motivation

Software is under constant scrutiny and often suffers from attacks nowadays. Buffer overflow attacks are one of prevalent and persistent security problems. As reported by the recent statistics of CERT advisories in Table 1 [1], software flaws over 50% are caused by buffer overflows since 2000 to 2004. Till June 2009, 38% of the 60 most severe vulnerabilities posted by CERT/CC were resulted from buffer overflows [2]. Even in recent years, the ubiquitous vulnerability still accounts for large amount in advisories of security organizations. In Table 2 which is derived from the recent data of US-CERT technical cyber security alerts [3], we summarize the number of security alerts related to buffer overflows explicitly for the last six years. The table shows that the percentage of explicitly buffer overflow related alerts is still high from 2004 to 2006, while the percentage declines afterward. One possible reason caused the interesting phenomenon is that many alerts from 2007 to 2009 are obtained from updates of software providers, and they do not provide much detail information about patched vulnerabilities. In general, a significant part of software vulnerabilities is resulted from buffer overflows, no matter directly or indirectly.

Table 1. Summary of recent CERT advisories (Last updated: 2004)

Year	Advisories	Buffer Overflow related Advisories	Percentage of Buffer Overflows
1996	27	5	19%
1997	28	15	54%
1998	13	7	54%
1999	17	8	47%
2000	22	3	14%
2001	37	19	51%
2002	37	21	57%
2003	28	18	64%
2004	9	7	78%
Total	218	103	47%

Table 2. Summary of US-CERT technical cyber security alerts

Year	Alerts	Explicitly Buffer Overflow Related	Percentage of Buffer Overflows
2004	27	12	44%
2005	22	12	55%
2006	39	15	38%
2007	42	11	26%
2008	38	2	5%
2009	15	2	13%
Total	183	54	30%

If a buffer overflow occurs in a program without appropriate bound checking, that will cause program crashes, undesired behaviors, or even unauthorized access to victim computers. In the worst case, an attacker can run malicious code on the victim machine

and even gain the control over the machine as an administrator. Thus, the damage due to the software vulnerability can make a great security impact on the whole system.

In order to mitigate the risks associated with buffer overflows, several different approaches for detecting and eliminating the vulnerabilities have been proposed. The existing detection techniques include statically analyzing source code and detecting buffer overflows during program execution.

1.1.1 Static Analysis

Static approaches exploit information provided in program semantics to perform source code analysis, such as identifying the usage of vulnerable functions and checking out-of-bound buffer accesses. Static analysis tools provide an effective way for vulnerability detection, because they check source code automatically without test cases. However, the fact of high false positive rate makes static tools have unacceptable performances in practical usage. For example, Splint [4], a static tool for checking vulnerabilities and errors in C programs, has an average false-positive rate of 50% [5].

1.1.2 Dynamic Analysis

The basic idea of dynamic analysis is to execute programs under test and detect the vulnerabilities at run-time. The existing tools can be divided into executable monitoring tools and compiler-based tools. Executable monitoring tools wrap the binary executable directly, and intercept the function calls of memory operation. So that memory manipulations in program can be monitored. For instance, Valgrind [6] is a binary instrumentation framework for dynamic program analysis, such as memory debugging and memory leak detection. Compiler-based tools insert instrumentation code to program source for monitoring at compile time. By retrieving the run-time information from instrumented code, dynamic analysis tools can check whether potential vulnerabilities exist or not. C Range Error Detector (CRED) [7] performs buffer overrun detection by maintaining a data structure for memory objects.

Dynamic detection is capable of finding out buffer overflows without any false positives, which static tools cannot achieve. But the accurate result is at the cost of slow performance, since a program needs to be instrumented with extra checking code for

run-time monitoring. So the program under dynamic test is significantly slower than the original one.

1.2 Problem Description and Objective

Although static and dynamic analysis tools have their own pros and cons, combination of both is a good way to take advantages of the two different approaches. Static testing has been well-developed, while there are still research issues in dynamic testing. In an automatic dynamic testing framework, detection and input generation approaches are both required. However, most dynamic detection tools do not provide an efficient way to generate test cases capable of reaching the condition of buffer overflows in the program under test. In this paper, we propose an efficient approach of test case generation that leads the execute path to where buffer overflows occur. The proposed approach is based on search-based testing and a data flow analysis technique called Chaining Approach. If the search finds test data which executes the target statement but fails to trigger the buffer overflow condition, the Chaining Approach performs data flow analysis to identify some paths other than current one. By traversing these paths, buffer overflow can occur with more chances. Two optimization strategies are introduced to reduce the overhead of the Chaining Approach, and the performance of the whole search process can be improved. A static checker is also integrated into our testing framework to prevent from paying too much run-time overhead on unnecessary testing target. Briefly, the goal of this paper is to detect buffer overflow vulnerabilities via efficient automatic generation of test data. We evaluate the new approach by applying it to three simple test objects and a real vulnerability. In the comparison of different configurations of optimization strategies, the search with appropriate strategies outperforms the original search.

The main contributions of this paper are listed as follows:

1. We propose an efficient approach of test data generation for buffer overflow detection. Using search-based testing to generate test cases and invoking the Chaining Approach as a backup strategy for search failures, two optimization strategies are presented to improve the whole search process. The experiment

results show that the proposed approach reduces the overhead of the Chaining Approach, and improves the performance of the test case generation.

2. We introduce an approach to improve the data flow analysis technique: the Chaining Approach. The proposed optimization strategies can be applied to the test case generation not only for buffer overflow detection but also for general search-based test data generation. According to the characteristics of the program under test, the level of improvement could vary. As shown in the evaluation results, the strategies should be applied based on the control structure or data flow information of the program under test to gain the best improvement.

In the next section, the related works are discussed. Section 3 provides some background about our approach. In section 4, we present the overview of the proposed approach. And the followings are sections for implementation of the testing framework and experiment results. In final section are discussion and conclusion.



2. Background

This section presents an overview of background to our approach, including search-based test data generation, computation of the objective function for covering buffer overflows, and the concept of the Chaining Approach.

2.1 Search-based Test Data Generation

In recent years automatic test data generation has been a topic of interest for many researchers, and there are two approaches with respect to generation methods are well-developed: constraint-based testing (CBT) [8-11] and search-based testing (SBT) [12]. While CBT models the input generation as a logical formula solving problem whose solutions are test input to a program, SBT represents test data finding as an optimization problem by searching the input space for relevant test data. This paper focus on search-based testing that is used to generate test data for buffer overflow detection as we will illustrate in later sections. As most works of SBT do, we describe automatic test generation as a process that finds program input so that the program execution is driven to a specific target statement, i.e. a selected code element. For example, where a buffer overflow may happen is chosen as a target for testing.

