BUFFER OVERFLOW VULNERABILITY DIAGNOSIS FOR COMMODITY SOFTWARE

by

Jiang Zheng

M.S., Northeastern University, Boston, USA, 2001

Submitted to the Graduate Faculty of

the Department of Computer Science in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2008

UNIVERSITY OF PITTSBURGH

COMPUTER SCIENCE DEPARTMENT

This dissertation was presented

by

Jiang Zheng

It was defended on

Sept. 19th 2008

and approved by

BUFFER OVERFLOW VULNERABILITY DIAGNOSIS FOR COMMODITY SOFTWARE

Jiang Zheng, PhD

University of Pittsburgh, 2008

ABSTRACT

Buffer overflow attacks have been a computer security threat in software-based systems and applications for decades. The existence of buffer overflow vulnerabilities makes the system susceptible to Internet worms and denial of service (DDoS) attacks which can cause huge social and financial impacts.

Due to its importance, buffer overflow problem has been intensively studied. Researchers have proposed different techniques to defend against unknown buffer overflow attacks. They have also investigated various solutions, including automatic signature generation, automatic patch generation, etc., to automatically protect computer systems with known vulnerabilities. The effectiveness and efficiency of the automatic signature generation approaches and the automatic patch generation approaches are all based on the accurate understanding of the vulnerabilities, the *buffer overflow vulnerability diagnosis (BOVD).* Currently, the results of automatic signature generation and automatic patch generation are far from satisfaction due to the insufficient research results from the automatic BOVD.

This thesis defines the automatic *buffer overflow vulnerability diagnosis* (BOVD) problem and provides solutions towards automatic BOVD for commodity software. It targets on commodity software when source code and symbol table are not available. The solutions combine both of the dynamic analysis techniques and static analysis techniques to achieve the goal.

Based on the observation that buffer overflow attack happens when the size of the destination buffer is smaller than the total number of writes after the data copy process if the buffer overflow attack happens through a data copy procedure, the diagnosis results return the information of the size of destination buffer, the total number of writes of a data copy procedure and how the user inputs are related with them. They are achieved through bound analysis, loop analysis and input analysis respectively. We demonstrate the effectiveness of this thesis approach using real world vulnerable applications including the buffer overflow vulnerabilities attacked by the record-setting Slammer and Blaster worms.

This thesis also does the complete case study for buffer overflow vulnerabilities which may have independent interests to researchers. Our buffer overflow case study results can help other researchers to design more effective defense systems and debugging tools against buffer overflow attacks.

Keywords: software security, vulnerability diagnosis, vulnerability defense, buffer overflow, loop analysis, bound checking.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PREFACE

First of all, I want to thank my advisor at University of Pittsburgh, Dr. Jose Carlos Brustoloni, who accepted me during my most difficult time of choosing research directions. His selfless support makes me ultimately obtaining my Ph.D. degree realistic. The same appreciation is for my other Ph.D. committee members, Bruce Childers, Shi-Kuo Chang and James Joshi.

I also want to appreciate Dr. Dawn Song, my outside committee member, who introduced me to this fascinating research area of software security. Beyond that, I learned how to work independently from her.

I also want to thank Dr. Rami Melhem, Dr. Milos Hauskrecht and Dr. Youtao Zhang who gave me guidance through my Ph.D. exploration process, Dr. Ahmed Amer whose smartness in research makes my first independent work a reputable publication and whose kindness make me confident on research.

I am very lucky to have chance to do co-op at CISCO Systems and do summer intern at Google Inc. during my graduate study. My mentor at CISCO Systems Inc., Albra Bonardi and my host at Google Inc., Ken Lin guided me through the projects of real-world industries. I really enjoy the real-world industry experiences which opened my eyes to broader areas. Here, I want to appreciate my co-workers, Andrew Joen, Fang Fang, Shaomei Wu, Jun Liu, Junmin Gao who are part of a lot of unforgettable memories in my mind.

I also want to appreciate my parents, Mr. Zheng and Ms. Yao. They gave me a healthy body, optimistic attitude, and strong motivation to obtain the Ph.D. degree. Without their love and constant support, it will be impossible for me to get my Ph.D. degree.

I also want to appreciate my son Andrew Z. Li who brings a lot of happiness during my stressful Ph.D. study process. His loveliness and his happiness have magic power to support me.

