Auditing buffer overflow vulnerabilities using program analysis and data mining techniques

Bindu Madhavi Padmanabhuni

2016

Padmanabhuni, B. M. (2016). Auditing buffer overflow vulnerabilities using program analysis and data mining techniques. Doctoral thesis, Nanyang Technological University, Singapore.

https://hdl.handle.net/10356/68915

https://doi.org/10.32657/10356/68915

AUDITING BUFFER OVERFLOW VULNERABILITIES USING PROGRAM ANALYSIS AND DATA MINING TECHNIQUES

BINDU MADHAVI PADMANABHUNI

SCHOOL OF ELECTRICAL AND ELECTRONIC ENGINEERING

2016

AUDITING BUFFER OVERFLOW VULNERABILITIES USING PROGRAM ANALYSIS AND DATA MINING TECHNIQUES

BINDU MADHAVI PADMANABHUNI

BINDU MADHAVI PADMANABHUNI

School of Electrical and Electronic Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Doctor of Philosophy

2016

Acknowledgment

Life as a research student was filled with myriad of enriching experiences. The skills learnt, the challenges faced and the conversations held have significantly contributed towards enhancing my critical-thinking and problem-solving skills. I have many cherished moments that are fondly etched in the memory and am indebted to all those who made them possible.

I would like to express my sincere and profound gratitude to my supervisor, Prof. Tan Hee Beng Kuan for initiating me into the field of research and nurturing me throughout the course of study. I am extremely grateful for his invaluable guidance, advice and continuous support. Under his tutelage, I have become a fine researcher and better writer.

I also take this opportunity to thank Dr. Chia Tee Kiah for sharing his expertise and giving insightful suggestions on various research ideas. Special mention of thanks goes to Vivek, Le Ha Thanh, Deepak and Lei Lei Win for the wonderful tête-à-têtes, debates and camaraderie during my time in the Computer Security Lab.

I would also like to thank my fellow students Kaiping, Shar, Ding Sun, Mahinthan and Charlie for the rigorous and fruitful discussions pertaining research ideas and issues at hand.

I am greatly indebted to my parents, especially my mother who has always been my pillar of strength, parents-in-law and sisters for their care, encouragement and love. Special thanks to my husband, Sridhar, for his practical and emotional support and my children Shyam and Saket for brightening up my days with their love and innocence.

TABLE OF CONTENTS

SUMMARY

This thesis presents approaches for mitigating Buffer Overflow (BO) vulnerabilities, one of the most prevalent and dangerous vulnerabilities from source code as well as x86 executables.

Current approaches to mitigate BO vulnerabilities can be broadly classified into three types namely: defensive coding, vulnerability detection before deployment and run-time exploit prevention after deployment. Defensive coding approaches include using safe versions of known vulnerable C string library functions, validating input thoroughly before propagating and performing bounds checks on arrays before writing to them. Vulnerability detection approaches help identify BO bugs in the program source code or, very rarely, using binary executables. Run-time detection techniques add instrumentation code to detect and intercept exploits during run-time.

While these approaches help mitigate and address some BO vulnerabilities, continuous presence of BO bugs in present day vulnerability reports suggests possible limitations in existing approaches or difficulty in their adoption. Defensive coding practices are helpful in developing more secure programs but they may not work when the source code is unavailable or should remain unaltered. Static vulnerability detection techniques, due to their conservative nature, are prone to generate large number of false positive cases. Run-time prevention techniques are effective in thwarting some known exploit patterns on deployed applications. But given the nature of BO vulnerabilities they typically fall short in preventing diverse exploit forms in addition to incurring high overhead costs. It should also be noted that majority of the approaches in literature only address mitigating BO bugs from source code whereas commercial-off-the-shelf and third party components are mostly distributed in the form of binaries.

Hence, it is evident that despite having wide-ranging solutions, BO vulnerabilities continue to surface often due to inherent drawbacks in present mechanisms or difficulty in using them. Therefore, complementary or alternative solutions which are effective and easy-to-use are needed to comprehensively address BO vulnerabilities. It is also imperative to devise mechanisms for auditing BO bugs from executables. Based on these observations, in this thesis, we propose three novel approaches for auditing BO vulnerabilities namely: test case generation, vulnerability auditing through hybrid analysis for addressing BO vulnerabilities in source code and vulnerability prediction methodology using static code attributes for predicting BO bugs in x86 executables.

The objective of the thesis is to use program analysis for auditing BO vulnerabilities. Hence our focus is on developing methodologies that support detection before deployment. Testing mechanisms help to prove bugs when revealing inputs can be generated thereby assisting the end-users to directly patch them up without further auditing efforts. Current testing approaches use heavy-weight techniques like symbolic execution to generate test cases. We therefore start with a light-weight test generation approach to prove bugs by leveraging on static analysis information. One drawback of testing schemes is that it is not possible to infer on vulnerability status of statements not proven vulnerable. Majority of static detection tools in literature model buffer usage in the program in form of constraints to generate warnings of possible overflows by solving the constraints. They suffer from scalability issues and generate too many false warnings. Inspired by defect and vulnerability prediction studies, which use easily obtainable code attributes like lines of code and complexity metrics to predict software defects and vulnerabilities, we explore on possible alternatives using program analysis information and data mining techniques for auditing BO bugs. We hence proposed a hybrid auditing methodology that first tries to prove bugs using dynamic analysis by test inputs generated as per prescribed rules. If a statement cannot be proven vulnerable, it is predicted using prediction models built using static code attribute data and machine learning algorithms. While both the aforementioned approaches can only work in the presence of source code, to plug in the dearth of solutions that cater to binary vulnerability analysis, we further proposed binary BO prediction methodology using static code attributes that can be extracted from x86 executables to form a comprehensive range of BO auditing techniques.

All the proposed approaches use program analysis information for auditing buffer overflows. Vulnerability prediction based proposals in the thesis for source code and x86 executables make use of data mining techniques on attribute data collected from static analysis information of the programs to predict bugs.

We first propose a novel light-weight rule-based test generation methodology for detecting BO vulnerabilities. The approach uses information collected during static analysis to automatically identify constraints on input and uses different input lengths based on buffer size to generate test cases. Computationally intensive techniques like constraint solving are currently not scalable to large systems in the context of vulnerability detection. Since our proposed approach doesn't involve any symbolic execution or constraint solving, it can be considered as light-weight and practical alternative to existing BO test generation solutions that can be easily incorporated into software development life cycle for early detection of bugs.

Subsequently, we propose a hybrid approach for BO auditing using static-dynamic analysis and machine learning. To counter BO exploits developers typically perform size checks before buffer filling operations and validate input before processing. An application is vulnerable if such defense mechanisms are absent or inadequate. In contrast to existing methods solely aiming to prove vulnerabilities, examining defenses implemented in the code can be useful in predicting bugs by providing probabilistic observations of statement's vulnerability if it cannot be proven vulnerable through dynamic analysis. Hence in our hybrid BO auditing approach we use static analysis information to gather code attributes capturing such defense mechanisms implemented in the code. Information gathered during attribute collection is also used for generating test inputs. The goal of the test inputs is to trigger overflows and confirm their vulnerability through dynamic analysis. For those statements that are not proven vulnerable using dynamic analysis, prediction models are built to predict vulnerabilities based on static code attribute data and machine learning algorithms.

Majority of existing BO solutions can only work in the presence of source code. Very few tools are available for analyzing binaries as it is challenging to extract and analyze information from binaries. Hence much of vulnerability detection research is limited to source code analysis. It is extremely difficult to use static analysis alone to infer BO vulnerabilities precisely from binaries. Therefore by combining prediction method and static analysis we provide an important avenue for addressing this problem in x86 executables through our binary BO prediction methodology proposal. This thesis also evaluates the proposed approaches and demonstrates that they are useful and effective.

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

1.1 Motivation

Software security is utmost concern in today's highly connected world for ensuring data and systems integrity. Given our ever-increasing reliance on computing system's services and functionality, any security risks to their availability, integrity and/or confidentiality could have serious and severe consequences. Code injection vulnerabilities such as buffer overflows (BO), SQL injection (SQLI), cross site scripting (XSS) are commonly found in applications with inadequate input validation. Among them, BO vulnerabilities, despite being known for a long time, are highly prevalent and are listed as third in CWE/SANS Top 25 Most Dangerous Software Errors [1]. Payment Card Industry (PCI) Data Security Standard (v 3.0) recommends applications to be tested for BO to develop and maintain secure systems and applications [2]. Recent bug reports reveal that BO bugs are regularly found in wide ranging applications from web browsers and graphics drivers to media playing software [3]. A simple keyword search on National Vulnerability Database [4] shows that 140 BO bug reports were published in the first half of 2015, out of which 94 (67%) were with a high severity rating indicating the security risks involved with the BO bugs. Many techniques were proposed in literature to handle this security threat. They can be classified into defensive coding, vulnerability detection, vulnerability testing, and run-time detection and recovery techniques.

BO bugs occur due to poor programming practices. Applying defensive coding techniques with security in mind is the best solution to eliminate them. Choosing programming languages like Java or environments like .NET that perform runtime bounds checking eliminates the problem. Apart from safe versions of C [5, 6], other defensive coding measures suggested for mitigating BO vulnerabilities in C/C++ include using secure versions of vulnerable C library functions, thoroughly validating input before propagating it in the program, and performing bounds checking before buffer filling operations [7]. Though effective, it can be seen that the onus of incorporating the defensive measures dwells on the development team. Consequently, due to their labor-

intensive nature, they are liable to be error-prone and are also hard to enforce into applications that have already been deployed.

Most static BO detection approaches [8-10] track buffer declaration and usage in the form of integer range constraints (e.g., buffer allocation size, used buffer size) to reason about out-of-bounds accesses through constraint solving. They raise warnings when an out-of-bound access is detected. Model checking methods [11, 12] convert this buffer range-checking problem into error-checking problem by generating constraints representing the checks and finding out paths reaching the error constraints. Some tools [13-15] need programmers to provide desired features in the form of pre- and post-conditions to generate constraints on vulnerable program constructs for bug detection. This necessitates the end-user to supply annotations for checking. Static detection tools suffer from high false alarms as the nature and precision of inference and analysis sensitivity affects their detection outcome.

Test generation methodologies proposed for BO detection include evolutionary algorithms [16, 17], mutation [18], fault-injection [19] and combinatorial testing mechanisms [20]. Symbolic evaluation techniques [21-23] also were proposed where path conditions are represented in the form of input constraints to generate test inputs for exploring the program paths containing potential vulnerable statement constructs. Some methods represent each element of the input string by a symbolic value whereas others represent only a part of the input symbolically. Full symbolic evaluation stresses the constraint solver as it requires huge memory to keep track and solve for the symbolic values while partial symbolic evaluation may need manual intervention in determining the initial representation size and optimizing it, if it is not adequate. Both these techniques suffer from path and state space explosion problems.

Stack-smashing and heap-buffer metadata exploits are most commonly known BO attacks. Therefore, many run-time detection techniques were proposed to prevent stack and heap-smashing attacks [24-30]. Most of the approaches for preventing stack-smashing attacks check for the modification of return address after a function call returns. If the return address is modified, they detect attack and halt execution. These methods cannot offer protection from exploits targeting other sensitive variables like pointers on the stack. Techniques for preventing heap-smashing attacks track buffer creation through dynamic memory allocators (e.g. *malloc*) by maintaining their size and allocated address ranges for detecting overflows in heap. These run-time methods can help prevent some but not varied BO attack forms on deployed applications [31], thereby, leaving bountiful avenues for exploiting those which cannot be handled by these techniques.

BO Endurance approaches [32-37] typically let the program to continue execution instead of halting it by either allocating more buffer space or by relocating vulnerable buffers to protected memory areas when an overflow is detected during run-time. They usually incur large overheads due to instrumentation and tracking purposes. Network based techniques [38, 39] look for presence of attack code in packets or use past-exploit signatures to prevent exploits. Although automated signature generation approaches were proposed [40-43], substantial losses might have already been incurred before signature generation.

Present BO mitigation approaches are mainly oriented towards source code based bug detection. Very few solutions exist that cater to binary or executable code [44-47]. Binary analysis is deemed harder because much of the source code level semantics and type information is lost during the executable generation process. In addition to that, there are not many tools available that can analyze binaries to detect vulnerabilities. Therefore binary BO detection or to that matter vulnerability detection is very much an unchartered territory. Binary vulnerability analysis is necessary and important when incorporating third party components whose source code is not distributed with the executables. It can also help in detecting bugs in proprietary software or legacy code.

It can be seen from the above discussion that existing techniques for detecting BO bugs do not focus on systematic reporting of defensive measures implemented in the program for the potential vulnerable statement constructs. Therefore, even though they may either prove or generate warnings to help in vulnerability detection, security auditors will still need to examine defensive measures implemented in the program by inspecting the chunks of program code to determine their adequacy.

Vulnerability prediction using data mining is a relatively new field of study [48, 49] as compared to defect prediction research. Defect prediction approaches [50-52] in literature predict software bugs by using data mining techniques on software code attributes such as those representing lines of code (LOC), McCabe's [53] or Halstead's [54] code complexity metrics. Some studies use product's vulnerable component's function call information in software version histories to predict vulnerabilities in new components [55]. One of the major drawbacks of such methods is that they report vulnerabilities or defects at a module or component level, which is coarse-grained. This requires considerable manual review efforts to locate the vulnerable code in the reported vulnerable module/component.

In summary, we can see that existing BO mitigation techniques suffer from one or more of the following drawbacks: (1) labor intensiveness, (2) high false alarms, (3) scalability issues (4) run-

time overheads, (5) lack of program defensive measure application information support, (6) work mainly in the presence of source code and are (7) coarse-grained. Therefore, end-users may find it hard to adopt these existing approaches.

1.2 Objectives

The objective of the proposals made in this thesis is to investigate the application of program analysis, empirical knowledge and data mining techniques for mitigating BO vulnerabilities from source code as well as x86 executables. Motivated by the observations that existing approaches in literature either suffer from inherent drawbacks or have issues limiting their usage by development teams, we propose novel approaches to augment and improve over existing vulnerability detection techniques. We develop tools that implement the proposed approaches so that they can be applied on real-world applications and help developers in auditing vulnerabilities before deployment.

1.3 Overview of Our Research

Program analysis plays a pivotal role in software-engineering tasks as it supports in analyzing the program to collect information for inferring program correctness. On the other hand, data mining techniques have shown promising outcomes in predicting software defects. In recent years, studies [56, 57] have shown that security properties of software can be inferred by combining empirical knowledge with program analysis information more effectively than conventional techniques.

Therefore, in this research, we explore the use of program analysis and data mining techniques for mitigating BO vulnerabilities. From our survey on BO approaches [58], we observed that current techniques in literature have some form of shortcomings or weaknesses. For example, static detection approaches have high false alarms whereas run-time detection techniques suffer from large instrumentation overheads. Most of the techniques can only work in the presence of source code, which is a further hindrance in checking for bugs in binary executables. Hence, in this thesis, we propose three approaches for mitigating BO vulnerabilities - *test generation, vulnerability prediction,* and *vulnerability auditing*. Each approach handles different aspects of alleviating BO vulnerabilities by contributing towards bug auditing before deployment phase.

The first proposal automatically generates test cases using information obtained from static analysis to alleviate manual test design needs as well as expertise and optimization efforts needed to operate heavy-weight tools like those using symbolic evaluation. Since test generation techniques help in identifying bugs straight away, they are preferred over other auditing schemes

that flag potential vulnerabilities which need additional manual auditing efforts to confirm the tool warnings. One potential drawback of testing techniques is that it is not possible to infer about statements not shown vulnerable. Existing detection tools model buffer usage in form of constraints and solve for them to raise warnings of possible overflows. They suffer from scalability issues and have very high false alarm rates. We therefore proposed a hybrid vulnerability auditing approach using static-dynamic analysis and data mining that either tries to prove bugs or predicts them based on static code attribute data using machine learning algorithms. Since most of BO detection techniques in literature need the presence of source code, we proposed a vulnerability prediction methodology using static code attributes obtained from disassembled binaries to plug in this gap. The proposed static code attributes in the prediction based approaches can be easily extracted with the help of a static analysis tool. Hence these prediction models could provide an economical and effective alternative to heavy-weight techniques by providing probabilistic observations of statement's vulnerability.

All the three proposed approaches are light-weight and are grounded on information retrieved by performing program analysis. The last two methods also involve data mining techniques. In all these approaches we use domain knowledge to discover and reflect interesting patterns pertaining BO vulnerability mitigation in the form of suitable models to be used for test generation, vulnerability prediction, and auditing purposes.

The thesis first presents source code based test generation approach that generates test cases by extracting potential input constraints from static program analysis information. It will later be shown in the thesis that buffer size and input predicate information obtained from static analysis can be used to effectively detect vulnerabilities.

Next in the thesis we propose a source-code based hybrid vulnerability auditing approach by combining static and dynamic program analysis with machine learning. Although the test generation approach in the previous proposal supports bug detection, not much can be known about statements that were not proven vulnerable by it. Therefore, the hybrid auditing approach presents a comprehensive solution that either tries to prove vulnerabilities using test inputs or predicts them using static code attributes representing defensive measures implemented in the code for their safety. It makes use of empirical knowledge obtained from BO studies to identify and extract patterns that result in BO vulnerabilities and defensive features that prevent from resulting in vulnerabilities. These patterns are modelled in the form of static code attributes. Information collected from the attribute collection phase is used to generate test inputs for revealing BO bugs through dynamic analysis. Confirmed cases can be fixed by developers without further

verification. Statements whose vulnerability is not confirmed by dynamic analysis are predicted by mining static code attributes using machine learning techniques. It will be shown in the thesis later that the proposed hybrid approach is practical and successful in BO auditing.

Both the above approaches can only be used in the presence of source code. To enable auditing in binary executables, we next present a novel vulnerability prediction approach using static code attributes obtained from binary disassembly. Binary analysis is difficult due to lack of tools for supporting it and information loss or retrieval difficulty caused by compilation. To overcome these challenges, we propose code attributes capturing vulnerability patterns that can be easily extracted from disassembled binaries. We also present the theory and heuristics employed in our binary static analysis framework to handle the impediments encountered in the course of analyzing binaries. We then show that the proposed approach is effective in predicting BO vulnerabilities from x86 executables with high recall and accuracy.

Therefore, our binary vulnerability prediction approach complements the above two source code based approaches by extending the state-of-the-art vulnerability prediction methodology to x86 executables. More importantly, our work provides inexpensive yet effective alternative and complementary solutions to existing research works in addressing BO vulnerabilities which can be easily incorporated in the software development life cycle.

1.4 Major Contributions

In this thesis, we propose three novel approaches for mitigating BO vulnerabilities. These approaches have been published in international journals and reputable conferences such as *COMPSAC* and *AST*. The major contributions of this thesis are:

1) **Vulnerability prediction:** We propose a novel methodology for predicting BO bugs by proposing static analysis based attributes characterizing BO defensive measures and buffer usage patterns extracted from disassembled binaries. Most of the current approaches in literature can only help mitigate BO bugs when the source code is available. Such methodologies therefore are not viable in the absence of source code. This could be because binary analysis is considered challenging and very few tools are available that can help analyze executables. Therefore the thesis proposes an approach that can be used to automate bug detection in binaries whose source code is not readily available. It thus enables the prediction of BO vulnerability from x86 executables using static analysis possible. We hypothesize that the proposed static attributes are important indicators of BO vulnerabilities.

We conducted a series of experiments using benchmark programs to prove our hypothesis. This work has been published in [59].

2) **Test case generation:** We present an automated rule-based test generation approach for revealing BO bugs. The approach first identifies potential vulnerable statement constructs in the program source code to gather static analysis information pertaining them. We propose rules for generating test cases using this extracted information to reveal bugs. Our hypothesis is that static analysis information can be used to generate bug revealing inputs with light-weight rule-based approach. We evaluated the proposed approach on a set of vulnerable benchmarked application programs to demonstrate its effectiveness. Vital part of the work presented in this thesis has been published in [60].

3) **Vulnerability auditing:** We propose a hybrid auditing approach that captures the defense features implemented in the program source code using static and dynamic program analysis. In this work we examine the buffer size checking and input validation schemes implemented in the code before buffer filling operations to provide probabilistic observations of statement's vulnerability using machine learning, if it cannot be proven vulnerable through dynamic analysis. Information obtained from static analysis is used to collect proposed code attributes characterizing the defenses and also to generate test inputs for dynamic analysis. The test inputs aim to trigger overflows and confirm their vulnerability. Statements not proven vulnerable using dynamic analysis are predicted using machine learning algorithms built using static code attributes. Since the test inputs may themselves reveal bugs, it is an improvement over prediction only approaches as auditing efforts that would be otherwise needed to confirm predicted vulnerabilities are reduced. We hypothesize that proposed code attributes can help infer vulnerabilities and that the information obtained from static analysis can be used in generating bug revealing inputs which can considerably reduce the manual auditing efforts.

   We prove the effectiveness of our hypothesis by conducting experiments on various benchmark programs. A preliminary version of this work has been published in [61]. The work presented in this thesis has been published in IET Software [62].

1.5 Thesis Organization

This thesis is organized as follows. Chapter 2 presents the background knowledge about the techniques used in the thesis. It first presents the BO vulnerability and various techniques commonly used in exploiting it. This is followed by a discussion on program analysis techniques

as well as data mining-based learning schemes which were used in the approaches proposed in the thesis.

Chapter 3 discusses the works in literature that are related to those presented in the thesis. We discuss defensive coding strategies proposed to limit overflows, test generation techniques, software defect and vulnerability prediction techniques used to identify bugs or problematic code regions, vulnerability detection mechanisms that identify and locate vulnerabilities in programs, and attack prevention techniques that ward off security attacks at runtime.

Chapter 4 presents the light-weight rule-based test case generation approach. The proposed approach uses information collected from static program analysis and pre-defined rules to generate test cases. As the approach uses only static analysis information without any constraint solving it is termed as light-weight. We discuss on the considerations for proposing the rules and present the details needed to collect the static analysis information for triggering the rules. The approach can be easily adopted by development teams to aid in preliminary bug analysis and is a practical alternative to existing solutions which typically use heavy-weight symbolic execution or evolutionary algorithms for generating test inputs. We then evaluate the proposed approach on a set of C benchmark programs to show its effectiveness.

Chapter 5 presents our work on hybrid analysis for auditing BO vulnerabilities from source code. In this chapter, we propose a hybrid approach combining static and dynamic program analysis with machine learning for auditing bugs. We propose simple rules to generate test data to confirm some of the vulnerabilities through dynamic analysis. Confirmed cases can be patched straight away without further review. Statements whose vulnerability is not confirmed by dynamic analysis are predicted by mining static code attributes. We also present our prototype tool called *BOAttrMiner* that collects data from C programs. Using the tool, we conduct experiments on benchmark applications to demonstrate the usefulness of the hybrid approach.

Chapter 6 presents a novel methodology using static analysis and machine learning for predicting BO vulnerabilities from x86 executables by characterizing buffer usage patterns and defensive mechanisms implemented in the code. The proposed characterization attributes are extracted by statically analyzing disassembled binaries. Since it is hard to find publicly available and mature tool that can support binary static analysis, we built up on existing tools and analysis frameworks to derive necessary functionality for performing binary static program analysis. We first describe the challenges we encountered during the process along with the solutions for overcoming them. We next present the proposed static code attributes for binary BO vulnerability

prediction followed by experimental evaluation to establish the effectiveness of the proposed approach.

We summarize the contributions of this research in Chapter 7 and identify possible scope for future work.

Appendix A lists the notation conventions used in the thesis.

Chapter 2

BACKGROUND

This chapter provides background on BO vulnerability followed by outline of program analysis concepts and data mining techniques used in the thesis. Section 2.1 presents BO vulnerability and methodologies typically used to exploit it. Section 2.2 gives an overview of program analysis techniques along with detailed presentation of techniques employed in the approaches proposed in this thesis. Section 2.3 presents data mining techniques used for data analysis purposes.

Before discussing the BO vulnerability, we shall introduce some definitions and terms, which are used throughout the thesis. Our analysis is based on control and data dependency information computed using the *control flow graph* (CFG) of the program. CFG could be built from either program source code or assembly code of disassembled binary. A node in the CFG represents a program statement. In this thesis we use the terms node and statement interchangeably. *Input* refers to external input submitted to the application program.

A *potential vulnerable statement* (PVS) is a statement in the program such that its execution can result in an undesirable behavior if the variables referenced in it are not adequately constrained. BO occurs during program execution when an application writes beyond the bounds of a pre-allocated fixed-size buffer. This data overwrites adjacent memory locations and, depending on what it overwrites, the overflow can affect the program behavior. Lack of bounds-checking operations before filling the buffers permits this error to happen. Applications written in programming languages such as C or C++ are commonly associated with BO vulnerabilities because they allow overwriting any part of memory without checking whether the data written will overflow its allocated memory. We treat calls to vulnerable C library functions (e.g., *strcpy*, *memset*) and statements writing to buffers (e.g., arrays) as PVS in this thesis. Since BO attacks are triggered through external input, we may also address it as tainted data in this thesis.

Attackers can use BO vulnerabilities present in the program to launch denial-of-service (DoS) attacks, spawn a root shell, gain higher-order access rights (especially root or administrator privileges), steal information, or impersonate a user. BO vulnerability can be classified as code injection vulnerability as it can be exploited by sending malicious input to execute user-supplied

exploit code (e.g., shell code). Depending on the location of exploitable vulnerable buffers, BO vulnerabilities can be used to alter program control to execute user-supplied code or re-direct the program control flow to execute the code already existing in the program (e.g., *return-to-libc*). In the following section, we describe various forms of BO exploits.

2.1 Buffer Overflow Vulnerability

In 1998, the Morris worm, one of the first to strike the Internet, exploited a BO in the Unix finger daemon (fingerd) to propagate itself from one machine to another (http://en.wikipedia.org/wiki/Morris_worm). The 2001 Code Red worm took advantage of a BO weakness in the Microsoft IIS webserver and reportedly infected 359,000 systems within 14 hours (http://en.wikipedia.org/wiki/Code_Red_(computer_worm)). In 2003, SQL Slammer exploited a BO bug in the Microsoft SQL server, spread quickly, and launched a DoS attack on various targeted networks (http://en.wikipedia.org/wiki/SQL_slammer). Despite being known and studied for a long time BO vulnerability reports continue to surface often in the vulnerability and bug reporting databases often with a high severity rating indicating the prevalence of errors and seriousness of their presence.

To carry out an exploit, attackers must find suitable code to attack and make program control jump to that location with the required data in memory and registers. Attackers glean information about the vulnerable program code and its runtime behavior from the program's documentation and source code, by disassembling a binary file, or by running the program in a debugger. BO exploits generally target function activation records, pointers, or the management data of heap-based memory blocks.

2.1.1 Function Activation Record Exploits

This is a popular technique, which targets the function activation record. When program execution calls a function, a stack frame is allocated with function arguments, return address, the previous frame pointer, saved registers, and local variables. In the stack frame, return address points to the next instruction for execution after the current function returns. Attackers can overflow a buffer on the stack beyond its allocated memory and modify the return address to change program control to a location of their choice.

Instead of supplying executable code, an attacker can supply data to a C library function, such as *system*, that is already present in the program code. Such exploits are called *return-to-libc* attacks because they direct control to a C library function rather than to attacker-injected code.

They also alter the return address. *Return-to-libc* attacks are ideal for exploiting programs that have memory protection mechanisms, like non-executable stacks, because they do not execute attacker-supplied code. Fig. 2-1 shows examples of frame activation record and *return-to-libc* attacks. Consider the vulnerable code snippet in Fig. 2-1(a). If argument *str* to *proc* is longer than 31, then the local variable *buffer* gets overwritten and modifies the memory area next to it. Fig. 2-1(b) and (c) show the stack frame of *proc* before and after *strcpy(buffer,str)* respectively. Exploit 1 uses a string consisting of shell code and the memory address to which the attack code copies the shell code to cause a BO. When the function returns, it jumps to the shell code to spawn a shell. Exploit 2 in Fig. 2-1 is a *return-to-libc attack* on the code snippet that spawns a shell by overwriting the return address with the address of *system*. Fig 2-1(d) shows the stack frame after buffer overflow and before program returns to *system*. The attack code includes the address for *system* as well as its parameters. Fig 2-1(e) shows the stack frame after program returns to *system*.

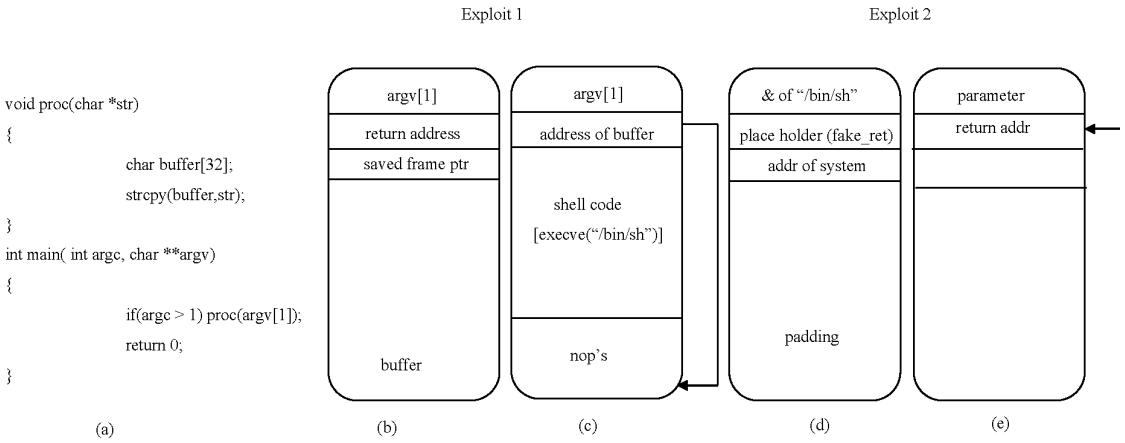


Figure 2-1. Frame Activation record exploit

Another target of frame activation record based exploits is the previous frame pointer. An attacker can build a fake stack frame with a return address pointing to the attacker's code. An overflow of the previous frame pointer will point to this fake stack frame. When the function returns, the attacker's code executes.

2.1.2 Pointer Subterfuge Exploits

Pointer subterfuge exploits involve modifying pointer values, such as function, data, or virtual pointers. They also can modify exception handlers.

Consider the following code snippet, which includes a buffer that has a function pointer allocated in the data section:

```
char buf[64];
int pfn(char *) = NULL;
void main(int argc, char **argv)
{
        …
        strcpy(buf,argv[1]);
        iResult = pfn(argv[2]);
        …
 }
```

Figure 2-2. Sample code snippet vulnerable to function pointer attack

An attacker can use *strcpy* to overflow the buffer and overwrite the function pointer. The overwritten pointer can point to the address of shell code or *system*. The attack takes place when the program calls the function pointer. Attackers can use pointer subterfuge in overruns of stacks, heaps, or objects containing embedded function pointers. This kind of attack is especially effective when the program uses methods for preventing return address modification because it does not change the saved return address.

An exploit can use data pointers to indirectly modify the return address. Such indirect overwriting schemes are useful if the program uses a protection mechanism like StackGuard [25] because they alter the return address without changing the canary—a value placed in the stack. When a buffer in the stack overflows, it will corrupt the canary. A program can use the canary as a check against BO exploit.

Another method for hijacking program control uses *longjump* buffers. The C standard library provides *setjmp*/*longjmp* to perform nonlocal jumps. Function *setjmp* saves the calling function's environment into the *jmp_buf* type variable (which is an array type) for later use by *longjmp*, which restores the environment from the most recent invocation of the *setjmp* call. An attacker can overflow the *jmp_buf* with the address of the attacker's code such that when the program calls *longjmp*, it will jump to the attacker's code.

Although not widely used, virtual-function pointer smashing is a threat even when a program uses anti stack-smashing protection, because such protection does not defend against overflow in the heap. C++ compilers use tables to implement virtual functions. These tables have an array of function pointers that the program uses at runtime to implement dynamic binding. Each instantiated object has a virtual pointer pointing to its virtual table as part of an object's header. By

making the virtual pointer point to an attacker-supplied virtual table with injected code, an attacker can transfer control to this code at the virtual function's next call.

Fig. 2-3 shows sample code vulnerable to virtual-function pointer smashing. This attack overflows the object's member variable *buffer* to modify the *vptr*, making it point to an attacker-supplied virtual table with injected code. Control will then transfer to this attacker supplied code with the next call to the virtual function.

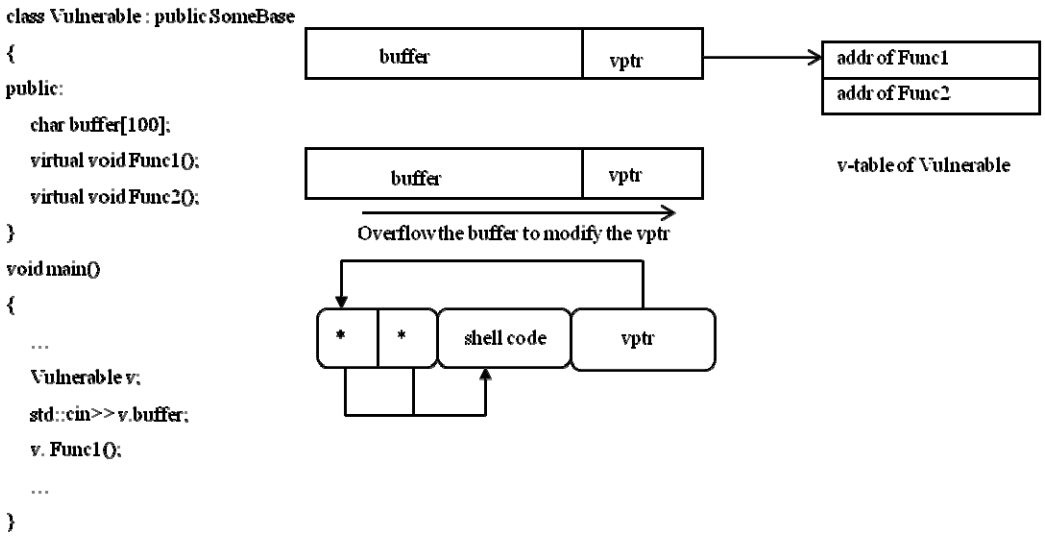


Figure 2-3. Sample code snippet vulnerable to virtual function pointer smashing

The Microsoft Windows Structured Exception Handling (SEH) mechanism is also another exploit target. When the program generates an exception, Windows SEH will catch it if the program has no handler or if the provided handler cannot process the exception. The function pointer for the exception handler is on the stack. By overflowing the stack buffer, an attacker can modify it to transfer control to another location.

2.1.3 Heap-based Exploits

Dynamic memory allocators such as *malloc* allocate memory on the heap dynamically during runtime. Linked lists manage memory blocks that are allocated and deallocated dynamically using *malloc* and *free*. The management data for each memory block, such as its size and pointers to other memory chunks, is stored in a linked-list-like data structure. The user data and management data are adjacent in a chunk similar to local variables and the return address on a stack. By overflowing user data in the memory block, an attack can corrupt the management data. However, modifying such data does not change program control because this data is not a pointer. Heap-

based exploits corrupt the metadata of heap-allocated memory blocks and use it to change other pointers.

It can therefore be seen from the above discussion that location of vulnerable buffer and means to trigger and exploit the overflow are important for launching attacks. Attackers typically try to alter code pointers to execute the attacker supplied code for gaining control over the machine. There are many resources available on the internet that sketch the details of identifying and exploiting BO bugs such as the famous Phrack magazine article about smashing the stack [63]. Therefore, before deploying an application it is prudent to conduct stringent checks to ensure security.

2.2 Program Analysis

Program analysis tools and techniques help in discovering facts about the program being analyzed. The analysis information thus obtained can be used for understanding the program behavior or for program optimization, verification, validation, error detection purposes. Based on the nature of analysis, program analysis[1] techniques can be broadly classified into static and dynamic program analysis. We use information obtained from static analysis techniques (such as control and data dependency analysis) in our proposed approaches. Our source code based hybrid auditing proposal also uses dynamic analysis approach of program instrumentation for dynamic testing. We therefore provide background information about the program analysis approaches used in the solutions proposed in this thesis along with a brief overview of other commonly employed program analysis techniques for BO detection in the following discussion.

2.2.1 Static Program Analysis

Static program analysis refers to techniques that analyze program code without executing it. Control and data flow analysis techniques are two main branches of static program analysis that help in identifying control and data dependencies of program points. They assist in software engineering tasks such as program optimization (e.g., redundant code elimination, infeasible path identification), program comprehension (e.g., which statement controls/affects the execution of a given statement, how does information flow to the statement within a program), testing and debugging (e.g., model checking). Due to their usefulness and effectiveness in identifying program dependencies, the control and data flow analyses form the basis for all the vulnerability mitigation approaches proposed in this thesis.

[1] Program analysis may be performed on source code or disassembled binary code. In this thesis, we consider both source code and disassembled binary program analysis.

Throughout the thesis, we use the terms and definitions in [64] for control and data flow analysis.

2.2.1.1 Control Flow Analysis

Control flow analysis helps in determining the hierarchy of the flow control in a program. For example, it helps in determining the order in which statements are executed and which statement controls the execution of other statements in the program. The flow control hierarchy thus determined is usually represented in the form of a directed graph called the *control flow graph* (CFG). The CFG represents all possible control-flow or execution paths in the program and is used in wide range of program analysis and software engineering tasks such as dependency analysis, program slicing and debugging.

Sinha et al [64] define CFG of a procedure P as a directed graph $G = (N, E)$ in which N contains one node for each statement in P, and E contains edges that represent possible flow of control between statements in P. It should be however be noted that instead of a statement, a node in N could also be used to represent a basic block. A basic block in the CFG is a sequence of statements in P with a single entry and exit statement at the start and end of the basic block respectively. If there are multiple procedures in the program, each procedure can be represented by its own CFG to compute dependency analysis either through context strings approach or summary-based approach. Another way of performing inter-procedural analysis is by constructing an *inter-procedural CFG* (ICFG), consisting of inter-connected CFGs, one for each procedure call by in-lining the called procedure at the call-site and performing dependency analysis on this ICFG as in intra-procedural CFG.

The CFG has two distinguished nodes, namely, the 'entry' and 'exit' nodes. An 'entry' node has no predecessors (i.e., 'in-coming' edges) whereas the 'exit' node has no successors (or 'out-going' edges) in the CFG. A predicate node in the CFG (or if using a basic block to represent a node in CFG, then the end statement of the basic block which is a predicate statement) has two successors with edges labelled 'true' and 'false' representing the 'true' and 'false' branches of the predicate respectively. All other nodes in the CFG have only one successor.

An path (n_1-n_k) in a CFG is defined as a sequence of nodes $W = n_1, n_2, ..., n_k$ such that $k \geq 0$, and such that, if $k \geq 2$, then *for* $i = 1, 2..k\text{-}1$, $(n_i, n_{i+1}) \in E$.

Control dependencies are computed using *post-dominator* information obtained from CFG. Node u in a CFG *dominates* node v if and only if every path from the 'entry' node to v in G

contains *u*. Node *u postdominates* node *v* if and only if every path from *v* to the exit node in *G* contains *u*.

Node *u* is *control dependent* on node *v* if and only if *v* has successors v' and v'' such that *u postdominates* v' but does not *postdominate* v''. In general, node *v* should be a predicate node (or a basic block with a predicate statement as end statement) for *u* to be control dependent on it. The control dependency information, computed using the *postdominator* relationships constructed from the CFG, can be represented in the form of a *control dependence graph* (CDG). The nodes in the CDG represent program statements and edges in the graph represent the *control dependence relations* between the statements.

2.2.1.2 Data Flow Analysis

Data flow analysis [65] is the process of collecting information about the flow of the data in the program. It can be used to answer various queries such as what might be the possible set of values at a program point, which program statement defines a variable used at another program point, the liveness of a variable at a point of interest etc. Various optimizations such as constant propagation, dead code elimination can be carried out through usage of data flow analysis. '*Reaching Definitions'* analysis and '*Live variable'* analysis are classic examples of forward-and backward data-flow schema respectively. '*Reaching Definitions'* analysis determines what definitions of a variable reach a particular program point whereas '*Live variable'* analysis determines if the value of a variable at a program point could be used along some path in the CFG starting from that point.

Node *u* is *data dependent* on node *v* if there exists a variable *x* such that *v* defines *x* and *u* uses *x*, and there is a definition clear path in *G* from *v* to *u*.

Data dependency relations are expressed in the form of *data dependence graph* (DDG) in which nodes represent program statements and edges connecting the nodes represent the *data dependency relations* between the statements.

We used the control and data dependency relations computed by CodeSurfer [66], a commercial code analysis tool, in our proposed source code-based BO mitigation approaches. For our binary analysis approach we used the disassembly produced by IDA Pro commercial disassembler [67] and CFG representation generated by Rose open source compiler infrastructure [68] to calculate control and data dependencies. For computing control dependency, we used the usual *postdominator* based approach. Data dependency relationships were calculated using standard iterative work-list algorithm using CFG of the procedure. The details of the algorithms

used for control and data dependency analysis are presented in Chapter 6 where we discuss our binary static analysis framework and approach for mitigating BO vulnerabilities from x86 executables.

2.2.2 Dynamic Program Analysis

Dynamic analysis is the analysis of the properties of a running program. Static analysis approaches examine the program to derive properties that hold over all execution paths whereas dynamic analysis derives properties that hold over one or more executions by examining a running program [69]. Programs are typically instrumented to extract useful run-time information to derive desired properties. Since static analysis derives its properties from observing all the execution paths in the program, static BO detection techniques generally raise too many false warnings as they approximate the buffer ranges to comprehensively encompass possible buffer manipulations in the program. In contrast, dynamic analysis cannot prove if the program satisfies the required properties but, in the presence of selected test inputs, it can examine and report property violations if they happen to occur during execution. Therefore, although it is not as exhaustive as its static analysis counterpart, dynamic analysis can help in improving the accuracy and understanding of the program behavior as it is derived from concrete program execution. It should be noted that as dynamic analysis incurs run-time overheads the improvement in accuracy could be coupled with higher costs than that of static analysis.

Dynamic program analysis has been used in applications such as anomaly detection [70-73] and test input generation [22]. The proposed approaches in this thesis are primarily based on static program analysis information and data mining techniques. Dynamic analysis is used in our source-code based hybrid BO auditing approach to improve accuracy and in our proposed rule-based test generation approach to prove its effectiveness. Both these approaches used program instrumentation to prove BO bugs with generated test inputs.

In the following discussion we provide the background information on other dynamic analysis approaches used for BO detection in literature.

2.2.2.1 Anomaly detection

Dynamic taint analysis has been used in run-time detection approaches in literature for handling BO exploits. These techniques [71, 72, 74, 75] treat external input as '*tainted'* and instrument the program such that result of any operation using '*tainted'* values is also set as '*tainted*'. When a '*tainted*' value is used in jump targets or control transfer instructions such as

function return address or program's function pointer, the safety policy assertion is determined to be violated and the attack is detected. These approaches instrument x86 executables either by binary re-writing or hardware modification without the need for source code to intercept BO attacks.