Based on search heuristics, the approaches of search-based testing formulate test data generation as an optimization problem. The program input domain forms a search space, in which a heuristic function generates a search landscape. The test data is obtained by searching for minimum points on the landscape. In the initial step of the search process, the program under test is given randomly generated inputs. By monitoring execution flow on that input, we can keep track of explored paths. An objective function is applied to measure how close the explored paths are to cover the selected target. In the following iterations of testing, the input data will be changed to be suitable for testing criteria based on the objective value of explored paths, and uncovered paths including the target might be explored. In other words, the program execution is guided toward the test target by the value evaluated in an objective function. However, for different types of structural coverage criteria, different approaches have been proposed. Goal-oriented approach [13] was developed to focuses on covering a

specified target statement, while most of previous works use path-oriented approaches which achieve statement coverage of a selected path. To avoid the requirement of path selection, goal-oriented approach classifies branches in control flow graph to prevent the execution of undesired branches that leads to failure of target coverage.

A key component of search-based testing is the search algorithm, which decides how to generate the input data. There have been various search algorithms to be utilized. Hill climbing [14], simulated annealing [15, 16], and evolutionary algorithms [17] have been adopted to generate test data. When considering the issue of selection of search algorithms, global optimization is more appropriate than local ones to prevent the trap of sticking in local optimum in the search landscape that could result in search failure. According to the idea, a global-optimized evolutionary algorithm is used in our proposed approach.

Evolutionary algorithms are genetic meta-heuristic search techniques that simulate evolution as an optimization procedure to evolve candidate solutions. Using operators inspired from genetics and natural selection, evolutionary algorithms can be applied to difficult or even undecidable problems and have well approximating solutions. Evolutionary testing (ET) is a testing process that applies evolutionary algorithms to test data generation. The burgeoning approach have been widely studied in the literature and applied to software testing, especially in structural test data generation [18]. In recent years, many researchers in this field use genetic algorithms, a particular class of evolutionary algorithms, as the search techniques to find test data. In genetic algorithms, solutions are encoded as genetic structure called individuals or chromosomes. Search process is driven by two operations: mutation and crossover. Mutation modifies solutions randomly, whereas crossover does exchange and recombination of different partitions between individual solutions. In selection phase, each solution is evaluated based on an objective function known as fitness function in evolutionary algorithms to decide which ones will be retained and others will be discarded in the following process.

2.2 Objective Function

As mentioned previously, an objective function measure how close an explored path on the generated test data is to cover the target statement. If some undesired branch is taken through an execution path, the objective function will give a value that indicates how far the program execution can take the alternating branch. In order to satisfy each branch predicate along the desired path, the objective function derived from the predicates that is assumed to be of the form $a \text{ op } b$, where a and b are arithmetic expressions and op is a relational operator. As Table 3 shown, the objective function is calculated based on the relation of both operator a and b in binary predicates, where K is a failure constant that is added to differentiate values under true and false predicates if the test data is undesired. This objective function measure how 'close' the predicate is to being true. The search of test input is guided by the obtained objective value, and the path along desired predicates can be traversed by generating appropriate inputs.

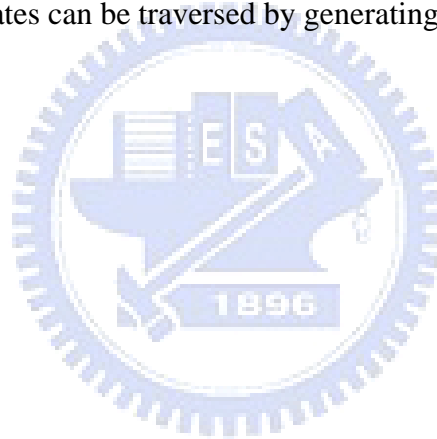


Table 3. Objective function calculation.

Relational predicate	Objective function f
boolean	If TRUE then 0 else K
$a = b$	If $abs(a - b) = 0$ then 0 else $abs(a - b) + K$
$a \neq b$	If $abs(a - b) \neq 0$ then 0 else K
$a < b$	If $a - b < 0$ then 0 else $(a - b) + K$
$a \leq b$	If $a - b \leq 0$ then 0 else $(a - b) + K$
$a > b$	If $b - a < 0$ then 0 else $(b - a) + K$
$a \geq b$	If $b - a \leq 0$ then 0 else $(b - a) + K$
$a \vee b$	$\min(f(a), f(b))$
$a \wedge b$	$f(a) + f(b)$
$\neg a$	Negation is moved inwards and propagated over a

CFG Node	
s	int void max(int a, int b)
	{
1	if (a > b)
2	return a; // target
	else
3	return b;
	}

Figure 1. Example function for objective value calculation.

In the example of Figure 1, node 1 is a branching node and node 2 is the target. If given the input ($a = 10, b = 20$), program execution will traverse the undesired branch (1, 3) without reaching the target. At this moment the objective value for the branching node 2 is calculated according to the functions of Table 3. For the predicate for node 2,

$a > b$, the objective function is $b - a + K$. Under assumption that K is 0, the objective value is $20 - 10 = 10$. Then the test data search process can change the program input and generate new inputs based on the calculated value, either in evolutionary algorithms or other search techniques. The zero objective value means the alternating branch is taken and the target is covered.

Search-based testing can achieve various testing criteria by taking advantage of objective value calculation, while the most often used criterion is branch or statement coverage. When applying search-based testing to detect particular vulnerabilities (e.g. buffer overflows), an intuitive idea is to set where these vulnerabilities are located as the target. This may lead to execution of the target statement but do not make vulnerabilities to be detected, because some specific paths are required to be traversed so that exceptional conditions related to these vulnerabilities can be exposed. For example, a buffer access where an overflow may occur is set as the test target. The existing approach could reach the target but may not make the buffer out of bound. In this case, additional terms need to be included in the objective function that guides the search toward regions of test input where buffer overflows are likely to be exposed.

2.3 The Chaining Approach

The existing methods that only use control information (e.g. control flow graph) about a program will have trouble in guiding the search for the correct solutions. The search failure due to data dependencies within the program shows that only control information is insufficient to handle various program structures. Take the function in Figure 2 as an example, where the target is the execution of node 5 that is influenced by a data dependence of variable b . To execute the target node, b has to be 1 that only happens when the input variable a is 0. The situation frequently occurs in many programs, but it cannot be handled by the methods making use of only control information. Actually, the search failure can be avoided in this case if data dependences related to the test target were also taken into consideration. In regard to this issue, the Chaining Approach [19], an extension of goal-oriented approach, uses data flow analysis to improve the chance of finding test data if control flow information fails to guide the search process.

CFG Node	
s	int ca_example(int a)
	{
1	int b = 0;
2	if (a == 0)
3	b = 1;
4	if (b == 1)
5	return 1; // Target
6	return 0;
	}

Figure 2. Example for the Chaining Approach.