Furthermore, I want to appreciate my teammates at BitBlaze project, from whom I learned a lot of things which build up my experiences in software security. They are all very smart guys. Among them, I want to especially appreciate David Brumley. David stimulates me each time when I felt frustrating. From him, I got a lot of inspirations. Several of my thesis contributions start from his insightful comments. Jim Newsome is another smart guy who gave me comments on how to present my work and how to write papers to attract other people. I learned a lot of experimental experiences from Zhenkai Liang, Ivan Jager, Pongsin Poosank and Ning Qu who are all very nice and patient. From them, I learned a lot of knowledge of the infrastructure of the BitBlaze project and system programming.

1.0 INTRODUCTION

Buffer overflow[73, 38] has become notorious since 1988, when Morris worm, one of the first significant computer break-ins which took advantage of a buffer overflow vulnerability, surfaced in software security community. The attacks caused by buffer overflow continue to be the major computer security threats. As a traditional exploit, buffer overflow allows attackers to inject malicious codes in the application at run-time. Those injected codes can help attacker to gain the access privilege of the host machine maliciously.

Even though buffer overflow has been intensively studied and researchers have proposed various mitigation techniques, it is still the most critical security threat these days. According to the statistics from CERT Coordination Center, the number of identified and cataloged vulnerabilities has increased over 213% in the last two years (2004-2006) as shown in Figure 1. Based on Xu's study [100], buffer overflow vulnerability is still the largest category of cataloged vulnerabilities. The existence of buffer overflow vulnerabilities makes the system susceptible to Internet worms and denial of service (DDoS) attacks which can cause huge social and financial impacts.

Due to its importance, during the last two decades, researchers have proposed various approaches[22, 30, 59, 81, 86, 62, 23, 58, 78, 43, 93, 92, 37, 36, 91, 11, 47, 20, 18, 21, 31, 48, 67, 68, 56, 42, 58, 78, 43, 100, 28, 15] to defend against buffer overflow attacks. However, each of them has its own limitations and can't be applied universally to prevent buffer overflow attacks. At the same time, researchers have also proposed many attack detection techniques[90, 72, 39, 76, 100, 88, 28, 102, 15, 22, 30, 59, 81, 86, 62, 23]. Using the above attack detection tools to monitor a program, an alarm will be raised once a malicious attack happens.

They have also investigated various solutions, including automatic signature generation,

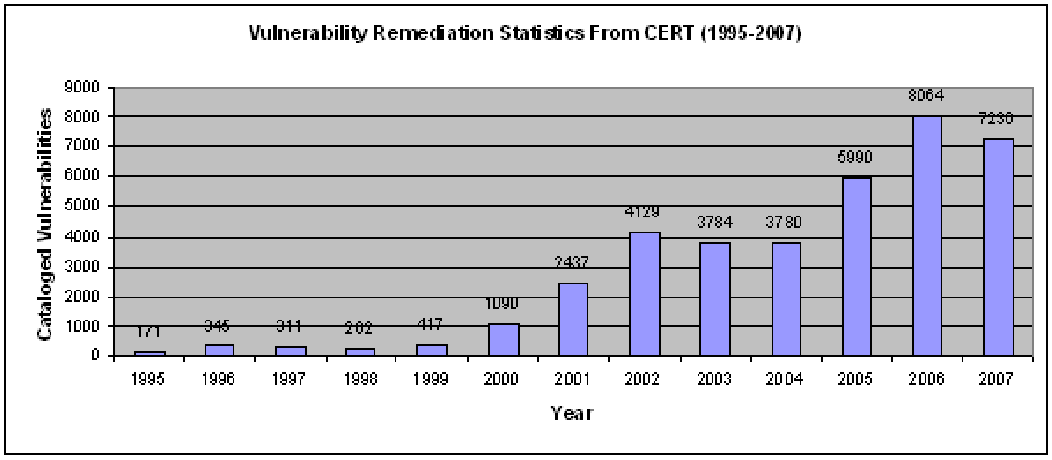


Figure 1: The Cataloged Vulnerabilities Data from CERT Coordination Center(CERT/CC) from 1995 to 2007