Source code based approaches [70, 76] implement run-time bounds checking by maintaining base address and limit of variables in stack, heap and global memory areas. During run-time when a pointer is dereferenced, its intended reference object is checked to see if it is in the same storage region as the variable from which it was derived to check for out-of-bounds accesses. Both these approaches were implemented as GCC compiler extensions.

2.2.2.2 Test input generation

Generating bug revealing test inputs from a large input space is difficult. Some binary buffer overflow test generation approaches use dynamic analysis to extract information pertaining binary to help in automated test input generation [22, 45]. Symbolic execution techniques are commonly used in automated BO test input generation. In these approaches, inputs to the program are represented as symbolic values. Path-based analysis is used to propagate expressions derived from symbolically executing each instruction in the program path. After executing a path, the path constraints involving input variables are solved using a constraint solver to get the set of possible inputs that can explore the path. If the path contains a bug that can be revealed by the generated test inputs, then the bug is detected.

Static analysis of binaries is difficult because much of the semantic and type information is lost during compilation. Therefore dynamic analysis can help in extracting information pertaining executables that cannot otherwise be obtained from static analysis [45, 77]. Hence binary test generation solutions use hybrid static-dynamic analysis to couple information obtained from the program analysis for better coverage and accuracy. In [45], dynamic analysis is used to resolve indirect jumps in the binary code and build a visibly pushdown automaton (VPA) reflecting control flow of the program. The model is later used for generating test inputs using symbolic execution. Loop-extended symbolic execution (LESE) [22] is another such dynamic analysis based binary test generation proposal that uses symbolic expressions computed from concrete execution to generalize dependency between program input and its effects on loops that were executed during the run.

It can be observed from the above discussion that symbolic execution approaches are computationally intensive and often suffer from path-explosion problems. Using dynamic analysis

to traverse program paths for information collection needs much optimization efforts and seed-tests to drive program control along various paths, which is an expensive proposition.

2.3 Data Mining

According to Han *et al* [78] data mining refers to extracting or mining knowledge from large amounts of data. The data to be mined could be stored in databases, data warehouses, files or other information repositories. Typical applications of data mining include pattern discovery, class description, identifying associations in data, similarity analysis and outlier analysis. By mining interesting patterns from existing datasets we can check for their presence in new and evolving datasets. This feature of data mining has been widely used in software quality analysis approaches such as defect and vulnerability prediction. In this thesis, we mine defensive countermeasures applied in source code and disassembled binary code to mine BO bugs. So in our context, the source code or the disassembled binary code represents the data whereas the BO vulnerability information is the knowledge we wish to extract from the data.

The main steps involved in the knowledge discovery process of data mining can be listed as:

1. Data pre-processing
2. Data analysis
3. Evaluation

In the following discussion, we present these techniques in detail.

2.3.1 Data Pre-processing

In this thesis, we used static program analysis techniques to extract program properties from source as well as disassembled binary code. Although we did not use pre-processing techniques in our thesis, we present here the techniques that are usually employed to deal with low-quality data. In most real-world applications, the extracted data could be incomplete, noisy or of low-quality. As the quality of the data determines the quality of mining results, data is generally pre-processed so as to improve the efficiency of mining process.

Some of typical pre-processing techniques applied are data cleaning, data integration, data transformation and data reduction [78]. Data to be mined can have unusual or erroneous values. The values for certain attributes could also be missing or might have also been defined differently across the data sources. Data cleaning is therefore applied on such low-quality data to handle noise

and inconsistencies. Data integration is the process of integrating data obtained from multiple sources.

Data transformation refers to transforming the data to make it conducive for mining effectively. One of the most-commonly employed data transformation technique is *normalization*. This is used when the range of values to be represented is huge. Therefore, data is scaled such that it falls within a specified range (e.g., [0,1]). This helps in preventing attributes with large values from out-weighing that with small ranges. Since the data is standardized it allows for efficient mapping of relationships between the attributes and could help improving the learning speed and mining abilities. *Normalization* is suitable when using distance-based mining methods such as clustering or nearest neighbor classification schemes. We use only classification algorithms in our proposed approaches and our distance-based classifier algorithms are able to handle the range of values in our datasets. Therefore we did not use any data transformation techniques in our thesis.

Data reduction techniques reduce the data such that the reduced representation produces same or almost same analysis results [78]. Our data size is not unusually huge to warrant the usage of data reduction techniques. Therefore although we have not used data reduction approaches to reduce the data size in our approaches, we were interested in knowing which attributes are most effective indicators in BO auditing. Hence we applied attribute subset selection technique, namely information gain, to find out the most effective attributes for BO auditing. Apart from such dimensionality reduction methods, numerosity reduction and data compression are other data reduction techniques typically used to reduce dataset representation.

2.3.2 Data Analysis

Analyzing data for knowledge extraction is the prime focus of the data mining process. After the raw data is pre-processed, the cleaned and consistent data is analyzed to mine knowledge for decision-making process. Classification, cluster analysis, association rule mining and graph mining are some of the data analysis techniques used widely. In this thesis, we use classification methods to build static and hybrid vulnerability auditing models.

Classification is a data analysis technique which predicts the class of a data instance using knowledge learned from data instances with known class information. As the class label of training data instance is provided to the learning algorithms, these forms of learning approaches are known as *supervised learning methods*. Data analysis techniques where the *class* label of training instance is not known are called *unsupervised learning methods* (e.g., clustering). *Supervised learning methods* are highly effective in detecting known patterns whereas

unsupervised learning helps in detecting unknown and interesting patterns. Since BO vulnerability is caused by lack of bounds checking, *supervised learning methods* that leverage on identifying patterns depicting presence of such defensive mechanisms can be useful in its auditing. Therefore we applied classification techniques in this thesis.

Data classification consists of two steps, namely *training* and *testing*. In the first step, data consisting of training samples from all pre-determined classes is given as input to the classification algorithm such as Naïve Bayes, decision tree or neural network. This is known as the training dataset. Each instance in the training dataset consists of an *n*-dimension attribute vector depicting the *features* describing the *classes* (e.g., lines of code, code complexity metrics which are used for software defect prediction). The attribute vector also contains an attribute depicting the class the training instance belongs to (e.g., vulnerable or non-vulnerable in our case). This attribute is called the class label attribute. Class label attribute is categorical in nature, discrete-valued and unordered. The classification algorithm then builds a model describing the classes in the training dataset by mapping input features to class labels. Depending on the classification algorithm used, the mapping can be represented as trees, rules or formulae.

In the second step, the model thus built from known class information in training dataset is used to predict the class labels of instances in test data. The performance of prediction models is estimated by comparing the predicted class label of the test instance with its actual class label. If the accuracy of the classifier is acceptable, then, in future, the models can be used to predict class labels of data instances for which the class information is not known. By using the prediction models built from extracting vulnerability information from known source code and binary executables, we can mine vulnerabilities is unknown or newly-developed programs.

2.3.3 Classifiers

In this thesis we used various classifiers to predict BO vulnerabilities by mining static code attributes obtained from source code and x86 executables. In the following discussion, we briefly outline the details of the machine learning approaches used in the study.

2.3.3.1 Naïve Bayes (NB)

Naïve Bayes is a statistical classifier based on Bayes theorem which assumes strict independence between the features. Given the training data, it estimates the posterior probability for each of the possible classes and assigns a given test instance to the class with highest calculated posterior probability. Bayes theorem is given by [78] as:

$$P(H|X) = \frac{P(X|H)P(H)}{P(X)}$$

Considering the BO vulnerability prediction problem, H is the hypothesis that an instance is vulnerable (or non-vulnerable), P(H|X) reflects the probability that X is vulnerable (or non-vulnerable) given the attribute value information of X. P(H) is the prior probability that any given instance is vulnerable (or non-vulnerable). P(X) is the prior probability that an instance X has same attribute values as that selected from set of training instances. P(X|H) is the posterior probability (or likelihood) that a instance with same attribute value information as X is vulnerable (or non-vulnerable). Using the theorem, probability of X belonging to each of the two classes (i.e., vulnerable and non-vulnerable) is calculated. Subsequently instance X is classified as belonging to the class with highest calculated probability. Bayesian classifiers were reported to achieve a performance comparable to decision trees and neural network classifiers. They are also said to have exhibited high accuracy and speed when applied to large databases [78].

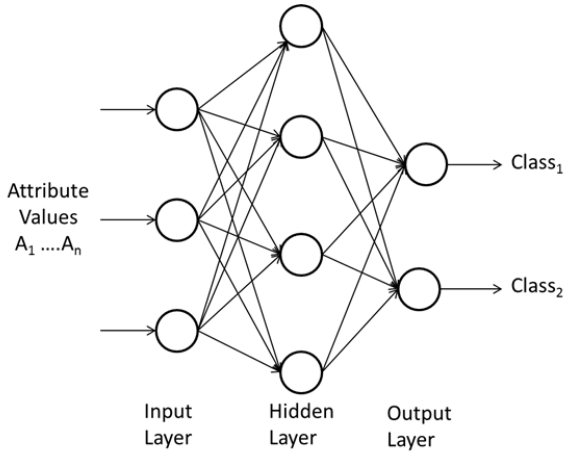
2.3.3.2 Bayes Net (BN)

Bayesian networks represent probabilistic distributions in a graphical manner. While Naïve Bayes classifiers assume that effect of an attribute/feature is independent of other attributes/features, Bayesian networks allow representation of dependencies among subsets of variables. The network is defined using a directed acyclic graph and a set of conditional probability tables. The nodes in the graph may represent the actual attributes or the hidden variables believed to form relationship [78] while the edges between the nodes represent probabilistic dependence among the respective variables. Although at times the network topology may be specified by an expert, several learning algorithms exist that learn the structure from the training data. They typically involve a function for evaluating the network based on the data and a method for searching through possible network structures. Once the network topology is determined, the next step is to calculate the conditional probability distributions of each of the variables in the graph given its parents. Inference queries can then be evaluated to determine class labels for classification purposes. Due to their ability to represent causal relationships graphically using probabilistic evaluations, Bayesian networks have been successfully applied in various domains such as medical diagnosis, bioinformatics, risk analysis etc.

2.3.3.3 Multi-layer Perceptron (MLP)

Multi-layer Perceptron is an artificial feedforward neural network which uses backpropagation learning algorithm to train the neural network. It is inspired by studies on biological neural

network structure which develops through learning to adapt to the environment. A MLP classifier consists of an input layer, one or more hidden layers and an output layer. Each layer is made up of units. The units in the input layer are fed with the attribute values of training instances. The nodes in the input layer are connected to those in hidden layer. The outputs of the hidden layer are either fed as inputs to another hidden layer or to the output layer. The connection between the units in the network have weights associated it.



Each neuron unit in the network aggregates and computes the weighted sum of its inputs. It then performs thresholding operation before propagating the data along the network layers to predict the class label that is emitted through the output layer. The backpropagation learning algorithm compares the network's prediction with that of the actual target value for the training data instances and adjusts the network weights to minimize the mean squared error between the predicted and target values. MLP classifier iteratively processes the training data and stops training when either the change in weights in previous iteration are smaller than specified threshold, or if the percentage of misclassified tuples is below threshold or if specified number of iterations have been completed. Due to their adaptive learning, fault tolerance and effectiveness in learning from training data, MLP classifiers were successfully employed on a wide variety of real-world data.

2.3.3.4 Simple Logistic (SL)

Simple logistic is a classifier for building linear logistic regression models. Linear regression models express a numeric class or outcome as a linear combination of attributes with predetermined weights [79] as follows:

$$\boldsymbol{x = w_0 + w_1 a_1 + w_2 a_2 + \cdots + w_k a_k}$$

where x is the class, $a_1, a_2, \ldots a_k$ are the attribute values and $w_0, w_1, \ldots w_k$ are the weights. Training data is used to calculate the weights by minimizing the sum of squares of the differences between predicted and actual values. Linear regression models may suffer when data used to predict exhibits non–linearity. Logistic regression transforms the original target variable using logit transformation by replacing $\boldsymbol{Pr[1|a_1,a_2,\ldots a_k]}$ with $\boldsymbol{log(Pr[1|a_1,a_2,\ldots a_k])/(1- Pr[1|a_1,a_2,\ldots a_k])}$. The target variable thus transformed is approximated using linear function like in linear regression as follows:

$$\boldsymbol{Pr[1|a_1,a_2,\ldots a_k] = 1/(1+exp(-w_0-w_1a_1-\ldots-w_ka_k))}$$

Instead of finding weights that minimize squared error as in linear regression, logistic regression which is an iterative algorithm uses training data to find weights that maximize the log-likelihood. Simple Logistic classifier is a decision tree structure with logistic regression functions at the leaves of the decision tree instead of a class label as in a conventional decision tree. The classifier uses attribute selection to include the most relevant attributes in the logistic regression models by imposing an error criterion that calculates the least error upon inclusion of the attribute. By combining the ability of logistic regression models to capture non-linear data patterns and the advantages of tree induction to fit a model to data, Simple Logistic classifier was reported to produce accurate and compact classifiers [80].

2.3.3.5 Sequential Minimum Optimization (SMO)

Support vector machine (SVM) uses non-linear mapping to transform original training data into a higher dimension to search for maximum margin hyperplane that can separate the class tuples in the data with maximum separation between the classes. The training instances of the classes that are on this hyperplane are called the support vectors and associated margin refers to the largest separation between the classes [78]. SVM algorithm finds this hyperplane and support vectors by formulating and solving this as a constrained convex quadratic optimization problem. For larger data with many training samples, solving the constrained convex quadratic optimization problem poses difficulties. SMO algorithm proposed by Platt [81] breaks the large quadratic programming (QP) problem into a series of smallest possible QP problems to overcome this issue. Smallest possible optimization problem in SVM QP problem involves two Lagrange multipliers. SMO algorithm proposes approach for choosing the multipliers to optimize and methods to solve for them. The breaking down of large QP problem and subsequent solving of the smaller QP problems speeds up the training process along with improving the scalability of the algorithm.

2.3.3.6 Evolutionary and Genetic Algorithms (EA and GA)

Evolutionary and Genetic Algorithms were used by test generation approach in Chapter 4 that we compared with our proposed rule-based test generation methodology. We therefore discuss the algorithms here briefly to give an idea of its working. Genetic and other related evolutionary algorithms use natural evolution as an optimization mechanism for solving engineering problems. They start with an initial population of individuals where each individual is a feasible solution to the problem. Each individual is characterized by a fitness function such that higher fitness implies better solution. This represents the survival of the fittest notion in evolution. Based on the fitness, parents for next generation are selected to generate offspring for the new generation. The offspring has combination of the properties of its parents. Crossover and mutation operations are used to create the offspring. In crossover operation the properties of parents are randomly recombined to generate offspring whereas mutation involves generating offspring from single parent by randomly changing some of the properties. The algorithm converges when the individuals in the population satisfy pre-specified fitness constraints. Performance of these algorithms is determined by accurate identification of solution parameters, and fitness evaluation. They are effective for finding optimal solutions for problems whose search space is large and which cannot be solved under other techniques. In the field of data mining, genetic algorithms have been used for classification purposes [78].

2.3.4 Evaluation

As described in the previous discussion once a prediction classifier model is built, the next step is to evaluate the accuracy and effectiveness of the model and its suitability in practical scenarios. Typically, model evaluation consists of data sampling and performance measure calculation by comparing actual and predicted class labels.

2.3.4.1 Performance measures

Existing works in software defect and vulnerability prediction studies [48, 51, 56, 82-84], generally use a confusion matrix (shown in Fig. 2-4) to assess the performance of defect or vulnerability prediction models built. From the confusion matrix, the following measures are calculated to assess model's performance:

- Probability of detection or recall (pd) = $tp/(tp + fn)$
- Probability of false alarm (pf) = $fp/(fp + tn)$
- Precision (pr) = $tp/(tp + fp)$

- Accuracy $(acc) = (tp + tn)/(tp + fp + fn + tn)$

	Classified Vulnerable	Classified Not-Vulnerable
Actual Vulnerable	True Positive (tp)	False Negative (fn)
Actual Not-Vulnerable	False Positive (fp)	True Negative (tn)

Figure 2-4. Confusion matrix for vulnerability prediction

pd measures how good the model is in finding actual vulnerable statements. *pr* measures how many of those classified as vulnerable by the prediction model are actual vulnerable statements. *pf* is a measure of false alarm predictions by the model. *pd* should be close to 1 so that all the vulnerable statements in the program are identified and *pf* should be close to 0 so that no false warnings are raised that would entail unnecessary auditing costs. *acc* measures the number of correctly classified instances as a percentage of total instances tested. In this thesis, we evaluate the performance of our prediction based auditing models using these measures.

2.3.4.2 Data sampling

Data sampling refers to the construction of training and testing datasets for model evaluation. Cross validation, random split-sampling, and bootstrapping are common data sampling techniques used for evaluating prediction models. In our study, we shall use *k-fold stratified cross validation* method which is generally used for evaluating classifiers. In *k*-fold cross-validation, the dataset is randomly divided into *k* partitions. The classifier prediction model is trained on *k*-1 partitions and is tested on the remaining partition. This process is repeated *k* times with every partition in the dataset being tested once, iterating until every partition has been served as the test set once. The evaluation results from *k* iterations are combined to get a final result. In a *stratified k-fold cross-validation*, the partitions are stratified such that the class distribution of the instances in each partition is approximately the same as that in initial data [85]. This approach has also been used by many software defect and vulnerability prediction studies [48, 51, 56, 83, 84].

Chapter 3

RELATED WORK

There is an extensive body of work on techniques for mitigating BO vulnerabilities in literature. Existing methods can be broadly classified into five types—defensive coding practices, vulnerability testing techniques, vulnerability prediction techniques, vulnerability detection techniques and runtime attack prevention techniques.

In this thesis, we have presented three types of techniques for mitigating BO threats—vulnerability testing, source-code based hybrid vulnerability auditing, and vulnerability prediction from binary executables. Hence, our works are related to the works from the above five categories. In this chapter, we compare the proposed techniques with these closely related works. As outlined in our survey on BO defensive techniques [58], we believe that BO threats can only be defended by an integrated approach combining different techniques at different stages of the software development cycle. Hence, the proposed approaches are intended not to replace, but to complement, the existing ones during various software development stages.

3.1 Defensive Coding Practices

Writing secure code by applying suitable defensive coding practices is the best solution for eliminating vulnerabilities. Since BO bugs are typically exploited due to absence of or inadequate input validation and bounds-checking mechanisms and usage of unsafe C library functions, enforcing defensive coding practices with security in mind is the best solution for countering BO threats. The onus of adapting and incorporating these practices rests with the development teams. In [86] the authors proposed a checklist, intended to be of use by developers and testers during software development for mitigating BO attacks. Shahriar et al [87] propose a rule-based approach for identifying and patching vulnerable code. The proposed rules can also be used as defensive coding practices for building secure software. The following measures were suggested in the literature:

Choice of Programming Language: Applications developed in C/C++ are greatly prone to BO bugs. C/C++ languages offer great degree of flexibility and high run-time performance. Hence

many commercial applications are still written in C/C++, but it leaves the onus of incorporating security in the hands of the developers. This is because C/C++ does not have any built-in security mechanisms for buffer bounds checking. Choosing programming language that performs runtime bounds checking is the best solution.

Using safe C/C++ dialects helps in alleviating the problem by remaining as close as possible to C/C++. While some safe dialects need only minimal intervention to compile the programs in the new dialect, others may need substantial modification to the programs. Cyclone [6] is a safe dialect of C that helps in preventing BO attacks by using dynamic bounds checks. While it can be deployed for projects to be developed it cannot be used for legacy code unless by code transformation or modification which involves high cost. CCured [5] extends C type system with explicit pointer types. It translates C programs into CCured and establishes all pointers either as safe, sequence or dynamic through a constraint-based type-inference algorithm and uses run-time checks when static analysis is not enough to determine the safety. It requires changes to source code but lesser than that of Cyclone and requires extra storage for sequential and dynamic pointers.

Secure C [88] is a source-to-source translator that uses shadow stack by allocating stack memory area dynamically to store stack arrays for handling frame activation record exploits. Apart from protecting some code pointers it also can deal with arrays declared globally or on heap but it cannot offer protection against certain pointer-overwrite based attacks. SMART C [89], another source-to-source translator, is an extension to C programming language which allows users to specify semantic macros for transforming source-level constructs. By specifying macro for array declaration statement construct to transform the declaration of array into a *struct* containing declared array as *struct* element along with a contiguous *Boolean* type element, overflows to array can be detected as they overwrite *Boolean* elements. Pointer and memory access detection techniques in literature [6] use the concept of "fat pointer" to maintain and update pointers to arrays with respective array bounds information. While solution like SMART C eases the process of such transformations by usage of semantic macros, the overheads for maintaining, updating and run-time detection pose a huge limitation in adapting them.

Safe versions of C library functions: Standard C run-time library functions like *strcpy*, *strcat* are unsafe because they do not perform any bounds checking before copying to the buffers [90]. While library function equivalents specifying the number of elements to be copied like *strncpy*, *strncat* exist, their usage too should be treated carefully as they can be used to copy part of source strings to destination buffer and do not specifically null-terminate the destination buffers. Using

safer versions of these functions such as *strcpy_s* and *strcat_s* is another good defensive coding practice. Such safe string library functions include libmib [91], which provides string library functions which automatically reallocate the destination strings as and when necessary, and Microsoft's BO handling specific StrSafe [92] libraries. Using *std::string* class is suggested for applications developed in C++. Proper usage of such library functions is effective in thwarting overflows related to unsafe string library function usage.

Input Validation: BO exploits are a subset of input manipulation vulnerabilities. Auditing the input thoroughly so that it meets the required specifications is an important defensive technique. In fact, the need for adequate input validation goes beyond security defense. Input validation is vital for the general needs in most systems for:

(1) Security enforcement: In addition to access control mechanisms, incorporation of adequate input validation ensures that any illegal input is rejected right away before causing any damage by the attackers on the software system and

(2) Data integrity enforcement: Though it is not possible to automatically check the accuracy of input data, input validation helps to filter out many input errors, thereby playing a big role in enforcing data integrity in the system according to requirements and business rules.

Bounds-checking: Lack of bounds checking operations before filling buffers results in the BO bugs. Performing bounds checking before every buffer write operation completely solves the problem.

While usage of these recommended practices is the best way to handle BO, it can be easily deduced that their incorporation and adoption is error-prone, labor-intensive and is difficult to be rigorous and comprehensive in-practice. To alleviate these problems automated tools and solutions were proposed as compiler options or extensions to help in the process of including run-time bounds checks. One of the earliest such solutions in literature was by Jones and Kelly [70] who introduced the concept out-of-bounds reference object for a pointer. In contrast to methods altering pointer representation in the program using a "fat pointer", the approach by Jones and Kelly defines a reference object of pointer as the buffer that the pointer intends to point-to. Under the assumption that result of any operation on the pointer must still leave it to point to same referential object, they use the information of the in-bounds pointer, reference object information and the result of the pointer operation to determine if the operation is still in-bounds. While the approach was effective, it reportedly slows down the system by 5 to 6 times and cannot handle cases where out-of-bounds pointer to object is stored to later retrieve in-bounds address as it violates the assumptions. To remediate this, CRED [76] allowed manipulations of out-of-bounds addresses by

maintaining a special out-of-bounds objected created for that value. Although this helps in handling the false alarm cases in [70], the overheads are still high at 11 to 12 times slow down for some of the tested benchmarks due to improvisations. Instead of maintaining a single big data structure for recording the referential object details of heap allocated program variables, Dhurjati and Adve [93] use the memory partitioning by Automatic Pool Allocation to maintain such structure for each created pool thereby reducing the search complexity time for reducing overheads. The drawback of this approach is that for non-heap data like global and stack objects the information needs to be entered manually and the approach may allow bound violations on pointers obtained from integer to pointer casting and unchecked code resulting from calls to external functions. The overheads of the approach are considerably lesser than that of [70, 76], although it records around 69% additional overhead for one of the test subjects.

Table 3-1. Summary of Defensive Coding Methodologies

Proposal	Description	Remarks
Pandey et al [86]	Proposes checklist to be of use by developers and testers during software development.	Intensive manual work and security training requirement.
Shahriar et al [87]	Proposes rules for identifying and patching vulnerable code.	Intensive manual work requirement for identifying and patching.
Cyclone [6]	Safe dialect of C that helps in preventing BO attacks by using dynamic bounds checks.	Manual work and dialect learning requirement. Cannot be used for legacy code.
CCured [5]	Extends C type system with explicit pointer types and establishes all pointers as safe, sequence or dynamic through a constraint-based type-inference algorithm and run-time checks.	Requires changes to source code and needs extra storage for certain pointer types
Secure C [88]	Source-to-source translator that uses shadow stack by allocating stack memory area dynamically to store stack arrays for handling frame activation record exploits.	Cannot offer protection against certain pointer-overwrite based attacks
SMART C [89]	Source-to-source translator which allows users to specify semantic macros for transforming source-level constructs.	Manual specification requirement
libmib [91] , StrSafe [92]	Safer versions of vulnerable C run-time library functions.	Manual incorporation and security training requirement.
Jones and Kelly [70]	Propose an out-of-bounds reference object for a pointer to detect overflows.	Slows down the system by 5 to 6 times
CRED [76]	Improves over [70] and allows manipulations of out-of-bounds addresses by maintaining a special out-of-bounds objected created for that value.	Slows down the system by 11 to 12 times on some cases
Dhurjati and Adve [93]	Use Automatic Pool Allocation for recording the referential object details of heap allocated program variables for reducing search complexity time and overheads.	Manual intervention needed for non-heap data information entry
Majority of approaches proposed in the literature are labor-intensive. Some require changes to the source code thereby incurring high costs while few others slow down the system considerably thus limiting their practical usage.		

Table 3-1 summarizes the defensive coding methodologies proposed in literature. It can be seen from the above discussion that application of defensive coding practices is important as they form the first line of protection in mitigating threats. However, although defensive measures such as

safe dialects, libraries and development frameworks are helpful in developing more secure programs, the drawback of such solutions is that developers need to learn and master these new development paradigms or environments. Despite the knowledge of security threats involved, BO bugs may still be present in the program due to the labor-intensive and error-prone nature of the methodology. Moreover, application of defensive coding may not always be feasible in practice as they are applicable only when actually writing (or planning to write) the code. In most cases it may require manual intervention and substantial modifications for rewriting the existing code. Their application is not possible in situations when the source code is unavailable or must remain the same. Therefore, our solutions would complement them in various stages of software development to test for the adequacy of defensive coding implementation in the programs.

3.2 Vulnerability Testing

Testing the programs for vulnerabilities helps in identifying and fixing them before deployment. In vulnerability testing, the program under test is executed with generated test inputs to observe the program output or behavior for detecting vulnerabilities. Since the programs are executed with the generated test inputs the techniques presented in the following discussion fall under dynamic program analysis. Testing approaches for detecting BO bugs can be chiefly classified into three categories, namely: symbolic evaluation, fault injection and evolutionary algorithms.

Typically, symbolic evaluation approaches for test generation maintain program states represented as logical expressions wherein symbolic values are used to represent external inputs. The expressions are updated as per code statement semantics along the program path. In case of a branch statement, the branch predicate is appended to the path-condition predicate. Constraint solvers are subsequently used to solve the path-condition constraints involving symbolic values to generate test inputs triggering vulnerabilities along the program path. Using this technique, bug is revealed if solution satisfying path-condition to sink exists. CUTE [94], EXE [21] are examples of such tools. Such path based test generation solutions typically suffer from path explosion problem as the size and complexity of the code increases.

Adaptive test generation techniques [95, 96] use feedback from program's responses to previous inputs to guide the test generation along other paths or interpret the input's effects for further fine-tuning to detect bugs. The solution in [96] uses static analysis information to identify possible delimiting characters that are parsed by the tester into parameters and mutated based on responses to previous inputs to generate new test cases for BO detection. It can be deduced that

this involves considerable manual efforts to alter and adapt the test cases from the feedback received from prior responses to detect bugs. DART [95], a directed automated testing technique, starts with a random input to gather constraints on input from predicate checks from the run to generate inputs that force execution on other paths in the subsequent run. While this may help in increasing the code coverage such technique alone may not be effective in detecting vulnerabilities such as buffer overruns.

Symbolic execution test case generation techniques for buffer overflows may use full-symbolic execution [21] by representing each character in input string with a symbolic value or may use partial symbolic execution [23] and represent only a part of input string by giving such symbolic values. Full-symbolic execution stresses the constraint solver as it introduces many symbolic values (since each character in input is represented by symbolic value) resulting in huge memory consumption in having to keeping track of all of them and solving large path constraints involving hefty number of such symbolic values. This may result in tool getting stuck in generating input not relevant to the error.

Splat [23] uses partial-symbolic representation along with associated symbolic length to remedy this. The premise of Splat for using partial-symbolic representation is that for many buffer overflows, most of the actual content of the buffer is not relevant, except for length of string and some small prefix of the string. While we agree with the assumption that length of string is more relevant, one cannot know before-hand the prefix size to be fixed. Splat authors suggest starting with a short prefix and gradually increasing it as the testing budget allows. This requires manual intervention for fixing initial prefix size and for optimizing it if initial size is not adequate.

Saxena et al [22] proposed loop-extended symbolic execution (LESE) for generating test cases from binary executables. Their approach introduces new symbolic variables for representing trip counts for each loop, links them to variables representing program input and combines these symbolic constraints with condition for violation of security policy for vulnerability checking of applications. Test input generation is done by reversing branch conditions with loop-dependent values and vulnerability discovery uses additional step of checking security predicate at each dangerous operation. It identified buffer overflows in real-world programs by sending an initial benign input and used that execution trace with grammar to discover vulnerability. One seemingly obvious limitation of LESE is that the initial concrete input used for dynamic analysis was close to the pattern of the exploit input. The authors suggest that if the tool is unable to generate exploit input from such nearer input it cannot generate the exploit input from an altogether unrelated

input. This approach is suitable for discovering vulnerabilities due to incorrect input-processing using loops and may not be suited for other purposes.

In [45] the authors use a combination of static dynamic analysis to guide binary test generation by prioritizing the paths to be explored. This is done by using the static analysis stage information to assign weights to represent the inter-procedural control flow graph of the program. In the first stage of the three stage process, dynamic analysis is used to identify indirect control flow transfers for building the inter-procedural control flow graph which is represented by visibly pushdown automaton (VPA). In the second stage, static analysis is performed on the VPA to identify possible bugs and to collect information pertaining buffer sizes and backward dataflow slices of sink statements. In the third stage this information is used to assign weights to VPA edges to prioritize paths to explore. The approach could generate inputs for all but one of the tested programs while the false positive rate of its static analysis component was reported to be 72%. This is attributed in the paper to using dynamic analysis and brute-force approach to build the VPA and authors suggest using more efficient approaches like function summaries to alleviate this.

Fault injection approaches corrupt input data and variables to generate test cases [97, 98]. These solutions use malformed data and large strings i.e., greater than the buffer length to trigger BO vulnerabilities. For integer variables extreme values in the integer range are used to trigger integer overflow vulnerabilities. Ghosh and Connor [19] propose algorithms for selecting susceptible buffers, creating buffer overruns and for analyzing the application for susceptibility from its source code. They select locations calling unsafe library functions on local buffers and non-library functions reading/copying user input as susceptible locations. The return address to overwrite is calculated and an attack string is prepared. This is repeated for each candidate location. They designed FIST tool to automate fault injection process on selected locations. Mutation based approaches are used to evaluate the quality of the test suite. In [18] the authors propose library functions, buffer size and null character assignment removal mutations for evaluating test suite's effectiveness for BO detection. Combinatorial black-box testing approach in [20] uses manual support to identify the input parameters affecting a statement and produces test inputs with all combinations of parameters thereby simulating the methods adopted by hackers. They suggest that using static analysis to identify the parameters could help in fully automating the test process.

Genetic algorithms [16] or variants of evolutionary algorithms [17] typically involve a pool of initial population (random set of candidate solutions), fitness function to measure the individual candidate's ability to provide desired solution, selection mechanism to select candidate solutions

for next generation (i.e., iteration) and approach to generate new individuals from older ones. Genetic algorithm solutions are effective when the search space is large with no available mathematical analysis. Their main drawback is that they are susceptible to misguided fitness function evaluation and subsequent arbitrariness introduced by the candidate selection process. Table 3-2 summarizes and reviews the vulnerability testing methodologies discussed.

Table 3-2. Summary of Vulnerability Testing Methodologies

Proposal	Description	Remarks
EXE [21]	Uses symbolic evaluation to maintain program states represented as logical expressions wherein symbolic values are used to represent external inputs. Path-condition constraints involving symbolic values are solved using constraint solver to generate test inputs triggering vulnerabilities along the program paths.	Full symbolic execution stresses the solver and leads to huge memory consumption. May suffer from path and state explosion issues.
Splat [23]	Uses partial-symbolic representation along with associated symbolic length by limiting the representation to some small prefix of the string.	Requires manual intervention for fixing initial prefix size and for optimizing it if initial size is not adequate.
LESE [22]	Introduces new symbolic variables for representing trip counts for each loop, links them to variables representing program input and combines these symbolic constraints with condition for security policy violation to generate test inputs.	Suitable for discovering vulnerabilities due to incorrect input-processing using loops and may not be suited for other purposes.
DART [95]	Starting with a random input the tool gathers constraints on input from predicate checks from the previous run to generate inputs that force execution on other paths in the subsequent run.	Good for increasing the code coverage but may not be effective for bug detection.
Liang et al [96]	Uses static analysis information to identify possible delimiting characters which are parsed by the tester into parameters and mutated based on responses to previous inputs to generate new test cases.	Intensive manual work requirement.
Babi et al [45]	Uses static dynamic analysis to build VPA and assigns weights to prioritize the edges to explore for guiding binary test generation.	Effectively generated inputs for all but one tested program although static component reported high false positive rate.
Fault Injection [97, 98]	Use malformed data to trigger bugs.	Focus is on violating protocol grammar. Other type of bugs may be missed.
Wenhua et al [20]	Uses manual support to identify the input parameters affecting a statement and produces test inputs with all combinations of parameters simulating the methods adopted by hackers.	Intensive manual work requirement.
Evolutionary Algorithms [16, 17]	Uses genetic and evolutionary algorithms to generate test inputs.	Good when search space is large. Needs domain expertise for setting up the problem. Susceptible to misguided fitness function and arbitrariness in candidate selection process.
Symbolic execution techniques suffer from path and state explosion problems and high space requirements. Fault injection mechanisms generate malformed data to trigger bugs and may fail to trigger other types of bugs involving non-trivial string constraints. Some proposals come with intensive manual efforts to operate the tools which are not suitable for practical purposes.		

It can be seen from the above discussion that the major advantage of test generation approaches is that there are no false positives, since the inputs help in revealing actual bugs. As the vulnerability is found along with the input to trigger it, the developers can use this information to fix the found bug straight away. Major challenges in test generation are that for symbolic execution based approaches, their effectiveness is typically determined by their ability to model underlying statement semantics effectively as well as their ability in handling path and state explosion issues. Fault injection mechanisms can help trigger bugs with perturbed and often extreme input values but they cannot handle non-trivial string constraints. Genetic algorithms and its variants need a lot of iterations to learn simple input constraints. Usage of static analysis information can help in minimizing their search space and in turn reducing the number of evolutions needed for generating bug inducing inputs.

Our test generation mechanisms in Chapter 4 and Chapter 5 use static analysis information as well as fault injection techniques to trigger BO inducing bugs. While Chapter 4 relies mostly on control and data dependency information for generating string inputs, the test generation methodology in hybrid auditing approach of Chapter 5 partitions input based on type (i.e., perturbs the input by injecting faults according to its type) as well as nature of input validation. Since the approaches do not try to emulate and solve for feasibility of all program paths they overcome scalability issues associated symbolic execution approaches. The solutions are mostly automated and do not need any further optimization as in partial symbolic execution based approaches. Therefore our solutions provide a light-weight, cheaper and effective means for test generation. Another drawback of test generation schemes is that while proven bugs can be readily fixed, it is not possible to know about the vulnerability status of statements not shown to be vulnerable. Our hybrid auditing approach in Chapter 5 overcomes this by predicting such statements instead of them being taken as safe.

3.3 Data Mining Techniques

Data mining models were used in our supervised vulnerability prediction techniques in Chapter 5 and Chapter 6. These models are similar to those built in software defect prediction studies [50-52, 99, 100] for software quality assurance. The results of the defect prediction can be used by quality assurance teams to allocate their resources effectively by concentrating on code segments predicted to be vulnerable. Due to the increasing complexity of software, such prediction based methodologies provide an important alternative and complimentary means of mitigating vulnerabilities at various stages of software development. In the following discussion we first present the existing works on software defect prediction followed by vulnerability prediction.

Defect prediction: Defect prediction studies in literature use machine learning techniques for predicting software defects. Organizations typically use the results of the studies to determine the software quality and estimate and direct the maintenance efforts. Due to the similarities between the approaches, vulnerability prediction can be thought of as a specialization of defect prediction methodology. One of the earliest studies on defect prediction is by Akiyama [101] who built a simple model using LOC. Later, comprehensive metrics like McCabe's cyclomatic complexity [53] and Halstead's complexity metrics [54] were proposed which were used as predictors of program defects [51, 102]. To evaluate the predictors, the studies use performance measures such as recall, precision, probability of false alarm and accuracy [51, 99]. Since these metrics can be easily obtained from specification, design and implementation, such static code attributes are widely used in predicting software defects.

In [51] a major study was performed to compare the performances of various classifier algorithms using public domain datasets. The results from it can be summarized as yielding predictors with a mean recall of 71% and probability of false alarm at 25%. Since such attributes are easy to collect and use, the authors suggest that they are effective and practical means for prioritizing resource-bound code inspection.

While Fenton and Neil [102] and Shepperd and Ince [103] argued against the reliability of static code attributes like LOC and complexity metrics, others suggest that such metrics can only offer limited accuracy [82, 104]. Study by Misirli and Kale [105] reveals that usage of defect predictors can decrease the code inspection efforts by 72%. Since there are no common or unanimous set of attributes that is applicable to all domains, several other metric measures were proposed to improve the predictive capabilities of classifiers.

With the advent of Object-Oriented (OO) programming, class-level metrics were proposed by Chidamber and Kemerer [106] in 1994, which are widely-used by tool vendors and researchers [107] to predict defects in OO systems. Component and file-level metrics were proposed in early 2000, wherein the latter tries to capture the changes to source files at different phases of software development.

With the growing popularity of version control systems, various process-based metrics were proposed to tap into the information pertaining development history like programmer experience level, code churn, defects found in reviews etc. [108-111]. Unlike static code attributes like LOC and complexity metrics, measuring process attributes is difficult as well as inconsistent. Another limitation for process-metrics usage is that for projects that are recently being developed or ones

that have less historical data, such predictive models cannot be used. Hence cross-project defect prediction models were proposed [112].

Similar to these defect prediction studies, we proposed and applied static code attributes in building vulnerability predictors in pure static as well as hybrid BO auditing approaches in Chapter 6 and Chapter 5 respectively. We used the classifiers such as Bayes algorithms and neural networks that were ranked among the best by these studies. We also applied the same performance measures, such as *recall* and *precision*, to evaluate our BO auditing models.

Contrary to software defect prediction, in this thesis, we predict BO security vulnerabilities in applications. As such, we proposed various attributes that are applicable to the vulnerability under consideration by catering to the suggestion of the usage of domain specific attributes. The proposed attributes can be easily collected with the aid of a static analysis tool, for predicting BO bugs making it cheaper yet effective complimentary solution for mitigating vulnerabilities analogous to using complexity metrics for defect prediction.

Vulnerability prediction: Vulnerability prediction using data mining is relatively new field. In this discussion we present prediction approaches that are closely related to ours as they too predict software vulnerabilities.

Shin et al [48] performed case-studies on Mozilla Firefox and Red Hat Enterprise Linux kernel using code metrics for complexity, code churn and developer activity to predict software vulnerabilities. They achieved a recall over 70%. Their assumption was that complex code is more prone to be vulnerable than simple one. We believe that this may not hold through over all domains. Given that BO bugs are susceptible to exploits even on simple code, we feel that attributes capturing problem domain characteristics are more appropriate for predicting vulnerabilities. Since the approach uses process-metrics such as developer activity, as has been discussed above, gathering such attributes is expensive and more over the collected attribute data may not be consistent. By contrast, the attributes proposed in this thesis are static code attributes which can be easily and consistently collected.

Neuhaus et al [55] predicted vulnerable software components using software version histories and corresponding known vulnerabilities. They mapped vulnerabilities to software components based on the imports or function calls of the past vulnerable components. This was then used to predict future vulnerabilities in new components in the current codebase. Such work is different from ours because we predict specific security vulnerabilities (i.e., BO) at the statement level as compared to predicting the overall vulnerabilities at a component level of precision. Their tool reported a 45% recall and 70% precision.

Shar and Tan [84] predict SQL injection and XSS vulnerabilities in PHP-based web applications by mining code attributes representing input sanitization and input validation regimes applied on the sinks. Instead of attributes representing user-defined input sanitization mechanisms which are more appropriate safety feature for their problem domain of web application security, we used code attributes capturing bounds checking and defensive measures suitable to audit buffer overflows in our proposal. They achieved 93% recall and 11% false alarm rates in predicting SQL injection vulnerabilities and 78% recall and 6% false alarm rates for XSS vulnerabilities on different problem domains and attributes.

Gegick et al [57] used recursive partitioning of data containing outputs from static analysis tool (i.e., alerts or warnings) along with code churn and LOC as features to predict vulnerable software components. They hypothesize that count and density of alerts along with other input variables can help discriminate between vulnerable and non-vulnerable components. Although their approach uses static code attributes like ours, the attributes represent wide-ranging program fault alerts generated by the static analysis tool and subsequent code audit and inspection outcomes of the alerts before using the dataset for classification purposes. This requires manual expertise to audit the alerts whereas our attribute collection mechanism is fully automated. Brief summary of vulnerability prediction methods presented in the above discussion is given in Table 3-3.

Table 3-3. Summary of Vulnerability Prediction Methodologies

Proposal	Description	Remarks
Shin et al [48]	Proposes code metrics for complexity, code churn and developer activity to predict software vulnerabilities under the assumption that complex code is more prone to be vulnerable than simple one	Gathering process metrics is expensive and may be inconsistent. Simplicity assumption may not hold for BO bugs
Neuhaus et al [55]	Predict vulnerable software components using software version histories and corresponding known vulnerabilities by mapping vulnerabilities to software components based on the imports or function calls of the past vulnerable components	Coarse-grained analysis
Shar and Tan [84]	Predict SQL injection and XSS vulnerabilities in PHP-based web applications by mining code attributes representing input sanitization and input validation regimes applied on the sinks	Fine-grained analysis. Usage of domain specific attributes resulted in good prediction accuracy
Gegick et al [57]	Use alerts or warnings from static analysis tool along with code churn and LOC as features to predict vulnerable software components	Coarse-grained analysis, needs manual expertise to audit the alerts before using dataset for classification
Existing approaches mostly predict security vulnerabilities in general and are often coarse-grained. Coarse-grained analysis needs security auditors to delve into the code deeply to identify the vulnerable statement requiring much manual effort.		