The basic idea of the Chaining Approach is to identify certain statements which define variables used in a "*problem node*", then construct abstract paths that lead to the execution of target by traversing the definition nodes prior to the problem node. A problem node is referred to as where search process has difficulties to find test data for preferred execution flow in control flow graph. Basically, the abstract paths referred to as "*event sequences*" is a sequence of executed nodes, where an event is an executed node. An event sequence consists of events and the order of each event means its order in an execution path. Formally speaking, an event is a tuple $e_i = (n_i, C_i)$ where n_i is a program node in control flow graph and C_i is a set of variables called "*constraint set*" in which each variable cannot be modified until the next event $e_{(i+1)}$, and an event sequence $E = \langle e_1, e_2, \dots, e_k \rangle$ is an sequence of events.

Extending from the goal-oriented approach, the Chaining Approach serves as a backup strategy employed if the original method failed to search for appropriate solutions. At the beginning, the initial event sequence $E_0 = \langle (s, \emptyset), (t, \emptyset) \rangle$ contains the start node s and the target node t that each has an empty constraint set. Once the search

process encounters a problem node p at which test input cannot be found to alter the execution flow toward the preferred branch, data flow analysis is applied to identify last definitions of variables used at node p . Let one of the last definition nodes is d_i . A new event e_d consists of node d_i and a appropriate constraint set that prevents the variable defined at d_i or previous nodes are redefined again and keeps a definition-free path. A new event sequence E_i (for $i > 0$) is formed by inserting new events and the problem event (p, \emptyset) . If the input search cannot find test data to execute the path indicated by an event sequence, another problem node occurs and a new event sequence is generated in the same way. All generated event sequences are organized in a tree structure. Root of the tree is initial event sequence with the first encounter problem node. During the traversal of a tree node, child nodes with new event sequences are formed when encountering a new problem node.

2.3.1 Example for the Chaining Approach

In the given example of Figure 2, the target is execution of node 5. Initial event sequence is $E_0 = \langle (s, \emptyset), (5, \emptyset) \rangle$. Without data flow information, methods without data flow information are hard to find input data to take true branch from node 4 where the predicate is $(b == 1)$. Because the objective function only takes the branch distance at node 4 into account, and is not aware of the data dependence that the b is only to be 1 when a is 0. So node 4 is marked as a problem node and is inserted into the event sequence:

$$E_0 = \langle (s, \emptyset), (4, \emptyset), (5, \emptyset) \rangle$$

Only variable b is used at node 4, then loop up the last definition for node 4 and two nodes are obtained: node 1 and 3. For each of both nodes, a new event consists of node itself and a constraint set with one element b . The effect of the constraint set is to keep the last definition of node 4 from redefinition. Two new event sequences are generated by placing event $(1, \{b\})$ and $(3, \{b\})$ before the problem node 4 in the event sequence E_0 separately:

$$E_1 = \langle (s, \emptyset), (1, \{b\}), (4, \emptyset), (5, \emptyset) \rangle$$

$$E_2 = \langle (s, \emptyset), (3, \{b\}), (4, \emptyset), (5, \emptyset) \rangle$$

Searching input for E_1 may not have any improvement since no more information is given to the search process, while E_2 gives more branch distance information at node 2 to guide the execution for covering the required node 3 that result in the true branch at node 4 executed.

2.3.2 Formal Description

The generation of an event sequence in the Chaining Approach is described as follows. Assume that $E = \langle e_1, e_2, \dots, e_i, e_{i+1}, \dots, e_m \rangle$ is the event sequence on which the search is finding input. Right after the event e_i is executed, a problem node p is encountered. Suppose that at a program node only one variable is defined. Let d to be one of the last definitions of problem node p , and $def(d)$ is a variable defined by d . Two new events are obtained from the problem node and its last definition: $e_d = \langle d, def(d) \rangle$ and $e_p = \langle p, \emptyset \rangle$. By inserting e_d and e_p into event sequence E , a new event sequence E' is generated. The position of e_p in E' is right after e_i , and e_d is in a certain position that decides subpath from d to e_i . Now the new event sequence E' is almost completed:

$$E' = \langle e_1, e_2, \dots, e_k, e_d, e_{k+1}, \dots, e_i, e_p, e_{i+1}, \dots, e_m \rangle.$$

Finally, the constraint set of some events in E' have to be modified to maintain the effect of last definitions. There are three cases should be applied the constraint set maintenance.

1. $C_p = C_i$, the constraint set of e_p is the same as the constraint set of the prior event e_i .
2. $C_d = C_k \cup def(d)$, the constraint set of e_d is the union of prior event's and one variable set defined at d .
3. $C_j = C_j \cup def(d)$, $k+1 \leq j \leq i$, merge $def(d)$ with C_j to keep $def(d)$ from redefinition between e_{k+1} and e_i .

3. Related Work

Even though evolutionary testing in previous work has advantages of global optimization as compared to traditional search techniques, it also suffers from a lack of data dependence information. P. McMinn et al. [20] have developed an approach that hybridizes evolutionary testing with the Chaining Approach to address the issue. Instead of taking only a target node into consideration, a new fitness function (i.e. objective function in ET) was defined to sum up how far each event e_i is to be executed in the event sequence of the length l :

$$fitness = \sum_{i=1}^l fitness(e_i)$$

The result of experiments showed the hybrid approach can handle cases with data dependences, even with flag variables.

The Chaining Approach is able to alleviate the problem caused by data dependencies inside the program under test. However, it cannot handle complex cases such as transitive data dependencies and the dependent situation inside a loop. Based on their previous work, P. McMinn et al. proposed an extended Chaining Approach [21] to handle the problem. The basic idea is using an "*influence set*" to keep track of all variables that affect the outcome at a problem node. In the event sequence generation process, definitions for all variables in the influence set are considered as nodes required to be executed prior to the problem node. Promising event sequences can be generated for complex data dependencies which are not possible to handle by the original approach. We use the similar combination of techniques in which the extended Chaining Approach to probe new promising paths and generate test data in evolutionary testing, but for vulnerability detection specifically.

There have been several researchers applying meta-heuristic search techniques to check safety property violation or to detect software vulnerabilities. The approach presented by Tracey et al. [22] is to use search-based test data generation to find test input which can result in safety property violation. By measuring how far an identified hazard condition is to be satisfied, the fitness function gives the search the guidance

toward critical area of input domain. The outcome of experiment show the approach can be applied to safety property testing or exceptional condition testing. Another work of Tracey et al. [23] focused on generating test data for exception testing. Similar to the authors' previous work, genetic algorithms and the fitness function are used. The evaluation presents an efficient performance at exception testing that exceptional conditions are raised successfully in simple examples and a program of aircraft engine management.

Grosso et al. [24] have used evolutionary testing to detect buffer overflow vulnerabilities. Potential vulnerabilities are identified by a static analyzer at first, and genetic algorithms are used to generate test data for triggering buffer overflows. Three fitness functions are defined to capture the testing criterion of buffer overflow coverage: vulnerable coverage fitness, nesting fitness, and buffer boundary fitness. The experiment results showed buffer boundary fitness, that is the fitness function using the distance from up limit of buffer access to buffer size, outperformed others. Weights of terms in the fitness function need manual effect to adjust their values, that keeps test data generation away from automation. In Grosso et al. [25], a dynamic fitness weighting method was proposed to address the issue. Via linear programming to solve a maximization problem, the dynamic fitness weighting is useful at fast detection of buffer overflow. As the evaluation presented, the dynamic weight fitness had better performance than previous basic boundary fitness. Our work has the same goal with both papers mentioned above, but we propose different approach that uses extended Chaining Approach to help the search to find test data and detects buffer overflow effectively.