automatic patch generation, etc., to automatically protect computer systems with known vulnerabilities. Exploits of vulnerable programs, including zero-day attacks on previously unknown vulnerabilities have caused havoc on the Internet and caused billions of dollars in estimated damage. *signature-based input filtering* is an effective and widely deployed mechanism to defend against such attacks. It matches program inputs against a set of signatures. It has been widely deployed due to its fast response and less intrusion to a system. Researchers have proposed many different approaches[60, 70, 40, 84, 61, 24, 26, 94, 35, 34]. However, none is satisfiable. Vulnerability-based signature approaches[24, 26, 94, 41, 34] which are believed to be the most effective signature techniques either generate the signature manually or assume the vulnerability condition is given or can't handle when the vulnerability doesn't happen through the unsafe usage of library functions. Automatic patch generation [82] applies some simple fix heuristics. The automatic patches generated using those heuristics are not effective. The effectiveness and efficiency of those approaches are all based on the accurate understanding of the vulnerabilities, the *buffer overflow vulnerability diagnosis (BOVD)*. Currently, the results of automatic signature generation and automatic patch generation are far from satisfaction due to the insufficient research results from the automatic BOVD. However, generating the patches and signatures by security engineers is an error-prone activity requiring heavy investment in time and manpower which may delay the protection time, e.g. patch generation time. They all **motivate** the research on automatic and accurate vulnerability diagnosis.

1.1 PROBLEM DEFINITION

A buffer overflow occurs when a store instruction writes outside the allocated buffer bounds. In the best case, a buffer overflow results in the program crashing due to an inappropriate memory dereference. In the worst case, a buffer overflow can be exploited to hijack control of a program.

When a new buffer overflow vulnerability is first exploited, a primary challenge is to diagnose the vulnerability. The expected buffer overflow vulnerability diagnosis results include 1) which buffer was overflowed, and 2) under what conditions the buffer will be overflowed. The running example demonstrated in Figure 23 is a buffer overflow vulnerable program motivated by several real world examples. We want to find out that the `chunk->data` is the vulnerable buffer, an integer field of a user input can cause the buffer overflow, and buffer overflow happens when the integer is greater than 36. Currently, this diagnosis process is done manually by security engineers which is slow, expensive and error prone. Given the results of automatic vulnerability diagnosis, we can then perform the accurate signature generation and patch generation automatically and efficiently. How automatic vulnerability diagnosis can benefit the whole software security research is demonstrated in Figure 2.

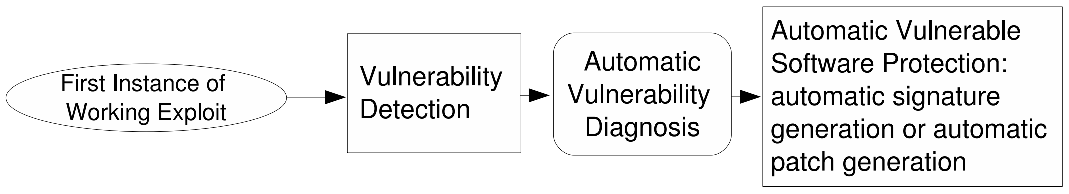

First Instance of Working Exploit
Vulnerability Detection
Automatic Vulnerability Diagnosis
Automatic Vulnerable Software Protection: automatic signature generation or automatic patch generation

Figure 2: Automatic vulnerability diagnosis in software security research

Automatic buffer overflow vulnerability diagnosis is important for enabling appropriate security responses to defend against zero-day attacks, such as creating an input filter (e.g., [24, 26, 94]) that filters out all subsequent exploits for the buffer overflow, and more effective automatic patch generation [82]. Unfortunately, in such scenarios we do not have access to the source code: defenses must be prepared from only the vulnerable program binary and a sample exploit. Such scenarios do not assume access to source code because many threats develop quickly (e.g., internet worms) and there is no time for an end-user to contact the vendor: defenses must be prepared immediately. Furthermore, when the application is legacy application or modern software application which tends to be built on third-party libraries or when the program segments are written in assembly code directly, the access to source code is unlikely. Due to all these reasons, we target the vulnerable program on binary programs.

In the automatic *buffer overflow vulnerability diagnosis* (BOVD) problem, we are given a compiled buffer overflow vulnerable program P and a working exploit E. The goal is to find out where the program is vulnerable, *buffer overflow vulnerability point* (BOVP) and why the program is vulnerable, *buffer overflow vulnerability condition* (BOVC).

The buffer overflow attack occurs by overwriting some critical data, for example, return address, function pointers, etc. We use the instruction that does the malicious write to pinpoint the vulnerability location of the given vulnerable program, which is the BOVP.