Overall the differences between existing vulnerability prediction approaches and those proposed in this Chapter 5 and Chapter 6 of the thesis are: (1) existing approaches predict security vulnerabilities in general or web application domain vulnerabilities as in [56, 84], whereas our approaches focus on mitigating BO vulnerabilities (2) we propose novel static code attributes by

taking into consideration the defensive measures implemented in the code for thwarting BO exploits (3) our fine-grained approaches target predicting bugs at statement level as opposed to predicting vulnerable components or program files. Therefore it is not possible to provide a direct comparison between existing approaches and ours due to difference in problem domain and granularities. Despite this, it shall be seen from Chapter 5 and Chapter 6 that our prediction based approaches too achieved high recall and low false alarm rates in auditing BO vulnerabilities.

3.4 Vulnerability Detection

3.4.1 Static Analysis-based Vulnerability Detection

BO vulnerabilities can be identified by statically analyzing program source code or disassembled binary code. Symbolic execution based approaches [8, 10] generally track buffer declaration in the program and model the buffer usage in the form of constraints to raise warnings when an out-of-bound access may happen. The nature and precision of analysis affects the outcome of the detection accuracy. Annotation based techniques [13-15] allow end-users to specify the desired properties as pre- and post-conditions for potential vulnerable statement constructs to generate and assert these constraints. Apart from tools using constraint solving, lexical analysis based approaches were also proposed in literature, which tokenize the code to identify and sort potential flaws by performing simple risk evaluation on source code [113-115]. Binary lexical analyses solution also exists [116]. Such tools can mainly help flag that vulnerable library functions were used in the code.

Vinod Ganapathy et al [8] model pointers to character buffers by four constraint variables to denote the maximum and minimum number of bytes allocated and used by the buffer. They model integer variables using the variable's maximum and minimum values. They detect overflow through constraint analysis when the maximum used value is greater than the allocated minimum or allocated maximum value for the buffer. However, the approach generates many false positives because of the flow-insensitive nature of the analysis. The nature of constraint modelling also causes some actual bugs to be missed resulting in false negatives too. BOON (Buffer Overrun DetectiON) [9] is another constraint-based inference tool which converts BO detection problem into integer constraint problem. It does so by identifying the strings in the program and gathering information about their allocated size and the number of bytes currently in use. Library functions are modelled based on how they use this information. Integer constraints are formulated based on the buffer usage and are analyzed to check if the allocated size is at least as large as the maximum

inferred used length. This tool also suffers from high false positives (around 90%) due to its flow and context insensitive range analysis.

LCLint is an annotation-assisted static analysis tool [117] that programmers can use to specify pre- and post-conditions. Constraints to solve are generated from the specified pre- and post-conditions. Constraints used to describe buffer ranges include *minSet*, *maxSet*, *minRead*, and *maxRead*. When the program calls an annotated function, it checks pre- and post-conditions to ensure safe access to buffers using these buffer range constraints. Annotation based approaches [13-15] differ in nature of annotation semantics and scope of detection. Manual specification of annotations is highly labor-intensive and hard to be accurate and comprehensive. CSSV [14] and Splint [13, 118] (a successor of LCLint) reported false positive rates of 50% and 75% respectively on tested programs.

Wang et al [11] use model checking to convert BO problem into error checking problem by asserting the constraints and finding out the reaching paths to this constraint error. ARCHER [10] uses a flow-sensitive inter-procedural symbolic analysis using bottom-up approach on call-graph to detect memory access errors by capturing constraints and propagating constraints upwards in call-graph. It reported average 25% false positive rate on programs used for evaluation. It would be seen from Chapter 5 and Chapter 6 that our approaches perform considerably better than ARCHER.

While most constraint-based solutions use in-built mechanisms for identifying potential vulnerable constructs to specify, propagate and solve for constraints pertaining the buffers, demand-driven analysis lets the user to specify a query on desired locations. Marple [119] is one such demand-driven path-sensitive analyzer which categorizes program paths to enable priorities when diagnosing root causes of vulnerable paths. The analysis identifies infeasible paths and examines buffers and classifies the paths that lead to buffer access as safe/vulnerable/overflow-input-independent/don't-know by raising queries on the buffers and propagating them backward along the control flow. Marple updates control flow at nodes where it can collect information about buffers from definition and allocation statements of the buffers and the ranges/values of program variables that impact buffer accesses. The query terminates when it reaches program entry or an infeasible segment, or when information gathered during propagation resolves the query. Since buffers are manipulated by pointers, lack of points-to-analysis in Marple results in false negatives, thereby missing some actual bugs. Warnings are prioritized to help in the diagnosis, but the approach should be fine-tuned to minimize the high number of don't know warnings on some cases, which may also contain actual bugs. Since the solution is demand-driven,

it needs the programmers to speculate and specify potential vulnerabilities, upon which the tool can provide the inference. Our hybrid auditing approach in Chapter 5 can complement such tools by providing probabilistic remarks on statements that it cannot prove vulnerable, thereby easing the manual effort needed to speculate which statements to query for.

Taint analysis based approaches also were proposed to detect BO bugs. These approaches mark the external input as tainted. The input taint information is propagated in the program to see if any tainted data reaches sensitive statement constructs such as vulnerable C library functions to flag vulnerabilities. Cova et al [47] use symbolic execution to propagate taint information for detecting vulnerabilities in x86 executables. By nature, such taint dependency algorithms raise warnings as long as sensitive statements are dependent on input-related data leading to false positives. Nagy and Mancoridis [120] proposed input coverage and distance metrics which measure percentage of statements tainted by user input and distance between the function and the origin of input data respectively, in conjunction with taint dependence analysis to detect bugs. If there is check on string length along the path leading to vulnerable statements they skip the dependent sub tree statements under the assumption that size checks are enforced, and hence the sink is safe. This assumption is not adequate to prove a statement's non-vulnerability and may lead to false negatives. Taint analysis based solutions can only help end-users to identify where checks must be added or enforced to prevent input manipulation exploits. As they do not take into account the input validation and BO defensive measures that may have been implemented for the sink they generate many false positives.

Studies [121, 122] for evaluating the effectiveness of static analysis tools suggest that the nature of inference (e.g., flow, context, pointer and intra/inter procedural analysis) effects the detection outcomes of the tools giving rise to false positives as well false negative outcomes. Buffer usage constraint representation further increases the complexity as it is difficult to accurately model complex operations involving buffer writes. Symbolic evaluation and constraint solving may affect the usability of the tool in development environments due to drawbacks pertaining scalability issues, advanced technical expertise requirement and manual auditing costs for dealing with large number of false alarms. The proposed approaches in the thesis are closely related to static analysis based vulnerability detection approaches as they use static analysis information obtained from analyzing program source code or binary disassembly to audit bugs. The vulnerability predictors that shall be presented in Chapter 5 and Chapter 6 overcome the challenges of existing approaches by capturing defensive measures implemented in the code to

build predictive models learned from past vulnerability information for inferring statement's vulnerability in inexpensive and effective way.

Shahriar and Zulkernine [122] suggest using novel inference techniques along with combining multiple inference techniques to overcome these challenges. Our vulnerability prediction based source code and binary BO auditing approaches can support in the process by providing probabilistic observations of statement's vulnerabilities. Comprehensive constraint based or demand-driven solutions can then be used on statements predicted vulnerable thus making cost-effective use of resources at disposal by the development or auditing teams.

3.4.2 Hybrid Analysis-based Vulnerability Detection

Apart from static analysis only based vulnerability detection approaches, hybrid frameworks were also proposed that make use of the complementary nature of static and dynamic analysis to detect bugs. Aggarwal and Jalote [123] use static analysis to scan source code for C pointer variables suspected to be aliased. Calls to unsafe C library function calls (*strcpy*) that use these suspicious variables as arguments are marked for dynamic analysis. Metadata for the marked function calls is generated by the tool to detect overflows during dynamic analysis by using test cases. This approach only handles overflows caused by pointer aliasing that limits its applicability.

Kumar et al [124] also use hybrid analysis to detect memory errors like buffer overflows, dangling pointers and memory leaks. Binary is first dynamically analyzed to get exact control flow sequence. During the process the tool also tracks calls to dynamic memory allocation functions (e.g. *malloc*) to gather information about such dynamically allocated memory blocks. The binary is then disassembled to get the assembly code and instrumented with information from previous step to build an Allocated Memory Table (AMT), which tracks where the pointers are pointing to and addresses and size of the block they are pointing to. In the third step, program slicing is performed on the assembly code to extract its runtime constraints. Program checker is then run on this code slice to detect memory related bugs. During this phase, whenever a write to heap buffer is made in the program, the tool retrieves the buffer start address and size information from AMT to detect heap overruns. Using dynamic analysis to identify instruction sequence needs test generation tool with high coverage. Since the approach can only detect heap overruns from information collected during the phase, it is limited by this factor. As the solution does not handle statically allocated buffers, overflows to such buffers cannot be detected. Table 3-4 outlines the vulnerability detection techniques discussed in Section 3.4.

Table 3-4. Summary of Vulnerability Detection Techniques

Proposal	Description	Remarks
Ganapathy et al [8]	Proposal models pointers to character buffers by four constraint variables to denote the maximum and minimum number of bytes allocated and used by the buffer. Constraint analysis detects overflows when maximum used value is greater than the allocated minimum or allocated maximum for the buffer.	Generates many false positives due to flow-insensitive nature of the analysis.
BOON[9]	Identifies strings in the program and gathers information about their allocated size and number of bytes currently in use. Integer constraints are formulated based on buffer usage to check if allocated size is at least as large as maximum inferred used length.	Suffers from high false positives (around 90%) due to its flow and context insensitive range analysis.
ARCHER [10]	Uses flow-sensitive inter-procedural symbolic analysis using bottom-up approach on call-graph to detect memory access errors by capturing and propagating constraints upwards in call-graph.	Reported average 25% false positive rate on programs used for evaluation.
CSSV [14] and Splint [13, 118]	These annotation based approaches allow programmers to specify pre- and post-conditions. Constraints to solve are generated from them. When annotated function is called, checks to ensure safe access are performed using these constraints.	Manual specification is labor-intensive. Both tools reported high false positive rates ($\geq$ 50%).
Marple [119]	Demand-driven path-sensitive analyzer which categorizes program paths to enable priorities when diagnosing root causes of vulnerable paths.	Needs the programmers to speculate and specify potential vulnerabilities, upon which the tool can provide the inference.
Cova et al [47], Nagy and Mancoridis [120]	Uses taint analysis to detect BO bugs by marking external input as tainted and propagating taint information in the program to see if any tainted data reaches sensitive statement constructs.	Taint analysis solutions generate many false positives as they do not take into account defensive measures that may have been implemented for the statement.
Aggarwal and Jalote [123]	Static analysis module of proposal scans source code for pointer variables suspected to be aliased. It marks calls to unsafe library functions using them as arguments to detect overflows during dynamic analysis by using test cases.	Applicability is limited as it handles vulnerability detection for specialized case.
Kumar et al [124]	Binary is dynamically analyzed to extract control flow sequence and track calls to dynamic memory allocation functions to gather information about memory blocks. Binary is then disassembled and instrumented with this information to track where the pointers are pointing to and respective addresses and block sizes. Subsequently, program slicing is performed to extract runtime constraints. Program checker is then run on this code slice to detect memory related bugs.	It can only detect heap overruns and not overflows to statically allocated buffers
Static analysis approaches in literature tend to suffer from high false alarms. Adoption of symbolic evaluation techniques in real-world scenarios could be further hampered by advanced skills needed to operate the tools as well as scalability issues. Annotation based proposals are labor-intensive as they need manual specifications. Hybrid analysis approaches proposed deal with specialized scenarios limiting their applicability.		

3.5 Runtime Attack Prevention

Many runtime techniques for defending against attacks use return address modification to detect overflows. Some proposals obtain information about the buffer bound estimates and instrument the code for runtime bounds checking against the gathered buffer bounds information to detect overflows. Recently taint analysis based frameworks have gained popularity, but maintaining the taint information results in very high overheads.

One of the most well-known frame activation record attack prevention solution is StackGuard [25]. It inspired many other such run-time prevention solutions which generally vary in the nature of storage and verification despite using similar principles for detecting stack smashing attacks. StackGuard is a compile time technique which inserts code to check for return address modification to protect attacks on return address. Stack Guard places a canary before the return address and checks if the value of the canary has been modified when the function returns. If yes, then BO has occurred, the attack is detected and program halts, thereby not transferring the program control to the attacker supplied return address and thus preventing the malicious code from running. This can defend only return address based attacks an even that is not completely fool proof as return address can be altered indirectly using a pointer without modifying the canary. Microsoft Visual Studio .NET 2002 introduced similar capability in the form of /GS compiler switch in its C/C++ compiler [125].

RAD [26] proposal copies the function return address to Return Address Repository global array during the function prologue and checks for modifications to this address in the function epilogue. This approach also, like StackGuard, protects only return address based attacks. Since StackGuard and RAD are compile-time techniques, they need recompiling of source code, which may not always be available. Therefore they cannot be used on legacy systems and vendor supplied binaries without source code.

Libverify [24] copies functions to heap memory and executes the functions from copied versions. It uses wrapper function to store the return address on function entry and verifies it on function return. In Stackguard, the verification code is injected at compile-time, whereas here it is done during run-time at the call of function/process by binary re-write of process memory and they use the return address itself as canary for comparison which reduces the binary instrumentation procedure because the stack offsets remain the same whereas in StackGuard the offsets have to be changed since canary is inserted before the return address. Libverify does not need source code as it is implemented as a dynamically loadable library that is preloaded with the process it needs to protect and the instrumentation code is present in the initialization routine but it leaves the canary stack itself unprotected.

An alternative approach to Libverify's load-time code instrumentation is to insert instrumentation code in the executable code itself. Nebenzahl et al [126] propose instrumentation of Windows PE executable to prevent from stack-smashing attacks. They use IDA Pro to disassemble the binary, and identify the details of functions like their call and exit locations, memory allocated to them etc., as the instrumented code is added at the function level to the

Windows binary at the start and before the return statement of the function. Since adding instrumented code before entry and after the function exit addresses requires modification of all memory references in the binary, it is done either by extending the last section in the binary or adding a new section to the binary and storing it there. Although it does not require source code, it cannot protect against attacks that target data structures other than the return address.

SmashGuard [127] proposes hardware modifications using modified micro-coded instructions for CALL and RET opcodes for defending attacks on return address. CALL instruction is modified to store a copy of return address and stack frame pointer on data stack and RET is modified to compare the one on the top of data stack with the one the program is trying to redirect execution back to. If an attack is detected, a hardware exception is raised and the process is terminated. Split Stack and Secure Return Address (SAS) proposal [128] uses two stacks, one for data and other for control information, so that overflowing of buffer and transferring control to this location can be avoided since these two will be on two different stacks and simultaneously changing both is very difficult (unlike Libverify). But it doesn't detect buffer overflows and neighboring locations of a buffer get corrupted when it overflows. The performance costs of the approach were reported to vary between 0.01 to 23% for split stack solution depending on the application under test. These two hardware modification approaches do not need recompilation of source code but they can't prevent against non-frame activation record exploits.

PointGuard [129] is a compiler extension to protect against code pointer and data pointer attacks of buffer overflows by encrypting pointers when stored in the memory and decrypting them when loaded into CPU registers thereby detecting invalid updates of pointers. But this encrypting and decrypting the pointers may have a significant effect on run-time performance with a reported maximum overhead of 21% on one of the tested programs.

Libsafe [24], implemented as a dynamically loadable library intercepts all calls to unsafe C library functions and in turn provides substitutes of these respective functions where similar functionality as the called library function is provided but limiting any overflows within the current stack frame, based on fact that a buffer cannot extend beyond stack frame so that return address is not overwritten. This method only protects those C library functions for which substitutes are provided and even for them, it allows the attacker to overwrite anything until the frame pointer. So it can't protect against attacks targeting function pointers and heap-based overflows. LibsafeXP [29] implemented as a dynamic shared library also provides wrapper functions for all buffer related vulnerable C library functions to check the possibility of out-of-bound accesses on buffers by estimating buffer sizes using the information from symbol table for

global buffers and intercepting calls to *malloc* family of functions for heap buffers but following Libsafe's approach of using frame pointer as upper bound for stack buffers and suffers the same setbacks as Libsafe. Unlike Libsafe, this can detect heap-based overflows too. Since it works in binary mode it doesn't need source code and debug information.

Other heap based detection solutions also use similar functionality by maintaining sizes of heap allocated buffers [30, 130]. Another variation of detecting heap-smashing incorporates canary verification or guard protection mechanisms [131-133].

TIED (Type Information Extractor and Detector) proposal [44] extracts debugging information from the binary executable to build a data structure containing information about global and local variables/buffers. It re-writes the binary to store this data structure. During the run-time whenever a vulnerable library function is called, it calls LibsafePlus (an extension of LibSafe) to determine if the source string can't overwrite the destination string. LibSafePlus makes use of information in the data structure. LibSafePlus intercepts calls to *malloc* to keep track of dynamically allocated buffers. Therefore heap-based exploits are also prevented. If it can't determine the size of a buffer then the protection offered by Libsafe is provided which itself suffers from drawbacks. This tool doesn't need source-code if it is compiled with a debug option. But it requires rewriting executables or shared libraries and modifying a number of existing sections for protecting them. The reported performance overhead is less than 10%.

Rinard [34] et al implemented a compiler that inserts dynamic checks into the generated code to detect all memory out-of-bound accesses. They present a novel approach of boundless memory blocks. They generate code to detect out-of-bound accesses and when it happens instead of letting it to corrupt the memory, they store it in a hash table and whenever this value is referenced they provide the stored value based on read address and allow the program to continue execution instead of crashing/halting.

Solar Designer, a linux kernel patch from Openwall Project [134] uses non-executable stack for preventing BO exploits targeting stack based malicious code execution. The attacker usually writes the malicious code in the buffers located on the stack and modifies the return address to point to this location to execute the code. The idea behind the proposal is that by modifying O.S to make stack non-executable, such exploits can be prevented. "Pax", a patch for linux kernel [135], also uses non-executable stack to prevent such exploits. But non-executable stack methods cannot defend against *return-to-libc* attacks and attacks on data segment or heap-based buffers. OpenWall also maps the shared library's address space such that their addresses always contain zero bytes to defend from *return-to-libc* attacks. But some instances like linux signal handlers, LISP interpreters

and nested functions in gcc need executable stack. So, non-executable stack approach is not viable solution for all applications. Pax also uses Address Space Layout Randomization (ASLR) and page-based protection mechanism to protect heap and stack. The idea behind ASLR is that most attacks need knowledge of addresses to carry out the attack and randomization makes it difficult to guess the addresses.

Dahn and Manchoridis [136] proposed using program transformation to prevent stack-based buffer overflow attacks on C source code. Since overflow of stack allocated arrays is the cause for stack-based BO attacks, they transform all stack allocated arrays into heap-allocated pointer-to-arrays and introduce *free* statements at end of every block that might have instructions to overflow the buffer. So, whenever an overflow occurs, it changes the program control to point to some address in the corrupted buffer, and here since the memory is being freed, a segmentation fault will occur since the return statement is attempting to dereference an invalid pointer. Their solution therefore needs the presence of source code and cannot offer protection against *return-to-libc* attacks.

Apart from canary or guard based and range maintenance based verification approaches, dynamic taint analysis based exploit detect frame works were also proposed in literature. Dytan [75] is a taint-analysis framework in which the user can specify the taint sources, sinks, and taint level. It accesses the user-supplied information, library, program CFG, and post-dominator tree information to identify the sources and sinks, and applies the taint level to the instruction operands by using Pin binary instrumentation tool to produce an instrumented executable. To detect BO, users can specify the external inputs to program as sources and control transfer instructions such as *jmp*, *ret* as the sinks to analyze if the input has tainted the target of these control instructions for detecting and preventing attacks. The taint markings associated with each byte, storing of CFGs, post-dominator tree information for program and its related libraries resulted in high space (of order of about 240 times for GZIP application) and time overheads. TaintCheck [72], Pasan [137] and Vigilante [74] are other such frameworks which use taint analysis or runtime flow analysis to detect attacks.

Kong et al [71] proposed using hardware supported instruction-level run-time approach using taint analysis to deal with non-control data attacks. Input data is treated as tainted. Instructions using them are treated as tainted instructions while the rest as taintless instructions. Taint information is propagated when a data value is arithmetically derived or copied from tainted data and a security alert is raised when tainted data reaches a taintless instruction.

SigFree [38] is a network-based prevention mechanism which works by checking for presence of exploit code in the request packet. The main idea behind it is that BO attacks need executable code to launch an exploit but client requests to a server do not contain executable code. Since it checks for the presence of code in the packet and is not based on vulnerability signature comparison, it can detect and block new attacks also. On the other hand, it is not effective against denial-of-service and *return-into-libc* attacks.

When attacks are detected, many detection methodologies halt the program operation, in effect causing denial-of-service and provide no mechanism for self-healing. DIRA [32] a tool that can repair itself from a detected attack, maintains a log that records memory updates to track data dependencies. When an attack occurs, it uses this log to restore the program's memory to its pre-attack state. DIRA incurs high runtime overhead with a reported 60% overhead on one of the tested programs because it must log each memory update and track each data dependency. Exterminator [33] is another runtime system for detecting, isolating, and correcting heap-based memory errors. Each object has metadata, which Exterminator uses for error isolation and detection before memory allocation. Based on the information from the error isolation algorithm, Exterminator generates a runtime patch for each error. For BO bugs, it pads the buffer with the maximum value encountered for this error. This approach does not need source code, and it is useful for testing or automatically correcting a deployed system. However, isolating and detecting the heap-based errors requires additional runs and increased memory consumption.

In [138] another dynamic recovery mechanism is proposed where they instrument all statically and dynamically allocated buffers and all accesses to these buffers in application source code, by moving static buffers to the heap by dynamically allocating the buffer when entering the function and freeing it while exiting the function. For memory allocation they use *pmalloc*, their version of *malloc*, which sets up write-protected guard pages around the buffers. Any overflow or underflow to these buffers will cause the process to receive a "Segmentation violation" signal that is handled by the instrumented code to recover from such failures by treating each function call as a transaction and aborting it and restoring the system to previous system state, same as the one before the function began, using the stored information in the memory when an attack is detected. The limitations of the approach are that I/O operations cannot be rolled back so the recovered system may not function in a consistent fashion and correctness of resulting computation is also not ensured. The proposal incurs memory usage overheads along with performance penalties of 20% on Apache benchmark and 440% on micro-benchmark used for evaluation. Brief review of various run-time attack prevention techniques proposed in the literature is given in Table 3-5.

Table 3-5. Summary of Run-time Attack Prevention Techniques

Proposal	Description	Remarks
StackGuard [25]	Places a canary before the return address and checks for its modification when the function returns.	Can defend certain return address based attacks. Needs recompiling code.
RAD [26]	Copies return address to a global array during the function prologue and checks for modifications in the function epilogue.	Can defend only return address based attacks. Needs recompiling code.
Libverify [24]	Copies functions to heap and executes them from copied versions. Uses wrapper function to store the return address on function entry and verifies it on function return.	Can defend only return address based attacks. Leaves the canary stack unprotected.
Nebenzahl et al [126]	Proposes instrumentation of Windows PE executable to prevent from stack-smashing attacks.	Can defend only return address based attacks.
SmashGuard [127]	Proposes hardware modifications using modified micro-coded instructions for CALL and RET opcodes.	Can defend only return address based attacks.
Split Stack and Secure Return Address [128]	Uses two separate stacks for data and control, so that overflowing of buffer and transferring control to this location can be avoided since these two will be on two different stacks.	High performance costs.
PointGuard [129]	Protects against code pointer and data pointer attacks of buffer overflows by encrypting pointers when stored in the memory and decrypting them when loaded into CPU registers.	High performance costs.
Libsafe [24]	Intercepts calls to and provides substitutes for unsafe C library functions by limiting overflows within the current stack frame.	Can't protect non library call sinks and heap-based overflows.
LibsafeXP [29]	Provides wrapper functions for buffer related vulnerable C library functions by estimating buffer sizes using symbol table information for global buffers, intercepting calls to dynamic memory allocation functions for heap buffers and using frame pointer as upper bound for stack buffers.	Allows the attacker to overwrite anything until the frame pointer for stack buffers. Needs symbol table information.
TIED [44]	Extracts debugging information from binary executable to build data structure containing information about global and local variables/buffers. Re-writes the binary to store this data structure and intercepts calls to dynamic memory allocation function calls to determine overflows.	Needs debug information.
Rinard et al [34]	Generates code to detect out-of-bound accesses and stores it in a hash table so that when it is referenced the stored value is provided based on read address to allow the program to continue execution instead of crashing	Can incur high performance costs.
Solar Designer [134]	Proposes non-executable stack for preventing stack based malicious code execution.	Cannot defend against *return-to-libc* attacks and attacks on data segment or heap-based buffers.
Dahn and Manchoridis [136]	Proposes using program transformation to prevent stack-based buffer overflow attacks on C source code	Needs presence of source code and cannot offer protection against *return-to-libc* attacks.
Dytan [75]	Uses taint-analysis of sinks to analyze if the input has tainted the target of control transfer instructions such as *jmp*, *ret* for detecting attacks.	High space and time overheads.
SigFree [38]	A network-based prevention mechanism which checks for presence of exploit code in the request packet.	Cannot defend from denial-of-service, *return-into-libc* attacks.
DIRA [32], Exterminator [33], Sidiroglou et al [138]	The approaches proposed various recovery and self-healing mechanisms that let the program to continue execution instead of terminating when an overflow occurs.	Very high memory overheads.
Runtime attack prevention techniques can help prevent only certain types of attacks. Some techniques need presence of source code. They typically incur large overheads.		

The main advantage of runtime prevention techniques is that they can help detect exploits by using actual run-time values. The drawback is that the protection offered by publicly available

solutions is limited to certain attack patterns [31, 139]. Furthermore, since instrumentation and run-time verification is typically needed to detect attacks, they incur high overheads and slow down the system considerably, at times, limiting their practical usage.

3.6 Summary

In this chapter, we have reviewed techniques for defending against BO exploits. These techniques can be classified into defensive coding practices, vulnerability testing techniques, vulnerability prediction techniques, vulnerability detection techniques and runtime attack prevention techniques. From our review, we identified that existing approaches suffer from one or more of the following weaknesses thus limiting them for comprehensively addressing BO bugs: (1) intensive manual efforts, (2) inaccuracy, (3) scalability issues, (4) high run-time overheads, (5) coarse-grained analysis and (6) lack of binary auditing methodologies.

Incorporating defensive coding practices with security in mind makes the code non-vulnerable but at the same time it is labor-intensive and error-prone. Vulnerability testing approaches can support bug detection by generating test cases to trigger bugs. Not much is known about statements not proven vulnerable by the test generation approaches. Since these approaches typically use symbolic execution or genetic algorithms to generate test cases, they require complex frameworks and occasionally manual efforts to drive the test generation process. Vulnerability detection approaches, particularly static analysis based techniques can identify most of the vulnerabilities, but the huge number of false alarms generated by them is a huge drawback in using such tools. Vulnerability prediction approaches identify vulnerable code so that code auditors can optimize their resources to dwell on such code. While being valuable, most such approaches are coarse grained as they predict vulnerabilities at module or component level instead of statement-level precision. Attack prevention techniques can intercept and prevent certain exploit types during run-time. These techniques need dynamic monitoring systems resulting in high overheads and cannot detect varied forms of BO exploits.

The techniques proposed in this thesis aim to address the limitations of these approaches. In contrast to existing test generation approaches, the light-weight approach proposed in Chapter 4 can be easily incorporated in the software development cycle for preliminary bug elimination. The prediction based source code and binary vulnerability auditing approaches proposed in Chapter 5 and Chapter 6 are inexpensive and effective in predicting bugs with high recall and precision. We believe that using them in conjunction with existing approaches would make effective use of resources and provide comprehensive defensive mechanism in mitigating BO vulnerabilities.

Chapter 4

DETECTING BUFFER OVERFLOW VULNERABILITIES THROUGH LIGHT-WEIGHT RULE-BASED TEST CASE GENERATION

BO exploits form a substantial portion of input manipulation attacks as they are commonly found and are easy to exploit. When present in the code, they can be exploited to compromise integrity, confidentiality and availability security constraints. Despite existence of many detection solutions ranging from static vulnerability detection to network-based signature detection, BO bugs are being reported intermittently in a multitude of applications suggesting either inherent limitations in current solutions or problems with their adoption by the end-users. Researchers have proposed static as well as run-time detection methods, test generation, Operating System and network based solutions to detect BO bugs. Static analysis techniques [9, 10, 118] generally generate too many false positives, which might be one of the reasons limiting their adoption by development teams. Run-time detection techniques [24, 25, 27, 128, 129, 131-133] add instrumentation code to detect exploits during run-time. Though effective, most such approaches come with significant run-time overheads and moreover cannot protect from various forms of buffer overflow exploits. O. S initiatives like making stack non-executable [134] helps in combating only stack-based exploits but cannot handle heap-based exploits. Network-based intrusion detection techniques generally use signatures of past exploits to detect attacks or employ dynamic taint analysis to track and detect exploits [72, 74], which involves huge costs in keeping track of tainted variables. While automated solutions exist to block network-based buffer overflow attacks without the need for exploit signature [38] they too are not fool-proof.

Test case generation methodology is preferable over other static vulnerability detection techniques as it helps in finding out actual bugs before deployment rather than raising warnings which needs further auditing to confirm the warnings. Static and dynamic analysis based mechanisms are two major test-case generation techniques used in literature. Static analysis based

techniques reason about the program on all possible run-time paths whereas dynamic analysis mechanisms execute the program and use the run-time behavior information from an initial well-formed input to generate test inputs exercising other program paths. Given the many number of paths in practical programs, dynamic analysis based input-generation may be highly expensive and not scalable or realistic in the context of large real-world programs. It is used to study the effect of input parameters on path selection so as to tweak them to explore other paths [95]. Static analysis is conservative and sound because it considers all possible paths. Therefore, in this chapter we propose static analysis based approach using program source code for generating test-inputs to detect buffer overflow vulnerabilities.

Test case generation approaches for buffer overflows include symbolic execution [21, 23] or genetic algorithms (GA) based on evolutionary computing [16, 17]. Symbolic execution approaches have higher test generation costs and may get stuck in finding valid input if the input length is sufficiently large. Genetic algorithms use evolutionary computing with code coverage parameters as fitness function for generating revealing test inputs. They usually need a number of iterations to learn input constraints. Combinatorial testing approaches [20] use all possible combinations of input parameters to generate test cases simulating the methods adopted by hackers. When used in string input generation context such evolutionary or combinatorial approaches could be cumbersome as the input space is large. Instead, using static analysis information to identify input checks can substantially aid in such test generation approach.

The above observations form the motivations for this work. To overcome afore mentioned challenges, in this chapter, we propose a novel light-weight rule-based test generation methodology for detecting BO vulnerabilities. The proposed approach uses information collected during static analysis to automatically identify constraints on input and uses different input lengths based on buffer size to generate test cases. Computationally intensive techniques like constraint solving are currently not scalable to large systems in the context of vulnerability detection. Moreover the end-users may find it difficult to adapt and apply them in typical usage scenarios. Since our proposed approach doesn't involve any symbolic execution or constraint solving, it can be considered as light-weight and practical alternative to existing solutions that can be easily adapted be development teams.

Contributions and Results

- We propose a novel light-weight rule-based approach for generating test cases automatically to detect buffer overflows.

- We propose rules for generating test inputs using information collected from static analysis to generate test inputs.
- We implemented a prototype tool called *BOTestGen* that automatically generates test cases as per proposed rules from C programs.
- We evaluated the rule-based test generation approach by conducting experiments on benchmark programs.
- In the experiments, our approach could generate revealing inputs for detecting known bugs along with reporting some undocumented bugs in the benchmark suites.

This chapter is organized as follows. Section 4.1 discusses the research hypothesis to be investigated in this study. We present the proposed rules for generating test inputs in Section 4.2. This is followed by the description of our framework and experimental set-up in Section 4.3 and 4.4 respectively. We present the evaluation results in Section 4.5. Section 4.6 concludes this chapter.

4.1 Research Hypothesis

BO vulnerabilities occur due to inadequate size checks or complete lack of it. Hence it is application-level vulnerability. Typically, improper input handling and size checking usually leads to exploits. External inputs affecting the application program can be supplied from command line, read from files or network etc. The inputs are processed and propagated further in the program to achieve program's objectives. Vulnerability generally arises when inputs are copied without proper size checks into program variables, leaving the program open to external attacks. In this study we limit ourselves to command line and file inputs and propose techniques for generating string inputs since buffer overflows are particularly susceptible to string based exploits.

From our analysis on real-life buffer overflows we observed that:

- A sufficiently large input (around two to three times) greater than the buffer size is typically enough to overflow buffers with no size checks before buffer filling operations. This is in tandem with existing symbolic evaluation based approaches which solve for the least input size that will overflow the buffer.
- Such length solving strategy alone is not enough when there are constraints on the input which must be satisfied to trigger overflows. Symbolic execution based techniques learn these constraints by appending path constraints for the path whereas genetic algorithms learn them by evolution.

- Static analysis can provide information pertaining input checks as it is derived from an all-path analysis of code.

The above observations lead us to our hypothesis (H1): Static analysis information can be used to identify input constraints for generating test inputs to reveal BO vulnerabilities in the application programs.

From H1, in next section, we propose our rule-based model using control and data dependencies of potential vulnerable statement to identify input predicates performing such checks and use them to generate test inputs without the need to do an all-path analysis or symbolic evaluation.

4.2 Rule-Based Test Case Generation

The proposed approach is grounded on control and data dependency information derived from the application program's CFG. Apart from control and data dependency information our approach only needs buffer size information which can be easily obtained using any commercial or open source static analysis tools. In this proposal we limit ourselves in proposing rules for generating string inputs while in practice input could be any type.

Before further discussion we introduce some terms used in this chapter. A sink *k* is a statement in a program such that the execution of *k* may lead to unsafe or incorrect operation if the values of variables referenced at *k* are not constrained correctly. A predicate node *p* in the CFG is called an input predicate node of sink *k* if the following properties hold:

- *p* refers to an input variable
- *k* is directly or transitively control dependent on *p*

Sink input predicates are useful in identifying input constraints that needs to be satisfied to reach this sink. Conventionally, fuzzing and evolutionary algorithms learn these constraints either by multiple trials or evolution respectively whereas adaptive techniques learn them through feedback from response to previous test inputs. Symbolic execution solutions do so by identifying and representing the path constraints in terms of symbolic input values. Identifying the sink input predicates using static analysis eases the process and enhances the accuracy. We present our proposed rules and motivations behind them in the following text.

4.2.1 Input Length (IL) Rule

Sending an input of length greater than destination buffer (i.e., buffer defined at the sink) size itself causes buffer overflows in some real-world cases. We generate inputs of three different lengths to trigger such overflows. Test inputs for revealing buffer overflows should be at least the length of destination buffer. This is because some sink functions automatically append a terminating null character to the supplied source string making it larger than the destination buffer size, and effectively causing the buffer to overflow. The rule for generating *Input Length* type test inputs is as follows:

> ***InputLength (IL) Rule***: Generate three different test input strings using a default character *'a'* of lengths equal to the *dstsz*, *dstsz+1* and *2×dstsz,* where *dstsz* is the size of buffer defined at sink *k*

We choose the first two sizes namely, *dstsz* and *dstsz+1* to detect possible off-by-one overflows. Off-by-one overflows happen when the buffer is overwritten by a single element. Some functions like *gets* append a null character to the input string making it to be greater than the size of buffer defined at the sink when copied using functions like *strcpy* which copy until a terminating character is met. We chose the last one, i.e., *2×dstsz*, as it is large enough to cause overflows if some input is consumed before writing to the buffer at sink node. One may use any sufficiently large input instead of this but the other two sizes namely *dstsz*, *dstsz+1* are recommended for detecting off-by-one overflow bugs. *Input Length (IL)* rule is triggered by default. This means that those three test strings with default character *'a'* are generated for every identified input-dependent sink node. If the input is received by reading character as in the case of *getchar*, it should be treated as a single character *'a'* instead of three different test string inputs. If multiple buffers with varying sizes are defined at the sink, then *dstsz* represents the maximum known buffer size.

4.2.2 Character Check (CC) Rule

Some sinks may need specific characters to be present in the input to trigger the bug along with large input. For example, consider the sample code in Fig. 4-1. Here the predicate *(*input != '&')* at line 5 is an input predicate node for sink at node 6 and is checking explicitly for character *'&'*. The bug in the code is triggered when a large input without *'&'* overflow character is given to the program. In some cases the bug might only be revealed when the character being checked in the predicate is present in the input. Hence, we generate two inputs when encountered with such

explicit *Character Check (CC)* nodes. One input string with character being checked and another without the character being checked at predicate node. Calls to C library functions like *strchr*, which check for the presence of character in a string, are also treated similarly when known character is passed as argument to *strchr* like functions.

> ***CharacterCheck (CC) Rule***: Generate three different test input strings using the character being used and another three without character being used in the input check, with lengths equal to the *dstsz*, *dstsz+1* and *2×dstsz,* where *dstsz* is the size of buffer defined at sink *k*

```
1.  int main(int argc, char **argv) {
2.    char *input = argv[1];
3.    char buf[32];
4.    char *p = buf;
5.     while ((*input!= '\0') && (*input != '&'))
6.    *p = *input;//BAD
7.     ...}
```

Figure 4-1. CharacterCheck example

For sink node at 6 in Fig. 4-1, the following test input strings will be generated: *'&'* (32 times), *'&'* (33 times), *'&'* (64 times), *'a'* (32 times), *'a'* (33 times) and *'a'* (64 times). It should be noted that *Character Check* rule does not generate input strings if a null-character is being used in the check. Since *Input Length* rule already generates input strings with *'a'*, so as to not generate too many inputs, the tool makes use of these inputs as the three input strings without character being used in the check unless if the *Character Check* rule is checking for character *'a'* wherein default character of *'b'* is used to generate such inputs. It should therefore be noted that the light-weight approach does not aim to mimic any complex strategies by looking for characters that are not used in any of the present *Character Check* predicates to generate input strings without character being used in the check. While one can always do so, we later show that such simplistic analysis too helps in detecting bugs successfully.

Apart from sink input predicates, this rule also applies to non-predicate nodes of sink performing input checks. Consider the code snippet presented in Fig. 4-2 extracted from man-1.5h1, a Red Hat Linux online documentation tool utility. Here although there is a size check present at statement 12 in the listing, it is flawed because instead of checking for *sizeof(tmp_section_list)/sizeof(char*)*, which evaluates to 100, it checks for *sizeof(tmp_section_list)* which is 400, allowing the buffer to be overflown if an input consisting of a sequence of *':"* greater than 100 is given to the application program.

```
1.  static char ** get_section_list (void) {
2.  int i;
3.  char *p;
4.  char *end;
5.  static char *tmp_section_list[100];
6.  ...
7.  i = 0;
8.  // colon_sep_section_list is data dependent on user input prior to this
9.  for (p = colon_sep_section_list; ; p = end+1) {
10. if ((end = strchr (p, ':')) != NULL)     *end = '\0';
11. tmp_section_list[i++] = my_strdup (p); //BAD
12. if (end == NULL || i+1 == sizeof(tmp_section_list))      break;
13. }
14. tmp_section_list [i] = NULL; //BAD
15. return tmp_section_list;
```

Figure 4-2. Man-1.5h1 Buffer Overflow Vulnerability

Note that sink at line 11 is not control dependent on predicate with *strchr* check. If we consider only sink predicates we will miss generating inputs for revealing such bugs. We therefore apply the *Character Check* rule to non-predicate nodes of sink performing input character checks to trigger such bugs.

We use the following approach to identify such non-sink predicate input checks:

1. Identify the sink inputs

2. For each such identified input, perform a forward slice from the input to retrieve predicates that are not sink predicates but are performing explicit character checks on input. These predicates are termed as *Sink Non-Predicate Input Checks.*

4.2.3 String Check (SC) Rule

It can be deduced that similar to character checks, predicates may also check for presence or absence of string literal inside an input string. So when sink input predicates or sink non-predicate nodes performing input checks are data dependent on C string comparison library functions like *strcmp* or string search functions like *strstr* with a constant string literal as argument, we generate two types of inputs as with *Character Check* cases: one with the string literal being checked present in the test input and other without it in the test input. We generate inputs such that the string literal being used in check is at the start of the sink input, at the end and one less than the end of string input to catch any off-by-one overflows.

For example if the size of buffer defined at sink is 4 and the string literal used in *String Check* is *"\\"*. The following inputs will be generated by the *String Check* rule for input with length *dstsz*: *"aaaa"*, *"\\aa"*, *"aa\\"* and *"a\\a"*. Similar inputs are generated for sizes *dstsz+1*and *2×dstsz*, resulting in a total of 12 test inputs being generated.

> ***StringCheck (SC) Rule***: Generate three different test input strings using the string being used in sink input predicate or sink non-predicate input check nodes, by placing the string literal being checked at 0, *(inpsz – len(str)),* and *(inpsz – len(str)) -1* positions of the input string when applicable, where *inpsz* is the length of input to be generated and *len(str)* is defined as the length of string literal. *inpsz* takes on values of *dstsz*, *dstsz+1*and *2×dstsz,* where *dstsz* is the size of buffer defined at sink.

4.2.4 Pattern Check (PC) Rule

Sink input may also be checked against a pattern. To accommodate this, we use the following approach for pattern check identification:

1. Identify sink input predicates performing character checks
2. For each such pair of character check nodes, get the input variables referenced by them. If they are referring to the same sink input variable and use same array index variable in the checks, then we suspect it as *Pattern Check*.

For a *Pattern Check (PC)* we output the pattern as test case for the end-user to generate test input. For example, an output pattern generated for a program was: *"Pattern with msg[i] == '\n' and msg[i+1]== '.' Character Checks"*. Manual intervention is needed to understand the generated test case pattern for this rule. Hence we suggest the end-user to apply this pattern like *String Check (SC)* rule on the other automated test inputs generated by the approach to detect overflows.

4.2.5 Specific Character at Specific Index Check (SCASIC)

An application may check for presence or absence of certain character at certain position of input string. At times, this information too can be got using the Abstract Syntax Tree (AST) of predicate using the source code analysis tools. In addition to straightforward instances of such checks involving constant index (e.g., *input[2] == '\'*) we use the following approach to identify indirect cases involving non-constant index. For each of sink input variables:

1. Perform a forward slice on the input to get all the statements that are dependent on the input. Let this statement set be represented by *'D'*.
2. For each of predicate *'s'* in *'D'*, *'s'* is said to be checking for specific character at specific index if the following properties hold:
    - *'s'* performs a constant check, like say *if(i==2)*, and the constant value is a whole number,
    - The variable referenced in *'s'* is used as index in any sink input predicate character checks

As has been observed before, the bug may be triggered if the character is present at the index or absent at the index. So we replace character referenced at sink input predicate node *'p'* with default character of *'a'* if it is present at that position and if not, we replace it with character being checked at *'p'*. This rule operates on existing test inputs to generate new inputs.