4. Method

4.1 Objective Function for Buffer Overflows

An objective function decides what input data the search will find to meet a given testing criterion. Common used testing criteria, such as branch coverage or statement coverage that try to cover specific code elements, is not sufficient for execution of a buffer overflow caused by an out-of-bound access. A potentially vulnerable buffer may be accessed in a legal range since exploitable input is not found to satisfy the exceptional condition. To define an objective function for buffer overflow coverage, relevant factors of the cause of buffer overflow is necessary to be taken into account. Three elements of the objective function are listed as follows:

1. *Out-of-bound distance*: the distance from the size of a buffer to the upper limit within the range of buffer access when the access is out of bound. Obviously, a buffer overflow occurs when the distance value is positive. The out-of-bound distance is defined in the similar form of objective functions in Table 3.

$$f_1 = \text{if } (L-BS) > 0 \text{ then } (L-BS) \text{ else } 0,$$

where L is upper limit of the range of buffer access, BS is the buffer size.

2. *Number of tainted variables*: the number of tainted variables that have a great influence on the access position of the buffer, e.g. buffer index. If more tainted variables are involved in the index expression of a buffer, the buffer could be controlled from input and result in buffer overflow with high likelihood.
3. *Depth of execution path*: the depth of the execution path that is exercised by current input data. This element of the objective function is useful in the case with continuous buffer accesses. Let a buffer access is wrapped inside a loop, and the buffer index is increased when looping. As the loop is exercised many times and the execution path becomes longer and longer, the buffer could be accessed out of bound with high probability.

From these three elements, an overall objective function can be derived:

$$F = w_1 \cdot OD + w_2 \cdot NTF + w_3 \cdot DEP,$$

where *OD* is the out-of-bound distance; *NTF* is the number of taint flow; *DEF* is the depth of execution path; *w1*, *w2*, and *w3* are weights of elements discussed as above.

According to different situations, the weight value can be adjusted in the objective value calculation. If index value could be tainted by input variables, increasing the weight of *NTF* improves the chance of finding exploitable input data. If the potential overflowed buffer is continuously accessed, the overflow condition could be raised with a high probability by increasing the weight of *DEF*. Grosso et al. [25] have used linear programming to solve the weights as a maximization problem to achieve automation.

4.2 Applying the Chaining Approach for Buffer Overflow Detection

All three terms in the objective function defined above could have some corresponding data-dependency relationships in the program under test:

1. The definitions of variables used in a buffer offset can affect the out-of-bound distance.
2. The tainted variables are actually the variables data-dependent on program input.
3. An increasing buffer offset could result in a buffer overflow. Usually this is caused by increase the offset inside a loop and the execution path is long, where the variables in the offset define themselves.

Because of these data-dependency relationships, we use the extended Chaining Approach with modification to achieve similar but more direct effects of the three terms.

4.2.1 Extra Out-of-bound Checking

Certain statements determine whether a buffer usage is out of bound or not. For example, definitions of an index variable could cause an overflow. A basic idea is to make these statements executed prior to the buffer access, so that program execution could be conducted to a buffer overflow. Let a buffer offset be the position where a buffer is accessed. Extended Chaining Approach is applied to figure out data

dependencies at the buffer offset, and construct event sequences that lead to a buffer overflow.

To make extended Chaining Approach to keep track of the data dependencies of the buffer offset, an extra bound-checked *if* statement is instrumented to wrap the vulnerable buffer access which is set as the target. If the target cannot be executed, the added *if* will be regarded as a problem node and the Chaining Approach will try to identify the statements that can make the out-of-bound condition satisfied.

As illustrated in Figure 3, a possible out-of-bound buffer access is transformed into a secure buffer access guided by an *if* statement which is an extra bound check by instrumentation. The new statement inside the *if* statement is set as the search target.

```
int add_bound_check ()
{
    ...
    if (access_range < 0 OR access_range >= buffer_size)
        return 1; // Set as search target
    else
        access_buffer; // Original statement
    ...
}
```

Figure 3. Bound check instrumentation

The extra bound check is also helpful to prevent the program crash which may result in abnormal state change in the search process. In test data generation for buffer overflow detection, the goal is the execution of an out-of-bound buffer access. The search process should be terminated when a buffer overflow is raised. Without run-time memory monitoring which slows down system performance significantly, however, a buffer anomaly is not acknowledged unless the program under test crashes. To identify buffer overruns, we instead add an out-of-bound check as a watch-point.

4.2.2 Optimization Strategies for the Chaining Approach

In a real program, there exists complex control flow and taint flow paths where statements that define buffer offsets reside, so a buffer overflow is triggered only when some long and specific paths through these critical statements are exercised. To generate the event sequences corresponding to the specific paths, the event sequence tree may grow into a large or complex structure. However, the original extended Chaining Approach could pay a lot of effort to traverse the tree and work on the event sequence of traversed node, but much effort of the tree traversal are not necessary. For the extended Chaining Approach dedicated to buffer overflow coverage, we proposed two optimization techniques to reduce the effort of the tree traversal and working on event sequences.

4.2.2.1 Look-Ahead Strategy

For a chain-like event sequence tree where the desired event sequence is at a deep node, default depth-first search (DFS) traversal can spend much time until reaching the desired node. In some cases that event sequences in a chain are similar that they share parts of events but differ in the number of a repeated event. The situation is common in the case with continuous buffer accesses for which an optimization technique called *Look-Ahead* strategy can be applied.

Let a potentially vulnerable buffer is continuously accessed inside a loop and the event sequence tree is chain-shaped. The event of increasing the index or pointer to the buffer is repeated in the chain. What the Look-Ahead strategy does is similar to ‘jump’ to a descent node in a certain level deeper than current node. And ‘jump’ again if the event sequence can be exercised successfully, or backtrack to the next of recent successful node. Actually, the strategy inserts multiple copies of the event with a self-defined node, and adjusts the number of the inserted events according to the result of exercising on the new event sequence. The process is repeated until a buffer overflow is found or the number of try-and-error reaches a limitation.

Look-Ahead strategy is integrated into the event sequence generation of extended Chaining Approach. To generate new event sequences, new events of last definitions for

a problem node are inserted. If one of event nodes has a self-defined statement, the node could be inserted into the old event sequence repeatedly to obtain new event sequences in the following process. Thus the self-defined node is identified as a *LA_node*. Let *LA_num* is the number of copies of a new event to be inserted. Initially, *LA_num* is set to 1 means that only one copy of self-defined event is inserted. If the same *LA_node* is encountered again in the next creation of new event sequences, multiple copies of the event with *LA_node* will be inserted. After exercising on the new event sequence, Look-Ahead procedure in Figure 4 is used to adjust *LA_num* to determine an appropriate number of copies for the event with *LA_node*.