Based on the observation that buffer overflow attack happens when the size of the destination buffer is smaller than the total number of writes after the data copy process if the buffer overflow attack happens through a data copy procedure, the BOVC contains the destination buffer size information, the total number of writes information and how the user inputs are related with them.

1.2 CHALLENGES.

Even though buffer overflow problem has been intensively studied and researchers have proposed various mitigation tools during the last few decades, the buffer overflow vulnerability diagnosis is still an open problem.

The Open Web Application Security Project (OWASP) has proposed Gray Box testing technique[1] to prevent stack buffer overflows. It searches for calls to insecure library functions like `gets()`, `strcpy`, `strcat()`, `strncpy()`, `memccpy()` etc. For `gets()`, `strcpy()` and `strcat()` which don't validate the length of source strings and blindly copy data into fixed size buffers, it trace back the source of function arguments and ascertain string lengths while reviewing code. For `strncpy()`, it traces back the size argument. If the size argument is related with user input, this is insecure and a stack buffer overflow may happen. This work is updated in Dec. 2006. However, there are several limitations on this Gray Box testing approach. First it targets insecure library function calls only, what if it happens in a code block that is programmer written? This Grey Box will fail to detect the possible buffer overflow attacks. Second, it targets stack buffer overflows when the destination buffers are on stack and have fixed sizes. What if the destination buffer is on heap? Especially what if the destination buffer size is controllable by user? The Gray Box testing approach can't handle those situations. This work still leaves a lot of open problems to the software security community, even though it is a very good trial to prevent buffer overflow attacks starting from understanding how a buffer overflow attack can happen.

The big challenges to solve the limitations of this Gray Box testing technique[1] and target the program on binary code level include but are not limited to the followings:

1.2.1 Handle programmer written data copy procedure.

Most of known buffer overflow happen when the size of the destination buffer is smaller than the total number of writes through a data copy process.

There are many known buffer overflow vulnerabilities that happen through unsafe usages of Standard C Library functions e.g. `strcpy()`. If a buffer overflow vulnerability happens through unsafe usage of known library functions, we can conclude through function summaries as we know the functionality of those functions in advance. If the data copying process isn't accomplished through those known library functions but a programmer implemented data copy procedure, what should we do? First, how do we locate a programmer written data copy procedure? Along a long sequence of codes, how can we know which part of the code is doing the data coping? Second, after we locate the programmer written data copy procedure, how can we understand the procedure? Is it similar to `strcpy()` or is it similar to `strncpy()`? What is its exact behavior? This problem is very difficult.

We answer the first question through loop detection. It is based on the observation that most of known buffer overflow attacks happen in loop context. We studied some standard library functions that are performing the data coping functionality including `strcpy()`, `strncpy()`, `memcpy()`, `memset()` etc. They are all implemented by some loop structures. The buffer overflow happens when the number of writes after the loop execution exceeds the destination buffer size limit. We then locate a programmer implemented data copy procedure through loop detection.

We answer the second question by coming up with a generalized data copy procedure and perform loop analysis on the generalized data copy procedure. In program modeling community, the loop behavior is believed to be undetermined. How to understand a programmer written data copy procedure is believed to be hard. This thesis answers this question by formulating a general data copy procedure to simplify loop behavior. We perform a series of analysis to understand it. Please refer to Section 3.3.1 for details.

This thesis work uses a lot of results of the compiler research community which includes loop detection[66], loop induction variable[66] detection, loop invariant variable[66] detection. The loop detection and loop induction variable detection are very complicated problems that have been studied very well by researchers. We studied them in detail and did the implementation.

Our loop detection can handle both of the reducible CFG (control flow graph) graphs and the irreducible CFG graphs. The irreducible graph happens in practice. We met irreducible graphs in our experiences. Black Hat[2] community did loop detection for their approach requirements. They can only handle reducible CFG graphs[3]. IDAPro's loop detection algorithm[4] considers the irreducible CFG graph situation. However it can't convert the irreducible CFG to reducible CFG. The behavior of a loop structure in an irreducible CFG is undetermined. Please refer Section 5.3.1 for details.

1.2.2 Find vulnerable buffer bound in binary code level.

To understand the BOVC when only the binary program is available is challenging. At a high level, the binary program itself does not indicate where and how large buffers are within the program. For example, Figure 3 shows the memory layout for a single function which has multiple local variables. Although we show the variable information in the diagram, in binary program itself the entire region would be viewed as a contiguous set of bytes.