For example, assume *dstsz* of 4 and having *Specific character at Specific Index Check* of *if(i==2)* and a *CharacterCheck* of *if(v[i] == '.')*. *IL* rule generates inputs *"aaaa"*, *"aaaaa"*, *"aaaaaaaa"* and *CC* rule generates inputs *"...."*, *"....."*, and *"……."* for input variable *'v'* referenced at predicate node in this example. After applying *SCASIC* rule we shall have the following inputs for *'v'*: *"aa.a"*, *"aa.aa"*, *"aa.aaaaa"*, *"..a."*, *"..a.."*, and *"..a….."* that are additionally generated.

> ***Specific Character At Specific Index Check (SCASIC) Rule***: For each of the test input values generated for sink input variable referenced at *'p'*, generate a new test input by replacing the character at index in input if it is not same as character being checked at *'p'*, or replace it with a default character *'a'* if it is same as that at *'p'*.

Table 4-1 summarizes the proposed rules. It identifies scenarios where the rule generates revealing inputs and conditions under which revealing inputs may not be generated or triggering the respective rules becomes unworkable in practice.

.

Table 4-1. Summary of Rules Proposed for Test Case Generation

Rule ID	Rule	Remarks
InputLength (IL)	Generate three different test input strings using a default character *'a'* of lengths equal to the *dstsz*, *dstsz+1* and *2×dstsz,* where *dstsz* is the size of buffer defined at sink *k*.	Helps reveal bugs where input of length greater than destination buffer size causes overflows. Fails when there are constraints on inputs that must be satisfied to trigger the bug.
CharacterCheck (CC)	Generate three different test input strings using the character being used and another three without character being used in the input check, with lengths equal to the *dstsz*, *dstsz+1* and *2×dstsz,* where *dstsz* is the size of buffer defined at sink *k*.	Supports reveal bugs where sinks may need specific characters to be present in the input to trigger the bug along with large input. Cannot handle cases where the character to be checked against cannot be derived using static analysis information. The rule will not be triggered under such circumstances.
StringCheck (SC)	Generate three different test input strings using the string being used in sink input predicate or sink non-predicate input check nodes, by placing the string literal being checked at 0, *(inpsz – len(str)),* and *(inpsz – len(str)) -1* positions of the input string when applicable, where *inpsz* is the length of input to be generated and *len(str)* is defined as the length of string literal. *inpsz* takes on values of *dstsz*, *dstsz+1*and *2×dstsz,* where *dstsz* is the size of buffer defined at sink.	Useful where sinks may need presence or absence of string literal inside an input string checked in string comparison library functions for revealing bug. If the bug can only be triggered due to its presence at some specific location in the input string other than that considered by the rule, it will be missed. Misses triggering the rule if the string literal being used in check cannot be determined by arguments to string comparison library functions.
Pattern Check (PC)	Identify sink input predicates performing character checks. For each such pair of character check nodes, get the input variables referenced by them. If they are referring to the same sink input variable and use same array index variable in the checks, the pattern is generated. Apply this pattern manually like *String Check (SC)* rule on the other automated test inputs generated by the approach.	This rule uses subscript based scheme for identifying pattern checks. Other forms of pattern checks (e.g. pointer and dependency based checks etc.) are missed by this. Heavy-weight tools using symbolic execution too may experience limitations in modeling such constructs.
Specific Character At Specific Index Check (SCASIC)	For each of the test input values generated for sink input variable referenced at sink input predicate *'p'* performing specific character at specific index check, generate a new test input by replacing the character at index in input if it is not same as character being checked at *'p'*, or replace it with a default character *'a'* if it is same as that at *'p'*.	This helps in revealing buggy sinks which require presence or absence of certain character at certain position of input string. Success depends on determining the position accurately through program analysis information.

4.3 Test Case Generation Framework

In this chapter we proposed a novel rule-based approach for generating test inputs to reveal BO vulnerabilities. The approach uses information collected from static analysis performed on program source code to learn input constraints for generating test inputs. Fig. 4-3 presents an overview of the proposed test case generation framework wherein the source code is first analyzed by the static analysis tool. The rule-based model uses the buffer size, control and data dependency information obtained from the static analysis phase for generating test inputs. It starts by identifying the sink statements in the program. We consider calls to vulnerable C library functions such as *strcpy*, *strcat*, *sprintf* etc. as sinks along with statements that write to arrays. In this study, we treat arguments supplied through command line as well as data read from file streams as inputs. For each identified sink, the approach generates test cases as per the proposed rules. Information gathered from static analysis is used to identify which rule to trigger by taking into account the code semantics and control and data dependencies of the statement.

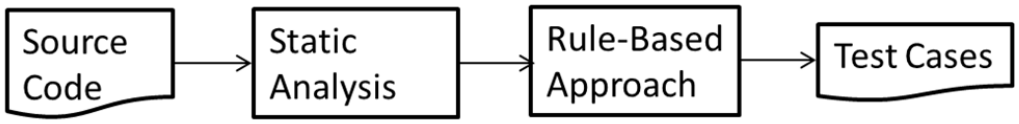


Figure 4-3. Overview of Proposed Approach

4.4 Prototype Tool and Benchmarks

We implemented the proposed approach in our prototype tool named *BOTestGen* written as a script to CodeSurfer [66]. CodeSurfer is a commercial C/C++ source code analysis tool, which can perform data and control dependency analysis. It statically analyzes the source code and builds a project representation from the code. CodeSurfer further provides an API for interacting/querying the representation. We identified vertices representing input and sink nodes by matching the vertices representing the known C library function calls performing such operations. CodeSurfer also stores information about the name and type of variable identified during project building phase. For array write vertices identification, we used this information. If the variable is statically allocated or if its size is known at compilation time, CodeSurfer knows the size of the variable and this can be retrieved using its API. We used this to generate test inputs of varying sizes as outlined by the rule-based test case generation model.

To generate test cases, the source code of the application is first analyzed by CodeSurfer. *BOTestGen* script is then run form the CodeSurfer project environment. *BOTestGen* starts by

identifying all the sink statement constructs in the program and retrieves the buffer sizes of sink destination variables. It uses the control and data dependency information of the sink to identify inputs affecting the sink and predicates performing input checks. This information is used in identifying the rules to trigger and automatically generate test cases as outlined by proposed rules.

The generated test cases are then written to a file along with the sink and input information corresponding to source code locations. The source code is instrumented for capturing BO bugs. This instrumented code is then compiled to generate executable with embedded BO checks. The executable is subsequently run by manually supplying the input generated by *BOTestGen* at specified input location to capture bugs.

To comprehensively evaluate the effectiveness of the proposed approach in detecting BO vulnerabilities we used programs from three benchmark suites [121, 140, 141] of varying sizes and complexity. These benchmarks were used in previous BO studies [17, 23, 87]. Table 4-2 shows the details of benchmark programs used in this experiment and the rules that triggered the bugs.

The first twelve programs in Table 4-2 come from Verisec suite [140] which consists of small code snippets capturing BO bugs from open source programs. The next seven are from Massachusetts Institute of Technology Lincoln lab (MITLL) benchmark [121]. MITLL benchmark contains programs developed from reported vulnerabilities in open-source network servers which were shown to challenge static BO detection tools. The BO vulnerabilities in the MITLL benchmark included buffers in varied locations (e.g., stack, bss segments) and are accessed through pointers, indices and functions. The last two real-world programs were from BugBench [141] benchmark. Presence of '+' in triggered rules column of Table 4-2 indicates that combination of rules was needed to reveal the bug, whereas ',' indicates that different bugs were revealed by invocations of comma separated rules. A '*' in the column indicates the tool was not able to generate any revealing input.

4.5 Evaluation

This section investigates our research hypothesis that the proposed rule-based generation approach can generate bug revealing inputs. Our tool was able to generate buggy inputs for all but 3 of the 21 programs ('f2', 's1' and 's3') suggesting that the proposed light-weight rule-based approach is a promising alternative to existing proposals. It can also be seen from the results that the approach can help identify the input constraints to effectively generate bug revealing inputs in most of the tested programs without the need for any symbolic execution or constraint solving.

Table 4-2. Results of Rule-Based Test Generation Approach for Benchmark programs

Program	Triggered Rules	CVE/BID Advisory
apache (get_tag - iter1_prefixShort_arr)	IL + CC	CVE-2004-0940
apache (escape_absolute_uri – simp2.)	CC	CVE-2006-3747
edbrowse (ftpls – strchr)	CC+SCASIC	CVE-2006-6909
gxine	IL	CVE-2007-0406
MADWIFI (giwscan_cb)	IL	CVE-2006-6332
OpenSER (fetchsms - istrstr)	SC	CVE-2006-6876
OpenSER (complete)	SC	CVE-2006-6749
samba (simp)	IL	CVE-2007-0453
SpamAssassin	PC	BID-6679
sendmail (buildfname – both_bad)	IL,CC	CVE-2003-0681
sendmail (mime_fromqp–arr)	IL,CC	CVE-1999-0206
sendmail (close-angle_ptr_two_tests)	CC	CVE-2002-1337
WuFTP (f1)	IL	CAN-2003-0466
WuFTP (f2*)	IL, *	CVE-1999-0878
WuFTP (f3)	IL,CC	CVE-1999-0368
Sendmail (s1)	*	CA-2003-07
Sendmail (s3)	*	CVE-1999-0206
Sendmail (s4)	IL	CVE-1999-0047
Sendmail (s5)	CC	CA-2003-12
polymorph-0.4.0	IL	-
ncompress-4.2.4	IL	-

It took a few seconds for programs in [140] and less than two minutes with MITLL programs [121] for *BOTestGen* to generate inputs. The time taken depends upon the number of sinks and the amount of sink input checks. Our tool also found out 2 apparently new bugs in 'polymorph-0.4.0' at lines 202 and 231 in polymorph.c. In case of 'f2' although our tool could not generate input to trigger the known bug, we could detect an undocumented bug not reported by Splat tool using symbolic evaluation [23], at line 94 in 'call_fb_realpath.c'.

Currently the tool caters only to string inputs as BO bugs are usually triggered by string inputs. Similar rules for handling other data types can be formulated to extend the rule-base. It can also been seen from the results that majority of the bugs were triggered by the *IL* rule followed by the *CC* rule. The promising findings could provide important avenues for detecting BO vulnerabilities in binaries using buffer size estimates gleaned from binary disassembly tools along with input check information retrieved from binary static analysis tools.

4.5.1 Comparison with Symbolic Evaluation Solution

MITLL programs (f1-s5) were used in evaluating Splat [23], a partial symbolic evaluation based test case generation tool. Splat timed out on 's1' and 's5', whereas our tool could generate exploit input for 's5'. Although the proposed rule-based approach was effective in generating bug revealing inputs in the benchmark programs it does not aspire the precision offered by the symbolic evaluation techniques that employ constraint solving for generating test inputs. Both 's1' and 's3' need a specific pattern to trigger the known bugs in the program. Given that our pattern check identification rule is based solely on array indexing scheme in contrast to pointer manipulation and other such dependencies, our tool fails to generate effective inputs for these programs. Despite using symbolic evaluation, Splat too missed generating the pattern for 's1', which highlights the limitations experienced by heavy-weight tools in reasoning some statement constructs.

The proposed approach is light-weight compared to Splat. Splat needs two parameters namely the address of the buffer storing the input as well as symbolic prefix to be specified for generating test inputs. The premise for partial symbolic representation in Splat comes from its assumption that for most cases actual content of the buffer is not significant and that some small prefix and the length of the string stored in buffer is adequate to generate revealing inputs. So the end-users need to choose and specify these parameters for generating inputs. Authors suggest that the user start with a small prefix and gradually increase the length as their testing budget allows. This process of specification, setting up and optimizing symbolic length requires considerable manual resources.

In contrast, the proposed rule-based approach automatically identifies program sinks to retrieve their sizes to generate test cases as per proposed rules rendering the solution to be viable and practical. Except for *PC* rule, generation of test inputs is automated without the need for any manual intervention.

4.5.2 Comparison with Evolutionary Algorithm

Table 4-3 compares number of test inputs generated by the proposed approach (RBTG) and results reported in [17] on programs evaluated by both the proposals. "EA", "GA" and "Random" represent number of generations taken by evolutionary genetic algorithm, genetic algorithm with constraint knowledge and random fuzzing approaches respectively to generate bug-revealing input. '*' indicates the tool was not able to generate revealing input. Our tool could generate bug revealing input for all of them and outperforms other approaches except in 'buildfname', where the large number of inputs is due to presence of many character checks. It was reported in [17] that GA module used the constraints obtained by analyzing the code statically to seed the initial population with these special characters. This helped it to converge faster than EA which further highlights the effectiveness of using static analysis information for test generation as opposed to random fuzzing methods.

In order to generate test inputs, the approach in [17] first statically analyzes the source code to generate vulnerability execution path from input sources to sinks using taint analysis techniques. After identifying the sequence of statements to be executed in the path, the source code is instrumented at statement level so as to gather runtime statistics of their execution frequencies which is later used to generate newer inputs from gathered statistics. This necessitates considerable resources to be employed for initial set-up and instrumentation purposes when compared to our light-weight approach.

Table 4-3. Performance Comparison of RBTG with EA, GA and Random Fuzzing

Program	RBTG	EA	GA	Random
mime_fromqp	6	154	20	370
buildfname	21	16	6	96
edbrowse(ftpls)	24	*	35	*

In summary, the results show that:

- By leveraging on static analysis information and code structure, bug revealing inputs can be successfully generated.
- The light-weight approach is effective and comparable to other test generation schemes, and improves over them in terms of cost and ease-of-use
- The approach can be easily adopted by development teams to aid in preliminary bug analysis
- Test generation alone is not enough to adequately detect BO bugs. While the proven bugs can be fixed, not much is known about the statements not proven vulnerable

4.6 Conclusion

The objective of our work in vulnerability detection is to aid in security auditing and testing by generating test cases for identifying vulnerabilities. In this chapter we proposed a novel light-weight rule-based test case generation approach for detecting BO vulnerabilities from source code. The approach is termed as light-weight since it does not involve any expensive symbolic evaluation or constraint solving.

The proposed rules in the approach are grounded on practical knowledge. Given a potential BO vulnerable statement, our approach begins by identifying the inputs referenced by the statement. It then uses control and data dependency information obtained from static analysis of the source code to identify and trigger the applicable rules for generating test cases.

We evaluated the proposed approach to check if the static analysis information can be used to effectively generate inputs for disclosing BO bugs. In our evaluation on various programs, our tool generated test inputs for detecting bugs in most of the programs along with finding few undocumented bugs. The methodology can be easily adopted by developers and is extensible. Our approach can also be extended to generate buggy inputs for binaries with information obtained from binary disassembly and binary static analysis

We also observed that despite it being light-weight, the approach is comparable to other testing approaches in terms of accuracy and efficiency. While it cannot be as precise as those using symbolic evaluation and constraint solving, it can be deduced that such heavy-weight approaches too often fail in reasoning and approximating statement constructs such as loop and pointer

accesses and may need manual support for optimizing them. While test generation helps in proving bugs when it can generate inputs to trigger them, it does not provide any support to infer on statements that are not shown vulnerable. Therefore, it would be useful to have an auditing mechanism that tries to prove vulnerabilities where possible and provide probabilistic observations of the statement's vulnerability if it cannot be proven to be vulnerable by test inputs.

Chapter 5

HYBRID VULNERABILITY AUDITING FROM STATIC-DYNAMIC ANALYSIS AND MACHINE LEARNING[2]

In the past, many static analysis based approaches were proposed to detect BO bugs. Though effective in detecting bugs, they tend to suffer from path and state explosion issues and generate too many false alarms thus needing huge auditing efforts for reviewing the generated warnings. Run-time detection techniques come with significant overheads for instrumenting and detecting bugs and can only help prevent certain attack forms. Test input generation methods [16-18, 21, 23], as seen in previous chapter, use genetic algorithms, symbolic execution or buffer size mutations for generating test inputs to reveal vulnerabilities. In Chapter 4, we proposed a rule-based test input generation approach using information obtained from static analysis. We showed that the approach is effective in revealing bugs. One drawback of test generation approaches is that not much can be known about the statements that are not proven vulnerable. Hence, we can see from the above that, solutions that are economical in terms of resources but effective in practice, offering viable alternative to current techniques, should be developed. To address this, in this chapter, we propose a hybrid approach combining static and dynamic analysis and machine learning for BO auditing.

The ideas developed in this chapter are partly inspired by existing work on software defect and vulnerability prediction, which use code attributes for predicting defects and vulnerabilities [48-52, 56, 84]. In supervised prediction approaches, researchers collect code attributes to characterize and predict defects or vulnerabilities. Since the attributes capture the code defect or vulnerability properties, they serve as an abstraction model in predicting them. The main advantage of data mining based prediction methods is that static code attributes such as lines of code, Halstead [54] and McCabe's [53] complexity metrics are easy to collect and with the presence of open-source tools like Waikato Environment for Knowledge Analysis (WEKA) [142], developers can readily use them for predicting bugs.

[2] The majority of this chapter is from paper published in IET Software and is subject to The Institution of Engineering and Technology Copyright.

Hence these prediction models could provide an alternative approach to mitigating vulnerabilities. Although the prediction approaches may not precisely identify vulnerabilities like sophisticated detection techniques using symbolic execution, they can be useful in providing probabilistic observations of vulnerable code sections. The security or code auditing teams can use this to focus their efforts on the code predicted to be vulnerable by the tool, thereby saving time and costs. Most of the supervised prediction techniques in literature were oriented towards predicting vulnerable components or modules [48, 55, 57] rendering them to be coarse-grained in nature. Some approaches [48] need process attributes like developer activity to predict. Such attributes are difficult to collect and could vary with respect to project teams, thereby, affecting the prediction performance. Therefore a vulnerability auditing framework that is fine-grained and at the same time accurate should be developed.

In [84] such a fine-grained framework was proposed that characterizes input validation and sanitization schemes used in web applications to predict SQL Injection and XSS vulnerabilities at statement level. It was shown that the fine-grained supervised prediction models could achieve high recall and accuracy. Since the solution predicts vulnerabilities at statement level precision, the auditing and review efforts that would otherwise be needed in the form of locating actual vulnerable statement from a predicted vulnerable module/component are not incurred. Therefore fine-grained solutions are preferable over coarse-grained counterparts.

Applications are vulnerable to BO if adequate bounds checking and input validation is not performed or is absent. Motivated by these observations, in this chapter, we propose static code attributes representing defensive measures implemented in the program for a statement to contain overflows. Since pure static analysis raises too many false alarms, we propose hybrid approach with input generation using information obtained from static analysis, as in previous chapter, and incorporate dynamic analysis in the auditing framework to augment approach's accuracy. Together, the static-dynamic analysis with machine learning helps in fine-grained vulnerability auditing by locating vulnerable code at statement level.

Contributions and Results

- We propose a novel set of static code attributes for BO prediction
- We outline test input generation using static analysis information by taking into consideration input type and validation
- We propose hybrid approach using dynamic analysis and machine learning on static code attributes for BO auditing

- We implemented a prototype tool called *BOAttrMiner* to automatically extract proposed attributes from C source code
- In our evaluation using standard benchmarks, our best classifier achieved a recall over 93% and accuracy greater than 94%. Dynamic analysis itself confirmed 34% of known vulnerabilities along with reporting 6 undocumented bugs, thereby reducing by third, otherwise needed manual auditing effort.

This chapter is organized as follows. Section 5.1 presents the research hypothesis to be tested in this chapter. Section 5.2 proposes the static code attributes that will be used for vulnerability prediction. Section 5.3 proposes the rules for test generation. We discuss the details of our hybrid auditing framework in Section 5.4. Section 5.5 describes our prototype tool, benchmarks used, experimental setup and design considerations. Section 5.6 evaluates the proposed hybrid auditing framework and Section 5.7 concludes the chapter.

5.1 Research Hypothesis

BO can be limited by performing bounds checking operations before filling the buffers or by filling the buffer based on its size. Therefore control and data dependencies of sink statements are useful in predicting vulnerabilities. Each sink statement may have different access mechanisms and features to handle overflows. To prevent input manipulation exploits, adequate input validation should be performed before propagating input variables to sinks.

To prevent BO vulnerabilities, developers generally use buffer size checks before filling operations or write to them by taking into account their size. Since input validation is performed before propagating external input in the program to prevent manipulation, observing BO defensive measures and input validation techniques can shed light on the statement's susceptibility to overflows. By analyzing the BO vulnerability reports and countermeasures to limit overflows we derived the following observations:

- Many exploits arise from improper identification of inputs capable of overflows. This results in missing the data and buffers that could be manipulated by inputs. Therefore, it is important to identify and enumerate all BO inducing inputs and sinks that consume the tainted data.
- Sinks differ in their nature of operation. While some perform copy operation others may perform concatenation, each entailing different measures to ensure their respective

safety. So, it is important to understand the nature of operation performed at sink to devise safety conformance strategies.

- Adequate input validation needs to be performed to confirm that the inputs meet the requirements.
- Proper usage of safer versions of vulnerable C string library functions by accounting for buffer size eliminates library function induced overflows.
- Size accounting can be done by checking using predicates (control dependency) or by using the *sizeof* operations or numerical size value (data dependency)
- Declaration based accounting allocates destination buffer space by considering source string length
- Proving the existence of vulnerability is preferable as it can be readily fixed

These observations lead us to our hypothesis *(H2)*: Hybrid static-dynamic analysis comprising static code attributes characterizing BO safety and defensive features implemented in the code can help predict vulnerabilities when test inputs cannot confirm their presence.

Based on these observations, we propose static code attributes to predict vulnerabilities. While the existing static analysis and test generation methods typically use heavy-weight symbolic evaluation to prove the safety adequacy of sink statements, we use light-weight data mining techniques on gathered static code attributes for providing probabilistic examinations when the sink cannot be proven vulnerable through dynamic analysis.

5.2 Static Code Attributes

In this proposal we use static code attributes to collect BO safety measures implemented for each sink. These attributes will be used for predicting vulnerabilities using machine learning algorithms and a couple of them are also used in selecting statements to be subjected to dynamic analysis. As outlined in the previous section, BO vulnerabilities happen due to inadequate or absence of input validation and bounds checking mechanisms before buffer filling operations. Therefore, bounds checking predicates and input validation checks can be useful in predicting vulnerabilities. Instead of size checks, buffers can also be filled by taking the buffer size into consideration. Hence we consider buffer size data dependencies also in our classification scheme along with bounds check predicates. Some statement constructs like *strncpy* have implicit safety

features available to limit filling operations. We take into account such statement specific characteristics too in our classification scheme.

Our static analysis is based on control and data dependency information computed using the CFG of the program. Each node in the CFG represents a program statement. We use the terms node and statement interchangeably in the discussion.

5.2.1 Sink and Input Classification

Different sinks may access different memory size and variable types. Sinks also differ with respect to the nature of operation performed at them. Such differences are to be captured in order to differentiate between sinks when performing size checks. We classify sinks based on their type of access (i.e., strings/memory blocks/array elements) and the nature of their operation (e.g., copy or concatenation) into the following types:

1) *String Copy* (e.g., *strcpy* calls)
2) *String Concatenation* (e.g., *strcat* calls)
3) *Memory alteration* (e.g., *memcpy* calls)
4) *Formatted string output* (e.g., *snprintf* calls)
5) *Unformatted string input* (e.g., *gets* calls)
6) *Formatted string input* (e.g., *sscanf* calls)
7) *Array element writes*: Any writes to an element of an array

Nodes 24 and 32 in Fig. 5-1 are Type 7 and Type 1 sinks respectively. Misidentification of inputs causes security issues by missing checks on the taint propagated. We therefore identify and classify inputs into following types based on their sources to study input taint propagation:

1) Command Line: Data submitted using command line (e.g., *getchar*)
2) Environment Variable: Data retrieved by reading environment variables (e.g., *getcwd*)
3) File: Data read from external files (e.g., *fgets*)
4) Network: Data read from network stream (e.g., *recv*)

In Fig. 5-1, nodes 4, 8, 13 are *Command Line* inputs for sink at 24 and its value is set to 3 for this sink. For sinks which are also input nodes, the respective input classification type is incremented by 1.

5.2.2 Input Validation and Buffer Size Check Predicate Classification

To prevent input related exploits, performing input validation on external input data forms the first line of defense in software system security. Code developers use various validation schemes to process external input so that it confirms to expected requirements before propagating it to sink statements. If adequate validation is performed to restrict input, the sink will be safe. Hence, we classify input validation methods implemented in the code for sink statements, to study their potential effects on input taint propagation. A predicate node *d* in the CFG is called an input validation node of sink *k* if:

i. Both *k* and *d* refer to a common input variable defined at the same node.

ii. *k* is control dependent on *d*.

For each sink *k* in the CFG of the program, we extract the input validation nodes, if any, by performing control and data flow analysis. Input validation should be performed based on the nature of the operation performed at sink. For example, though both *strcpy* and *strncpy* belong to the same sink type in our classification scheme, due to the different nature of the operations they require different types of validation since the number of characters to be copied is limited by argument value for a call to *strncpy* whereas it is limited by source string length in *strcpy*. Hence, the specified number of characters to be copied should be checked against destination buffer size in *strncpy*, whereas source string length should be validated against destination buffer size in *strcpy*.

```
1.    char *get_tag(char *tag, int tagbuf_len) {
2.      char *tag_val, c, term, *t;
3.      t = tag;  --tagbuf_len;
4.      do {c = getchar(); } while (isspace(c));
5.      if (c != '"' && c != '\'') return NULL;
6.      term = c;
7.      while (1) {
8.       c = getchar();
9.        if (t == tag + tagbuf_len) {
10.         *t = EOS;
11.         return NULL;}
12.       if (c == '\\') {
13.        c = getchar();
14.         if (c != term) {
15.           *t = '\\';
16.           t++;
17.           if (t == tag + tagbuf_len) {
18.             *t = EOS;
19.             return NULL;
20.           }
21.         }
22.       }
23.       else if (c == term) break;
24.       *t = c;
25.       t++; }
26.     *t = EOS;…}
27.   int main () {
28.     char tag[MAX_STRING_LEN];
29.     char buf[MAX_STRING_LEN];
30.     get_tag (tag, MAX_STRING_LEN);
31.     if(strlen(tag) < MAX_STRING_LEN)
32.       strcpy(buf,tag);
33.   ….}
```

Figure 5-1. Sample code snippet from Verisec suite

We classify input validation methods into following types according to the variables referenced at *k* and *d*:

1) *String Length of source buffer Check*: *d* refers to the string length of buffer referenced at *k* via call to library functions such as *strlen* (e.g., node at 31 performs this for sink at 32 in Fig. 5-1)

2) *NULL Check*: Some functions return NULL when the requested operation could not be performed. If *d* contains NULL as one of the operands of the operation involving input variable referred at *k* and *d*, it is performing a NULL Check. String termination character checks also come under NULL Check.

3) *EOF Check*: *d* has EOF as one of the operands involving the input variable referred at *k* and *d*. EOF macro flags either reaching of end of file during file reading operations or happening of an error.

4) *Character Check*: Input variable referred at *k* and *d* is a character or integer. Node 12 in Fig. 5-1 will be classified as doing this for the sink at 24.

5) *Character Occurrence in String Check*: Input variable referred at *k* and *d* is passed as argument to functions such as *strchr*, *strpbrk*.

6) *String Comparison*: Input variable referred at *k* and *d* is passed as argument to string comparison functions (e.g., *strcmp*).

7) *Other check*: *k* has validation node *d*, and *d* is not of the types discussed above (e.g., nodes 14, 23 for sink at 24 in Fig. 5-1).

Apart from input validation nodes, we also include sink predicate nodes *p* that perform buffer size or length checks on variable defined at *k*. We classify such nodes as follows:

1) *Size of destination buffer Check*: *p* refers to the size of buffer defined at *k*.

2) *Size of destination (Buffer -1) Check*: Off-by-one errors occur because buffers overflow by 1 element. Character strings are terminated by a special character. So when performing bounds check for such buffers, the size should be checked against *sizeof (buffer)-1* instead of *sizeof (buffer)*. Moreover, array element indexing starts from 0 instead of 1. So we have another attribute to represent this distinction.

3) *Size of destination (Buffer – x) Check*: Sometimes a few elements have already been written into the buffer defined at *k*. Such instances are captured using this attribute.

4) *String Length of destination buffer Check*: *p* refers to string length of buffer defined at *k* via call to library functions such as *strlen*.

Type 2 and Type 3 predicate nodes are variants of *Size of destination buffer Check* intended to catch off-by-one overflows and characterize checks on prior written buffers respectively. A control node may be classified into one or more types. Node 31 in Fig. 5-1 will be classified into Type 1 validation node and Type 1 predicate node as it is checking string length of source string with destination buffer size. We use code semantics and heuristics to distinguish between destination buffer size checks. When a predicate node is classified as performing one of the above mentioned 11 types of checks, the counter for the respective type will be incremented by one for that particular sink.

5.2.3 Data Buffer Declaration Statement Classification

A statement *q* is said to be data buffer declaration node for *k* if *k* defines *v*, and *q* declares *v*. Buffer declaration statement is also used as security measure in some code constructs. Consider the example in Fig. 5-2 where the destination buffer is declared taking into account the source length to prevent BO. We classify *q* into one of the following types:

1) Static Allocation: *v* is declared statically at *q* by a constant (e.g., node 28 in Fig. 5-1)
2) Dynamic Source Independent Allocation: *v* is dynamically allocated at *q*, but *q* is not data dependent on sink source
3) Dynamic Source Dependent Allocation: *v* is dynamically allocated at *q*, and *q* is data dependent on sink source (e.g., node 4 in Fig. 5-2).

When multiple declaration types are present we look out for the presence of source independent declaration followed by source dependent, static declaration and set the attribute value according to that priority to reflect the defensive measure application.

```
1.  void copystr (char* str) {
2.     char* buf;
3.     int len = strlen(str) + 1;
4.     buf =  (char*) malloc(len);
5.     strcpy(buf,str);
6.  }
```

Figure 5-2. Data Buffer Declaration Statement example

5.2.4 Sink Characteristics Classification

Some sinks have special features to limit write operations. As a defensive coding practice, developers use these features to stop overflows. Though both *strcpy* and *strncpy* perform string copy operation, they differ in the sense that the number of characters to be copied is specified in the latter but not in the former. We take into factor such safety features available for the sink in our classification scheme. In contrast to bounds checking predicates involving sink's control dependencies, limiting can also be achieved using data dependency. Therefore we include such safety features also and propose the following attributes:

1) *Number of Elements Copied within bounds*: This attribute is applicable for sinks like *memcpy* where the number of elements to be copied from source is specified in the sink. It is *true* (=1) if the specified number is not greater than the minimum known destination buffer size, and

false (=0) if otherwise, *unknown* (=2) when it cannot be evaluated and *not applicable* (=-1) for sinks where it does not apply or if the size is unknown.

2) *Array write index within bounds*: It is applicable to array write sinks with constant index value and known buffer size. It can have one of the three values: *true* (=1), *false* (=0) or *not applicable* (=-1). It is set to '-1' for sink node at 24 in Fig. 5-1, as the index value is not known.

3) *Format String Precision within bounds*: This is applicable to sink classification types 4 and 6. It can have one of the three values: *true* (=1), *false* (=0), or *not applicable* (=-1). It is set to '1' for sink node at 4 in Fig. 5-3 because the format string together with precision (precision value 37 + 2 extra characters in format string + '\0' = 40) is within the buffer size of 40. This helps in inferring that when format string precision is within bounds, the sink is safe.

4) *String Copy within bounds*: This is applicable for *strcpy* calls with string literal as source and known destination buffer size. It is *true* (=1) if the minimum known destination buffer size is greater than string length of the source and *false* (=0) if vice-versa, *unknown* (=2) if either the source is not a string literal or destination buffer size is not known. It is *not applicable* (= -1) if the sink is not a *strcpy* call.

```
1.  #define MAXSIZE 40
2.  void test(char *str) {
3.    char buf[MAXSIZE];
4.    sprintf(buf, "<%.37s>", str);//37 + '<' + '>' + '\0' = 40
5.    buf[MAXSIZE-1] = '\0';
6.    …
7.  }
8.  int main(int argc, char **argv) {
9.    char *userstr;
10.  if(argc > 1) {
11.   userstr = argv[1];
12.   test(userstr);
13.  …..
14. }
```

Figure 5-3. Sink characteristics example

5) *Data Dependent on destination buffer size*: If sink operands are data dependent on the size of buffer defined at the sink, this attribute value is incremented by one. These attributes are relevant when there is no predicate involved to check if BO happens but buffer size is taken into consideration before filling (e.g., *int l = sizeof(dst); snprintf(dst, l, "%s", src)*)

6) *Data Dependent on destination buffer size variant*: This is similar to the previous attribute but uses a variant of size of buffer defined at *k*. This happens when some elements are already written into the buffer prior to this.

7) *Is Character case conversion Sink*: This is applicable to array element writes. Consider the sink: **p = toUpper(*p)*. We set this attribute value for the sink to *true* (=1) because the operation being performed here is a case conversion. Such sinks are of interest because they operate on already filled buffers and the attribute helps in flagging that no further filling operation is performed here. It is *false* (=0) for non-conversion array element write sinks and is *not applicable* (=-1) for other sink types.

8) *Resets in Control Predicates*: This attribute counts the number of times the sink destination buffer pointer is reset in sink control predicates. It is relevant in representing sink operations which flush the buffer after write and are filled and flushed multiple times during execution. If the buffer is not reset after use, then, even if size checks are present it may lead to vulnerabilities.

9) *Post Size Check Increment*: This attribute characterizes scenarios when a predicate performs one of the size checks presented in Section 5.2.2, and there is an increment on index or pointer to buffer defined at *k*. Nodes at 9 and 17 exhibits this behavior for sink at 24 in Fig. 5-1. If there are multiple increments after bounds check and the number of post-increments isn't considered correctly in the check, it may lead to vulnerabilities.

Sink characteristics attributes are extracted based on sink identification. For example though both *strcpy* and *strncpy* are classified as being of the same type (i.e., Type 1) in the sink classification scheme, the attribute *Number of Elements Copied within bounds* in the sink characteristics classification is applicable for *strncpy* only and helps distinguish between them. Attributes taking on values 1/0/2/-1 are declared on a nominal scale in our machine learning tool and are treated as such by machine learning algorithms.

5.2.5 Target Attribute and Attribute Vector

Since we predict BO vulnerability, the target attribute or the class label attribute for our problem is "*Vulnerable?*". The proposed auditing approach is fine-grained as we predict the vulnerability of a sink, with the sink being an individual instance in the dataset provided to the data mining algorithms. We proposed altogether 27 static code attributes (including the target attribute - "*Vulnerable?*"). Table 5-1 summarizes the proposed code attributes. "Nom" in the attribute description entry of Table 5-1 indicates nominal data type whereas "Num" indicates numeric data type attribute. Therefore each database instance representing a program sink is characterized by a 27-dimensional attribute vector. The value of each attribute in the attribute vector represents the quantity or measure evaluated by the attribute. The attribute vector for sink at

24 in Fig. 5-1 is (7,3,0,0,0,0,0,0,1,0,0,2,2,0,0,0,1,-1,-1,-1,-1,0,0,0,0,2,FALSE) matching (*Sink*, *Command Line*, .., *String length of source buffer*, …, *Data Buffer Declaration Type*, …, *Post Size Check Increment* , *Vulnerable?*). The dataset prepared by collecting the attribute vectors for sinks not proven vulnerable by dynamic analysis is given to the data mining algorithms for predicting their vulnerability.

Table 5-1. Static Code Attributes

Attribute Name	Description
Sink	Set to '1' for String Copy, '2' for String Concatenation, '3' for Memory Alteration, '4' for Formatted String Output, '5' for Unformatted string input, '6' for Formatted string input or '7' for Array element write (Nom)
Command Line	Number of nodes that access data from command line (Num)
Environment Variable	Number of nodes that access data by reading from environment variables (Num)
File	Number of nodes that access data from files (Num)
Network	Number of nodes that access data from network stream (Num)
String Length of source buffer Check	Number of string length of source buffer checks performed by validation nodes (Num)
NULL Check	Number of data presence checks performed by validation nodes (Num)
EOF Check	Number of EOF macro checks performed by validation nodes (Num)
Character Check	Number of character / integer checks performed by validation nodes (Num)
Character Occurrence in String Check	Number of character occurrence in a string checks performed by validation nodes (Num)
String Comparison	Number of string comparison checks performed by validation nodes (Num)
Other check	Number of validation nodes performing input validation checks other than the 6 above (Num)
Size of destination buffer Check	Number of sink predicate nodes performing destination buffer size checks (Num)
Size of destination (Buffer -1) Check	Number of sink predicate nodes performing destination buffer size checks to prevent off-by-one errors (Num)
Size of destination (Buffer – x) Check	Number of sink predicate nodes performing destination buffer size checks for possibly filled-in buffers (Num)
String Length of destination buffer Check	Number of string length of destination buffer checks performed by sink predicate nodes (Num)
Data Buffer Declaration	Set to '1' for Static, '2' for Dynamic Source Independent or '3' for Dynamic Source Dependent Allocation (Nom)
Number of Elements Copied within bounds	Set to '1' if number of elements to be copied is within bounds, '0' if vice-versa, '2' if it can't be determined or '-1' if specification is not present for the sink type (Nom)
Array write index within bounds	Set to '1' if array write index is within bounds, '0' if vice-versa or '-1' for non-array write sinks (Nom)
Format String Precision within bounds	Applicable to sink types 4 and 6. Set to '1' if format string precision specified is within bounds, '0' if vice-versa or '-1' if NOT-APPLICABLE (Nom)
String Copy within bounds	Applicable to *strcpy* calls with string literal as source. Set to '1' if string literal's length is within bounds, '0' if vice-versa, '2' if unknown or '-1' if NOT-APPLICABLE (Nom)
Data Dependent on destination buffer size	Number of nodes that are data dependent on destination buffer size (Num)
Data Dependent on destination buffer size variant	Number of nodes that are data dependent on destination buffer size variant (happens for prior written buffers) (Num)
Is Character case conversion Sink	Applicable to array element writes. It is set to '1' if case conversion operation is performed at sink, '0' if not and '-1' for other sink types (Nom)
Resets in Control Predicates	Counts the number of times the sink destination buffer pointer is reset in sink control predicates (Num)
Post Size Check Increment	Counts the number of nodes performing post size check index or pointer increments (Num)
Vulnerable?	Target attribute: Vulnerable (=TRUE) or Non-Vulnerable (=FALSE) (Nom)

5.3 TEST INPUT GENERATION RULES AND DYNAMIC ANALYSIS

In this section we describe the sink selection criterion for dynamic analysis, rules proposed for test input generation as well as the dynamic analysis methodology.

5.3.1 Test Input Generation

In the proposed hybrid auditing approach, the program sinks are run with generated test inputs to confirm vulnerabilities. In addition to predicting unconfirmed vulnerabilities, static code attribute values are also used to determine whether automated test inputs are generated for the sink statements. For example, consider a character buffer, *str* of size 10. When a write is made to the sixth element of *str*, one can infer that there can be no overflow. Since we capture such information too in our code static attributes, we made use of the attribute values so as to eliminate such evident statements for dynamic analysis, thereby reducing additional overheads that would have been otherwise incurred in testing such statements.

To date, we have automated string and DNSQuery input generation. Inputs representing paths are hand-crafted taking into consideration the file system structure. We use input type and input validation information to generate test inputs. For example, consider *Character Check* node: *if (v == '\')* for which two inputs are generated: one using default character *'a'* (e.g., "*aaaaaaa*") and another using the character *'\'* in the validation node (e.g., "\\\\\\\") as it represents an input constraint.

Rules for generating test inputs are as follows:

1) For input type *char** or *int**, generate three strings using default character *'a'* of lengths equal to the *dstsz*, *dstsz+1* and *2×dstsz*, where *dstsz* is the maximum known size of buffer defined at sink.

2) For input type *char** or *int**, and having explicit *Character Check* or *Character Occurrence in String Check* with known character, generate three strings using the character in validation node, with lengths equal to the *dstsz*, *dstsz+1* and *2×dstsz*.

3) For input type *char** or *int**, and having *String Comparison* validation with string literal argument generate three input strings using the string literal used in the validation node by placing it at 0, *(inpsz-len(str))* and *(inpsz-len(str))-1* positions of generated input (when applicable) where *inpsz* is the length of input string to be generated and takes on values *dstsz*, *dstsz+1* and *2×dstsz* and *len(str)* is the length of string literal.

4) For input type *char** or *int** and passed as argument to case-checking C library functions (*isUpper* or *isLower*), generate inputs using upper, lower and both cases for alphabetic characters.

5) If input is *int* and not received from functions like *getc* or *getchar* set it to value $2 \times dstsz$.

6) If input is *DNS Query*, generate query using different domain names and different values for *RDLen* (ie., one input with *RDLen* equal to actual value of length of resource record used in the query and another with a value less than the actual value) in *DNS Query*.

7) If input is of type *Path*, generate and use strings as described in (1) through (4) as well as generate different paths such that the path string length is equal to the *dstsz*, and greater than *dstsz*.

8) For input of type *Path*, create a symbolic link such that the string length of symbolic path is less than *dstsz* but the actual path string length is greater than the *dstsz*.

9) For *Path* input, generate an invalid path such that the path string length is greater than *dstsz* (e.g., *"/aaaa"*)

10) For *Internet Address* input, generate two inputs, valid address and zero network address (i.e., 0.0.0.0).

It can therefore be seen that input generation uses static analysis information and non-confirming inputs, like real-world exploits, for revealing vulnerabilities. Input is partitioned based on type and validation characteristics (where applicable) although not complete as it considers only subset of all possibilities (e.g., rule 10). While it is not intended to be as exhaustive as combinatorial or symbolic evaluation methods, it makes use of information from static analysis to try revealing bugs for enhancing the accuracy.

5.3.2 Dynamic Analysis

Dynamic analysis in the proposed approach is performed by instrumenting source code at sinks to capture possible overflows with buffer size and other relevant information obtained from static analysis. The program is tested with inputs generated for sinks to detect bugs. To reduce the dynamic analysis efforts we don't generate inputs when the code attributes infer sink to be safe. Fig. 5-4 presents the inference criteria and the guidelines used in the instrumentation. The source code instrumented to output the BO detection result is compiled and tested with generated inputs to intercept and flag overflows. Manual intervention is needed for instrumentation and interpreting the information output by prototype tool.

Sink Test Input Generation Selection Criterion:

- Sink classification type is 1/3/4/5 and the attribute value of *Number of Elements Copied within bounds* is 1.
- Sink classification type is 7 and the attribute value of *Array write index within bounds* is 1.