Let LA_node be a node where exists a variable of the influence set defines itself inside a loop, LA_num be the Look-Ahead number, and LA_base be the base number

Let $last_index$ be the index of the last executed event in the event sequence, $prev_insert_point$ be the index of the event after which the new events are inserted in the previous iteration, and $current_node$ be the current node of the event sequence tree

Look-Ahead procedure

If (LA_node is identified)

If $last_index \leq prev_insert_point$

$current_node \leftarrow parent(current_node)$

reset LA_node and LA_num

Else if ($last_index < (prev_insert_point + LA_num)$)

$current_node \leftarrow parent(current_node)$

reset LA_node

$LA_num \leftarrow last_index - prev_insert_point$

else

$LA_num *= LA_base$

Endif

Endif

$parent(n)$ returns the parent node of node n

Figure 4. Procedure of Look-Ahead strategy

In the procedure, the current event sequence is divided into three segments. Which segment the last executed event is located decides how to adjust the value of LA_num . $last_index$ is where the last executed event located in the event sequence. If the last executed event cannot execute more events after the $prev_insert_point$, the following

traversal is backtracked to the parent of the current node and related state variables are reset since even one new event cannot be exercised. If the last executed event is after *prev_insert_point* but does not exercise all the inserted new events, the next traversal is also backtracked to the parent of the current node. However, *LA_num* is set based on the affordable amount of the insert event. If all of new events are executed, *LA_num* is increased to insert more copies of the new event. Finally, the extended Chaining Approach generates new event sequences by inserting *LA_num* copies of the event with *LA_node*.

Example

Consider the function *look_ahead_ex()* in Figure 5. A buffer overflow may occur at node 5, which is set as the target. The last definition of index is at node 2 of a *for* loop in which an *if*-statement is used as a test for the quit from the loop. When the input *buf* has the content of eight Ns, the index *i* can be increased to 8 and result in the out-of-bound access at node 5. The probability of generating the exact input value is low, therefore, the index variable *i* is put into the influence set and extended Chaining Approach is invoked. As illustrated in Figure 6, the generated event sequence tree is like a chain. The initial event sequence is $E_0 = \langle s, 5 \rangle$. The buffer overflow cannot be triggered via E_0 , then the last definition for variable *i* at node 5 is identified at node 2. A new event sequence is generated by inserting an event of node 2 into E_0 :

$$E_1 = \langle s, 2, 5 \rangle$$

However, working on E_1 still has nothing to do with detecting the buffer overflow. Since the last definition node for node 2 is itself, each new event sequence is formed by adding an event of node 2 into the parent's event sequence. Finally, the whole tree grows into the chain shape. Instead of sequential traversal in DFS strategy, Look-Ahead strategy is used to make the traversal more quickly.

CFG Node	
s	int look_ahead_ex(char buf[8])
	{
1	int i;
2	for (i = 0; i < sizeof(buf); i++)
3	if (buf[i] != 'N')
4	break;
5	buf[i] = 'M'; // Possible buffer overflow!
6	return 0;
	}

Figure 5. Example function for Look-Ahead strategy

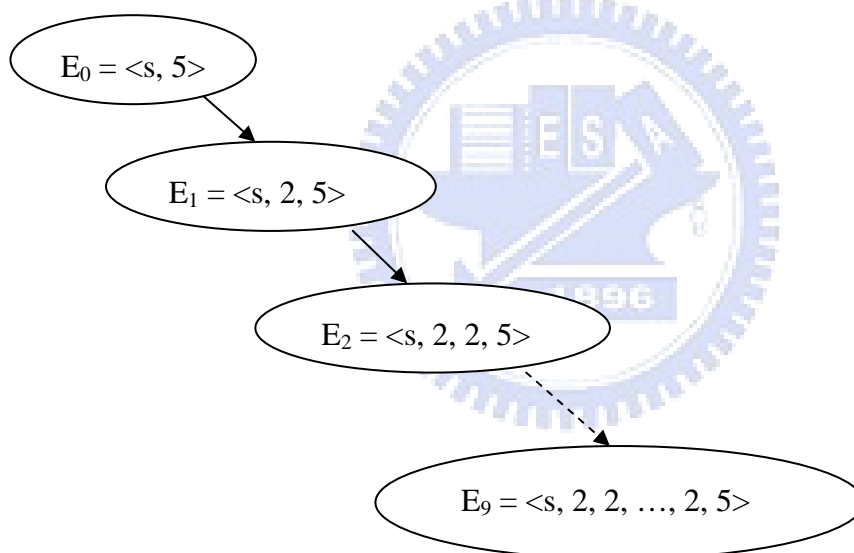


Figure 6. Event sequence tree derived from the example of Look-Ahead strategy.

It is now demonstrated how the Look-Ahead strategy is applied to the example of Figure 5. Assume the base number LA_base has the value of 5. The first new event to be inserted is the event with node 2 at which the variable i is increase by one. So the event for node 2 is identified as Look-Ahead node (LA_node). Because the first generated event sequence E_1 is exercised successfully, further event sequences would be desired.

Note that LA_num has been increase to 5. And the new event sequence can now be generated that include five more events than E_1 :

$$E_2 = \langle s, 2, 2, 2, 2, 2, 2, 5 \rangle \text{ (with six 2s)}$$

E_2 is feasible, but at node 5 there are still no any buffer overflows detected. In event sequence generation, the node 2 is encountered and the event for node 2 is identified again since the definition of i uses itself. And LA_num is multiplied by LA_base and increased to 25. The following event sequence will be generated from E_2 :

$$E_3 = \langle s, 2, 2, \dots, 2, 5 \rangle \text{ (with thirty-one 2s, where } 6+25=31)$$

This event sequence provides too many events for node 2 so that it is infeasible to execute. At most eight events for node 2 are affordable in this case. Assuming the last executed event is the eighth in the repeated events, the Look-Ahead strategy would adjust LA_num and a new event sequence is generated for the previous event sequence E_1 :

$$E_4 = \langle s, 2, 2, 2, 2, 2, 2, 2, 2, 5 \rangle \text{ (with eight 2s)}$$

This event sequence requires node 2 to be executed eight times, which results in the buffer overflow at node 5.

4.2.2.2 Tainted-First Strategy

If a buffer is tainted from input data, it can be controlled by malicious users and become vulnerable. Similarly, if a buffer offset at a buffer access statement is tainted from input data, a malicious user could control the access at an arbitrary position either out of bound or not. In other words, the tainted buffer offset has a high probability to cause a buffer overflow. Actually, a taint flow from input variables to the buffer offset is a relationship of data dependencies, from which relevant event sequences can be generated. With slight modification, extended Chaining Approach is able to figure out

the taint flow, and generate tainted event sequences which can be seen as taint flow paths.