The existing research on local variable recovery for binary programs can't guarantee the results. The IDAPro disassembler is believed to be the most advanced tool for hostile code analysis, vulnerability research and software reverse engineering. Its techniques can recover 83% of the local variables and 0% of the fields of heap-allocated objects. The Value-Set Analysis (VSA)[19] claims that they can correctly identify 88% of the local variables and 89% of the fields of heap-allocated objects. The IDAPro is an industry product and the VSA[19] is academic work. These two approaches are the best results so far. So, there is no perfect solution on this problem.

However, in the BOVD problem, we don't need to recover all of the local variables or fields of heap-allocated objects. We only want to identify the size of the vulnerable buffer. We solve this problem by studying runtime traces. This is based on the observation that the runtime traces contain all of the memory operation history which includes the function call, function return, memory allocation, memory deallocation, memory access etc. Please refer to Section 3.3.2 for details.

1.2.3 Extract the relationship of user inputs with buffer overflow factors.

The problem of extracting the relationship of user inputs with the factors that can cause buffer overflow is equivalent to the path explosion problem in software testing and software security research. This problem has also been intensively explored by researchers[50, 34, 35, 26, 27, 51, 49]. Among them, the latest results are by Godefroid et al.[50], Brumley et al.[26] and Bouncer[34].

Brumley et al.[26] propose using weakest precondition to generate vulnerability-based signatures. In their work, they assume the vulnerability point and the vulnerability condition (e.g. the factors that can cause buffer overflow) are given. Their major contribution is on the path explosion. This work enhances the forward symbolic execution methods[24]. The forward symbolic execution method does not scale well to large and real programs as the number of execution paths can be exponential to the number of exploring depth. However, calculating the weakest precondition is very difficult for programs with loops or recursion and when the formula generated is very complicated.

Bouncer[34] improves the Vigilante[35]. Vigilante[35] uses the symbolic execution[101, 27] and path slicing[55] to extract the conditions to reach the vulnerability point on one execution path. Bouncer[34] contributes on symbolic summaries for common library functions and generation of alternative exploits guided by symbolic execution for the path explosion.

Godefroid et al.[50] proposes a fuzz testing technique. Compared to the blackbox fuzz testing approach[41] which applies random mutations to well-formed inputs and test the program on the resulting values, it performs white-box fuzz testing by applying a new search algorithm. This new algorithm is designed to partially explore the state space of large applications;maximize the number of new tests generated from each symbolic execution;apply heuristics to maximize code coverage; resilient to divergences.

This is an independent research area. We leave this as a future research work to explore the input analysis in depth. Our current implementation is a simple approach where only the exploit executed path is examined. Please see Section 8.2 for the discussion of how to improve the existing approach towards generating zero false negative and zero false positive signatures.

1.3 APPROACH OVERVIEW.

In this thesis, we provide solutions towards automatic BOVD for commodity software when source code and debugging information are not available. We combine both of the static binary program analysis techniques and dynamic taint analysis techniques to achieve this challenging goal.

Researchers[65, 85, 99, 80] have addressed intensively on finding the BOVP. We use the previously proposed dynamic-taint-analysis[72, 76, 39] to do the attack detection. Using the dynamic taint analysis, a huge class of exploits can be detected for commodity software. We then search back the collected runtime traces to find out the latest instruction that writes the security critical data, which is the BOVP.

Based on the observation that buffer overflow attack happens when the size of the destination buffer is smaller than the total number of writes after the data copying process if the buffer overflow attack happens through a data copy procedure, the BOVC contains the destination buffer size information, the total number of writes information and how the user inputs are related with them. The above BOVC results assume the buffer overflow happens through a data copy procedure. This thesis can also identify the case when the buffer overflow attack happens through an unchecked array index instead of a data copy procedure.

The BOVC is achieved through three steps analysis. The three steps analysis include loop analysis, bound analysis and input analysis. The bound analysis answers the size of the destination buffer information. The loop analysis answers the total number of writes information of programmer implemented data copy procedure. For buffer overflow attack happens through standard C library functions (library data copy procedures), we answer the total number of writes information trough function summaries. The input analysis answers how the user inputs are related with the size of the destination buffer and the total number of writes.

Our loop analysis is based on the observation that most of the real world buffer overflow attacks happen through loop context. We use loop detection to locate a programmer implemented data copy procedure and we propose a generalized data copy procedure and apply loop analysis on it to answer the total number of writes question.