Instrumentation Guidelines for sink *k* with buffer *b* of size *dstsz* defined at *k* :

- If sink is a null terminating C library function (*e.g., strcpy, strcat , sprintf*) instrument after the sink by the check *strlen(b) ≥ dstsz*
- If sink is a C library function having specification for number of elements to be copied to destination buffer, *n*, instrument the sink by the check *n > dstsz* (*e.g., strncpy, memcpy, snprintf*)
- If sink is an array-write at index *i,* then instrument the sink by the check *n ≥ dstsz*
- If sink is written via pointer *p* to buffer *b*, then instrument the sink by the check *(p-b) ≥ dstsz*

Figure 5-4. Test Input Generation Selection criterion and BO check instrumentation guidelines

5.4 Hybrid BO Auditing Framework

The proposed hybrid approach tries to either prove the statement's vulnerability through dynamic analysis or predict it from its static code attributes characterizing BO defensive features. Information from attribute collection phase is used to generate test inputs for dynamic analysis. Dynamic analysis phase was incorporated to increase the accuracy, as static code attributes alone cannot precisely predict BO vulnerabilities. The test input generation in this approach is also light-weight, as it only uses information gathered during static attribute collection. We use the input type and input validation information in the proposed BO defense characterization scheme to partition the input space for test generation along with domain knowledge. We used data mining algorithms on collected static attribute data to predict statement's vulnerability if it is not shown to be vulnerable through dynamic analysis.

Fig. 5-5 depicts the proposed approach wherein the source code is first analyzed statically to collect code attributes for each sink. During the data collection process the collected information is also used to generate inputs as per rules proposed in Section 5.3.1. At the end of data collection, the code attributes are used to determine if the sink should be selected for dynamic analysis. If the sink selection outcome results in an affirmative, dynamic analysis is performed on instrumented source code with bug revealing code inserted at the sink using the generated test inputs. If a statement cannot be shown vulnerable through dynamic analysis or was not selected for test input generation, it is then predicted using the prediction models built from learning static code attributes.

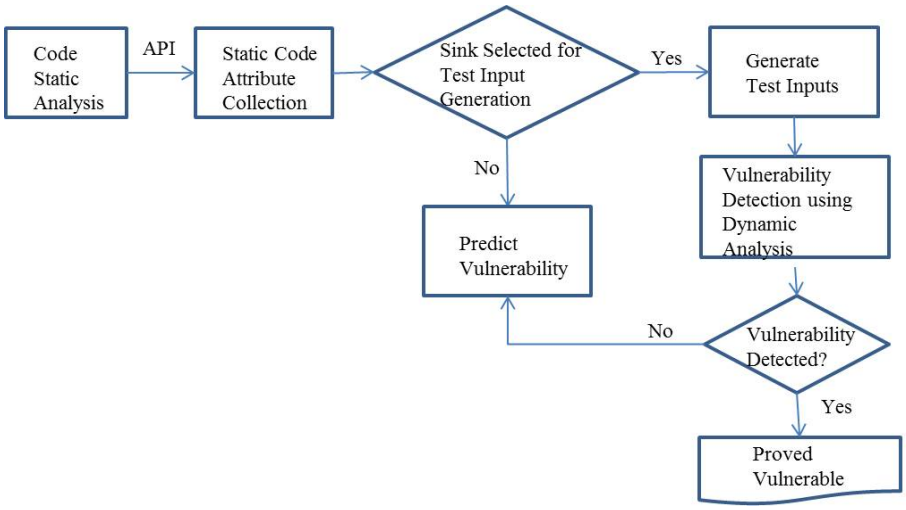


Figure 5-5. Overview of hybrid auditing approach

5.5 Experiments

In this chapter we audit BO vulnerabilities using hybrid analysis and machine learning. To facilitate adoption, we used off-the-shelf tools for performing static analysis (GrammaTech's CodeSurfer) and data mining (WEKA). Fig. 5-6 summarizes the steps involved in the proposed hybrid approach for ease of understanding. For evaluation, we used MITLL and BugBench benchmarks used in previous BO studies [22, 23, 87, 143]. Sections 5.5.4 and 5.5.5 present the details of the benchmarks used in the evaluation and the experimental set up.

1. Analyze the source code using CodeSurfer static analysis tool
2. Identify the sink statements in the program
3. **for** each sink
   i. use the control and data dependency information to gather proposed static code attributes as described in Section 4.2
   ii. **if** (!(((sink type ∈ {1/3/4/5}) && (Number of Elements Copied within bounds == 1)) or ((sink type == 7) && (Array write index within bounds == 1))))
      a. Generate test inputs as outlined in Section 4.3.1
      b. Output Sink Attributes
      c. Output Test Inputs, if any generated
   iii. **else**
      a. Output Sink Attributes
   iv. **endif**
4. **endfor**
5. Manually instrument sinks in the program source code using the guidelines outlined in Figure 4.4
6. Generate an executable using the instrumented source code produced in step 5
7. Run the executable generated in step 6 with the test inputs generated in step 3 (ii-a) for the program sinks and store details of sinks detected vulnerable
8. **for** each sink in the program
   i. **if** sink is detected vulnerable in step 7, it is proven vulnerable.
   ii. **else** add the sink static code attributes generated in step 3 (ii-b) to program data file for prediction.
   iii. **endif**
9. **endfor**
10. Predict the sinks in program data file added in step 8 (ii) using classifier algorithms in off-the-shelf tool WEKA

Figure 5-6. Algorithm Summarizing Steps involved in the Proposed Hybrid Approach

5.5.1 Static Analysis

To collect attribute data and generate string inputs we implemented a prototype tool called *BOAttrMiner* as a script to CodeSurfer [66]. Sink selection rules are also included in *BOAttrMiner*. DNSQuery input generation is written in separate C code. *BOAttrMiner* uses CodeSurfer API to identify known C library functions for classification purposes. Since the project representation of program being analyzed is built by CodeSurfer during program compilation time, it knows the variable type if it can be determined during the compilation. If the size of the variable can be determined during the project building phase, CodeSurfer allows the end-user to retrieve this also using its API. Our tool made use of the type and dependency information of identified sinks to generate test inputs. In total, *BOAttrMiner* generates three files. "*SinkAttributes*" file contains sink location in source code and its corresponding code static attribute values for all the sinks in the program. "*TestInputs*" file contains test inputs along with corresponding input and sink location in the source code for statements selected for dynamic analysis. "*DynamicAnalysisInfo*" file contains destination buffer information for instrumentation purposes. For each sink in the program, *BOAttrMiner* extracts the attribute values as per classification proposed in Section 5.2. Attribute vector for each sink consists of sink type, 25 other attributes and target attribute – "*Vulnerable*?". Hence it can be seen that like size and code complexity attributes used in defect prediction studies our proposed attributes too can be collected easily making the models affordable and practical.

5.5.2 Dynamic Analysis

We manually ran BO check instrumented programs produced as outlined by guidelines in Section 5.3.2 by supplying test inputs generated as per prescribed rules in Section 5.3.1, using information in files generated by *BOAttrMiner* to detect vulnerabilities. Since vulnerability information is available for benchmark programs in [121, 141] we instrumented only sinks defining vulnerable buffers during evaluation.

5.5.3 Data Mining

If the sink is not proven vulnerable by dynamic analysis or not selected for test input generation, it is predicted using classifier models built from learning code static attributes.

5.5.3.1 Classifiers

Classification algorithms, as mentioned in Chapter 2, are *supervised learning methods* which learn input-output mapping using the training data instances with target/class label information to generalize the acquired knowledge for predicting the target class of test instances. In literature many classification algorithms were proposed such as support vector machines, neural networks, simple decision trees, nearest neighbor algorithms, statistical method based techniques, as well as ensemble setups. Different classification algorithms may perform differently on the same dataset due to variations in their intrinsic learning methodologies. To assess predictive capability of our attributes we used three well-known machine learning algorithms: BayesNet (BN), Multi-Layer Perceptron (MLP) and Sequential Minimal Optimization (SMO) by performing a 10-fold cross validation (CV). Different classifiers were used in the study to evaluate the effectiveness of the proposed approach across various learning schemes but not to compare between classifier performances. The implementations of the classifiers used in evaluation are available for immediate usage in the WEKA data mining tool kit [142]. The working of these classifiers was discussed in Chapter 2. We hereby present the information pertaining vital parameter values used for building the classifier models in WEKA.

BN classifier represents probabilistic distributions in a graphical manner by learning the structure of the network from the training data. It involves using a function for evaluating the network based on the data and a method for searching through possible network structures. Upon determining the network structure conditional probability distributions of each of the variables in the graph are calculated to determine class labels for classification purposes. To build a BN model we used the default parameter values set in WEKA, which are:

```
searchAlgorithm = K2;

estimator = SimpleEstimator;
```

Setting the parameter "`searchAlgorithm`" to K2 would allow the BN model to use K2 algorithm for searching network structures. Likewise setting the parameter "`estimator`" to SimpleEstimator would cause it to be used for estimating the conditional probability tables of a Bayes network once the structure has been learned. The algorithm estimates probabilities directly from data.

MLP classifier is inspired by studies on biological neural network structure which develops by learning to adapt to the environment. It uses backpropagation learning algorithm to train the artificial feedforward neural network by iterating over the training data such that the network weights are adjusted to minimize the mean squared error between the predicted and target values. Similar to building the BN model described above, to build a MLP model too we used default parameter values set in WEKA, which are:

```
No. of hidden layer = 1;

No. of nodes per hidden layer = (no. of attributes + no. of classes)/2;

trainingTime = 500;

learningRate = 0.3;

momentum = 0.2;
```

Training of the MLP classifier is stopped when the number of iterations specified as per "`trainingTime`" is completed which is set to default value of 500 epochs. "`learningRate`" represents the amount by which the weights are updated whereas "`momentum`" represents the momentum applied to weights during updating.

SMO algorithm is used for training support vector machine classifier models. SVM uses non-linear mapping to transform original training data into a higher dimension to search for maximum margin hyperplane that can separate the class tuples in the data with maximum separation between the classes. As with other models described above, here too we use default parameter values in WEKA for building the SVM model using SMO algorithm which are:

```
C = 1.0;

epsilon =  1.0E-12;
```

```
kernel = PolyKernel;
```

The complexity parameter represented by '`C`' is set to '1.0'. This parameter is used to trade off wide margin between the classes with a small number of margin failures to finitely penalize data points that cross boundaries. Parameter "`epsilon`" is a slack variable that permits margin failure and is set to default value specified by WEKA. Kernel function represented by parameter "`kernel`" in the classifier model measures the similarity or the distance between the input value and the stored training instance. WEKA sets this to "`PolyKernel`" which allows non-linear model mapping. Both SMO and MLP algorithm implementations in WEKA transform nominal attributes into binary ones and normalize the attributes by default. Default settings in WEKA were used as we did not intend to optimize any particular classifier. The objective of the study was to know what accuracy can be expected using the predictive capability of the proposed static code attributes. Further research on the area could concentrate on optimizing classifiers.

5.5.3.2 Performance Measures

To measure the performance of hybrid auditing approach, we used the measures recall or probability of detection (*pd*), probability of false alarm (*pf*), precision (*pr*) and accuracy (*acc*) calculated as per the formulae defined in Section 2.3.4.1. Since we proposed a hybrid approach involving static and dynamic analysis, sinks proven vulnerable by dynamic analysis were added to the true positive (*tp*) count of WEKA generated classifier output to calculate hybrid analysis performance measures for classifiers.

5.5.4 Experimental Design – MITLL Benchmark

This benchmark was developed by Zitser et al [121]. It contains 14 programs with and without BO bugs (i.e., patched versions) developed from reported vulnerabilities in three real-world open source network servers. We used some of the MITLL benchmark programs accepting string input in the previous chapter for test generation. The programs have vulnerabilities due to unsafe C string manipulation library function calls and also due to direct array accesses using pointers or an index. Table 5-2 shows the statistics of the programs used for evaluation. Bind (DNS server) has 4 model programs, Wu-ftpd (ftp server) has 3 and Sendmail (email server) has 5 programs respectively. The cumulative total of relevant statistics for each application, namely Bind, Wu-ftpd and Sendmail, is presented at the end of the application for ease of understanding. Each of the three application programs (Bind, Wu-ftpd and Sendmail) has several bugs reported in CVE and CERT data bases and accordingly captured in the model programs. The input format to the model

program varies and is given in the third column of Table 5-2. Column 5 reports the number of sinks in the program and the sixth column shows the number of known bugs in the program. The last column (# of Detected Bugs) in Table 5-2 shows the number of bugs detected by our dynamic analysis component. The underlined digits in the last column show the identified undocumented bugs in the programs.

Table 5-2. MITLL Benchmark Statistics

Application	Program	Input Format	LOC	Sinks	Known Bugs	Detected Bugs
Bind	b1	DNS Query	1.15K	4	1	1
	b2	DNS Query	1.39K	6	1	1
	b3	DNS Query, Integer	0.28K	4	1	1
	b4	String, Internet address	0.69K	10	2	1
Bind Total			**3.51K**	**24**	**5**	**4**
Wu-ftpd	f1	Path	0.42K	23	4	4 + 1
	f2	Path	0.94K	17	1	1 + 1
	f3	Path	1.01K	56	19	7
Wu-ftpd Total			**2.37K**	**96**	**24**	**14**
Sendmail	s1	String, Integer	0.93K	67	28	-
	s2	Password	0.99K	23	2	1 + 1
	s3	String	0.60K	10	3	-
	s4	String	0.61K	19	4	4
	s5	String	1.11K	16	3	-
Sendmail Total			**4.24K**	**135**	**40**	**6**

Since Bind dataset is small, Bind and Wu-ftpd datasets were combined to enable 10-fold CV application. There are 120 sinks in the combined dataset of which 18 were found vulnerable by dynamic analysis. Hence the remaining 102 sinks were predicted. Sendmail application has 135

sinks, out of which 6 are proven vulnerable by dynamic analysis. Of the 6 proven vulnerable, 1 bug is a previously unknown bug. The 129 remaining sinks (= 135-6) whose vulnerability status is unknown will be predicted similarly using 10-fold CV.

5.5.5 Experimental Design– BugBench Benchmark

As full-scale case studies we tested our proposed approach on real–world programs with BO vulnerabilities. We used programs from BugBench benchmark suite to conduct this test. BugBench consists of buggy applications with various types of software bugs collected by Liu et al [141]. Table 5-3 gives the statistics of the buggy programs used in the study. ncompress-4.2.4 is an application from Red Hat Linux for file compression and decompression whereas polymorph-0.4.0 is a GNU utility for converting Win32 file names to Unix. Both the programs have known overflows in their stack buffers. Our dynamic analysis found out all the known bugs in both the applications along with finding 3 undocumented bugs in polymorph-0.4.0. To predict the statements whose vulnerability is not known from both these application's datasets, we used Sendmail prediction dataset prepared as described in Section 5.5.4 as the training dataset. In other words, to evaluate the performance of our approach on ncompress-4.2.4, we used Sendmail dataset for training the prediction models and ncompress-4.2.4 dataset comprising of 42 (= 43-1) unknown sinks as the test dataset. Polymorph-0.4.0 was also evaluated similarly. This experimental design is in line with our hypothesis that by training the prediction models from applications with known vulnerability information, we should be able to predict statements in newly developed applications.

Table 5-3. BugBench Benchmark Statistics

Application name	**Description**	**Input Format**	**LOC**	**Sinks**	**Known Bugs**	**Detected Bugs**
ncompress-4.2.4	Compression and decompression program (Linux)	String	1.93K	43	1	1
polymorph-0.4.0	Win32 to Unix filename converter	String	0.70K	38	3	3 + 3

5.6 Evaluation

The objective of the evaluation is to answer the following research questions:

1. *"How effective is the hybrid approach in auditing buffer overflows?"*
2. *"Given the static code attributes representing sink type, inputs and safety features, can we predict the sink's vulnerability?"*
3. *"Are test inputs generated using information collected from static analysis effective in vulnerability detection?"*
4. *"How does the proposed hybrid approach compare with existing approaches in literature?"*

5.6.1 Results and Undocumented Bugs

The evaluation was conducted using Intel 2.50 GHz 8 GB RAM PC. Performance of classifiers on datasets prepared as described in Section 5.5.4 and 5.5.5, using 10-fold CV are presented in Table 5-4. Average classifier performance measure over all datasets is given in last row of Table 5-4.

As we integrate multiple approaches we did not measure theoretical complexity but considered computer time. It took *BOAttrMiner* thirty and sixteen minutes for 's1' patched version and 'ncompress' programs respectively to generate files and around a minute each for the rest of the evaluated programs.

On average, all classifier models achieved good results with (*pd* > 96% and *pf* < 22%). Individually too minimum recall over all classifiers on all datasets is 90% suggesting that the hybrid approach could correctly audit more than 9 out of 10 bugs. MLP is the best classifier (based on average *acc*) followed by SMO and BN. Considering individual values, MLP reports *pd* over 93% and *pf* below 6.5%, which is close to desirable values. BN reports high *pf* compared to others. This could be due to the reason that Bayesian network algorithm assumes strict independence between the attributes, but it can be seen from our classification scheme that there are certain dependencies between the attributes (certain attributes are only applicable for certain sinks). The results of rest of the classifiers are comparable to those obtained in the benchmarked vulnerability prediction studies with (*pd* > 70%) reported for developer activities based prediction [48], and (*pd* = 78%, *pf* = 6%) for XSS and (*pd* = 93%, *pf* = 11%) for SQL Injection vulnerabilities reported in [84].

Table 5-4. Results of Hybrid Auditing Approach (with performance measures in %)

Data	**Classifier**	***pd***	***pf***	***pr***	***acc***
Bind +Wu-ftpd	BN	96.7	23.6	58.8	81.7
	MLP	93.5	2.2	93.5	96.7
	SMO	96.7	3.3	90.9	96.7
Sendmail	BN	97.5	9.6	81.6	92.6
	MLP	97.5	6.4	86.9	94.8
	SMO	90.2	8.5	82.2	91.1
ncompress	BN	100	40.4	5.5	60.5
	MLP	100	2.4	50	97.7
	SMO	100	0	100	100
polymorph	BN	100	12.5	60	89.5
	MLP	100	0	100	100
	SMO	100	9.4	66.7	92.1
Average	BN	98.6	21.5	51.5	81.1
	MLP	97.8	2.8	82.6	97.3
	SMO	96.7	5.3	85	95

Dynamic analysis identified 25 of 73 known bugs (≈ 34% of bugs) along with detecting 6 undocumented bugs (details presented in Table 5-5). It detected bugs in 9 of 12 programs from MITLL benchmark and all known bugs in BugBench programs. Hence, the results show that proposed approach is effective with high recall and low false alarms and that the static code attributes are good indicators of BO vulnerabilities. The test inputs generated are effective in

revealing vulnerabilities. If detected by dynamic analysis, no auditing is necessary, just patching of the code by the developers.

Table 5-5. Undocumented BO Cases in Benchmark Subjects

Program	FileName	Line #
f1 (Wu-ftpd)	mapped-path-bad.c	207
f2 (Wu-ftpd)	call_fb_realpath.c	94
s2 (Sendmail)	util-bad.c	187
polymorph-0.4.0	polymorph.c	200, 202, 231

5.6.2 Comparison with Static Analysis Approaches

In this section we compare the performance of MLP classifier of the proposed approach (PA) with static analysis tools. Table 5-6 presents the details of tools used for comparison.

Table 5-6. Description of Static Tools Used for Comparison with Hybrid Approach

Tool	Detection Strategy
Splint [13]	Intra-procedural analysis by monitoring buffer creation and accesses in form of annotated constraints and solving the constraints to detect BOF
ARCHER [10]	Call-graph bottom-up flow-sensitive inter-procedural symbolic constraint analysis
BOON [9]	Inter-procedural flow-insensitive symbolic constraint analysis on vulnerable C library functions.
UNO [144]	Inter-procedural flow-sensitive model-checking
Polyspace [145]	Inter-procedural abstract interpretation
PA	Hybrid Program Analysis with machine learning that harnesses the static and dynamic analysis information.

Performance measures of various static analysis approaches on MITLL benchmark are computed from buggy and corresponding patched sink results reported in [146] and given in Table 5-7. It can be seen from Table 5-7 that PolySpace and Splint perform comparatively better than other tools. ARCHER and BOON have small non zero recall on Sendmail and Bind+Wu-ftpd respectively, showing that they detect some bugs without false positives. UNO neither detected bug nor raised warnings on safe cases, so it achieved accuracy over 50% in both the test subjects. While PolySpace shows a minute higher recall over MLP classifier of Proposed Approach in Sendmail application, it has very high false alarm rate with less precision. Hence it can be seen that proposed approach is better than static analysis tools with ($pd > 93\%$ and $pf \leq 6.4\%$) for the best classifier, MLP.

Table 5-7. Comparison of Hybrid Approach Results with Static Analysis Tools

Benchmark	**Approach/Tool**	***pd***	***pf***	***pr***	***acc***
Bind+Wu-ftpd	Polyspace	56.75	47.37	53.84	54.67
	ARCHER	0	0	NA	50.67
	Splint	63.16	38.46	38.46	62.34
	BOON	2.7	2.63	50	50.67
	UNO	0	0	NA	50.67
	MLP Classifier - PA	93.5	2.2	93.5	96.7
Sendmail	Polyspace	97.8	90.56	48.94	51
	ARCHER	2.12	0	100	54
	Splint	6.38	5.66	50	53
	BOON	0	0	NA	53
	UNO	0	0	NA	53
	MLP Classifier - PA	97.5	6.4	86.9	94.8

In [143] a path-sensitive analysis by specifying sink safety constraints was proposed. Upon identifying sink the tool raises query backwards to determine safety constraint satisfaction. It uses symbolic execution to track and update buffer usage and reports faulty path segments for which safety constraint is determined false. Table 5-8 compares our results with [143]. While it found out 3 more bugs than our light-weight approach, it may not detect the bug at 207 in 'f1' wherein compiler's placement of adjacent buffers renders it to be vulnerable as the source buffer at line 207 was overflown by write to its adjacent buffer. Such bug can only be detected by dynamic analysis and conventional static analysis as in [143] would term it as safe. It needs the user to specify the faults (i.e., the sinks) to detect their vulnerability status. Our hybrid approach can also be used to complement such demand-driven analysis by employing it to identify probable vulnerabilities instead of labor-intensive user-specifications. The demand-driven approach as in [143] can then be applied to confirm the vulnerability status leading to cost-effective solution combination.

Table 5-8. Comparison between Path-Sensitive Approach and Proposed Hybrid Approach

Program	# Bugs Detected by Approach in [143]	# Bugs Detected by Proposed Approach
f1(Wu-ftpd)	4	5
s2(Sendmail)	4	2
polymorph-0.4.0	8	6

5.6.3 Comparison with Test input generation approaches

LESE [22] and Splat [23] use constraint solving on symbolic inputs to generate test cases for revealing vulnerabilities. Splat uses partial-symbolic representation of input string with associated symbolic length. This needs manual intervention for fixing initial prefix size and optimizing it, if inadequate. LESE symbolically represents the number of types a loop executes and links it input fields expressing loop-dependent program values in terms of input. It may not be effective in detecting bugs with loop-independent bug triggering input grammar. In contrast to these heavy-weight techniques we used a rule-based approach taking buffer size, input type and validation into account by generating conforming and non-confirming inputs using information obtained from static analysis.

Although they both used the programs in the MIT LL benchmark we used for our evaluation, a direct comparison of performance measures is not possible, since, unlike us, they did not consider each sink but whole program while reporting results. Furthermore, as the benchmark is designed to be self-contained, programs in BIND1 and BIND2 of benchmark which parse DNS packets also include code to generate DNS packet. In our evaluation, we modified it in line with BIND3 for DNS packet receiving. So, it is not clear from Splat as to which buffers (and what parts) are considered as input. We treated the whole DNS Query packet as input.

LESE included the exploit input in their result but the presented exploit input cannot be used to exploit all the known vulnerabilities marked as /* BAD */ in the benchmark code. Hence it appears that they considered the whole program while reporting the exploit input result. In that regard, our inputs reveal bugs in 9 of 12 programs without any symbolic evaluation while Splat and LESE detected in 10 and 12 programs respectively. Using such test input techniques we can only infer on sinks proven vulnerable but nothing is known about rest of the sinks. We believe that our approach is better since we predict such sinks with a high recall instead of them being taken as safe.

5.6.4 Limitations

While the proposed hybrid method is effective in auditing buffer overruns, it has some limitations. The proposed approach uses machine learning algorithms for training the prediction models using static code attributes. Although the proposed static code attributes can be easily collected using any static analysis tool, sufficient known vulnerability data is needed to train the prediction models. The rule-based test input generation is successful in generating revealing inputs as shown by the evaluation results, but it fails to generate inputs for complex cases (for example ones which expect certain patterns in the input and other dependencies that can only be identified by heavy-weight analysis and modeling). The proposed rules can be augmented with static analysis information with look out for possible patterns, but this would incur additional overheads that are hard to justify in most real-world cases.

5.6.5 Threats to Validity

For every empirical study it is important to be aware of potential threats to the validity of obtained results. In this chapter, we use hybrid approach involving dynamic analysis and machine learning for auditing buffer overflows. Training and testing procedures may influence the outcome of data mining experiments resulting in threats to internal validity. We used 10-fold cross validation to cancel out any factors that may influence the results due to training and testing

dataset preparation. External validity threats refer to factors preventing the generalization of results. It is hard to get many publicly available real-world programs with known vulnerability information for training the models. We therefore used benchmarked applications for evaluation. To determine the generalization capabilities, the prediction model was trained on one benchmark application (Sendmail) and tested on other benchmark applications (ncompress and polymorph).

5.7 Conclusion

In this chapter, we proposed a light-weight hybrid analysis approach for auditing buffer overflows by combining static and dynamic analyses, and machine learning. Our tool starts with static analysis to collect the proposed static code attributes for all identified sinks in the program. Upon collection, some of the attribute values are used to select sinks for dynamic analysis. The information from data collection phase is also used to generate test inputs. Selected sinks are tested using generated inputs on executables compiled from BO check instrumented code. For sinks where no overflow is detected, a machine learning classifier built on code attributes is used to predict overflows. Confirmed vulnerable cases can be directly patched by the developers without further auditing.

In our experiments, the dynamic analysis could detect 34% of known vulnerabilities. This is considerable given that no symbolic evaluation was used. Overall, the hybrid approach yields with the best classifier, MLP, accuracy above 94% and recall over 93% with false alarm rates less than 7% suggesting that more than 9 out of 10 actual vulnerabilities were audited correctly. These results are significantly better than many other static analysis tools reported in the literature and publicly available for comparison.

Since the proposed attributes can be easily collected, our approach is practical while being efficient, making it an effective alternative solution to vulnerability detection. When experiencing new patterns for exploiting vulnerabilities, the models can be re-built with attributes capturing them and the solution is hence extensible. While we make no claims that the proposed hybrid auditing approach is intended to replace existing detection approaches since it can only provide probabilistic remarks based on past training data with known vulnerability information, it can also be used as a complimentary aid to first find out probable bugs so as to use heavy-weight approaches like formal proving or symbolic execution to determine the existence of real bug. Thus, the hybrid approach offers a cheaper yet effective method for mitigating vulnerabilities.

Chapter 6

VULNERABILITY AUDITING FROM BINARY EXECUTABLES

Developers adopt or implement defensive coding methods, such as buffer size checks and input validation, to guard against BO. Buffer size checks ensure that buffers are not overflown whereas input validation checks the user input against required properties such that only inputs satisfying the properties are accepted for further processing. If the defensive methods are adequate, BO vulnerabilities can be avoided. By characterizing the defensive coding methods it has been shown in Chapter 5 that BO vulnerabilities can be audited effectively from source code. Although there are a number of approaches for detecting vulnerabilities from source code very few do so from binary executables [46, 47, 77, 147-150]. Commercial-off-the-shelf components are generally distributed in binary form. Therefore, before using such third-party components it is imperative to check for presence of vulnerabilities. Complier optimization too can sometimes render changes to executables from what is intended in source code [151]. Analyzing executables is highly desirable in such scenarios as it can help uncover potential vulnerabilities. While the advantages of analyzing binaries are well-known, very few tools and frameworks [152-154] are available for analyzing binaries as it is deemed challenging since the source code level semantics information is not available in the executable. There is no notion of variables or variable types in binaries as the memory is accessed directly through addresses or indirectly through register based address expressions. Hence much of vulnerability detection research is limited to source code analysis.

In this chapter we propose a novel approach for BO auditing through binary analysis. By developing techniques to infer security related information from binary executables, we support the growing need of security analysts in understanding binaries. After fleshing out the methodologies used for enabling binary static analysis we investigate the use of static analysis and machine learning for predicting BO vulnerabilities from binaries in this chapter.

As can be deduced from the discussion of BO in previous chapters, it is extremely difficult to use static analysis alone to infer BO vulnerabilities precisely from binaries. This is because much

of the source code based information is either not available or is difficult to extract and comprehend from binaries. Combining prediction method and static analysis could provide important avenue for addressing this problem. Hence, in this chapter, we propose use of static analysis and machine learning for BO vulnerability prediction in executables.

Similar to characterizing buffer size and validation checks, examining the characteristics of buffer usage patterns and BO defensive measures implemented in the assembly code could be useful in predicting vulnerabilities. These observations coupled with the effectiveness of source-code based hybrid auditing approach motivated us to develop a binary vulnerability analysis framework by combining static analysis and machine learning using attributes that can be extracted from binary executables. Our attributes for BO characterization use only control and data dependencies and buffer usage information obtained from static analysis.

In addition to predicting BO bugs from binaries the attributes can also serve as pointers of defensive coding practices implemented in the program thereby aiding the end-users by supplying them with reverse engineering information needed to conduct security audits.

In light of this, in this chapter, we propose a binary analysis framework for statically analyzing binaries along with the novel methodology for providing probabilistic observations of a disassembled statement's vulnerability to BO. We first present the mechanism and algorithms used in the binary static analysis. This is then followed by the discussion of vulnerability characterization in terms of attributes that can be extracted from binary static analysis. As in source-code based auditing approach, here too we propose a fine-grained statement level vulnerability auditing approach. We then use static control and data flow analysis information to automatically extract all the proposed attributes from the disassembled binaries. Binary BO prediction models are then built using the attribute data collected from programs with known vulnerability information. By training the prediction models on binaries with sufficient known vulnerability information we can predict vulnerabilities in other applications whose binaries are available.

Contributions and Results

- Characterizing BO defenses implemented in the code from binary disassembly.
- Enable the prediction of BO vulnerability from x86 executables using static analysis possible.
- Static analysis based code auditing approach for predicting BO vulnerabilities from x86 executables.
- Prototype tool for extracting binary disassembly information from IDA Pro Disassembler.

- Prototype tool for static analysis of x86 executables
- Prototype tool for predicting BO vulnerabilities from x86 executables
- Evaluation of the proposed approach based on binaries of six C language-based benchmark programs. In the experiments, using our proposed approach, the best prediction model achieved excellent results with recall of 75% and precision of 84% with an accuracy of 94%.

This chapter is organized as follows. Section 6.1 presents the concepts and back ground for understanding the binary analysis framework. Section 6.2 provides the details of the framework. We present our BO characterization scheme for predicting vulnerable x86 statements in Section 6.3. The experimental set up used in the evaluation is detailed in Section 6.4. We evaluate the proposed approach and discuss the findings in Section 6.5. Section 6.6 concludes the findings from the chapter.

6.1 Binary Analysis

Software applications are typically distributed in the form of binaries to prevent end-users from comprehending proprietary algorithms and to protect intellectual property rights of software owners. Presence of malware adds another dimension to this by covertly gathering and siphoning off sensitive information. As binaries contain all the instructions that are executed by the program during run-time, analyzing binaries can help in addressing any perceived problems before deployment. The primary challenge with the binary analysis is to comprehend the program nature by looking at the binary representation. Although there are tools such as disassemblers that help in the process, reverse engineering binaries and comprehending necessary information from them needs substantial resources and high level of expertise. It would therefore be advantageous to have tools that help in automating the process with minimal supervision.

Existing binary analysis tools can be classified into analysis and instrumentations frameworks. Binary analysis tools may either apply pure static analysis like proprietary CodeSurfer/x86 platform [152] or combined static-dynamic analysis as in BitBlaze [153] for program analysis. There are also binary analysis frameworks like Rose [68], BAP [154] and Vulcan [155] that provide means for building customized program analysis tools. Vulcan depends on symbol table information such as that present in PDB file generated by Microsoft VC++ compiler for the analysis, which limits its application. BAP, on the other hand, disassembles x86 and ARM binaries and represents the assembly instruction's side effects in an intermediate language (IL) for enabling syntax-directed analysis. Both BitBlaze and BAP let the end-users to perform or adapt analyses

and optimizations on IL. We used binary analysis framework of Rose compiler infrastructure (developed by Lawrence Livermore National Laboratories) to build our static analysis tool. Rose [68] provides API for performing control flow analysis on the disassembled code. It also provides different representations of x86 assembly instructions for performing a variety of analyses such as constant propagation and symbolic execution which can be used for data dependency analysis. Binary instrumentation frameworks like Valgrind [156, 157], Pin [158], DynamoRio [159] and Paradyn project's Dyninst [160] support fine-grained instrumentation of user-level applications. Although the instrumentation tools cannot by themselves identify any program bugs, they are used in dynamic analysis to gather statistics about the programs for aiding reverse engineers working on binaries. All these tools provide API for building plugins to intercept and analyze context information during the program run. They are generally used to capture information from program trace to enrich that obtained from static analysis.

6.1.1 Binary Disassembly

In this section, the basics of executable generation are presented first followed by the description of binary disassembly and the challenges encountered during the process. Subsequently, the solutions adopted to overcome them are sketched out.

Executable or program building process generally involves multiple stages such as pre-processing, compiling, assembly and linking as shown in Fig. 6-1. In the pre-processing phase, the header files are included; macros are expanded and conditional compilation is performed. During compilation the source code is parsed and tokenized to build an intermediate representation (IR) of the program from the tokens. The representation thus built is used to perform a variety of analyses like syntax checking, control and data flow analysis to optimize the generated code in terms of memory, speed etc. The compiler takes the output of pre-processor to generate assembly code. During the assembly phase this assembly code is translated to machine instructions. Some of the source code level information is lost during the assembly process (e.g., variable and label names unless debug information is specifically made available). Besides, since the program data also is stored in the binary form just like program code, distinguishing code from data becomes difficult during disassembly. As code comments in high-level constructs are lost during the assembly, comprehending the code becomes tough. In the final stage of the program building or executable generation, the linker combines the object files generated by the assembler with libraries (if any) to produce an executable by resolving references to external symbols, assigning addresses to program functions and variables and further revising code and data to reflect this address assignment.

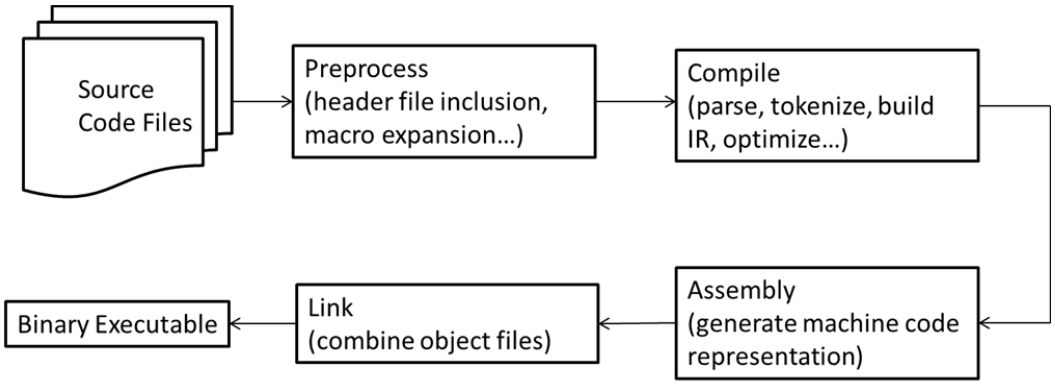


Figure 6-1. Steps involved in executable generation

Machine code is directly executed by the machine but is not comprehensible to humans. Therefore software reverse engineers typically use tools like disassemblers or decompilers to help in reversing binaries. Disassembly is the extraction of assembly instructions from its machine code representation whereas decompilation is the process of reconstructing higher level representation from program's assembly level representation. It should be noted that given the loss of information during the program building stage, decompilation often is very difficult and infeasible to be accurate in practice as it depends on the richness of information present in the machine code. Obfuscated code further complicates this by making it even more difficult to decompile. In this chapter, we therefore propose auditing approach by considering disassembled binary.

There are two types of static disassembly namely: linear sweep and recursive traversal. A linear sweep disassembler sequentially parses the binary code starting from the first byte in binary's text segment. It assumes that next instruction to decode begins right after the prior disassembled instruction. Thus it attempts to disassemble one instruction after the other without taking any notion of the control flow. Since code and data are sometimes interspersed in binary, it may generate errors in disassembly by interpreting data as code. Similarly, the algorithm also suffers when obfuscation approaches insert junk bytes to fool the disassembler. Presence of variable length assembly instructions imposes further challenges.

Recursive traversal disassembly overcomes this drawback by taking into account the control flow of the program to determine the binary parsing. The algorithm starts by using linear sweep and disassembles until a branch instruction is encountered. Upon disassembling and subsequently identifying the branch's control flow successors, it starts to disassemble at those addresses. The issue with recursive traversal is that the branch targets have to be identified correctly in order for the disassembly to be accurate. Presence of indirect jumps and calls in the executable hamper its

accuracy. This may result in its inability to identify reachable code or may erroneously cause the disassembler to interpret data as code when the set of targets are overestimated.

We used IDA Pro [67], a popular commercial disassembler, which employs recursive traversal algorithm for binary disassembly in our framework. To handle the algorithm's inability to follow indirect paths, IDA Pro employs proprietary heuristics to identify code pointers. We made use of IDA Pro in our study as it identifies program functions and builds a CFG for each identified function along with a call graph for the disassembled executable showing the caller and callee relationship. IDA Pro also provides Software Development Kit (SDK) to access its internal programming interfaces in the form of API. Using the SDK, the disassembly information can be accessed and operated upon by the end-user.

IDA performs sophisticated analysis to determine the stack frame layout for every function it disassembles by monitoring the behavior and operations performed on the variables in the function to determine local variable area size and memory references to them. It assigns names to the local variables based on their location relative to previous frame pointer value on the stack. For example in Fig. 6-2 when the function *sub_4011a0* is called, a stack frame is created with return address pushed initially onto the stack frame. During the prologue, as depicted by the first three instructions in the left of Fig. 6-2, the previous stack frame pointer value is pushed onto the stack frame. The function has one local variable. As the function local variables are named by their relative location to previous frame pointer on stack, it can be seen that the local variable *var_4* is at an offset of 4 bytes from pushed frame pointer and hence named accordingly.

We rely on IDA Pro's advanced analysis for identifying global and local variable addresses/offsets and sizes for our static analysis. Although our static analysis framework is built on Rose, the information provided by IDA Pro is richer. Therefore IDA Pro is used to extract needed information from binaries. The variables in the executable may be global, local or heap variables. Global variables can be accessed from anywhere in the program and have fixed addresses (within *.data* or *.bss* segment) whereas the scope of local variables is limited to the stack frame of the function in which they are allocated. Heap variables are dynamically allocated and deallocated and hence their addresses cannot be determined statically. We used IDA Pro's SDK to identify all statically known addresses and stack offsets from the disassembly. Since there might be holes in these addresses we treat a variable as starting from the statically known start address (or offset) to next known address (or stack offset) as in [161].

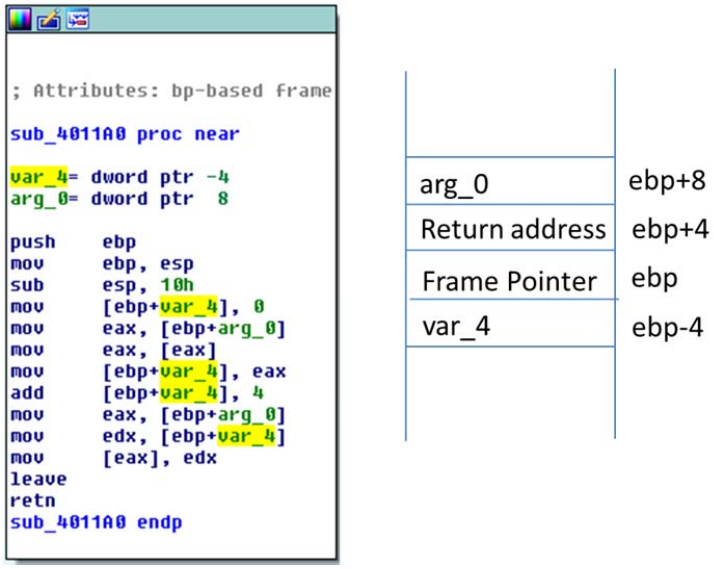


Figure 6-2. Example depicting IDA Pro's disassembly and function stack layout for a program function

Consider the stack frame layout of a program as depicted in Fig. 6-3. Here, the program has local variables *r*, *s*, *str*, *a*, *b* etc. which were declared in the program and allocated space on the stack. Whenever the variable is referred to in the disassembly it is accessed using its offset from the stack pointers *ebp* or *esp*. Since frame pointer was used for accessing them in the disassembled program we present using that notation in our discussion.

The address space for *r* ranges from *ebp-4* to *ebp-1* (both inclusive), thereby resulting in 4 bytes being allocated to it. Similarly variables *s*, *str*, *a*, *b* and *c* are each allocated 4, 32, 40, 40 and 4 respectively which can be calculated from the stack layout offset differences. Unlike local variables which are accessed using stack pointers and corresponding offsets in function stack frame, global variables have a fixed statically known address and are hence accessed using the address directly. Their sizes are also identified similarly.

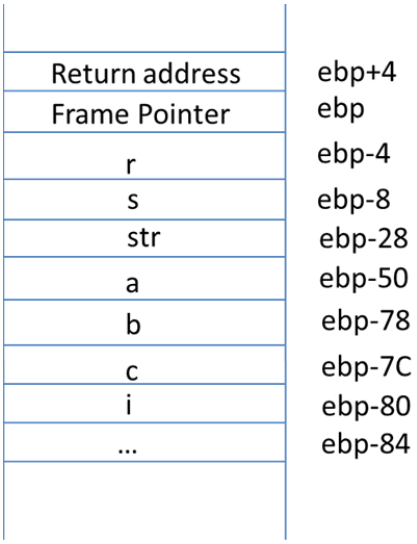


Figure 6-3. Example for variable size determination from statically known stack frame offsets

After identifying the functions, program variables and constructing the CFG, the next task is to build tools for performing static analysis. Most importantly the tool should have the following basic functionalities for the disassembled binary:

- Identification of variables defined at a statement
- Identification of variables referenced at a statement
- Identification of control dependency for a statement
- Identification of data dependency for a statement

A statement in a disassembled binary is an assembly instruction. An assembly instruction generally does not bear a one-to-one correspondence with a source code instruction. More often than not, multiple assembly instructions are needed to represent a high-level language statement construct. An assembly instruction is a low-level language construct representing the operations performed by a processor during the course of execution. Each assembly instruction is composed of a mnemonic representing the processor command and a list of operands. The mnemonics coupled with the architecture and operand position determines if the operand is defined in the instruction or referenced. Some instructions like *push, pop* affect program state variables that are not in the operand list. Therefore, in order to perform program analysis, the semantics of each instruction has to be accurately represented to model the instruction's behavior. We developed static analysis tool built on top of functionality provided by binary analysis component of Rose to achieve this objective. Information extracted from IDA Pro is given as input to this binary static

analysis tool. We next explain the algorithms used in determining the control and data dependencies of disassembled statements.