Let a potentially vulnerable buffer to be accessed with an offset in a program. Assume that there exists a taint flow from input variables to the buffer offset. Extended Chaining Approach is applied to handle the data dependencies related to the buffer. Before the process of event sequence generation, data flow analysis with respect to program input is done to realize which variable definitions have used input variables. If a new event has a node where program input is used to define other variables, the event sequence generated from the new event is referred to as a *tainted event sequence*. Since it is more possible to find buffer overflows via the tainted event sequences, they would be chosen in a high priority.



CFG Node	
s	int Tainted_First_ex(char *s) {
	int first, last;
	u_int i;
1	i = 0;
2	while (isdigit(*s)){
3	i = i*10 + (*s - '0');
4	s++;
	}
5	first = i;
6	s++;
7	i = 0;
8	while(isdigit(*s) {
9	i = i*10 + (*s - '0');
10	s++;
	}
11	last = i;
12	if (first >= vec_max)
13	first = vec_max - 1;
14	if (last >= vec_max)
15	last = vec_max - 1;
16	while (first <= last)
17	vec[first++] = i;
18	return 0;
	}

Figure 7. Example for Tainted-First strategy

Example

For example, the function in Figure 7 is based on code from sendmail 8.11.0-5. The input string *s* is converted to two index value of *first* and *last*. The bound checks for the index variables are not proper because of a type casting side effect at node 12 and 14. Therefore, both checks can be bypassed by making indices negative. So it is possible to

overflow the buffer at node 17 where a negative index is regarded as a very large positive value.

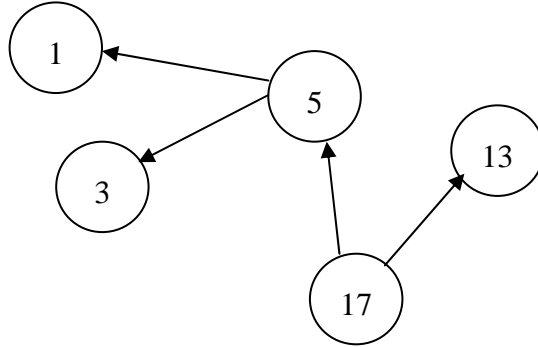


Figure 8. Data dependency graph of the example function for Tainted-First strategy.

The data dependencies at node 17 are shown through the graph in Figure 8. If consider the tainted paths prone to the buffer overflow at node 17 and the data dependency relationship for the offset *first*, the three nodes {3, 5, 17} have to be passed orderly. The generated event sequence tree is illustrated in Figure 9. E_5 is the first tainted event sequence to be identified since it is the first one that has the event of node 5, where the input variable s is involved in the definition. Thus, the search would exercise on E_5 prior to other event sequences, and the buffer overflow can be detected earlier than using the DFS strategy.

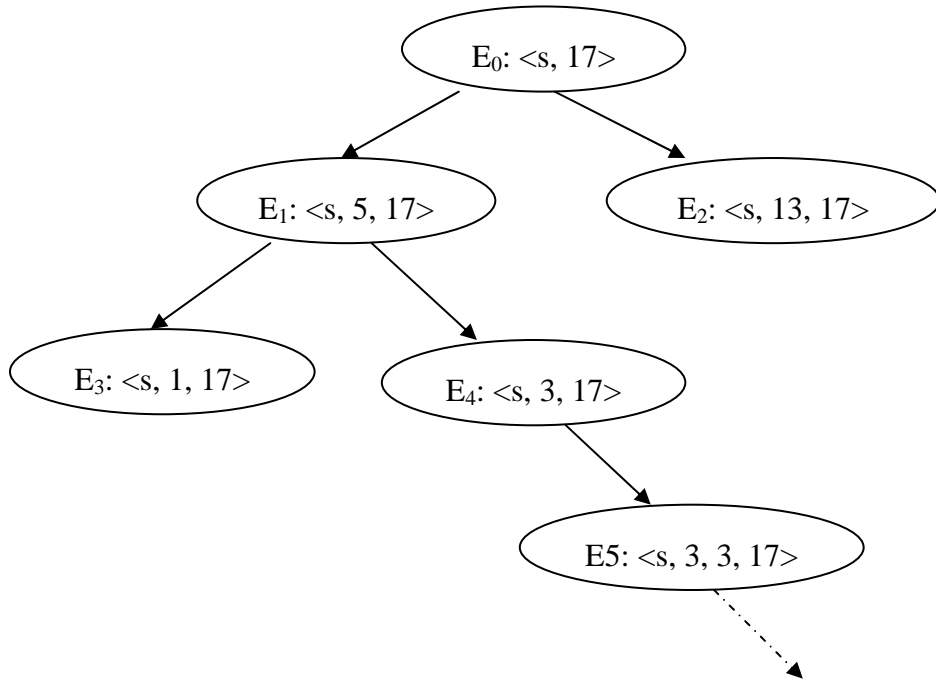
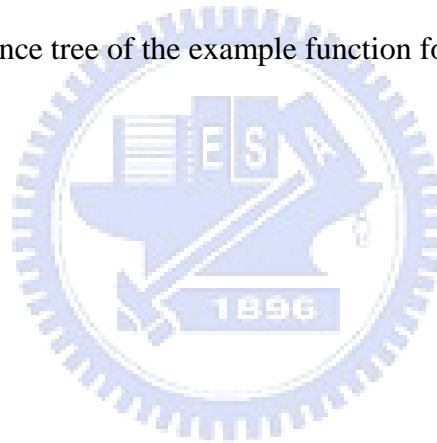


Figure 9. Event sequence tree of the example function for Tainted-First strategy.



5. Implementation

The main components in our testing system are outlined in Figure 10 and explained in the remainder of this section. The system consists of two main components: static frontend and search-based test data generation. The latter is what we have implemented and developed the approach mentioned in the previous section. Through a static analysis of program source code, a set of potentially vulnerable code statements (referred to as suspects) are produced in the form of source code location (e.g. line number and buffer name). Provided with the program source and a suspect, search-based testing component uses heuristic techniques to generate test data on which the suspect is manifested through the reproduction of a buffer overflow.

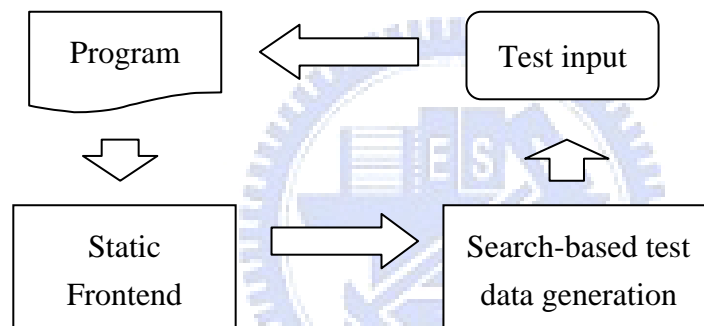


Figure 10. Test input generation process

5.1 Static Frontend

The static checking tool *splint* [4] is used as the frontend in our system to analyze the program under test and produce suspects in which buffer overflows could happen. Because static tools usually suffer from a high positive rate, the dynamic analysis approach, that is search-based testing, is used in combination with the static frontend. Thus, the reported suspects will be set as test targets of search-based testing to identify real vulnerabilities.

5.2 Dynamic Component: Search-based Testing

Program instrumentation is required to analyze a program at run-time. We use CIL

[26] to instrument programs under test for information collection and computation for search-based test generation. CIL is an OCAML application for program analysis and transformation for C programs. Control flow and data flow graphs are constructed based on the corresponding analysis provided by CIL. The extra bound checking for buffer overflows is also inserted via the program analysis tool.

The main parts of search-based testing have been implemented in C++. To facilitate the graph computation for control and data information, boost graph library [27] is included to take advantages of available graph search algorithms. As for input generation, a genetic algorithm library for C++ called GALib [28] is used to develop the required functions. The input search process is integrated into the objective value calculation in GALib. The basis of our approach, extended Chaining Approach proposed in [21], is deployed as a backup strategy in search-based testing.

We have encountered two problems when using GALib to perform search-based testing. The first is solution convergence: when all genomes in the population become similar, the search process will stop since no better solutions could occur. This often cause search failures if the genomes converge before the exact solution is found. The second problem is that the genetic algorithms may spend too much generations or iterations to search for the right solution when input domain is large but the expected input resides in a small area.

In our implementation, a local search algorithm is hybridized with the genetic algorithm to address the issue. If the solution convergence occurs, Hill Climbing is invoked to find better test data. If the number of execution iterations exceeds a specific threshold, Hill Climbing is also invoked to search for other test data in case of that the search is trapped in the current state.

6. Results

A set of case studies is designed to show how the proposed optimization strategies can improve the performance of the search-based testing with extended Chaining Approach with respect to buffer overflow detection. Three simple test objects are used as subjects of the experiments for the evaluation of different strategy configurations. To test the practical effectiveness of our approach, we choose a real vulnerability and perform the experiment.

6.1 Test Objects

Each of these test objects has one-buffer-overflow vulnerability which is instrumented with an extra bound check and selected the true branch as the test target. The probability of entering each of targets is low enough since the buffer overflows occur only when some specific inputs are given.

Continuous Access

This test program can be seen in Figure 5 and was introduced in Section 4.2.2.1. There exists a continuous buffer access wrapped inside a loop, where the index is increased. A possible buffer overflow is located at node 5. Here the bounds check has been forgotten, and the mistake could result in an off-by-one overflow.

Taint Flow

The control structure of this test object is similar to the program of “Continuous Access”. The vulnerable buffer is also accessed after leaving the loop. However, inside the loop is a branch of taint flow while another branch is not tainted. What could be tainted is a definition for a buffer offset that decides the position of buffer access.

Command Selection

This program performs a command selection task and takes two arrays of five integer values. Each integer is read from an input array to determine which command will be executed. What the first command does is to check if the value read from another input array is positive. If the value is positive, an index is increased by one and a buffer is written. The buffer write is vulnerable because of a lack of bounds checking.

Multiple-level nested if statements make the control structure of the program complex and difficult for input search.

tTflag Buffer Underrun: CVE-2001-0653

The published vulnerability is the original version of the example in Figure 7. The `tTflage()` function can parse a command-line value and store the parsing result in a trace vector. The input parameter is a string that has the form of “x-y”, where x and y are transformed from characters to integers. Both x and y are assigned as the index values for the buffer access. Although there have been two bound checks to limit the values of x and y, it is possible to bypass the first check with a negative value. The problem is that the check does not limit the lower bound of x to zero. A very large value of x can be seen as negative and pass the check, and make the buffer overflowed.

6.2 Experimental Setup

We ran each experiment with a fixed setting of the GALib’s evolutionary algorithm. There are 50 individuals per generation that form only one population. The search process terminates if no solution has been found after 1000 generations. Real-valued encodings are used to represent valuables in programs.

Each experiment with each test object was repeated 20 times for four different strategy configurations:

1. Original Approach: the extended Chaining Approach is used without any strategies enabled
2. Look-Ahead: Only Look-Ahead strategy is enabled.
3. Tainted-First: Only Tainted-First strategy is enabled.
4. Look-Ahead & Tainted-First: Both of Look-Ahead and Tainted-First are enabled.

The setting of the Chaining Approach is also set to fixed values. The maximum depth of event sequence tree is limited at 15. Look-Ahead number is set as 5. If the best objective value is not improved over two generations, Hill Climbing is used to search for other solutions. Furthermore, if the algorithm cannot get better objective value over 60 generations, the Chaining Approach is invoked to seek better solutions.

6.3 Evaluation Results

Each experiment performed on each test object has a successful search rate of 100%, no matter with what optimization strategies. This shows that the Original Approach is sufficient to handle all test programs, and the proposed strategies do not reduce the success rate.

Table 4 shows the average number of objective evaluations for four strategy configurations. To realize how the strategies improve the original approach, the comparison of objective evaluations of strategies based on Table 4 is illustrated in Table 5. Notice that each strategy configuration is compared to Original Approach. The experiment results with respect to each test object are discussed below.

Continuous Access

This program features a continuously increased buffer index, resulting in the chaining mechanism could insert the same self-defined events to generate new event sequences. Thus, the characteristic of continuous buffer access make both Look-Ahead enabled configurations have obvious improvement over others.

Taint Flow

In the loop of the test object, taking the taint branch or the other without taint affects the search performance. With Tainted-First, the desired event sequences with taint flow are chosen prior to other sequences, so the buffer overflow is raised in a less number of objective evaluations.

Command Selection

It is required for the buffer overrun to following the taint flow from the input command string and increase the buffer index. Although Look-Ahead and Tainted-First can make the search to reduce a certain number of iterations, the improvement is not obvious since the complex control structure.

tTflag Buffer Underrun: CVE-2001-0653

The buffer index required two critical conditions to be out-of-bound. First, the index must bypass the bounds check or it will be redefined as a constant value. That is, the program execution should follow taint flow. Second, the index has to be large enough

and cause a sign change after the type casting in the bound check. Tainted-First is helpful in the first condition and Look-Ahead is for second one. The row data of tTfalse() in Table 5 shows that each of both strategies has its benefit, and the performance of combining both strategies is even better.

Table 4. Average objective evaluations for each test object against different strategy configurations.

Test Object	Original Approach	Look-Ahead	Tainted-First	Look-Ahead & Tainted-First
Continuous Access	34854	13894	34963	13904
Taint Flow	21202	19917	15103	12709
Command Selection	39413	31555	34009	33805
tTflag() in sendmail	48931	29256	38129	27406

Table 5. Comparison of objective evaluations of strategies to the original approach without any optimization strategies.