6.1.2 Intra-procedural Control Dependency Analysis

Control dependency analysis helps in identifying the predicates or checks that must be satisfied before reaching the program statement. Control dependency relationships can be calculated from the CFG of the application procedure. Sinha et al [64] define CFG of a procedure *P* as a directed graph *G = (N, E)* in which *N* contains one node for each statement in *P*, and *E* contains the edges that represent possible flow of control between statements in *P*. Instead of a statement, a node in *N* could also be used to represent a basic block. A basic block in the CFG is a sequence of statements in *P* with a single entry and exit statement at the start and end of the block respectively.

The CFG has two distinguished nodes, namely, the 'entry' and 'exit' nodes. An 'entry' node has no predecessors whereas the 'exit' node has no successors in the CFG. A predicate node in the CFG (or basic block in CFG in which end statement is a predicate statement) has two successors with edges labelled '*true'* and '*false*' representing the '*true*' and '*false*' branches of predicate respectively. All other nodes in the CFG have only one successor.

Control dependency successors are computed using the post-dominator information obtained from the CFG. Post-dominator tree is constructed by performing dominator analysis on Reverse CFG of the CFG. Reverse CFG of a CFG is computed by reversing the edges and swapping the 'entry' and 'exit' nodes of original CFG. Node *u* in *G dominates* node *v* if and only if every path from the 'entry' node to *v* in *G* contains *u*. Node *u postdominates* node *v* if and only if every path from *v* to the 'exit' node in *G* contains *u*. Node *u* is *control dependent* on node *v* if and only if *v* has successors v' and v'' such that *u postdominates* v' but does not *postdominate* v''.

Using the immediate post-dominator information obtained from the post-dominator tree and the control-flow information of predicate node retrieved from CFG, control dependency successors are typically calculated using the algorithm presented in Fig. 6-4. The theory and algorithms for calculating dependency relations are discussed in detail in [162, 163].

```
CFG Edge from X to Y with label l.
X depicts predicate node with more than one successor.
IPDOM(X) returns immediate post-dominator of X.
CDsucc calculates the control dependency successors.

Procedure CDsucc (X->l Y)
      If Y ≠ IPDOM(X) then Z = Y
            While Z ≠ IPDOM(X) do
                        Z is control dependent on X with label l
                        Z = IPDOM(Z)
            endwhile
      endif
end CDsucc
```

Figure 6-4. Algorithm to calculate control dependency relations from program's post-dominator and CFG information

6.1.3 Intra-procedural Data Dependency Analysis

Data dependency analysis helps in discovering the statements that a given statement depends upon. Given a potential vulnerable statement, it is imperative to know how the external data is propagated with in the program. To perform data dependency analysis, the first step is to identify the variables defined and referenced in the assembly instruction. An assembly instruction has two parts: code and data. The code (i.e., the instruction mnemonics) specifies the operation performed by the instruction whereas the operand required by the code can be in the instruction itself, in an internal register or in the memory. Although the mnemonics and the operand placement typically can help in identification of variables defined and referenced at the instruction, it can be construed from the above discussion that some instructions have variables defined or referenced at them that are not explicit by looking at the operands. Therefore the variables defined or referenced have to take into account the semantics of the operation performed by the instruction. For example a *pop ecx* instruction, in addition to defining the instruction operand *ecx* also modifies *esp* by incrementing it even though *esp* is not an explicit instruction operand.

Apart from careful representation of instruction semantics another important impediment in binary analysis is the nature of variable access through addressing modes. While in some cases the variable being accessed is easily identifiable, in vast majority, the memory address of the variable is expressed in the form of a register expression, which can only be computed when the contents of the registers are known to help determine the variable being accessed.

Assembly language supports different addressing modes [164] to specify the instruction operands (as shown in Fig. 6-5) namely:

- Register addressing mode: Registers contain the data to be manipulated by the instruction (e.g., *mov eax, ebx* copies the content of *ebx* register to *eax*, thereby referencing *ebx* and defining *eax*)
- Immediate addressing mode: Data is specified in the instruction itself (e.g., *mov eax, 100*)
- Direct addressing mode: Operand specified needs access to memory and is generally limited to simple or non-pointer variables (e.g., *mov eax, myvar*)
- Indirect addressing mode: Operand specified needs access to memory and is usually used for accessing complex variables like arrays or variables pointed-to by pointer variables (e.g., *mov ecx, [eax]*);

Operands located in memory are accessed by specifying the address of the memory variable as a register reference expression *(base + index * scale +displacement)*. The objective of different addressing modes for memory is to handle different data structures and higher-level language constructs effectively. Register indirect addressing mode using base register is generally used for pointer dereferencing. In the *base+displ* addressing mode, the effective address of the memory variable is expressed as a sum of address in the *base* register and signed *displacement* value. Indexed addressing mode is typically used to access array elements with the value in *index* register enumerating the element to be accessed and that in *scale* depicting the size of array element type.

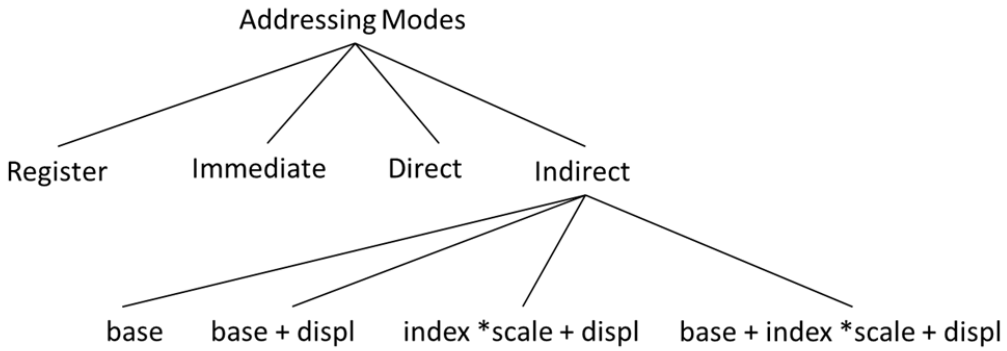


Figure 6-5. Assembly Addressing Modes

Live variables analysis and reaching definitions analysis are two forms of data flow analysis involving variables. Liveness analysis involves analyzing if a variable is used in the future whereas reaching definitions analysis is performed to identify what definitions of a variable reach

a particular statement. Since our auditing approach requires data dependency for computing BO static attributes, our binary static analysis framework only caters to reaching definitions analysis.

A definition of a variable is a statement that defines the variable. Since each assembly instruction has its own address, the dependency is represented using the unique statement address as its label for the definition. Statement *u* is *data dependent* on statement *v* if there exists a variable *x* such that *v* defines *x* and *u* uses *x*, and there is a definition clear path in CFG from *v* to *u*.

Data flow information for reaching definitions can be collected by setting up and updating the definitions and uses of variables at various points in the program. This is represented formally for a statement '*S*' [162] by the following equation:

$$out[S] = gen[S] \cup (in[S] - kill[S])$$

and can be interpreted as *"the information at the end of a statement is either generated within the statement, or enters at the beginning and is not killed as control flows through the statement"* [162]. Since data in the program flows along control paths, data dependency analysis is affected by the control flow constructs. Therefore the CFG of the function is used to carry out the analysis. To determine the memory variables accessed by the indirect addressing mode we perform symbolic evaluation in conjunction with reaching definitions analysis on the program's CFG for calculating data dependency relations.

Consider an example C code snippet as shown in Fig. 6-6 and its corresponding disassembly. It should be noted that the executables for the examples in the following discussion were generated by including the debug information for ease of understanding while our prototype tool does not make such an assumption of its availability.

The code initializes a pointer, modifies the pointed-to element directly without pointer usage and subsequently dereferences the pointer. In the absence of memory tracking and symbolic evaluation it is not possible to know what variable the pointer '*c*' is referring to at statement 6 of source code in Fig. 6-6. This is because if we only track register accesses, one can only determine based on load operation of instruction *move eax, [ebp+c]* that contents of '*c*' are copied to *eax*, but it is not possible without simultaneously tracking memory variables to determine that '*c*' has the address of variable '*r*'. Therefore memory accesses (loads and stores) also need to be taken into account in data dependency analysis. By doing so and tracking memory as well as register states symbolically, it can be determined that pointer variable '*c'* has the address of local variable '*r'* as its value. When the pointer is dereferenced and copied to variable *'d'* at statement 6 in C source code, it translates to last three instructions in the corresponding disassembly, wherein the value in

variable '*c'* is first copied into register *eax*. At the end of the instruction *move eax, [ebp+c]*, *eax* has as value the address of variable '*r*'. *mov ecx, [eax]* uses register indirect addressing where the value of the memory address pointed by *eax* is copied into *ecx* register. In effect, this causes *ecx* to be copied with value 10, the contents of '*r*'. The last instruction *mov [ebp+d], ecx* copies the value 10 into local variable '*d*'. Hence to compute data dependency relations, in our framework, we maintain a memory state containing entries for registers, flags and program variables. Usage of symbolic evaluation helps in representing the effects of the disassembled instruction on this state, thereby helping to identify definitions and uses of operands to calculate data dependencies.

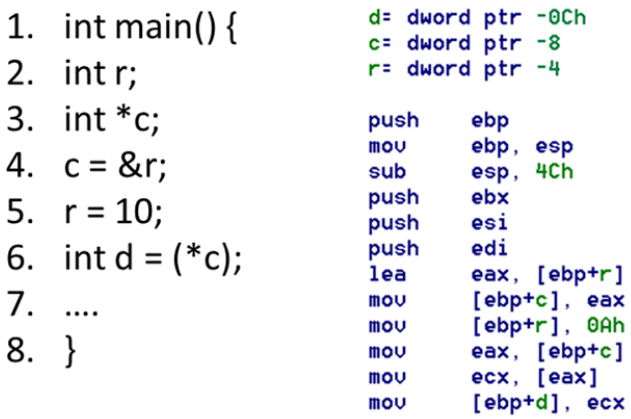


Figure 6-6. Example code depicting the need for symbolic evaluation and memory tracking for data dependency calculation

It should however be noted that no path feasibility analysis is performed during the static analysis in our binary analysis framework. Usage of symbolic evaluation could also help in determining indirect jumps or calls that can be calculated by constant propagation as shown in statement 8 of Fig. 6-7. Similar to pointer initialization in Fig. 6-6 by statement *lea eax, [ebp+r]*, wherein the effective address of variable *r* is copied to register *eax*, the statement 8 in the souce code listing of Fig. 6-7 when translated to binary is represented as *mov foo, offset InitArray*. This results in copying the address of procedure *InitArray* into global function pointer variable *foo*. Since we track memory variables too during the symbolic evaluation, after the instruction is evaluated, entry for memory variable *foo* in the memory state will be populated with address of function *InitArray*. After pushing the call parameters onto the stack when subsequent call is made through function pointer by the instruction *call foo*, the call instruction symbolic semantics retrieves the address of function being called and transfers the program control to that instruction. Hence usage of symbolic evaluation for static analysis helps in identifying such control transfers which cannot be directly obtained from call graph of statically disassembled binary.

```
1.  void (*foo)(int *,int,int);
2.  void InitArray(int *a,int size,int element);
3.  int main() {
4.  int r, a[10], b[10];
5.  char str[32];
6.  scanf("%d",&r);
7.  InitArray(a,10,r);
8.  foo = &InitArray;
9.  foo(b,10,5);
10. int k = b[3];
11. …
12. }

13. void InitArray(int *a,int size, int element) {
14. for(int m=0; m <size; m++) a[m] = element;
15. }
```

Figure 6-7. Example code snippet showing call using function pointer

To perform symbolic evaluation, the definitions and dependencies for each statement in the CFG are to be maintained for calculating data dependencies. Since the basic block represents a statement sequence with single entry and exit statements, instead of maintaining state for each statement, which is memory exhaustive, associating and maintaining state with basic block is adequate. The basic block's symbolic state represents the registers, flags and memory variables in terms of symbolic expressions which are solved to answer questions pertaining memory accesses in the form of indirect addressing modes. Calls to library functions are handled as per call semantics. For example when a call to *strcpy* is made, the first argument is identified as being defined and the second argument as being referenced. Dynamically allocated variables are identified by symbolic values returned by calls to library functions such as *malloc*. Global variables are identified by their data segment addresses and local variables by their stack offsets. At each instruction, the uses and definitions are captured along with its effect on symbolic state and propagated to control flow successors until the iterative reaching definitions algorithm converges. Fig. 6-8 shows the iterative intra-procedural data dependency algorithm.

1. **decl** *worklist*: vector of nodes representing BB in CFG
2. **Proc**: IntraProcDD()
3. Add entry BB of CFG to *worklist*
4. **while** *worklist* is not empty,
    a. Get the front BB node *n* in the *worklist*
    b. Identify and retrieve the predecessors *m* in CFG for *n*
    c. Retrieve and merge the state and definitions of *m* predecessors to get final state and definitions to start processing *n*
    d. Process and propagate state and definitions by symbolically executing all the instructions in *n*
    e. If this is the first instance of processing *n* or if the definitions in the *n* have changed after this processing instance from prior processing of *n*, include the successors of *n* into the *worklist* to propagate these changes
5. **end while**
6. **end proc**

Figure 6-8. Intra-procedural iterative worklist algorithm for calculating data dependencies

When a basic block has multiple predecessors the states and dependency information from all predecessors is merged to get the final state from which the instructions in the basic block are evaluated symbolically. For container variables like arrays we calculate the data dependencies by considering the whole container instead of individual elements. This is generally in-line with the precision offered by commercial code analysis tools.

Inter-procedural analysis is similar to intra-procedural analysis except that it is performed taking into consideration program's call graph. Inter-procedural data dependency analysis can be done either by replacing each call with the CFG of the respective callee function at the call instruction in the caller or by stepping into the callee with the caller's memory state as the entry state of the callee. In practical cases, a procedure may call multiple procedures and may also be called multiple times by other procedures. Therefore a value-based approach as mentioned above is unrealistic in practice.

Scalability and efficiency in terms of memory space and time constraints are vital in inter-procedural analysis. Therefore in our framework, we use summary flow functions to capture the summary of variables defined and referenced in the callee and use them to represent call effects in the caller when encountered with a call to callee. Given that such a summary flow function should represent the effect of multiple calls to the callee, the generated summary should be context-independent and at the same time parametrized such that context-dependent information can be used to incorporate the calling context upon call to the callee function. To do so we incorporate summarized functions concept in our framework. This enables scalability and efficiency while compromising little on accuracy.

6.1.4 Inter-procedural Analysis

6.1.4.1 Control dependency

To calculate inter-procedural control dependency the dependencies of callee's entry block are updated to register its dependence with that of the caller's basic block containing the call instruction. As and when a summarized function is called from the caller, the callee's entry block's control dependencies are updated with the basic blocks that the caller instruction's basic block is immediately control dependent upon.

6.1.4.2 Data dependency

To compute inter-procedural data dependencies, the function to be summarized is analyzed using the intra-procedural worklist algorithm presented in Fig. 6-8. This helps in gathering known definitions and unknown references of the program variables. To include the calling context, we need to identify the function arguments and record dependencies affected by them. Since global variables and dynamically allocated variables are not limited by the scope of the function in which they are used or defined, their states and dependencies should be propagated upwards in the call graph. Given that local variables have function scope and are visible only in the function in which they are defined, after the function returns, their memory state is not needed and can be done away with. *eax* register is typically used to store the return value of the function when the function returns a value. Therefore the summary should also have a state entry for *eax* register. Thus to model inter-procedural data dependency analysis, we need to:

- gather and propagate dependency information pertaining local variables throughout the function
- gather and propagate dependency information pertaining global and dynamically allocated variables throughout the executable's call graph
- gather and propagate dependency information pertaining registers throughout the executable
- maintain summary symbolic state for the global and dynamically allocated variables, *eax* and argument values

When a function *x* is called by function *y*, its dependency and symbolic summary state information *calleedep* and *calleess* are retrieved respectively. Let *callerdep* and *callerss* be the dependency and symbolic summary state information at the call point in *y*. The calling context (i.e., the argument

values and stack addresses of arguments) at the call point is obtained from *callerss*. Since *calleedep* and *calleess* represent the inherited data dependency and state information of the callee, the synthesized dependency and state information can be derived by incorporating the calling context in conjunction with inherited summary into *callerdep* and *callerss*. Fig. 6-9 shows the steps involved in inter-procedural data dependency analysis.

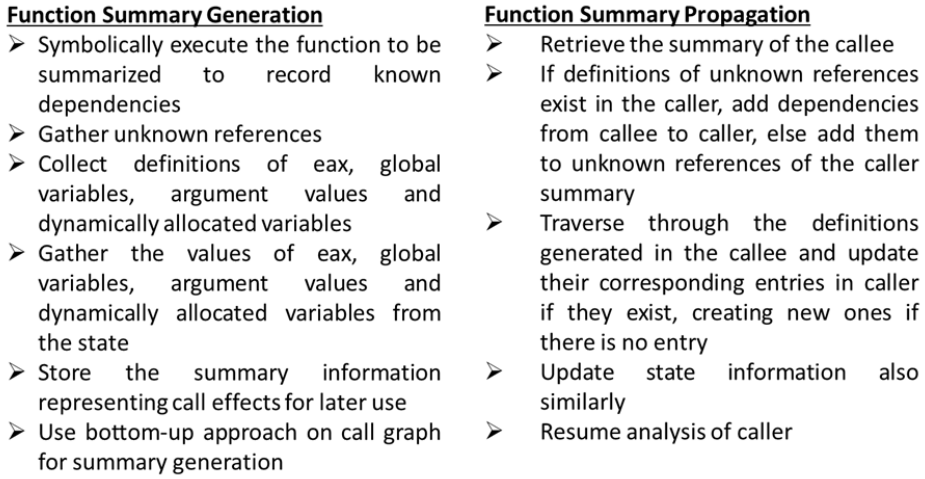


Figure 6-9. Steps involved in inter-procedural data dependency analysis using function summaries

It can therefore be observed that summarizing functions does not require traversing the CFG of caller function for gathering contexts since the summary generation is parametrized and context-independent to capture the effects of calling context. Such approach is scalable as it restricts the influences between caller and callee to acceptable levels, though not as accurate as can be obtained through value-based analysis. Hence to be scalable we impose trade-offs in terms of accuracy.

6.2 Overview of Binary Analysis Framework

We used binary static analysis tool built on top of Rose to gather code attribute information for predicting vulnerable BO statements in binary disassembly. Rose is an open source compiler infrastructure that supports analysis of source code and binary executables. It runs on linux O.S. Since the precision and assumptions of the analysis may affect vulnerability prediction, in this section, we describe the binary analysis tool and outline how the binary analysis challenges were addressed.

As mentioned in the previous section, in order to perform data dependency analysis, the semantics of each instruction have to be precisely represented to model the instruction's behavior.

Rose provides x86 assembly instruction semantics that caters to this requirement. Instruction semantics in Rose have two halves: the abstract semantics and the semantic policies. Abstract semantics define each x86 instruction in terms of abstract RISC-like operations. When instantiating the abstract semantics object we give it a semantic policy. Rose has many semantic policies that can be attached to the abstract semantics object. The split architecture allows end-users to have one implementation that knows about each individual x86 instruction, with many ways of interacting with the instructions. Different semantic policies are used to accomplish different goals (ex: constant propagation policy, symbolic semantics policy). Since symbolic execution is needed to identify memory accesses, we used symbolic semantics policy provided by Rose for our analysis.

This enables us to capture its effect on machine state and also helps in propagating constants and identifying those indirect branches or calls that can be determined statically through constant propagation. It is important to note that we do not evaluate path feasibility criteria or keep track of buffer usage in the program by representing them in form of constraints as in source code based BO detection solutions. Constraint solver is only used in the analysis to identify memory variables being accessed by given variable address to determine data dependencies.

As discussed previously, assembly instructions use different addressing modes to access memory based on variable type in high level-language construct. The address is specified in the form of register expression *(base + index * scale +displacement)*. If the values contained by base and displacement are statically known, the variable being accessed can be identified to container level accuracy. Fig. 6-10 shows the overview of the binary static analysis frame work wherein binary is first disassembled using a disassembler. The disassembler helps in identifying the function as well as variable boundaries in the binary disassembly. This information is stored in files which are later given as inputs to binary program analysis tool, "BinAnalysis", built on top of Rose framework. It computes the control and data dependency relationships and provides API for querying the computed dependency information given an instruction address. We outline the details of the framework in the following discussion.

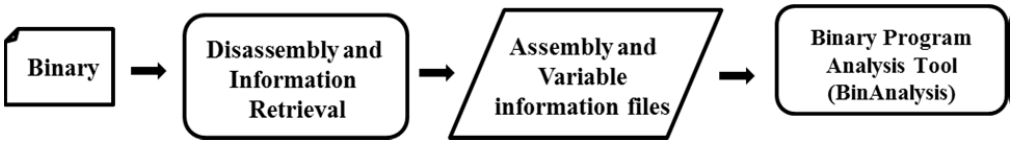


Figure 6-10. Binary Static Analysis Framework Overview

6.2.1 Disassembly and Variable Information

The first stage of binary analysis involves function boundary identification and variable information gathering. Although the binary analysis framework provided by Rose too has provisions for function identification it does not analyze disassembled code to identify possible variables. We therefore used IDA Pro to identify functions along with retrieving information pertaining variable size estimates. We wrote a plugin called *Extractor* to extract this information from disassembled binary and store them into files for later parsing by Rose. For global variables, we need address as well as size, whereas local variables and function arguments are represented by the function address, stack offsets and size. IDA Pro provides information of any structures that it identifies. So, we capture this information too through *Extractor*. Fig. 6-11 shows an example with the format used in storing information pertaining identified global and local variables.

```
<globalvariable>                    <localvariable>
<address>4264760</address>          <funaddress>4264560</funaddress>
  <size>26</size>                   <offset>-120</offset>
  <isStruct>0</isStruct>            <size>4</size>
</globalvariable >                  <isStruct>0</isStruct>
<globalvariable>                    </localvariable>
  <address>4264786</address>        < localvariable>
  <size>169</size>                  <funaddress>4264560</funaddress>
  <isStruct>0</isStruct>            <offset>-116</offset>
<globalvariable >                   <size>32</size>
                                    <isStruct>0</isStruct>
                                    </localvariable>
```

Figure 6-11. Global and Local variable information representation

Extractor generates 3 important files: global and local variable information and disassembly dump files. The variable information files are used for identifying memory accesses whereas all the disassembled function instructions are written into the disassembly dump file. These files are given as inputs to the binary static program analysis tool, *BinAnalysis*, built on top of Rose binary analysis framework. Rose interprets these disassembled functions and builds an Abstract Syntax Tree (AST) for each of the functions and subsequently constructs their CFGs.

6.2.2 Control Dependency

Second stage of binary static analysis starts with Rose framework parsing and interpreting the assembly code generated by IDA Pro to build AST and generate CFG. This is then used for calculating intra-procedural control dependency for each of the function *CFGs* built. Rose provides API for building post dominator relationships from *CFG*. This is used to find out

immediate post dominators. From this control dependency successors were calculated using standard approach described in Section 6.1.2. The call graph generated by IDA Pro may be incomplete due to indirect calls (for example, calls made using function pointers). We, therefore augment inter control-dependency as and when a call to known function is made. Fig 6-12 shows control dependency output of a disassembled program. The first dependency in the output shows that basic block with address *411284* depends on basic block with address *41127b*. The type indicates the label on the edge between the predicate and its control dependent successor, with value '0' representing the *false* and '1' representing the *true* branch labels. Basic blocks *41129c* and *4112d2* are loop headers; therefore they are shown as dependent on themselves.

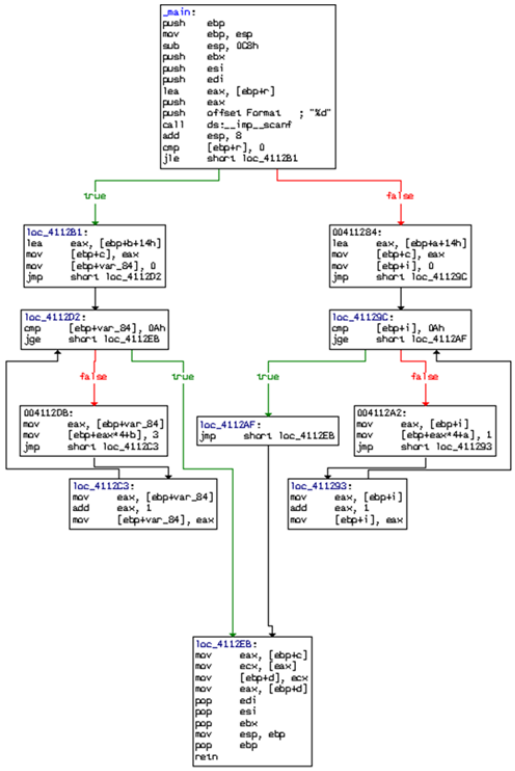


Figure 6-12. Control Dependency output

6.2.3 Data Dependency

In the third stage of our approach container level inter-procedural data dependency analysis is performed using *CFGs* and variable information retrieved from the first stage of analysis. By container level we mean that for container data accesses (e.g., arrays) our tool computes data dependency in terms of the whole container basis. For *struct* type variable, when the *struct*

information is available from IDA Pro, we apply the *struct* offsets onto the variable information to identify individual element boundaries of *struct* variable.

Rose provides policy for symbolic semantics which can emulate execution of a single basic block of x86 instructions. Each basic block is associated with a symbolic state representing the state of registers, memory and flags (except instruction pointer, which is maintained separately by the policy). Fig. 6-13 depicts the structure of a basic block's symbolic state.

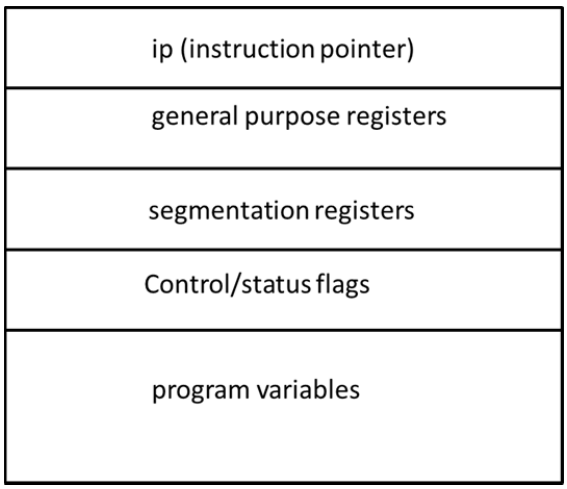


Figure 6-13. Layout of basic block's symbolic state

Yices SMT solver [165] is used to answer various questions such as when two memory addresses can alias one another. Each x86 instruction is associated with semantics to emulate how it manipulates the symbolic state.

```
case x86_add: {
        if (operands.size()!=2)
          throw Exception("instruction must have two operands", insn);
        switch (numBytesInAsmType(operands[0]->get_type())) {
          case 1: {
            Word(8) result = doAddOperation<8>(read8(operands[0]), read8(operands[1]), false, policy.false_());
            write8(operands[0], result);
            break;
          }
          case 2: {
            Word(16) result = doAddOperation<16>(read16(operands[0]), read16(operands[1]), false, policy.false_());
            write16(operands[0], result);
            break;
          }
          case 4: {
            Word(32) result = doAddOperation<32>(read32(operands[0]), read32(operands[1]), false, policy.false_());
            write32(operands[0], result);
            break;
          }
          default:
            throw Exception("size not implemented", insn);
            break;
        }
        break;
      }
```

Figure 6-14. Abstract semantics for add instruction in Rose

Consider the abstract semantics for *add* instruction in Fig. 6-14 which determines that the instruction has two operands before proceeding to symbolically evaluate it. We augmented the emulation process for whole *CFG* instead of the basic block level analysis provided by Rose. Additionally, we also modelled calls to library functions using library function prototypes. We now present other extensions done by us to those provided by Rose infrastructure.

We handled merging of states (when a basic block has more than one predecessor) by assigning a new symbolic value if the variable value in the two predecessor states is different. A mapping is maintained from the new symbolic value to the merged values it represents. If the function has more than one return block, states of all return blocks are merged to get a final return state. We used standard work list algorithm described in Section 6.1.3 for calculating intra-procedural data dependency analysis. Whenever a read or write operation is performed the dependency information is updated to reflect the *use* and *def* relationships between the program statements. Each basic block in the CFG is associated with a state and dependency information objects.

Looking back at the data dependency example in Fig. 6-6, for the pointer de-referencing and copying instruction *mov ecx, [eax]* with address *411269* our tool generates the following output as it tracks memory as well as register state to calculate dependency relations. The output shows that the instruction *411269* is dependent upon the instruction with address *411266* that moves the contents of pointer '*c*' to *eax* as well as the instruction *41125f* that defined variable '*r*' (that '*c*'

points to) with an immediate value of 10. The "*name*" in the output shows the variable (or register/flags) using which the instruction is dependent upon.

```
insn 0x411269:mov    ecx, DWORD PTR ds:[eax]
     Depends on:
     1. 0x411266:mov    eax, DWORD PTR ss:[ebp + 0xf8<-
0x08>]; name: $x86_eax; whoWrote: 9
     2. 0x41125f:mov    DWORD PTR ss:[ebp + 0xfc<-0x04>],
0x0000000a; name: (add[32] -4[32] -4[32]); whoWrote: 10
```

In order to perform value-based inter-procedural analysis, the callee function state should be set to that of the caller at the point of call and proceed as usual with intra-procedural analysis for callee. This approach though precise, is not efficient as real life applications contain many procedures and procedure calls. Efficiency and scalability assume more significance in inter-procedural analysis than intra-procedural analysis. We therefore used function summaries for handling calls, since this makes the approach to be scalable, while compromising little on precision.

Function summary is context-independent summary generated to represent the call effect (that is, there is only single summary for the function regardless of the calling context). When the summarized function is called from a caller, based on calling context and synthesized function summary, control and data dependency information is updated and propagated back to the caller. Caller resumes analysis after propagation at instruction subsequent to call.

To generate function summary, the function to be summarized is symbolically executed to record known dependencies and gather unknown references. During analysis the definitions of registers, flags, global variables, argument values and dynamically allocated variables are collected and an end state of function is generated by gathering the values of global variables, function argument values, dynamically allocated variables and *eax* from the function's return block state. Fig. 6-15 represents properties captured by the function summary.

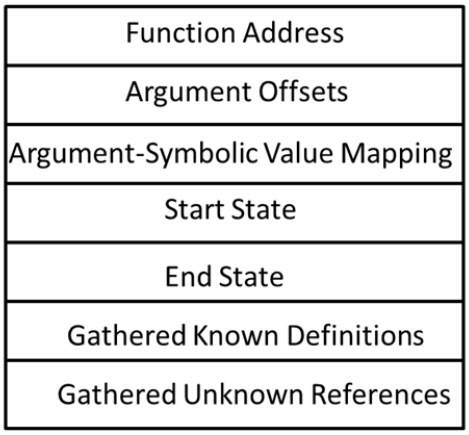


Figure 6-15. Function Summary characteristics

To apply function summary, at the call instruction in the caller function, the summary of the callee is retrieved first. If definitions of unknown references in the callee summary exist in the caller, dependencies from callee unknown reference instruction to instructions defining those in that of caller are added. If no instruction exists that define these variables/registers in the caller, they are added to unknown references of the caller summary to be propagated up in the call graph and updated when the definition is found. We traverse through the definitions generated in the callee and update their corresponding entries in caller, if they exist, or create new ones if there is no entry. State information is also updated similarly. The tool then resumes with analysis of next instruction in caller. To facilitate function summaries usage, the leaf nodes in the call graph are summarized first before going on to next level and so on.

Consider the inter-procedural example code snippet in Fig. 6-7, where the array to be initialized, its size and value with which it is to be initialized are passed as arguments to *InitArray* via call from *main* first using a direct call at node 7 and later via call through function pointer *foo* at node 9 with arrays '*a*' and '*b*' as arrays to be filled at respective calls. Fig. 6-16 shows the disassembly of the instructions from source code statements 6 through 10 on the left side and CFG with disassembled instructions of *InitArray* on the right side to exemplify the function summary usage.

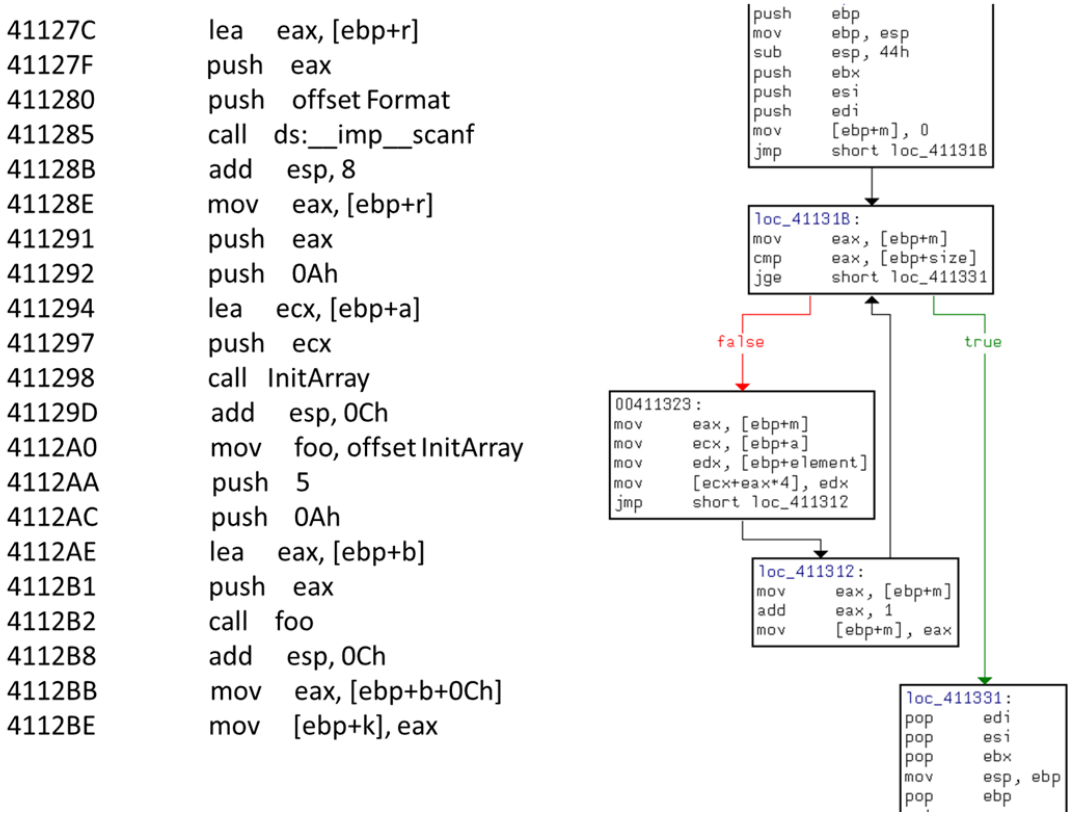


Figure 6-16. Inter-procedural data dependency analysis example

Consider the first call to *InitArray* at instruction address 411298. The arguments to *InitArray* are pushed from right to left in C source code onto the stack at instructions *411291*, *411292* and *411297* corresponding to '*r*', immediate value of 10 and effective address of local variable '*a*'. Since the function summary approach works from bottom-up of the call-graph, the summary of *InitArray* is computed first. As function summary is context-independent, during the process of summary generation we begin with an initial symbolic state for the function to be summarized and all its associated arguments. The number of function arguments is retrieved using the function stack frame information obtained from IDA Pro. We insert symbolic memory variables representing the function arguments during the function prologue. The first instruction of *InitArray* is the function prologue instruction, *push ebp*, which reads and pushes the value of previous stack frame pointer onto the stack. This instruction actually is dependent upon the instruction which prior defined *ebp* in the caller. As the dependency cannot be retrieved using current known information, it is added to unknown references of callee. This helps to populate the dependency when the calling context becomes available while analyzing the callee *main* during its summary generation. The next instruction in *InitArray* is *mov ebp, esp*. Since the *push ebp* instruction before this instruction modified *esp* register by decrementing it by 4, the tool adds dependency for this as

it can be derived from current state and definitions. Later in the analysis, when the comparison between local variable '*m*' and function argument '*size'* is made in second instruction of basic block with address *411318*, i.e, 41131e: *cmp eax, [ebp + size]*, as the definition for function argument '*size'* is not available, it is added to unknown references similarly. When write is made to the array element at instruction: *41132c:mov [ecx + eax*0x04], edx* it is identified as writing to the memory address with function argument value and is hence recorded in the definitions generated in the basic block and propagated down the CFG. At the end of analysis, the state and definitions of *eax*, global, dynamically allocated variables and function argument values (i.e., arguments passed as pointers) and unknown references are collected for generating the function summary.

When *main* is being analyzed, at instruction *411298* and later at *4112b2*, call to *InitArray* is made. Then the tool retrieves the prior generated summary of *InitArray* to add dependency of unknown references if information pertaining them is available. For the first instruction *push ebp* in *InitArray* the tool generates the following output:

```
insn 0x411300:push   ebp
     Depends on:
     1. 0x411298:call   0x00411005; name: $x86_esp; whoWrote:
9
     2. 0x411271:mov    ebp, esp; name: $x86_ebp; whoWrote: 9
     3. 0x4112b2:call   DWORD PTR ds:[0x00416120]; name:
$x86_esp; whoWrote: 9
     4. 0x411336:pop    ebp; name: $x86_ebp; whoWrote: 9
```

Since a *call* instruction in *main* pushes the return address onto the stack and *push* instruction in *InitArray* reads the stack pointer, *esp*, prior to pushing the previous frame pointer onto the stack at *411300: push ebp* instruction on *InitArray*, it is dependent on call instructions to *InitArray* at *411298* and *4112b2* using *esp* register. Another point to note here is that, while during the first call the read to *ebp* is dependent on *main* function prologue instruction with address *411271* defining *ebp*, during the second call, before *InitArray* returns to *main*, it writes *ebp* with previous frame pointer value popped from the stack at *411336: pop ebp* instruction. Therefore *411300: push ebp* instruction is dependent upon instructions *411271* and *411336* due to *ebp* reads. When a function is called multiple times, the information from different calling contexts must be merged to identify the final dependencies. Parameterized summary approach provides means for doing this. The following gives the tool output for read to function argument '*element*' at *411329*.

```
insn 0x411329:mov    edx, DWORD PTR ss:[ebp + 0x10]
     Depends on:
     1. 0x411301:mov    ebp, esp; name: $x86_ebp; whoWrote: 9
     2. 0x411291:push   eax; name: -204[32]; whoWrote: 10
     3. 0x4112aa:push   0x05; name: -200⌈32⌉; whoWrote: 10
```

Now consider the case where writes to variables defined in the callee are propagated up in the caller. Since array '*b*' is defined in *InitArray*, when a read to its third element is made at *4112bb* in caller *main*, as the tool updates the caller's dependencies and state information for variables defined and propagated upward in the call graph, at the call point using the generated summary, it is able to track the dependency to give the following output:

```
insn 0x4112bb:mov    eax, DWORD PTR ss:[ebp + 0xb8<-0x48>]
     Depends on:
     1. 0x411336:pop    ebp; name: $x86_ebp; whoWrote: 9
     2. 0x41132c:mov    DWORD PTR ds:[ecx + eax*0x04], edx;
name: (add⌈32⌉ -4⌈32⌉ -84⌈32⌉); whoWrote: 10
```

Thus it can be seen that function summaries are scalable and effective means of performing dependency analysis. To summarize, in this section we presented the details of the binary static analysis tool and the approximations carried out in our program analysis. It can be deduced from the discussion that given the lack of high-level information and intrinsic complexity and impreciseness associated with binary analysis, it is not possible to use static analysis alone for inferring BO vulnerabilities precisely from executables. Therefore, using machine learning and static analysis together provides an alternative yet important means of handling this issue.

6.3 Static Code Attributes

In this section, we characterize possible defenses implemented in the code for preventing buffer overflows and propose static code attributes for representing them. BO prediction models are then built using this collected attribute data. We use the control and data dependencies of potential vulnerable statement computed by the binary analysis tool to collect the data. Static analysis information is computed using *CFG* of the disassembled binary. A node in the *CFG* represents a program statement in disassembled binary. Before further discussion we introduce few terms used in the chapter.

We define sink *k* as a potential vulnerable program statement in binary disassembly such that the execution of *k* may lead to unsafe operation if values of variables referenced at *k* are not

limited properly. We treat calls to C library functions and statements that write to containers as sinks in this chapter.

A predicate node *d* in the *CFG* is called an input validation node of sink *k* if:

i. Both *k* and *d* transitively reference to a common input variable defined at the same node.

ii. *k* is control dependent on *d*.

A predicate node *p* in the *CFG* is called an input dependent predicate node of sink *k* if:

i. *p* refers to an input variable that *k* is not directly or transitively data dependent upon.

ii. *k* is control dependent on *p*.

Input dependent predicate nodes are important in depicting that some constraints on non-sink inputs are to be met before reaching the sink.

For ease of understanding we present a source code example in Fig. 6-17 with BO sinks at nodes 6, 12 and 17 to explain the characterization scheme. *Source* refers to variables referenced at sink and *destination* refers to variable (i.e., buffer) defined at sink. It should be noted that the *source* and *destination* can represent more than one variable (e.g., function arguments passed by reference).

6.3.1 Input and Source Characterization

To counteract input-related exploits, it is important to enumerate external inputs that the sink is data dependent upon and study their characteristics. We treat data submitted using command line, read from external files or values from environment variables as input.

1) *Input Count*: This attribute counts the number of sink inputs. This attribute value is set to 1 for sink at node 12 in Fig. 6-17, as it is data dependent on input node at 17.

2) *Inputs with Limiting*: This attribute counts the number of sink inputs which have been received by imposing length restrictions. It is relevant in flagging that buffer sizes were considered when receiving external inputs. This attribute value is set to 0 for sink at 12 in Fig. 6-17 as input at 17 does not impose any length restrictions.

3) *Environment Dependent Input*: This attribute counts the number of sink inputs that access data by reading environment variables and indicates that sink is dependent upon them apart from external inputs.

4) *Is Source a Buffer*: This attribute is used to identify if *source* is a buffer or not. It can have one of the two values, namely, *TRUE* (=1) or *FALSE* (=0). This is set to '1' for sink at 12 in Fig. 6-17, whereas sink at 6 has this attribute value set to '0'.