Test Object	Original Approach	Look-Ahead	Tainted-First	Look-Ahead & Tainted-First
Continuous Access	100%	40%	100%	40%
Taint Flow	100%	94%	71%	60%
Command Selection	100%	80%	86%	86%
tTflag() in sendmail	100%	60%	78%	56%

7. Conclusion

Search-based testing is an intelligent and efficient approach to generate test data for structural testing. As a backup mechanism for search failures, the Chaining Approach helps to handle the problems due to data dependencies. The well-developed test data generation approach has been applied to buffer overflow detection. This paper shows that search-based testing can be improved based on the characteristics of programs under test. The proposed optimization strategies, i.e. Look-Ahead and Tainted-First, reduce the overhead of the Chaining Approach, and also reduce the overall cost of the search process.

We performed the experimental studies with a real vulnerability of sendmail and three simple test objects. These test objects have different program structures and characteristics, so the performance of each strategies configuration varies. In the worst cases, it degenerates to the original approach. However, it was shown that our approach does decrease the number of objective evaluations in all cases, either in an individual strategy or a combination. The way to achieve the best improvement is to choose the suitable strategy configuration based on specific program features.

There are many interesting ways to extend this work. The test data generation for buffer overflow detection requires the run-time monitoring to retrieve the memory status. Via a run-time memory checker, e.g. Beagle [29] or MemCheck of Valgrind, search-based testing can be applied to detect more types of buffer overflows.

Another interesting avenue is hybridizing search-based and constrain-based testing. There have been researchers proposed the integration to take the both advantages. Tao Xie et al. [30] presented an approach based on dynamic symbolic execution, also known as concolic testing, assisted with fitness-guided path exploration. Kobi et al. [31] have proposed an integration framework using evolutionary and concolic testing for object-oriented programs, in which one of test data generation methods is responsible for class method sequences and the other handles unit testing. Inspired from the previous work, an integration idea is to use search-based testing to find a path toward the specific vulnerability and cocolic testing is invoked to find test data that satisfy the

vulnerability condition.



References

- [1] N. Tuck, B. Calder and G. Varghese, "Hardware and binary modification support for code pointer protection from buffer overflow," in *Proceedings of the International Symposium on Microarchitecture*, pp. 209-220, 2004.
- [2] CERT, CERT/CC advisories, <http://www.cert.org/advisories/>, January 2004.
- [3] US-CERT, Technical cyber security alerts, <http://www.us-cert.gov/cas/techalerts/>, June 2009
- [4] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE Software*, pp. 42-51, 2002.
- [5] M. Zitser, R. Lippmann and T. Leek, "Testing static analysis tools using exploitable buffer overflows from open source code," *ACM SIGSOFT Software Engineering Notes*, vol. 29, pp. 97-106, 2004.
- [6] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 89-100, 2007.
- [7] O. Ruwase and M. S. Lam, "A practical dynamic buffer overflow detector," in *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, 2004,
- [8] P. Godefroid, N. Klarlund and K. Sen, "DART: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.
- [9] K. Sen, D. Marinov and G. Agha, "CUTE: A concolic unit testing engine for C," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 263-272, 2005.
- [10] C. Cadar and D. Engler, "Execution generated test cases: How to make systems code crash itself," in *Proceedings of the 12th International SPIN Workshop on Model Checking of Software (SPIN'05)*, 2005,
- [11] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill and D. Engler, "EXE: Automatically generating inputs of death," in *CCS '06: Proceedings of the 13th ACM Conference on Computer and Communications Security*, pp. 322-335, 2006.

- [12] P. McMinn, "Search-based software test data generation: a survey," *Software Testing, Verification & Reliability*, vol. 14, pp. 105-156, 2004.
- [13] B. Korel, "Dynamic Method of Software Test Data Generation," *Software Testing, Verification & Reliability*, vol. 2, pp. 203-213, 1992.
- [14] B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering*, vol. 16, pp. 870-879, 1990.
- [15] N. Tracey, J. Clark, K. Mander and J. McDermid, "An automated framework for structural test-data generation," in *Proceedings of the 13th IEEE international conference on Automated software engineering*, pp. 285-288, 1998.
- [16] N. Tracey, J. Clark and K. Mander, "The way forward for unifying dynamic test-case generation: The optimisation-based approach," in *International Workshop on Dependable Computing and its Applications (DCIA)*, 1998, pp. 169-180.
- [17] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas and K. Karapoulios, "Application of genetic algorithms to software testing (application des algorithmes genetiques au test des logiciels)," in *5th International Conference on Software Engineering and its Applications*, pp. 625-636, 1992.
- [18] P. McMinn and M. Harman, "A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2007)*, pp. 9-12, 2007.
- [19] R. Ferguson and B. Korel, "The chaining approach for software test data generation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 5, pp. 63-86, 1996.
- [20] P. McMinn and M. Holcombe, "Hybridizing evolutionary testing with the chaining approach," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2004)*, 2004,
- [21] P. McMinn and M. Holcombe, "Evolutionary testing using an extended chaining approach," *Evolutionary Computation*, vol. 14, pp. 41-64, 2006.
- [22] N. Tracey, J. Clark, J. McDermid and K. Mander, "Integrating safety analysis with automatic test-data generation for software safety verification," in *Proceedings of the 17th International Conference on System Safety*, pp. 128-137, 1999.

- [23] N. Tracey, J. Clark, K. Mander and J. McDermid, "Automated test-data generation for exception conditions," *SOFTWARE—PRACTICE AND EXPERIENCE*, pp. 61-79, 2000.
- [24] C. Del Grosso, G. Antoniol, M. Di Penta, P. Galinier and E. Merlo, "Improving network applications security: A new heuristic to generate stress testing data," in *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation*, pp. 1037-1043, 2005.
- [25] C. Del Grosso, G. Antoniol, E. Merlo and P. Galinier, "Detecting buffer overflow via automatic test input data generation," *Computers and Operations Research*, vol. 35, pp. 3125-3143, 2008.
- [26] G. C. Necula, S. McPeak, S. P. Rahul and W. Weimer, "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," *LECTURE NOTES IN COMPUTER SCIENCE*, pp. 213-228, 2002.
- [27] Boost, "Boost Graph Library," <http://www.boost.org/doc/libs/release/libs/graph/>
- [28] M. Wall, "GAlib: A C++ library of genetic algorithm components," *Mechanical Engineering Department, Massachusetts Institute of Technology*, 1996.
- [29] C. Tsai and S. Huang, "Detection and diagnosis of control interception," in *9th International Conference on Information and Communications Security (ICICS)*, 2007,
- [30] T. Xie, N. Tillmann, P. de Halleux and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," Technical Report MSR-TR-2008-123, Microsoft Research, 2008.
- [31] K. Inkumsah and T. Xie, "Evacon: A framework for integrating evolutionary and concolic testing for object-oriented programs," in *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, pp. 425-428, 2007.