5) *Is Source Null Terminated*: Non-null terminated buffers may cause BO in some program constructs. For example, consider vulnerable C library function like *strcpy* which looks for terminating null character in a *source* string and copies it to the *destination* including the terminating *null* character. Hence, it is important that strings in the program be null-terminated before further access. We therefore use this attribute to reflect this defensive measure. This attribute can have one of the three values. It is *FALSE* (=0) if none of the variables represented by *source* buffer is null-terminated at least once in the program, *TRUE* (=1) when the *source* buffer (or all the variables represented by it) are null-terminated at least once in the program and *SOMETIMES* (=2) when only some of the variables represented by *source* buffer are null-terminated. This value is set to 2 for sink at 12 in Fig. 6-17 because one of the buffers that the *source* (*srcStr*) points to is null terminated (since *gets* null-terminates *localBuf* at 17) whereas the other (i.e., *globalBuf*) is not null-terminated.

```
1.  # define BUFSZ 32
2.  char globalBuf[BUFSZ];
3.
4.  void initializeBuf(char *bufToFill, size_t sz) {
5.  for(size_t count =0; count < sz; count++)
6.  bufToFill[count] = 'a';//OK
7.  }
8.
9.  void copy(char *srcStr) {
10. char dstStr[BUFSZ];
11. if(strlen(srcStr) < BUFSZ)
12. strcpy(dstStr, srcStr); //OK
13. }
14.
15. int main(int argc, char **argv) {
16. char localBuf[BUFSZ];
17. gets(localBuf);//BAD
18. initializeBuf(globalBuf, BUFSZ);
19. if(argc > 1) {
20. if(strcmp(argv[1],"Global") == 0)
21. copy(globalBuf);
22. else copy(localBuf);
23. }
24. … }
```

Figure 6-17. Example code to explain attributes characterization

6.3.2 Sink Characteristics

Depending on type of operation, sinks, may need diverse forms of bounds checking mechanisms and defensive measures. For example, a *strcpy* call needs to be preceded by bounds check to see if the destination can accommodate source string, whereas lengths of both *source* buffer as well as *destination* should be compared with *destination* size before a string concatenation operation. Some sinks like *strncpy* have in-built BO defense mechanisms to help limit buffer filling operations. We use attributes capturing these defense measures in our sink characteristics classification.

1) *Operation Type*: We classify the sinks based on the operation being performed into one of the following five types:
    1. *Copy*: Calls to C library functions which perform copying operations (e.g., *strcpy*, *memcpy*).
    2. *Concatenation*: Calls to C library functions which perform concatenation operations (e.g., *strcat*, *strncat*).
    3. *Formatted Write*: Calls to C library functions which perform formatted write operations (e.g., *sprintf*, *snprintf*).
    4. *UnFormatted Write*: Calls to C user input library functions which perform unformatted write operations (e.g., *gets*, *fgets*).
    5. *Array Write*: All other non -library call statements that write to containers (e.g., sink at node 6 in Fig. 6-17).
2) *Has Defensive Limiting*: This attribute is used to flag usage of secure versions of C library functions where number of elements for filling destination is specified in the sink statement (e.g., *strncpy*, *snprintf*). It can have one of three values namely, *FALSE* (=0), *TRUE* (=1), and *NOT-APPLICABLE* (=2). It is set to *NOT-APPLICABLE* for *Array Write* sinks. This attribute has value of '0' for sinks at 12 and 17 in Fig. 6-17.

6.3.3 Destination Buffer Characterization

In this proposal we study the destination usage patterns in predicting BO vulnerabilities. We therefore propose the following attributes to capture destination characteristics.

1) *Is Destination Null Terminated*: The purpose of this attribute is similar to *Is Source Null Terminated* attribute in *Input and Source Characterization* and likewise can have one of the three values. It is *FALSE* (=0) if none of the variables represented by *destination* is null-terminated at least once in the program, *TRUE* (=1) when the

destination (or all the variables represented by it) are null-terminated at least once in the program and *SOMETIMES* (=2) when only some of the variables represented by *destination* are null-terminated. This attribute value is set to 1 for sinks at 12 and 17 in Fig. 6-17.

2) *Multiple Writes to Destination*: This attribute is used to gather information pertaining existence of multiple writes to destination. When multiple writes exist, checks should be made on available space and care should be taken that further filling operations do not overflow buffer. It can have one of the three values. It is *FALSE* (=0) if none of the variables represented by the *destination* was written more than once in the program, *TRUE* (=1) when the *destination* (or all the variables represented by it) are written more than once in the program and *SOMETIMES* (=2) when only some of the variables represented by *destination* are written more than once in the program.

3) *Location*: This attribute depicts the buffer location and can have one of the following four values namely:

1. *Global*: *Destination* is a global variable.
2. *Local*: *Destination* is a local variable. Sink node at 12 in Fig. 6-17 is classified as this type.
3. *Mixed*: *Destination* is represented by global, local or heap variables.
4. *Heap*: *Destination* is a heap variable.

4) *Same Source and Destination Size*: In some coding constructs both *source* buffer and *destination* have same size and *source* buffer size is taken into consideration while filling it before it is copied to the *destination*. Hence we use this attribute to capture such defensive measures. It can have one of the three values namely *TRUE* (=1) when the variables represented by *destination* and *source* have same size, *FALSE* (=0) when the *destination* and *source* do not have same size even once and SOMETIMES (=2) when otherwise. This attribute value was set to '2' for sink at 12, since the IDA Pro plugin identifies the size to be 40 instead of 32 for global variable due to holes in compiled code.

5) *Declaration*: Buffer declaration is used as defensive measure in some coding constructs where *destination* is allocated based on *source* buffer before filling operation. We therefore include this attribute to gather such information. It can have one of the following 5 values:

1. *Static*: *Destination* is declared statically. All the sinks in Fig. 6-17 are classified as this type.
2. *Dynamic Source Dependent*: *Destination* is declared dynamically and the declaration statement is data dependent upon sink *source*.
3. *Dynamic Source Independent*: *Destination* is declared dynamically and the declaration statement is not data dependent upon sink *source*.
4. *Dynamic Constant Size*: *Destination* is declared dynamically using constant size.
5. *Mixed*: If *destination* is declared through more than one of the four above mentioned types, then it is set to this value.

6.3.4 Control Dependency Characteristics

It is important to carry out bounds checking operations before filling buffers to prevent overflows. Therefore predicates checking for buffer size and length are important in predicting BO vulnerabilities. In addition to this, all inputs should be thoroughly validated before propagation to safeguard the system from input-manipulation attacks. We therefore include attributes depicting control dependency characteristics of sink to capture this information.

1) *String Length of Source*: This attribute counts the number of sink predicate dependent nodes that refer to string length of buffer referenced at sink, *k* via call to *strlen*. Node 11 in Fig. 6-17 is performing this check for sink at node 12.
2) *Size of Source*: This attribute counts the number of sink predicates that refer to size of buffer referenced at *k*. Node 11 in Fig. 6-17 is classified as performing this check for sink at 12 as both *source* buffer and *destination* have same size.
3) *String Length of Destination*: This attribute counts the number of sink predicate dependent nodes referring to string length of buffer defined at *k* via call to *strlen*.
4) *Size of Destination*: This attribute counts the number of sink predicates that refer to size of buffer defined at *k*. Node 11 in Fig. 6-17 is performing this check for sink at 12.
5) *Source Dependent Loop Termination*: Buffers are sometimes filled in loops. If the loops are terminated by taking into account only *source* without buffer size the application could be susceptible to overflows. To examine such constructs we include this attribute. It is *TRUE* (=1) when sink statement is in a loop and loop termination directly or transitively refers to sink *source*, *FALSE* (=0) when otherwise and *NOT-APPLICABLE* (=2) when sink statement is not in loop. This is set to '1' for sink at 6 in Fig. 6-17 due to *count*, which is a sink *source*.

6) *Validation*: This attribute counts the number of sink validation nodes. Node 11 is the validation node for sink at 12 in Fig. 6-17 and hence this attribute value is set to '1' for this sink.
7) *Input Dependent Predicates*: This attribute counts the number of sink input dependent predicates. It is set to '2' for sink at 12 due to input predicates at 19 and 20 in Fig. 6-17.
8) *Source Buffer Predicates*: This attribute counts the number of sink predicates which are directly or transitively dependent on buffer referenced at *k* and helps in inferring on number of checks on sink *source* buffer.
9) *Destination Buffer Predicates*: This attribute counts the number of sink predicates which are directly or transitively dependent on buffer defined at *k* and helps to infer on checks performed on *destination*.

6.3.5 Data Dependency Precision Characterization

It can be seen from Section 6.2.3 that container-level precision and function summaries introduce some imprecision in data dependency algorithm as trade-off for scalability. We therefore include attributes that capture this imprecision and observe their effect in predicting vulnerabilities. We introduce few terms here pertaining data dependency ambiguity.

If statement *'s'* writes to container *'c'*, and if *'c'* has writes to it such that at least one write is data dependent on an input variable and there exists another that is not data dependent on any input variable, then container *'c'* is termed as ambiguous. Conversely, if writes to *'c'* are not ambiguous with respect to input then *'c'* is termed as non-ambiguous. If either all the writes to *'c'* are data dependent on some input, or if all of them are independent of input or if *'c'* is only written once in the program then it is considered non-ambiguous. The following attributes are used to capture data dependency precision ambiguities:

1) *Inter Procedural Destination Buffer Access*: This attribute is used to capture effect of function summaries on destination dependencies. It is *TRUE* (=1) if *destination* is accessed and written inter-procedurally in the program and *FALSE* (=0) if vice-versa. This is set to '1' for sink node at 6 in Fig. 6-17.
2) *Number of Destination Writes*: This attribute counts the total number of writes to variables represented by the *destination* other than the write at sink node.
3) *Is Destination Buffer Ambiguous*: This attribute is set to *TRUE* (=1) if *destination* is ambiguous container and *FALSE* (=0) if vice-versa. This attribute value for sink at 12 in Fig. 6-17 is set to '0' since there is only one write to *destination*.

4) *Is Source Ambiguous*: This attribute is set to *TRUE* (=1) if *source* buffer is ambiguous container and *FALSE* (=0) if vice-versa. It is set to *NOT-APPLICABLE* (=2) when there are no writes to *source* (e.g., global variables initialized at declaration, constant sources). This attribute takes on value of '2' for sinks at 6 and 17 in Fig. 6-17.
5) *Sink Dependent Ambiguous Containers*: This attribute counts the number of ambiguous containers that the sink is directly or transitively dependent upon.
6) *Sink Dependent Non-Ambiguous Containers*: This attribute counts the number of non-ambiguous containers that the sink is directly or transitively dependent upon.
7) *Predicates Referring To Ambiguous Containers*: This attribute counts the number sink predicates that refer to ambiguous containers.
8) *Predicates Referring To Non-Ambiguous Containers*: This attribute counts the number sink predicates that refer to non-ambiguous containers. This is set to '1' for sink at 12 due to the predicate at 11.

All non-numeric attributes like those representing *TRUE*, *FALSE*, *NOT-APPLICABLE* are declared as nominal in our dataset and are treated as such by WEKA algorithms despite their numeric values. Table 6-1 summarizes the proposed 30 static code attributes inclusive of target attribute-"*Vulnerable*?". "Nom" in the attribute description entry of Table 6-1 indicates nominal data type whereas "Num" indicates numeric data type attribute.

Table 6-1. Static Code Attributes for Binary BO Prediction

Attribute Name	Description
Input Count	Number of sink inputs (Num)
Inputs with Limiting	Number of sink inputs received by length restrictions (Num)
Environment Dependent Input	Number of sink inputs which access data by reading from environment variables (Num)
Is Source a Buffer	Set to '1' if source is a buffer, '0' if otherwise (Nom)
Is Source Null Terminated	Set to '0' if none of the variables represented by source buffer is null-terminated at least once in the program, '1' if vice-versa, '2' when only some of the variables represented by source buffer are null-terminated (Nom)
Operation Type	Set to '1' for Copy, '2' for Concatenation, '3' for Formatted Write, '4' for Un Formatted Write and '5' for Array Write operations (Nom)
Has Defensive Limiting	Set to '1' for library function sinks where number of elements for filling destination is specified, '0' if vice-versa and '2' for array write sinks (Nom)
Is Destination Null Terminated	Set to '0' if none of the variables represented by destination is null-terminated at least once in the program, '1' if vice-versa, '2' when only some of the variables represented by destination are null-terminated (Nom)
Multiple Writes to Destination	Set to '0' if none of the variables represented by the destination was written more than once in the program, '1' if vice-versa, '2' when only some of the variables represented by destination are written more than once in the program (Nom)
Location	Set to '1' for Global, '2' for Local, '3' for Mixed and '4' for Heap (Nom)
Same Source and Destination Size	Set to '1' when the variables represented by destination and source have same size, '0' when the destination and source do not have same size even once or '2' when it happens sometimes (Nom)
Declaration	Set to '1'for Static declaration of destination buffer, '2' for Dynamic Source Dependent, '3' for Dynamic Source Independent, '4' for Dynamic Constant Size or '5' for Mixed declarations (Nom)
String Length of Source	Number of sink predicate dependent nodes that refer to string length of buffer referenced at sink (Num)
Size of Source	Number of sink predicates that refer to size of buffer referenced at sink (Num)
String Length of Destination	Number of sink predicate dependent nodes referring to string length of buffer defined at sink (Num)
Size of Destination	Number of sink predicates that refer to size of buffer defined at sink (Num)
Source Dependent Loop Termination	Set to '1' when sink is in a loop and loop termination directly or transitively refers to sink source, '0' when otherwise and '2' when sink is not in loop (Nom)
Validation	Number of sink validation nodes (Num)
Input Dependent Predicates	Number of sink input dependent predicates (Num)
Source Buffer Predicates	Number of sink predicates that are directly or transitively dependent on buffer referenced at sink (Num)
Destination Buffer Predicates	Number of sink predicates directly or transitively dependent on buffer defined at sink (Num)
Inter Procedural Destination Buffer Access	Set to '1' if destination is accessed and written inter-procedurally in the program or '0' if vice-versa (Nom)
Number of Destination Writes	Number of writes to variables represented by the destination other than the write at sink (Num)
Is Destination Buffer Ambiguous	Set to '1' if destination is ambiguous, '0' if vice-versa (Nom)
Is Source Ambiguous	Set to '1' if source buffer is ambiguous container, '0' if vice-versa or to '2' when there are no writes to source (Nom)
Sink Dependent Ambiguous Containers	Number of ambiguous containers that sink is directly or transitively dependent upon (Num)
Sink Dependent Non-Ambiguous Containers	Number of non-ambiguous containers that sink is directly or transitively dependent upon (Num)
Predicates Referring To Ambiguous Containers	Number of sink predicates that refer to ambiguous containers (Num)
Predicates Referring To Non-Ambiguous Containers	Number of sink predicates that refer to non-ambiguous containers (Num)
Vulnerable?	Target attribute: Vulnerable (=TRUE) or Non-Vulnerable (=FALSE) (Nom)

6.4 Experiments

In this chapter, we build classifier models for predicting BO vulnerabilities from binaries. Since we predict buffer overflows, calls to string or memory block manipulation functions like *strcpy*, *memcpy* are regarded as sinks. All container writes are also treated as sinks since we do not have type information for container to determine if they are *"char"* arrays or not. For each of the sinks identified, our tool collects attributes as per the BO characterization scheme proposed in Section 6.3 using the information obtained from static analysis tool. In this experiment we evaluated four well-known classifiers on dataset with known vulnerability information.

We used off-the-shelf tools like WEKA and IDA Pro to facilitate adoption and built-upon existing tools like ROSE binary analysis infrastructure to help replication. Fig. 6-18 shows the overview of the vulnerability prediction system. We used IDA Pro for disassembling binaries. IDA Pro provides access to its internal data structure through API enabling users to write plugins which can be executed by IDA Pro. We wrote an IDA Pro plugin called *Extractor* which writes the disassembly of all the functions identified by IDA Pro in the executable to a file. *Extractor* also gathers variable address/stack offset and size information and data segment boundary addresses and writes them to Xml files. These files are inputs to our *BinAnalysis* tool built on top of ROSE binary analysis framework.

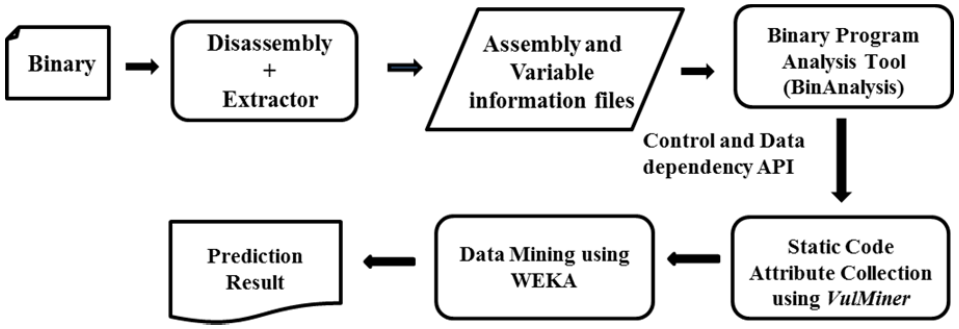


Figure 6-18. Overview of Vulnerability Prediction System

BinAnalysis is a static program analysis tool for computing control and data dependencies. While performing static analysis, *BinAnalysis* simultaneously identifies sinks and collects data about buffers (i.e., containers) defined at sink. To identify buffers, the tool uses library function call semantics and variable access information (e.g., index based addressing mode) collected during static analysis. *VulMiner* uses the API of *BinAnalysis* to extract code attributes. This dataset is then modified to add sink vulnerability information and is given as input to WEKA data mining tool for training and testing BO prediction models.

6.4.1 Benchmark

For our experimental evaluation we used applications from MITLL benchmark [121]. Table 6-2 gives the statistics of 6 programs used in the evaluation. As the programs are designed to be self-contained, 'b1' and 'b2' programs include code to generate input. We modified it in line with 'b3' to receive input and also removed a user-defined function (*newstr*) which is not called by any other user-defined function in the program. LOC column in Table 6-2 represents the program size in C source code and Sinks column represents the number of potential vulnerable statements identified by binary static analysis tool. The number of sinks in binary dataset is higher than actual buffer writes in the source code since IDA Pro could not identify few *struct* types in some binaries and hence all writes to *struct* variables were taken to be sinks. Given that we deal with binaries which lack such information, this is to be expected.

Table 6-2. Benchmark statistics

Name	LOC	Sinks	Bugs
b1(NXT-Bug)	1.15K	32	1
b2(SIG-Bug)	1.39K	40	1
b3(Iquery)	0.28K	10	1
f1(mapped chdir)	0.42K	21	4
f2(off-by-one)	0.94K	24	3
f3(realpath)	1.01K	57	19

6.4.2 Data Collection

BinAnalysis identifies sinks while performing data dependency analysis. *VulMiner* queries transitive control and data dependencies along with information collected during static analysis for each of the identified sinks to extract attribute values based on BO defense characterization scheme described in Section 6.3. This is then modified to add known sink vulnerability information retrieved manually to prepare the final dataset. The dataset thus prepared is given as input to data mining tool for building vulnerability prediction models.

For each sink in the program, *VulMiner* extracts the attribute values as per characterization scheme described in Section 6.3. There are, in total, 30 attributes consisting of "*Input Count*" and 28 other attributes along with the target attribute -"*Vulnerable*?". Hence, the attribute vector for sink node at 12 in Fig. 6-17 is (1,0,0,1,2,1,0,1,0,2,2,1,1,1,0,1,2,1,2,1,0,0,0,0,0,0,2,0,1,*FALSE*) corresponding to (*Input count, Inputs With Limiting, .., Operation Type,.., Declaration, String Length Of Source Buffer,.., Predicates Referring To Non-Ambiguous Containers, Vulnerable?*).

6.4.3 Experimental Design

There are in total 184 sinks, out of which 29 are vulnerable. We used standard 10-fold CV on the dataset prepared as outlined in Section 6.4.2.

6.4.4 Classifiers

To assess our approach we used four well-known machine learning algorithms, each with different principles and yielding different prediction models in the WEKA data mining tool kit: naïve Bayes (NB), Multi-Layer Perceptron (MLP), Simple Logistic (SL) and Sequential Minimum Optimization (SMO). These algorithms use features or attributes to predict a class (in our context BO vulnerabilities). The details of the classifier algorithms were presented in Chapter 2. Similar to hybrid source-code based auditing approach in Chapter 5, which used default parameter settings set by WEKA for building classifier models, here too we do not attempt to tune the parameter values to optimize any classifier. We used MLP and SMO classifiers with same default parameter values as in Chapter 5. The settings for MLP are:

```
No. of hidden layer = 1;

No. of nodes per hidden layer = (no. of attributes + no. of classes)/2;

trainingTime = 500;

learningRate = 0.3;

momentum = 0.2;
```

The settings for SMO are:

```
C = 1.0;

epsilon =  1.0E-12;

kernel = PolyKernel;
```

Naïve Bayes is a statistical classifier based on Bayes theorem which uses training data to estimate the posterior probability of the test instance belonging to each of the target classes (in our case: *Vulnerable* and *Non-Vulnerable*) and assigns it to the class with highest probability. As with others, here too we used default parameter values set in WEKA given by:

```
useKernelEstimator = False;

useSupervisedDiscretization = False;
```

If set to 'True', the first parameter would allow usage of a kernel estimator for numeric attributes instead of normal distribution. We used the default value instead, which uses normal distribution. Similarly, setting the "`useSupervisedDiscretization`" to 'True' enables supervised discretization for converting numeric attributes to nominal ones, but again we use the default setting.

Simple Logistic classifier is a decision tree with logistic regression functions at its leaves. It uses LogitBoost with simple regression functions for fitting the logistic regression models. It is an iterative algorithm and the stopping criterion can be specified by using a fixed value or by cross-validation option. To build the SimpleLogistic model the following vital parameters have to be specified:

```
errorOnProbabilities = False;
heuristicStop = 50;
maxBoostingIterations = 500;
numBoostingIterations = 0;
useCrossValidation = True;
```

Parameter "`errorOnProbabilities`" when set to 'True' uses error on the probabilities as error measure to determine the number of iterations. When "`heuristicStop`" is greater than zero, the LogitBoost iterations are stopped if no new error minimum has been reached in the last "`heuristicStop`" iterations. Parameter "`maxBoostingIterations`" allows end-user to set the maximum number of iterations while "`numBoostingIterations`" is used to set a fixed number of iterations for the LogitBoost. "`useCrossValidation`" when set to 'True' indicates that the optimal number of iterations to perform is to be determined by cross-validation. As mentioned previously, we do not fine tune the parameter values for building the classifier models to optimize any classifier as our objective is to highlight the predictive capabilities of the proposed static code attributes by using the default parameter settings.

6.5 Evaluation

6.5.1 Objective

The objective of evaluation is to answer the following research questions:

1. *"Is proposed approach combining machine learning and static analysis effective in predicting vulnerable sinks?"*

2. *"How do binary BO prediction models compare with source code static analysis tools?"*
3. *"Which static code attributes are best indicators of BO vulnerabilities?"*

We used the performance measures that were previously used in evaluating source code based hybrid auditing models for evaluating the binary prediction models too.

6.5.2 Results

The focus of our experiment is to evaluate the accuracy of proposed approach. Different prediction models were used as it cannot be known prior which model will yield better performance. The result of all classifiers using 10-fold CV on dataset prepared as described in Section 6.4.3 is depicted in Table 6-3.

Table 6-3. Performance Measures for Classifiers

Model	***pd***	***pf***	***pr***	***acc***
NB	89.7	15.5	52	85.32
MLP	75.9	5.2	73.3	91.85
SL	62.1	2.6	81.8	91.85
SMO	75.9	2.6	84.6	94.02

It can be seen from Table 6-3 that tested classifiers had a recall over 60%, which means that 6 out of 10 actual vulnerabilities were predicted correctly. Except NB which had an accuracy of 85.32% all the classifiers tested had accuracy over 90%. While accuracy of NB is lower than rest, it has the highest recall rate of 89.7% along with highest false alarm probability of 15.5%.

SMO reported the best performance with 75.9% recall and 84.6% precision suggesting that more than 7 out of 10 bugs were predicted correctly and more than 8 out of 10 statements predicted to be vulnerable are indeed vulnerable. This could be due to the fact that SMO builds binary support vector machine, which constructs a hyper-plane to maximally separate instances belonging to different classes. MLP also had recall of 75.9% but had lesser precision than SMO on the dataset. The results suggest that proposed static analysis cum machine learning approach is effective in predicting vulnerable sinks from disassembled binaries.

6.5.3 Comparison with Static Source Code Analysis Tools

In this section we compare performance of our prediction models with that obtained by static source code analysis technologies using the results reported in [146]. We give the performance measures of the tools with positive recall in Table 6-4 to provide an insight of their functioning on

the benchmark programs. It can be seen from Table 6-4 that Splint [13] has highest recall of all the tools followed by PolySpace [145]. BOON [9] could detect only one bug in all the 6 tested programs. UNO [144] and ARCHER [10] could not detect any bugs but they also did not raise any false warnings in patched versions of buggy programs. Hence their performance measures were not included in Table 6-4 as recall is zero. Splint and PolySpace also have high false alarm probabilities and none of the tested tools had a *precision* over 60%.

Table 6-4. Performance Measures for Static Source Code Analyzers

Tool	***pd***	***pf***	***pr***	***acc***
Splint	61.11	40.54	59.46	60.27
PolySpace	55.56	48.65	52.63	53.42
BOON	2.78	2.70	50.00	50.68

These results answer our second research question by showing that our binary BO classifiers are effective in predicting BO vulnerabilities with a recall higher than 62%, which is better than the best performing static source code analysis tool, Splint, with recall of 61.11%. Our best classifier, SMO, has a notable recall of 75.9%. It can be inferred from the above discussion that using static analysis alone to detect BO bugs from binaries could be difficult and that in the absence of source code, our approach using prediction method and static analysis can be considered as an effective and practical alternative to BO detection.

6.5.4 Attribute Ranking using InfoGainAttributeEval in WEKA

To answer our third research question we ranked the attributes using *InfoGainAttributeEval* attribute selection algorithm in WEKA. Fig. 6-19 depicts the outcome of attribute information gain ratio for 15 best ranked attributes. It can be seen from the figure that, as expected, attributes representing defense practices like input validation, usage of safe C library functions, sink operation type and source and destination buffer predicates along with destination size checks are important in BO prediction. Presence of data dependency ambiguity attributes suggests that container-level precision also effects prediction.

```
0.27786     Validation
0.19146     Predicates  Referring  To  Non-Ambiguous
Containers
0.18436     Has Defensive Limiting
0.17212     Predicates   Referring   To   Ambiguous
Containers
0.14389     Operation Type
0.14296     Size of Destination
0.12873     Sink Dependent Non-Ambiguous Containers
0.12656     Source Buffer Predicates
0.12286     Location
0.11937     Declaration
0.10353     Is Source Null Terminated
0.09236     Same Source And Destination Size
0.09044     Is Destination Null Terminated
0.08503     Destination Buffer Predicates
0.04561     Is Destination Buffer Ambiguous
```

Figure 6-19. InfoGainAttributeEval Outcome for 15 best ranked attributes

6.5.5 Limitations and Threats to Validity

While the proposed approach is effective in predicting BO vulnerabilities from binaries, it has certain limitations. Our approach uses machine learning algorithm for training prediction models using static code attributes extracted from binary disassembly. Although the static code attributes can be easily collected, it needs a binary analysis tool to do so and requires sufficient known vulnerability data train the prediction models. We rely on IDA Pro to identify function boundaries and variable offsets which, though significant, is not fool-proof. Our data dependency analysis introduces container-level precision (which is generally in-line with that offered by commercial source code analysis tools) and function summaries, which may cost us in precision. IDA Pro variable identification and data dependency computation may effect sink identification. In spite of these expected binary analysis impediments we believe that our approach provides an effective and practical alternative of addressing binary vulnerability detection. We used 10-fold cross validation on benchmark programs to cancel out threats to validity for evaluating the generalization abilities of our prediction models.

6.6 Conclusion

In this chapter, we proposed a novel methodology using static analysis and machine learning for predicting BO bugs from x86 executables by characterizing buffer usage patterns and defensive mechanisms implemented in the assembly code. Although vast numbers of approaches were proposed for BO detection from source code there are only few that can audit bugs from

binary executables. Since most third-party components are distributed in the form of binaries it is important to make sure that they are bug free before incorporating into the software systems. It is extremely challenging to comprehend information pertaining security from binaries. While tools like disassemblers and decompilers are available in the market to support in the process of security auditing, it needs considerable expertise to use them to extract such information. Therefore it would be highly desirable to have an automated tool that can help predict bugs from binaries.

In reality, very few binary analysis frameworks exist and more often than not, they are either proprietary in nature (thus not available for usage by public) or are associated with compilers that build the executables they can help analyze. Therefore, their application is hampered by these underlying limitations. Some frameworks use dynamic analysis information for analyzing. While binary instrumentation tools for performing dynamic analysis are available, generating inputs to reach the program paths containing sinks and building security auditing frameworks using the dynamic analysis information may not provide an inclusive accurate solution. Hence performing static program analysis is better way of handling such drawbacks.

We therefore proposed a static analysis based solution to predict BO bugs. Since there are no readily available open-source tools that can statically analyze executables, we built upon existing tools to derive the necessary functionality. We used IDA Pro's remarkable capabilities to overcome the challenges in disassembly by using its proprietary algorithms to identify program functions and variable size estimates. Since Rose framework has well-built instruction semantics to succinctly express the x86 assembly instruction behaviors and AST as well as CFG building capabilities we made use of these provisions to build our program analysis framework on top of functionality provided by Rose.

We proposed and applied a container level, parameterized and summarized approach for computing data dependencies by compromising a little on accuracy to meet our auditing needs. Our tool starts with IDA Pro disassembling the executable and *Extractor* plugin gathering and storing information required for program analysis into files. These files are then given to Rose to interpret the program functions in the disassembly to build their CFGs for control and data dependency analysis. Our *BinAnalysis* framework analyzes the CFGs to compute control and data dependency relationships. During the process it also collects information for BO auditing by identifying sink statement constructs and gathering relevant statistics for our proposed characterization. *VulMiner* later queries *BinAnalysis* to retrieve information collected and computed during static analysis for each of the identified sinks to extract attribute values based on proposed BO defense characterization scheme.

Performance measures of prediction models from our experimental evaluation using benchmark programs were encouraging. Our best classifier model had a recall of 75% with 84% precision and 94% accuracy suggesting that our proposed attributes are effective indicators in predicting BO vulnerabilities. Our approach can be used to automate bug detection in third-party components or binaries whose source code is not readily available. The algorithms and approximations suggested for program analysis can be incorporated for detecting bugs in other binary based frameworks. By outlining and implementing the binary program analysis components and characterizing BO defenses implemented in the code from binary disassembly, we enabled the prediction of BO vulnerability from x86 executables using static analysis possible. The results suggest that the classification scheme is successful in auditing bugs. Hence, we have made automated bug auditing from binaries practical and resourceful.

Chapter 7

CONCLUSIONS AND RECOMMENDATIONS

Software permeates many aspects of today's life. Securing software is therefore of greatest concern so as to ensure integrity, availability and confidentiality of the software systems. In this thesis, we have proposed three different approaches for auditing BO bugs: vulnerability testing, source-code based hybrid vulnerability auditing, and vulnerability prediction from binary executables. Each approach caters to different aspects of BO mitigation. Vulnerability testing helps to identify bugs when revealing inputs are generated. Existing test generation approaches typically use heavy-weight techniques like symbolic execution or genetic algorithms which need considerable resources for setting up or optimizing them. We therefore proposed a light-weight source-code based approach that uses static analysis information to identify input constraints for generating inputs. While statements proven vulnerable by generated inputs can be patched up, test generation schemes do not have means to infer on sinks not proven vulnerable. Therefore we proposed hybrid source-code based auditing approach which either tries to prove presence of bugs by dynamic analysis using test inputs or predicts them using classifier models built from learning static code attributes representing BO defensive counter measures applied in the code. Lastly, we proposed a vulnerability prediction approach to predict bugs from disassembled binaries to overcome the lack of solutions that cater to binary vulnerability auditing. Since the proposed code attributes in the prediction based approaches can be easily collected using static analysis tools, they provide an economical and effective alternative to sophisticated detection techniques like those using symbolic execution and constraint solving which suffer from scalability and accuracy issues.

As mentioned previously in Chapter 3, an integrated approach combining different techniques at different stages of software development is needed to handle the threats posed by BO bugs. Hence the proposed approaches can also serve as complimentary solutions for auditing BO bugs in tandem with the existing techniques. Fig. 7.1 shows such integrated system for defending from BO exploits. By applying defensive coding measures during the development phase and performing adequate input validation and size checking schemes before filling buffers it is possible to

eliminate BO bugs. During the testing and auditing phase before deployment, the proposed vulnerability testing solution in Chapter 4 could be used to generate test inputs for revealing bugs. Since this approach can only generate string inputs, this should be followed by hybrid vulnerability auditing approach proposed in Chapter 5 which takes into account other input types too for further auditing. Since IL, CC and SC rules were comprehensively covered by the test generation tool in Chapter 4; inputs generated by these rules should be ignored when using the hybrid vulnerability auditing approach subsequently. As the hybrid approach either tries to prove bugs or predicts them, the prediction results from the tool could be used to focus auditing efforts on statements predicted to be vulnerable. Static code detection tools proposed in literature such as that in [8, 10] could be used to support the process by using them to verify the status of the statements predicted vulnerable by the hybrid approach. Vulnerability prediction approach proposed in Chapter 6 could be used to predict bugs in disassembled binaries when the source code is unavailable. After deployment, run-time detection tools should be employed to intercept and detect attacks during run-time, thus, realizing the ideology of using a combination of techniques to mitigate BO vulnerabilities at various stages of occurrence.

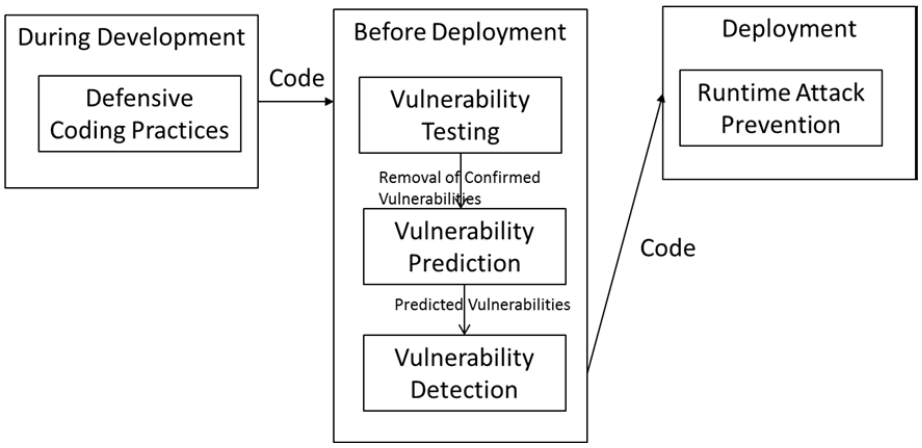


Figure 7-1. Overview of integrated use of BO mitigation techniques during software development process

This chapter is organized as follows. Section 7.1 summarizes the contributions made by this thesis towards academic research. Section 7.2 discusses the contributions made towards industry practice. Section 7.3 outlines some of the future research directions.

7.1 Contributions towards Academic Research

BO is one of the most serious vulnerabilities in the history of cyber security. Researchers have proposed wide-ranging solutions from simple static analysis to complex run-time detection and

exploit prevention solutions to handle them. Despite these, BO vulnerability reports continue to surface regularly in vulnerability databases reflecting the limitation or difficulty in adaptation of current solutions. Existing approaches suffer from one or more of the following limitations thereby preventing them from exhaustively addressing this security issue as outlined in Chapter 3:

1) *Intensive manual effort*—Defensive coding practices can help in completely eliminating the vulnerability. But the onus of adhering and incorporating them vests with the developers who have to apply them manually. Annotated and specification-based static analysis approaches need developers to specify the security policies in terms of pre- and post-conditions to ensure the buffers do not overflow. Dynamic taint analysis approaches also may require user to specify sinks and inputs to determine taint propagation for generating alerts when security policy is violated. Test case generation approaches also may need manual support to either identify inputs or optimize algorithm parameters for generating adequate test data. The end-users of such tools, typically developers, towards whom the solutions are intended, are often not well-trained in security and underlying tool usage and optimization. Therefore, these works are labor-intensive as well as error-prone.

2) *Coarse-grained analysis*—Most of existing vulnerability prediction approaches can only predict vulnerabilities unto software module or component level. This necessitates the security auditors to put in lot of manual effort to dig into them for identifying the vulnerable code sections.

3) *Inaccuracy*—Static analysis based detection approaches usually generate a lot of false positives as they tend to overestimate the buffer usage in the program. This requires too much of time and efforts in auditing the many warnings generated by the tool for determining actual bugs and thus seriously undermines the tool's effectiveness.

4) *Scalability Issues*—Most of static analysis approaches use symbolic evaluation and constraint solving to detect overflows. Modelling buffer usage in real-world programs using such algorithms could face scalability issues. Dynamic analysis based frameworks for test generation and detection are computationally expensive as they either model input effects and tweak them to direct program flow towards other paths for adequate path coverage or track input taint in the program, both of which are not scalable. This limits their adoptability in practice.

5) *Runtime overhead*—Runtime detection and exploit prevention techniques are highly efficient solutions for securing already-deployed applications. But most of them look

for specific attack patterns and can only handle those for which they are devised for. They also incur large overheads during runtime.

6) *Lack of binary auditing* —Overwhelming majority of the existing approaches in literature can only work in the presence of source code. This drawback poses severe limitations when the source code for the binaries to be audited is not available. There is also a dearth of tools and solutions that can help in binary analysis. With ubiquitous presence of viruses, worms, trojans it is just a matter of time that systems are compromised if the binaries are not tested for vulnerabilities. Automated solutions for binary bug detection are therefore highly necessary.

Based on these observations we presented three novel approaches in this thesis —rule-based test case generation, vulnerability auditing through hybrid static-dynamic analysis and binary vulnerability prediction to defend against BO vulnerabilities. Our research work mitigates the above mentioned shortcomings of existing research works. The ideas presented in this thesis have been published in [59-62, 166]. Summarizing our research work, its contributions towards academic research are:

1) *Minimal manual support*—The proposed approaches in the thesis need minimal manual support. For the rule-based test case generation, the proposed approach automatically identifies inputs and sinks to generate test cases. Except for pattern based test cases where the pattern is output as test case, all the other test inputs are generated without the need for any manual intervention or supervision to be readily used for testing. The data mining based source code and binary auditing approaches, like any other prediction approach, need users to gather and supply past vulnerability information labelled data for training the prediction models. Upon completing the training and building of prediction models, predicting future vulnerabilities is fully automated without any further need for manual intervention.

2) *Fine-grained analysis*—In contrast to existing vulnerability prediction research, all proposed approaches in the thesis perform fine-grained analysis. Our vulnerability prediction approaches characterize defensive measures implemented in the code for each potential vulnerable statement construct. As such, we can audit vulnerabilities at statement level. This improvement from its corresponding coarse-grained counterparts needed us to explore on novel set of domain-specific attributes contrary to the oft-used software attributes such as code complexity metrics to predict vulnerabilities.

3) *Accuracy*—To mitigate the accuracy issues associated with static analysis based approaches in detecting vulnerabilities, we proposed two novel solutions to cater to source code as well as binary vulnerability auditing. Firstly, we augmented hybrid source-code analysis with empirical knowledge and data mining techniques for predicting vulnerabilities, resulting in good accuracy (recall over 93% with false alarm rates less than 7%). Secondly, our proposed binary solution achieved promising results with a recall over 75% and false alarm rates less than 3%.

4) *Scalability*—Our solutions are scalable as they make use of program analysis, empirical knowledge and data mining techniques. Most symbolic execution techniques for test generation and vulnerability detection are not scalable as they suffer from path and state explosion problem. Our source code based approaches do not involve any symbolic execution or constraint solving. While our binary program analysis uses symbolic execution for computing dependency relations, we proposed and applied approximations that help scale the solution by compromising little on accuracy. Application of rule-based test generation and data mining techniques for source code and binary auditing renders the approaches to be scalable as they do not solve for path constraints or prove program correctness through program state exploration. We thus mitigated the scalability issues associated with current approaches.

5) *No runtime overhead*—The focus of our research is to detect and predict vulnerabilities from source code and binary executables. Therefore, unlike runtime detection and exploit prevention approaches, our work does not involve any intrusion detection. Thus, there are no runtime overhead costs associated with the proposed approaches in the thesis.

6) Binary vulnerability auditing—To overcome the need of binary program analysis frameworks and security auditing approaches, we proposed our binary analysis framework built using readily available tools and resources along with the approximations applied in the framework to overcome the challenges involved in binary analysis. By proposing and characterizing the defenses implemented in assembly code we enabled prediction of BO vulnerabilities from binary executables possible. The proposed binary framework can also be used to analyze binaries to support in reverse engineering.

7) *Experimental evaluation*—Using our prototype tools, we evaluated the approaches proposed in the thesis comprehensively. We were able to generate test cases for

detecting bugs in 18 of 21 benchmark programs along with reporting some undocumented bugs using our light-weight rule-based approach in Chapter 4 without the need for any symbolic evaluation or constraint solving. The dynamic analysis component of our hybrid source code auditing approach in Chapter 5 could detect 34% of known vulnerabilities along with reporting couple of unknown bugs on the benchmark programs. In all, the hybrid approach achieved near ideal results with a recall over 93% with false alarm rates less than 7% for the best classifier. Our binary bug prediction approach has a good prediction result with recall over 75% and false alarm rates less than 3% that is comparable to those obtained in other prediction studies.

Table 7-1 summarizes the contributions of BO mitigation approaches proposed in the thesis. It outlines the gaps addressed by each of them as well as their strengths and weaknesses. In summary, our overall research can be concluded as follows. The rule-based test generation approach helps in detecting bugs effectively. We started with test generation approach for mitigating bugs as it can help in revealing them straight away without the need for further auditing. As with any other test generation approach, while the proven bugs can be readily fixed not much is known about statements that are not proven vulnerable. Hence we proposed hybrid auditing approach to either prove vulnerability through dynamic analysis by partitioning input space according to the input type and validation or provide probabilistic remarks about the statement's vulnerability when it cannot be proven vulnerable to comprehensively audit each potential vulnerable statement construct in the program. Since the source code of the program may not always be available, we further proposed binary vulnerability prediction methodology to comprehensively mitigate BO vulnerabilities.

Table 7-1. Summary of Approaches Proposed in the Thesis

Proposed Approach	Strategy	Remarks
Rule-based test case generation	Uses proposed rules to generate test cases by automatically identifying program sinks and respective inputs. Static analysis information is used to identify which rule to trigger. Useful for preliminary bug analysis. Does not suffer from path or state explosion issues typically associated with symbolic evaluation techniques.	Scalable as it uses static analysis information to identify input constraints. Fine-grained, light-weight approach without run-time overheads as it does not involve any intrusion detection strategies. Needs manual support only to generate inputs for identified pattern checks. Test inputs are generated automatically for the rest. Apparent drawback is presence of false negatives and the need for the availability of source code to generate test inputs.
Hybrid vulnerability auditing using static-dynamic analysis and machine learning	Tries to prove vulnerabilities using dynamic analysis with help of test inputs generated as per prescribed rules. If a statement cannot be proven to be vulnerable, it predicts it using prediction models built from learning static code attributes characterizing BO defenses implemented in the code. Proposed code attributes can be easily collected making it economical and viable alternative.	Scalable, source-code based fine-grained methodology with good accuracy. Needs some manual intervention for proving bugs through code instrumentation and to gather past vulnerability information labelled data for training the prediction models. Can also be used as complimentary aid to first find out probable bugs and use heavy-weight approaches like formal proving or symbolic execution to determine the existence of real bugs.
Vulnerability auditing from binary executables	Characterizes and proposes static code attributes capturing BO defensive measures implemented in the assembly code for the sink to predict vulnerabilities from disassembled x86 executables.	Scalable, fine-grained approach that helps in auditing binaries. Needs minimal manual support to collate and supply past vulnerability information labelled data for training the prediction models. Upon building them the solution is fully automated. No intrusion detection is involved; hence has no run-time overheads. The approximations and approaches used in analyzing binaries may affect the prediction outcome.

7.2 Contributions towards Industry Practice

Protecting organization's reputation is of utmost importance to its stakeholders. In the perspective of software industry, customer trust and confidence are critical to the enterprise since security breaches in the developed software severely undermine business operations. The adverse effects of security breaches not only effect the organization, but spill onto the end-users of the software. Therefore software should be developed by thinking of and incorporating security sensitive measures in all stages of software development life cycle. This thesis contributes to enhancing software security measures by proposing approaches for mitigating BO vulnerabilities, a serious and common application security threat.

Thinking about security and incorporating the security measures since beginning stages of software development cycle helps in building secure software. Microsoft's Security Development Lifecycle [167] outlines the following practices for ensuring secure software building:

1) *Core Security Training*: It is important that development team is well-versed in all the security aspects of the product that may affect product's users. Therefore training the stakeholders in security information is of pivotal concern and priority.

2) *Establish Security and Privacy Requirements*: By defining security and privacy requirements early in the development life cycle, it is possible to evaluate key milestones as well as minimize disruptions.

3) *Create Bug Bars*: Defining minimum acceptable levels of security and privacy requirements at the onset of the project helps the team to understand the risks associated and plan accordingly to apply them throughout the project.

4) *Establish Design Requirements*: This phase should identify security and privacy concerns so as to minimize schedule disruptions and project expenditure in the future.

5) *Attack Surface Analysis and Threat Modeling*: This could be accomplished by modeling the perceived threats and planning mitigation approaches to reduce the risks associated with the threats.

6) *Secure Implementation*: Using approved tools and associated checks, implementing the secure coding practices learned during the security training phase (such as deprecating unsafe function usage throughout the project) and conducting code reviews ensures that secure coding practices are being followed.

7) *Secure Verification*: To ensure that secure design and implementation practices were followed during the software development, it is important to perform vulnerability and penetration testing on the implemented code.

8) *Secure Release Phase*: By creating an incidence response plan, it is possible to address new threats encountered during the software release. Secure release phase must also involve final security review and product security documentation to include information regarding security treatment to help in post-release servicing.

9) *Security Response*: When security vulnerability in the deployed product is detected, it is important to ensure that proper steps are taken for mitigating the risks associated by executing the incidence response plan and generating and distributing security patch to the product.

Software Assurance Forum for Excellence in Code (SAFECode) [168] also outlines another step, *Security Research*, for investigating on new threat vectors and researching on novel innovative technologies to mitigate them. The development teams can benefit by employing the research findings to fortify software from such attacks.

This thesis provides in-depth details about the nature of the BO vulnerability, commonly used exploit patterns along with comprehensive discussion about of the state-of-the-art techniques used in mitigating it, thereby encompassing the information needed for training the project team about the security vulnerability. Armed with this knowledge, the developers can anticipate possible threats and decide on suitable tools and methodologies to use for building secure software. The proposed prediction based vulnerability auditing approaches could help the code auditors to review the code effectively by focusing efforts on the code predicted vulnerable thus saving time and costs instead of reviewing the whole code chunk. Since the thesis proposes methodology for analyzing and auditing bugs from binary executables too, it contributes significantly to less-traversed path of binary analysis by advancing techniques for comprehending binaries, and supporting bug detection from executables. Therefore, in summary, the vulnerability information and vulnerability mitigation approaches proposed in the thesis contribute to many of the above established industry practices for building and delivering secure software.

7.3 Recommendations

In this section we outline the key research areas for future work that can enhance the work presented in this thesis.

We presented rule-based approach for test generation in Chapter 4 and Chapter 5. The proposed rules could help detect many bugs but there is still room for improvement. While the rules are primarily oriented towards string input generation, we suggest using similar techniques for other variables based on their type information and possible state perturbation for revealing bugs. For example, if the input variable is an integer, test inputs could be generated using maximum and minimum possible values for the integer type. Such inputs could help in triggering bugs where integer overflow could result in BO vulnerabilities. If the external data is read from files or network streams, rules could be included to generate malformed files or network data that violates protocol under consideration to possibly trigger bugs. The approach in Chapter 4 can detect patterns using index-based modelling. Similar measures for pointer-based patterns and indirect dependencies through program variables could be researched upon.

In our study, we have conducted several experiments to evaluate the proposed approaches using test subjects differing in their functionality with sizes ranging from relatively small to large scale. Nevertheless, the subjects used may still not be representative of different BO vulnerabilities. Hence an important future scope is to replicate and re-evaluate the approaches on other open-source as well as industrial scale applications.

Presently function-summaries are used for inter-procedural data dependency algorithm. The precision and accuracy of the data dependency algorithm could be improved by using rigorous algorithms for detecting memory variables and modelling the inter-procedural effects more precisely.

The approaches proposed in the thesis consider BO vulnerabilities in general. They do not address related bugs like buffer underflow and integer overflow resulting in BO bugs. Buffer underflow occurs during execution when the program writes to memory location before that of the buffer due to index or pointer decrement operations performed on the buffer. Numerical errors using integers may include integer overflows, underflows, sign conversion errors etc. Since numbers may be used for buffer allocation, indexing or for specifying the bounds for performing string operations, integer overflow bugs, if present in the code, can be used as base for BO exploits. Applications written in C are also vulnerable to format string bugs apart from errors described above. Format string vulnerability occurs when the application does not properly validate external input that is allowed to influence arguments to string formatting functions like *printf*. By supplying specially crafted inputs it is possible to gain higher order privileges on compromised machines. Approaches proposed in the thesis could be extended to postulate rules

and code attributes capturing these bug patterns for comprehensive auditing of C/C++ language related bugs.

Another possible future direction is to explore the application of similar techniques to detect other bugs like path traversal, cross site request forgery, URL redirect etc. by proposing attributes for characterizing them as they are also exploited due to inadequate input validation.

Mining graphs to distinguish between safe and vulnerable programs is another enhancement of proposed techniques. This could be done by characterizing the program graphs looking for the presence of safe or vulnerable constructs for inferring upon the application's vulnerability using data mining techniques through graph similarity measures. Paths leading to sink can be characterized by attributes representing defensive measures applied along the paths to protect the sink. For BO, path-wise attributes such as those representing nature of size checks, input validation etc. performed along them could be proposed to see if they are performed on all the paths leading to the sink or only on some of the paths or none. Graph similarity measures could be postulated and the resulting data can be mined using data mining techniques to differentiate between safe and vulnerable programs. Such a comprehensive modeling of program graphs can be helpful in predicting code behavior.

We have presented approaches for auditing BO vulnerabilities from source-code as well as binary executables respectively. Researchers could extend these frameworks by combining complementary strengths of these techniques with respective formal proving approaches to reduce the manual review efforts. For instance, works integrating multiple approaches like vulnerability testing, formal proof and prediction could be carried out to prove the presence or absence of vulnerability where possible and predict the rest. Such an integrated framework could prove to be an interesting avenue for vulnerability detection.

APPENDIX A

Table A-1. Notations used

Symbol	Meaning
BO	Buffer Overflow
CFG	Control Flow Graph
DD	Data Dependency
CD	Control Dependency
DDG	Data Dependence Graph
CDG	Control Dependence Graph
LOC	lines of code
sink	security-sensitive program statement or operation
tp	true positive
tn	true negative
fp	false positive
fn	false negative
pd	probability of detection or recall
pf	probability of false alarm
pr	precision
acc	accuracy
BN	Bayes Net classifier
NB	Naïve Bayes classifier
SL	Simple Logistic classifier
MLP	Multi-Layer Perceptron classifier
SMO	Sequential Minimum Optimization classifier

AUTHOR'S PUBLICATIONS

[1] Padmanabhuni, B.M., and Tan, H.B.K.: *Auditing Buffer Overflow Vulnerabilities using Hybrid Static-Dynamic Analysis.* IET Softw., **10**(2), pp. 54-61 (2016).

[2] Padmanabhuni, B.M., and Tan, H.B.K.: *Buffer Overflow Vulnerability Prediction from x86 Executables Using Static Analysis and Machine Learning*, in *Proceedings of IEEE 39th Annual Computer Software and Applications Conference,* pp. 450-459 (2015).

[3] Padmanabhuni, B.M., and Tan, H.B.K.: *Light-Weight Rule-Based Test Case Generation for Detecting Buffer Overflow Vulnerabilities*, in *IEEE/ACM 10th International Workshop on Automation of Software Test (AST),* pp. 48-52 (2015).

[4] Padmanabhuni, B.M., and Tan, H.B.K.: *Predicting Buffer Overflow Vulnerabilities through Mining Light-Weight Static Code Attributes*, in *IEEE International Symposium on Software Reliability Engineering Workshops,* pp. 317-322 (2014).

[5] Padmanabhuni, B.M., and Tan, H.B.K.: *Auditing Buffer Overflow Vulnerabilities Using Hybrid Static-Dynamic Analysis*, in *Proceedings of IEEE 38th Annual Computer Software and Applications Conference*, pp. 394-399 (2014).

[6] Padmanabhuni, B.M., and Tan, H.B.K.: *Defending against Buffer-Overflow Vulnerabilities.* Computer, **44**(11), pp. 53-60 (2011).

BIBLIOGRAPHY

1. *CWE - 2011 CWE/SANS Top 25 Most Dangerous Software Errors*. Available from: http://cwe.mitre.org/top25/index.html#CWE-120
2. *PCI Security Standards Documents*. Available from: https://www.pcisecuritystandards.org/security_standards/documents.php
3. *OSVDB: Open Sourced Vulnerability Database*. Available from: http://osvdb.org/
4. *NVD - Home*. Available from: https://nvd.nist.gov/
5. Necula, G.C., McPeak, S., and Weimer, W.: *CCured: type-safe retrofitting of legacy code.* SIGPLAN Not., **37**(1), pp. 128-139 (2002).
6. Jim, T., Morrisett, J.G., Grossman, D., Hicks, M.W., Cheney, J., and Wang, Y.: *Cyclone: A Safe Dialect of C*, in *Proceedings of General Track of the annual conference on USENIX Annual Technical Conference*, pp. 275-288 (2002).
7. Cowan, C., Wagle, P., Pu, C., Beattie, S., and Walpole, J.: *Buffer overflows: attacks and defenses for the vulnerability of the decade*, in *Proceedings of DARPA Information Survivability Conference and Exposition*, pp. 119-129 (2000).
8. Ganapathy, V., Jha, S., Chandler, D., Melski, D., and Vitek, D.: *Buffer overrun detection using linear programming and static analysis*, in *Proceedings of 10th ACM conference on Computer and communications security*, pp. 345-354 (2003).
9. Wagner, D., Foster, J.S., Brewer, E.A., and Aiken, A.: *A first step towards automated detection of buffer overrun vulnerabilities*, in *Network and Distributed System Security Symposium,* pp. 3-17 (2000).
10. Xie, Y., Chou, A., and Engler, D.: *ARCHER: using symbolic, path-sensitive analysis to detect memory access errors*, in *Proceedings of 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 327-336 (2003).
11. Lei, W., Qiang, Z., and Pengchao, Z.: *Automated Detection of Code Vulnerabilities Based on Program Analysis and Model Checking*, in *Proceedings of Eighth IEEE International Working Conference on Source Code Analysis and Manipulation,* pp. 165-173 (2008).
12. Hart, T.E., Ku, K., Gurfinkel, A., Chechik, M., and Lie, D.: *PtYasm: Software Model Checking with Proof Templates*, in *Proceedings of 23rd IEEE/ACM International Conference on Automated Software Engineering,* pp. 479-480 (2008).
13. Evans, D., and Larochelle, D.: *Improving Security Using Extensible Lightweight Static Analysis*. IEEE Softw., **19**(1), pp. 42-51 (2002).
14. Dor, N., Rodeh, M., and Sagiv, M.: *CSSV: towards a realistic tool for statically detecting all buffer overflows in C.* SIGPLAN Not., **38**(5), pp. 155-167 (2003).
15. Hackett, B., Das, M., Wang, D., and Yang, Z.: *Modular checking for buffer overflows in the large*, in *Proceedings of 28th international conference on Software engineering*, pp. 232-241 (2006).
16. Grosso, C.D., Antoniol, G., Merlo, E., and Galinier, P.: *Detecting buffer overflow via automatic test input data generation.* Comput. Oper. Res., **35**(10), pp. 3125-3143 (2008).
17. Rawat, S., and Mounier, L.: *An Evolutionary Computing Approach for Hunting Buffer Overflow Vulnerabilities: A Case of Aiming in Dim Light*, in *Proceedings of 2010 European Conference on Computer Network Defense,* pp. 37-45 (2010).
18. Shahriar, H., and Zulkernine, M.: *Mutation-Based Testing of Buffer Overflow Vulnerabilities*, in *Proceedings of 32nd Annual IEEE International Computer Software and Applications Conference,* pp. 979-984 (2008).

19. Ghosh, A.K., and O'Connor, T.: *Analyzing Programs for Vulnerability to Buffer Overrun Attacks*, in *Proceedings of 21st National Information Systems Security Conference*, pp. 274-382 (1998).
20. Wenhua, W., Yu, L., Donggang, L., Kung, D., Csallner, C., Dazhi, Z., Kacker, R., and Kuhn, R.: *A combinatorial approach to detecting buffer overflow vulnerabilities*, in *Proceedings of 41st IEEE/IFIP International Conference on Dependable Systems & Networks (DSN),* pp. 269-278 (2011).
21. Cadar, C., et al., Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., and Engler, D.R.: *EXE: automatically generating inputs of death*, in *Proceedings of 13th ACM conference on Computer and communications security*, pp. 322-335 (2006).
22. Saxena, P., Poosankam, P., McCamant, S., and Song, D.: *Loop-extended symbolic execution on binary programs*, in *Proceedings of eighteenth international symposium on Software testing and analysis*, pp. 225-236 (2009).
23. Xu, R.-G., Godefroid, P., and Majumdar, R.: *Testing for buffer overflows with length abstraction*, in *Proceedings of 2008 international symposium on Software testing and analysis*, pp. 27-38 (2008).
24. Baratloo, A., Singh, N., and Tsai, T.: *Transparent run-time defense against stack smashing attacks*, in *Proceedings of annual conference on USENIX Annual Technical Conference*, pp. 1-13 (2000).
25. Cowan, C., Pu, C., Maier, D., Hintony, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., and Zhang, Q.: *StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks*, in *Proceedings of 7th conference on USENIX Security Symposium, Volume 7*, pp. 63-78 (1998).
26. Tzi-cker, C., and Fu-Hau, H.: *RAD: a compile-time solution to buffer overflow attacks*, in *Proceedings of 21st International Conference on Distributed Computing Systems,* pp. 409-417 (2001).
27. *Stack Shield: A "stack smashing" technique protection tool for Linux*. Available from: http://www.angelfire.com/sk/stackshield/
28. Hong, H., Xian-Liang, L., Li-Yong, R., Bo, C., and Ning, Y.: *AIFD: A Runtime Solution to Buffer Overflow Attack*, in *Proceedings of 2007 International Conference on Machine Learning and Cybernetics,* pp. 3189-3194 (2007).
29. Zhiqiang, L., Bing, M., and Li, X.: *LibsafeXP: A Practical and Transparent Tool for Run-time Buffer Overflow Preventions*, in *2006 IEEE Information Assurance Workshop,* pp. 332-339 (2006).
30. Dongfang, L., Zhenglin, L., and Yizhi, Z.: *HeapDefender: A Mechanism of Defending Embedded Systems against Heap Overflow via Hardware*, in *Proceedings of 9th International Conference on Ubiquitous Intelligence & Computing and 9th International Conference on Autonomic & Trusted Computing,* pp. 851-856 (2012).
31. Wilander, J., and Kamkar, M.: *A comparison of publicly available tools for dynamic buffer overflow prevention*, in *10th Network and Distributed System Security Symposium*, pp. 1-14 (2003).
32. Smirnov, A., and Chiueh, T.-C.: *DIRA: Automatic Detection, Identification and Repair of Control-Hijacking Attacks*, in *12th Network and Distributed System Security Symposium,* pp. 1-17 (2005).
33. Novark, G., Berger, E.D., and Zorn, B.G.: *Exterminator: automatically correcting memory errors with high probability.* SIGPLAN Not., **42**(6), pp. 1-11 (2007).
34. Rinard, M., Cadar, C., Dumitran, D., Roy, D.M., and Leu, T.: *A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors)*, in *Proceedings of 20th Annual Computer Security Applications Conference,* pp. 82-90 (2004).

35. Gang, C., Hai, J., Deqing, Z., Bing Bing, Z., Zhenkai, L., Weide, Z., and Xuanhua, S.: *SafeStack: Automatically Patching Stack-Based Buffer Overflow Vulnerabilities.* IEEE Transactions on Dependable and Secure Computing, **10**(6), pp. 368-379 (2013).
36. Gao, Q., Zhang, W., Tang, Y., and Qin, F.: *First-aid: surviving and preventing memory management bugs during production runs*, in *Proceedings of 4th ACM European conference on Computer systems*, pp. 159-172 (2009).
37. Lee, T.-R., Chiu, K.-C., and Chang, D.-W.: *A Lightweight Buffer Overflow Protection Mechanism with Failure-Oblivious Capability*, in Hua, A., and Chang, S.-L. (Eds.): *Algorithms and Architectures for Parallel Processing*, Springer Berlin Heidelberg, pp. 661-672 (2009).
38. Xinran, W., Chi-Chun, P., Peng, L., and Sencun, Z.: *SigFree: A Signature-Free Buffer Overflow Attack Blocker.* IEEE Transactions on Dependable and Secure Computing, **7**(1), pp. 65-79 (2010).
39. Hsu, F.-H., Guo, F., and Chiueh, T.-C.: *Scalable network-based buffer overflow attack detection*, in *Proceedings of 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pp. 163-172 (2006).
40. Brumley, D., Newsome, J., Song, D., Wang, H., and Jha, S.: *Towards Automatic Generation of Vulnerability-Based Signatures*, in *Proceedings of 2006 IEEE Symposium on Security and Privacy*, pp. 2-16 (2006).
41. Nanda, S., and Tzi-cker, C.: *Execution Trace-Driven Automated Attack Signature Generation*, in *Proceedings of Annual Computer Security Applications Conference,* pp. 195-204 (2008).
42. Xu, J., Ning, P., Kil, C., Zhai, Y., and Bookholt, C.: *Automatic diagnosis and response to memory corruption vulnerabilities*, in *Proceedings of 12th ACM conference on Computer and communications security*, pp. 223-234 (2005).
43. Zhenkai, L., and Sekar, R.: *Automatic generation of buffer overflow attack signatures: an approach based on program behavior models*, in *Proceedings of 21st Annual Computer Security Applications Conference,* pp. 215-224 (2005).
44. Avijit, K., Gupta, P., and Gupta, D.: *Binary rewriting and call interception for efficient runtime protection against buffer overflows.* Softw. Pract. Exper., **36**(9), pp. 971-998 (2006).
45. Babi, D., Martignoni, L., McCamant, S., and Song, D.: *Statically-directed dynamic automated test generation*, in *Proceedings of 2011 International Symposium on Software Testing and Analysis*, pp. 12-22 (2011).
46. Christodorescu, M., and Jha, S.: *Static analysis of executables to detect malicious patterns*, in *Proceedings of 12th conference on USENIX Security Symposium, Volume 12*, pp. 1-18 (2003).
47. Cova, M., Felmetsger, V., Banks, G., and Vigna, G.: *Static Detection of Vulnerabilities in x86 Executables*, in *Proceedings of 22nd Annual Computer Security Applications Conference*, pp. 269-278 (2006).
48. Shin, Y., Meneely, A., Williams, L., and Osborne, J.A.: *Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities.* IEEE Trans. Softw. Eng., **37**(6), pp. 772-787 (2011).
49. Shin, Y., and Williams, L.: *An initial study on the use of execution complexity metrics as indicators of software vulnerabilities*, in *Proceedings of 7th International Workshop on Software Engineering for Secure Systems*, pp. 1-7 (2011).
50. Koru, A.G., and Liu, H.: *Building Effective Defect Prediction Models in Practice.* IEEE Softw., **22**(6), pp. 23-29 (2005).
51. Menzies, T., Greenwald, J., and Frank, A.: *Data Mining Static Code Attributes to Learn Defect Predictors.* IEEE Trans. Softw. Eng., **33**(1), pp. 2-13 (2007).

52. Lessmann, S., Baesens, B., Mues, C., and Pietsch, S.: *Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings.* IEEE Trans. Softw. Eng., **34**(4), pp. 485-496 (2008).
53. McCabe, T.J.: *A complexity measure.* IEEE Trans. Softw. Eng., **SE-2**(4), pp. 308-320 (1976).
54. Halstead, M.H.: *Elements of Software Science.* Elsevier Science Inc. (1977).
55. Neuhaus, S., Zimmermann, T., Holler, C., and Zeller, A.: *Predicting vulnerable software components*, in *Proceedings of 14th ACM conference on Computer and communications security*, pp. 529-540 (2007).
56. Shar, L.K., Tan, H.B.K., and Briand, L.C.: *Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis*, in *Proceedings of 2013 International Conference on Software Engineering*, pp. 642-651 (2013).
57. Gegick, M., Williams, L., Osborne, J., and Vouk, M.: *Prioritizing software security fortification throughcode-level metrics*, in *Proceedings of 4th ACM workshop on Quality of protection*, pp. 31-38 (2008).
58. Padmanabhuni, B.M., and Tan, H.B.K.: *Defending against Buffer-Overflow Vulnerabilities.* Computer, **44**(11), pp. 53-60 (2011).
59. Padmanabhuni, B.M., and Tan, H.B.K.: *Buffer Overflow Vulnerability Prediction from x86 Executables Using Static Analysis and Machine Learning*, in *Proceedings of IEEE 39th Annual Computer Software and Applications Conference,* pp. 450-459 (2015).
60. Padmanabhuni, B.M., and Tan, H.B.K.: *Light-Weight Rule-Based Test Case Generation for Detecting Buffer Overflow Vulnerabilities*, in *IEEE/ACM 10th International Workshop on Automation of Software Test (AST),* pp. 48-52 (2015).
61. Padmanabhuni, B.M., and Tan, H.B.K.: *Auditing Buffer Overflow Vulnerabilities Using Hybrid Static-Dynamic Analysis*, in *Proceedings of 2014 IEEE 38th Annual Computer Software and Applications Conference*, pp. 394-399 (2014).
62. Padmanabhuni, B.M., and Tan, H.B.K.: *Auditing Buffer Overflow Vulnerabilities using Hybrid Static-Dynamic Analysis.* IET Softw., **10**(2), pp. 54-61 (2016).
63. One, A.: *Smashing the stack for fun and profit.* Phrack Magazine Online. **7**(49).
64. Sinha, S., Harrold, M.J., and Rothermel, G.: *Interprocedural control dependence.* ACM Trans. Softw. Eng. Methodol., **10**(2), pp. 209-254 (2001).
65. Allen, F.E., and Cocke, J.: *A program data flow analysis procedure.* Commun. ACM, **19**(3), pp. 137-147 (1976).
66. *CodeSurfer® - Technologies | GrammaTech Static Analysis*. Available from: http://www.grammatech.com/research/technologies/codesurfer
67. *IDA: About*. Available from: https://www.hex-rays.com/products/ida/
68. *ROSE compiler infrastructure*. Available from: http://rosecompiler.org/
69. Ball, T.: *The concept of dynamic analysis.* SIGSOFT Softw. Eng. Notes, **24**(6), pp. 216-234 (1999).
70. Jones, R.W.M., and Kelly, P.H.J.: *Backwards-compatible bounds checking for arrays and pointers in C programs*, in *International Workshop on Automatic Debugging,* pp. 13-26 (1997).
71. Kong, J., Zou, C.C., and Zhou, H.: *Improving software security via runtime instruction-level taint checking*, in *1st workshop on Architectural and system support for improving software dependability*, pp. 18-24 (2006).
72. Newsome, J., and Song, D.: *Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software*, in *12th Annual Network and Distributed System Security Symposium,* pp. 1-17 (2005).
73. Suh, G.E., Lee, J.W., Zhang, D., and Devadas, S.: *Secure program execution via dynamic information flow tracking.* SIGARCH Comput. Archit. News, **32**(5), pp. 85-96 (2004).

74. Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., and Barham, P.: *Vigilante: end-to-end containment of internet worms.* SIGOPS Oper. Syst. Rev., **39**(5), pp. 133-147 (2005).
75. Clause, J., Li, W., and Orso, A.: *Dytan: a generic dynamic taint analysis framework*, in *Proceedings of 2007 international symposium on Software testing and analysis*, pp. 196-206 (2007).
76. Ruwase, O., and Lam, M.S.: *A Practical Dynamic Buffer Overflow Detector*, in *Network and Distributed System Security Symposium,* pp. 159-169 (2004).
77. Caselden, D., Bazhanyuk, A., Payer, M., McCamant, S., and Song, D.: *HI-CFG: Construction by Binary Analysis and Application to Attack Polymorphism*, in Crampton, J., Jajodia, S., and Mayes, K. (Eds.): *Computer Security – ESORICS 2013*, Springer Berlin Heidelberg, pp. 164-181 (2013).
78. Han, J., Kamber, M., and Pei, J.: *Data mining : concepts and techniques*. Morgan Kaufmann, 3rd ed (2012).
79. Witten, I.H. and Frank, E.: *Data Mining : Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Series in Data Management Systems, 2nd ed (2005).
80. Landwehr, N., Hall, M., and Frank, E.: *Logistic Model Trees.* Machine Learning, **59**(1), pp. 161-205 (2005).
81. Platt, J.C.: *Fast training of support vector machines using sequential minimal optimization*. Advances in kernel methods, MIT Press, pp. 185-208 (1999).
82. Arisholm, E., Briand, L.C., and Johannessen, E.B.: *A systematic and comprehensive investigation of methods to build and evaluate fault prediction models.* J. Syst. Softw., **83**(1), pp. 2-17 (2010).
83. Mende, T.: *Replication of defect prediction studies: problems, pitfalls and recommendations*, in *Proceedings of 6th International Conference on Predictive Models in Software Engineering*, pp. 1-10 (2010).
84. Shar, L.K., and Tan, H.B.K.: *Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization patterns.* Information and Software Technology, **55**(10), pp. 1767-1780 (2013).
85. *Data mining: concepts, methodologies, tools, and applications*. IGI Global (2013).
86. Pandey, S.K., Mustafa, K., and Ahson, S.I.: *A Checklist Based Approach for the Mitigation of Buffer Overflow Attacks*, in *Proceedings of Third International Conference on Wireless Communication and Sensor Networks,* pp. 115-117 (2007).
87. Shahriar, H., Haddad, H.M., and Vaidya, I.: *Buffer overflow patching for C and C++ programs: rule-based approach.* SIGAPP Appl. Comput. Rev., **13**(2), pp. 8-19 (2013).
88. Nishiyama, H.: *SecureC: control-flow protection against general buffer overflow attack*, in *Proceedings of 29th Annual International Computer Software and Applications Conference,* Volume 142, pp. 149-155 (2005).
89. Jacobs, M., and Lewis, E.C.: *SMART C: A Semantic Macro Replacement Translator for C*, in *Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 95-106 (2006).
90. *Security Development Lifecycle (SDL) Banned Function Calls*. Available from: https://msdn.microsoft.com/en-us/library/bb288454.aspx
91. *Libmib Allocated String Functions*. Available from: http://www.mibsoftware.com/libmib/astring/
92. *About Strsafe.h (Windows)*. Available from: https://msdn.microsoft.com/en-us/library/ms647466.aspx
93. Dhurjati, D., and Adve, V.: *Backwards-compatible array bounds checking for C with very low overhead*, in *Proceedings of 28th international conference on Software engineering*, pp. 162-171 (2006).

94. Sen, K., Marinov, D., and Agha, G.: *CUTE: a concolic unit testing engine for C.* SIGSOFT Softw. Eng. Notes, **30**(5), pp. 263-272 (2005).
95. Godefroid, P., Klarlund, N., and Sen, K.: *DART: directed automated random testing.* SIGPLAN Not., **40**(6), pp. 213-223 (2005).
96. Liang, L., Fang, L., and ZhiQiang, H.: *Detecting Buffer Overflows Using Adaptive Test Case Generation.* International Journal of Advancements in Computing Technology, **3**(11), pp. 231-237 (2011).
97. Jorgensen, A.A.: *Testing with hostile data streams*. SIGSOFT Softw. Eng. Notes, **28**(2), pp. 1-6 (2003).
98. Xiao-Song, Z., Lin, S., and Jiong, Z.: *A Novel Method of Software Vulnerability Detection based on Fuzzing Technique*, in *Proceedings of International Conference on Apperceiving Computing and Intelligence Analysis,* pp. 270-273 (2008).
99. Menzies, T., DiStefano, J., Orrego, A., and Chapman, R.: *Assessing Predictors of Software Defects*, in *Workshop Predictive Software Models,* pp. 1-5 (2004).
100. Khoshgoftaar, T.M., and Seliya, N.: *Fault Prediction Modeling for Software Quality Estimation: Comparing Commonly Used Techniques.* Empirical Softw. Engg., **8**(3), pp. 255-283 (2003).
101. Akiyama, F.: *An Example of Software System Debugging.* Information Processing, **71,** pp. 353-379 (1971).
102. Fenton, N.E., and Neil, M.: *A Critique of Software Defect Prediction Models.* IEEE Trans. Softw. Eng., **25**(5), pp. 675-689 (1999).
103. Shepperd, M., and Ince, D.C.: *A critique of three metrics.* Journal of Systems and Software, **26**(3), pp. 197-210 (1994).
104. Menzies, T., Milton, Z., Turhan, B., Cukic, B., Jiang, Y., and Bener, A.: *Defect prediction from static code features: current results, limitations, new approaches.* Automated Software Engg., **17**(4), pp. 375-407 (2010).
105. Misirli, A.T., Bener, A., and Kale, R.: *AI-Based Software Defect Predictors: Applications and Benefits in a Case Study,* in *Proceedings of Twenty-Second Innovative Applications of Artificial Intelligence Conference,* pp. 1748-1755 (2010).
106. Chidamber, S.R., and Kemerer, C.F.: *A Metrics Suite for Object Oriented Design.* IEEE Trans. Softw. Eng., **20**(6), pp. 476-493 (1994).
107. Catal, C., and Diri, B.: *A systematic review of software fault prediction studies.* Expert Systems with Applications, **36**(4), pp. 7346-7354 (2009).
108. Nagappan, N., and Ball, T.: *Use of relative code churn measures to predict system defect density*, in *Proceedings of 27th international conference on Software engineering*, pp. 284-292 (2005).
109. Nagappan, N., Ball, T., and Murphy, B.: *Using Historical In-Process and Product Metrics for Early Estimation of Software Failures*, in *Proceedings of 17th International Symposium on Software Reliability Engineering*, pp. 62-74 (2006).
110. Bird, C., Nagappan, N., Murphy, B., Gall, H., and Devanbu, P.: *Don't touch my code!: examining the effects of ownership on software quality*, in *Proceedings of 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pp. 4-14 (2011).
111. Rahman, F., and Devanbu, P.: *Ownership, experience and defects: a fine-grained study of authorship*, in *Proceedings of 33rd International Conference on Software Engineering*, pp. 491-500 (2011).
112. Zimmermann, T., Nagappan, N., Gall, H., Giger, E., and Murphy, B.: *Cross-project defect prediction: a large scale experiment on data vs. domain vs. process*, in *Proceedings of 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 91-100 (2009).

113. Viega, J., Bloch, J.T., Kohno, Y., and McGraw, G.: *ITS4: a static vulnerability scanner for C and C++ code*, in *Proceedings of 16th Annual Computer Security Applications Conference,* pp. 257-267 (2000).
114. Kang, H., Kim, K., Hong, S., and Lee, D.: *A Model for Security Vulnerability Pattern*, in Gavrilova, M., Gervasi, O., Kumar, V., Tan, C.J.K., Taniar, D., Laganá, A., Mun, Y., and Choo, H. (Eds.): *Computational Science and Its Applications - ICCSA 2006*, Springer Berlin Heidelberg, pp. 385-394 (2006).
115. *Flawfinder*. Available from: http://www.dwheeler.com/flawfinder/
116. Tevis, J.-E.J., and John A. Hamilton, J.: *Static analysis of anomalies and security vulnerabilities in executable files*, in *Proceedings of 44th annual Southeast regional conference*, pp. 560-565 (2006).
117. Larochelle, D., and Evans, D.: *Statically detecting likely buffer overflow vulnerabilities*, in *Proceedings of 10th conference on USENIX Security Symposium,* Volume 10, pp. 1-13 (2001)
118. *Splint*. Available from: http://www.splint.org/
119. Le, W., and Soffa, M.L.: *Marple: a demand-driven path-sensitive buffer overflow detector*, in *Proceedings of 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pp. 272-282 (2008).
120. Nagy, C., and Mancoridis, S.: *Static Security Analysis Based on Input-Related Software Faults*, in *Proceedings of 13th European Conference on Software Maintenance and Reengineering,* pp. 37-46 (2009).
121. Zitser, M., Lippmann, R., and Leek, T.: *Testing static analysis tools using exploitable buffer overflows from open source code.* SIGSOFT Softw. Eng. Notes, **29**(6), pp. 97-106 (2004).
122. Shahriar, H., and Zulkernine, M.: *Mitigating program security vulnerabilities: Approaches and challenges.* ACM Comput. Surv., **44**(3), pp. 1-46 (2012).
123. Aggarwal, A., and Jalote, P.: *Integrating Static and Dynamic Analysis for Detecting Vulnerabilities*, in *Proceedings of 30th Annual International Computer Software and Applications Conference,* Volume 01, pp. 343-350 (2006).
124. Kumar, P.D., Nema, A., and Kumar, R.: *Hybrid analysis of executables to detect security vulnerabilities: security vulnerabilities*, in *Proceedings of 2nd India software engineering conference*, pp. 141-142 (2009).
125. *Compiler Security Checks In Depth*. Available from: http://msdn.microsoft.com/en-us/library/aa290051(VS.71).aspx
126. Nebenzahl, D., Sagiv, M., and Wool, A.: *Install-time vaccination of Windows executables to defend against stack smashing attacks.* IEEE Transactions on Dependable and Secure Computing, **3**(1), pp. 78-90 (2006).
127. Ozdoganoglu, H., Vijaykumar, T.N., Brodley, C.E., Kuperman, B.A., and Jalote, A.: *SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address.* IEEE Transactions on Computers, **55**(10), pp. 1271-1285 (2006).
128. Xu, J., Kalbarczyk, Z., Patel, S., and Iyer, R.K.: *Architecture support for defending against buffer overflow attacks*, in *2nd Workshop on Evaluating and Architecting System dependability,* pp. 1-8 (2002).
129. Cowan, C., Beattie, S., Johansen, J., and Wagle, P.: *Pointguard^{TM}: protecting pointers from buffer overflow vulnerabilities*, in *Proceedings of 12th conference on USENIX Security Symposium, Volume 12*, pp. 91-104 (2003).
130. Fetzer, C., and Zhen, X.: *Detecting heap smashing attacks through fault containment wrappers*, in *Proceedings of 20th IEEE Symposium on Reliable Distributed Systems,* pp. 80-89 (2001).
131. Novark, G., and Berger, E.D.: *DieHarder: securing the heap*, in *Proceedings of 17th ACM conference on Computer and communications security*, pp. 573-584 (2010).

132. Zeng, Q., Wu, D., and Liu, P.: *Cruiser: concurrent heap buffer overflow monitoring using lock-free data structures.* SIGPLAN Not., **46**(6), pp. 367-377 (2011).
133. Nikiforakis, N., Piessens, F., and Joosen, W.: *HeapSentry: Kernel-Assisted Protection against Heap Overflows*, in Rieck, K., Stewin, P., and Seifert, J.-P. (Eds.): *Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer Berlin Heidelberg, pp. 177-196 (2013).
134. *Linux kernel security hardening patch from the Openwall Project*. Available from: http://www.openwall.com/linux/
135. *Homepage of PaX*. Available from: https://pax.grsecurity.net/
136. Dahn, C., and Mancoridis, S.: *Using Program Transformation to Secure C Programs Against Buffer Overflows*, in *Proceedings of 10th Working Conference on Reverse Engineering*, pp. 323-332 (2003).
137. Smirnov, A., and Chiueh, T.-C.: *Automatic Patch Generation for Buffer Overflow Attacks*, in *Proceedings of Third International Symposium on Information Assurance and Security*, pp. 165-170 (2007).
138. Sidiroglou, S., Giovanidis, G., and Keromytis, A.D.: *A dynamic mechanism for recovering from buffer overflow attacks*, in *Proceedings of 8th international conference on Information Security*, pp. 1-15 (2005).
139. Wilander, J., Nikiforakis, N., Younan, Y., Kamkar, M., and Joosen, W.: *RIPE: runtime intrusion prevention evaluator*, in *Proceedings of 27th Annual Computer Security Applications Conference*, pp. 41-50 (2011).
140. Ku, K., Hart, T.E., Chechik, M., and Lie, D.: *A buffer overflow benchmark for software model checkers*, in *Proceedings of twenty-second IEEE/ACM international conference on Automated software engineering*, pp. 389-392 (2007).
141. Lu, S., Li, Z., Qin, F., Tan, L., Zhou, P., and Zhou, Y.: *Bugbench: Benchmarks for evaluating bug detection tools*, in *Workshop on the Evaluation of Software Defect Detection Tool,* pp. 1-5 (2005).
142. *Weka 3 - Data Mining with Open Source Machine Learning Software in Java*. Available from: http://www.cs.waikato.ac.nz/ml/weka/
143. Le, W., and Soffa, M.L.: *Generating analyses for detecting faults in path segments*, in *Proceedings of 2011 International Symposium on Software Testing and Analysis*, pp. 320-330 (2011).
144. Holzmann, G.J.: *UNO: Static Source Code Checking for UserDefined Properties*, in *Proceedings of 6th World Conference on Integrated Design and Process Technology,* pp. 1-27 (2002).
145. *Static Analysis Tools for C/C++ and Ada - Polyspace*. Available from: http://www.polyspace.com/downloads.html
146. Zitser, M.: *Securing software: An evaluation of static source code analyzers*. Master's Thesis, Massachusetts Institute of Technology (2003).
147. Wang, T., Wei, T., Lin, Z., and Zou, W.: *IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution*, in *Network Distributed System Security Symposium,* pp. 1-14 (2009).
148. Durães, J. and Madeira, H.: *A Methodology for the Automated Identification of Buffer Overflow Vulnerabilities in Executable Software Without Source-Code*, in Maziero, C., Gabriel Silva, J., Andrade, A., and de Assis Silva, F. (Eds.): *Dependable Computing*, Springer Berlin Heidelberg, pp. 20-34 (2005).
149. Hiser, J., Coleman, C., Co, M., and Davidson, J.: *MEDS: The Memory Error Detection System*, in Massacci, F., Redwine, S., Jr., and Zannone, N. (Eds.): *Engineering Secure Software and Systems*, Springer Berlin Heidelberg, pp. 164-179 (2009).

150. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., and Vigna, G.: *Automating mimicry attacks using static binary analysis*, in *Proceedings of 14th conference on USENIX Security Symposium, Volume 14*, pp. 1-16 (2005).
151. Balakrishnan, G., and Reps, T.: *WYSINWYX: What you see is not what you eXecute.* ACM Trans. Program. Lang. Syst., **32**(6), pp. 1-84 (2010).
152. Balakrishnan, G., Gruian, R., Reps, T., and Teitelbaum, T.: *CodeSurfer/x86—A Platform for Analyzing x86 Executables*, in Bodik, R. (Ed.): *Compiler Construction*, Springer Berlin Heidelberg, pp. 250-254 (2005).
153. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., and Saxena, P.: *BitBlaze: A New Approach to Computer Security via Binary Analysis*, in *Proceedings of 4th International Conference on Information Systems Security*, pp. 1-25 (2008).
154. Brumley, D., Jager, I., Avgerinos, T., and Schwartz, E.J.: *BAP: a binary analysis platform*, in *Proceedings of 23rd international conference on Computer aided verification*, pp. 463-469 (2011).
155. Edwards, A., H. Vo, and A. Srivastava, *Vulcan: Binary Transformation in a Distributed Environment*, *Technical Report (MSR-TR-2001-50),* pp. 1-10 (2001).
156. *Valgrind*. Available from: http://valgrind.org/
157. Nethercote, N., and Seward, J.: *Valgrind: a framework for heavyweight dynamic binary instrumentation.* SIGPLAN Not., **42**(6), pp. 89-100 (2007).
158. *Pin - A Dynamic Binary Instrumentation Tool*. Available from: https://software.intel.com/en-us/articles/pintool
159. *DynamoRio Dynamic Instrumentation Tool Platform*. Available from: http://dynamorio.org/
160. *Paradyn Tools Project*. Available from: http://www.paradyn.org/
161. Balakrishnan, G., and Reps, T.: *Analyzing Memory Accesses in x86 Executables*, in Duesterwald, E. (Ed.): *Compiler Construction*, Springer Berlin Heidelberg, pp. 5-23 (2004).
162. Aho, A.V., Lam, M.S., Sethi, R., and Ullman, J.D.: *Compilers : principles, techniques, & tools*. Pearson/Addison Wesley, Boston, 2nd ed. (2007).
163. Wolfe, M.J.: *High performance compilers for parallel computing*. Addison-Wesley, Redwood City, Calif (1996).
164. Dandamudi, S.P.: *Introduction to Assembly language programming : for Pentium and RISC processors*. Springer, New York, 2nd ed. (2005).
165. *The Yices SMT Solver*. Available from: http://yices.csl.sri.com/
166. Padmanabhuni, B.M., and Tan, H.B.K.: *Predicting Buffer Overflow Vulnerabilities through Mining Light-Weight Static Code Attributes*, in *IEEE International Symposium on Software Reliability Engineering Workshops,* pp. 317-322 (2014).
167. *Microsoft Security Development Lifecycle*. Available from: https://www.microsoft.com/en-us/sdl/
168. *Safecode | Driving Security and Integrity*. Available from: http://www.safecode.org/