Given the vulnerable buffer location and the overwritten security critical data (e.g. return address), we can calculate the coarse-grained bound result of the vulnerable buffer easily. However, this result can't protect variables located in between. Using this coarse-grained bound result, non-control-data attack[29] may happen in the future on the same vulnerability. Figure 3 compares the difference between coarse-grained bound analysis and fine-grained bound analysis results.

Our fine-grained bound analysis is achieved through memory traces analysis. This is

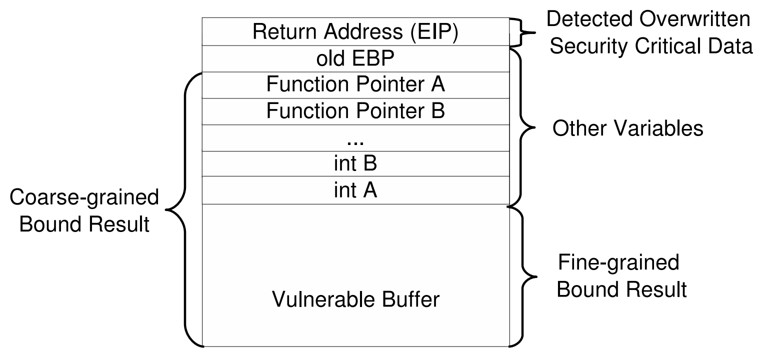


Figure 3: The comparison of coarse-grained and fine-grained bound analysis results.

based on the observation that our runtime traces contain all of the memory update information. Because we can overwrite those variables that have not been written, we can guarantee the fine-grained bound result for the given execution path. Also, because our bound analysis is independent of the location of the vulnerable buffer, we can handle the buffer overflow that happens anywhere including stack, heap, Data/BSS.

Our input analysis generates the dynamic slicing of the traces contains the malicious variable(s) and performs symbolic execution on it to answer how the user inputs are related with those malicious variable(s).

The direct application of our BOVD results is for the signature generation. We will use the generated signatures for the real world vulnerable applications to demonstrate the effectiveness of our approach in Chapter 5. In Chapter 8, we discuss the application of our BOVD results on automatic patch generation, automatic exploit generation and how to achieve zero false-positive and zero false-negative signature results by extending this thesis work.

1.4 PROBLEM SCOPE

Even though our approach is designed to do BOVD for commodity software when source code and debugging information are not available, it can be generalized to do the vulnerability diagnosis for most memory corruption attacks including buffer overflow, format string, integer overflow that triggers buffer overflow in the working exploit and double free. This thesis work targeted on binary programs, while it can also be implemented at source code level to do the software verification during the software testing stage. It would be an enhanced version of the Grey Box Testing tool[1].

1.5 CONTRIBUTIONS

In summary, this thesis makes the following contributions:

- It combines both of the static binary program analysis techniques and dynamic taint analysis techniques to achieve the BOVD. It extends the existing dynamic-taint-analysis attack detection tool to do the BOVP detection and performs three steps analysis towards understanding BOVC. The three steps analysis are loop analysis, bound analysis and input analysis.
- It uses loop analysis to understand programmer written data copy procedures under reasonable loop heuristic assumptions. This is based on the observation of all known buffer overflow attacks that most of them take place in loop context.
- It achieves fine-grained bound analysis result of the vulnerable buffer using novel memory trace analysis approach. This can be achieved based on the fact that the runtime traces contain all of the memory operation history. This memory operation includes function call, function return, variable push and pop, variable load and store, malloc, free etc.
- It performs buffer overflow vulnerability case study which may have independent interests to security engineers. There are six basic buffer overflow vulnerability cases that are based on several assumptions. The six basic cases can cover all of the real world vulnerable applications that this thesis work studied. Other cases by removing each of

the assumptions are also discussed. This result can help other researchers to design more effective buffer overflow response and debugging systems.

The organization of this thesis is as follows. In Chapter 2, the existing attempts on vulnerability diagnosis and the limitations of each approach are given. As the direct application of BOVD is for vulnerability-based signature generation, the research work on signature generation is also summarized. I also summarized the buffer overflow defense systems that researchers have developed, none of which can defend against the buffer overflow attacks completely. In Chapter 3, it describes the proposed BOVD solution of this thesis work which includes the dynamic analysis engine, the static analysis engine and the diagnosis engine. Among them, the diagnosis engine is the major contribution of this thesis work. After that, we discuss how the proposed BOVD approach can handle the vulnerability diagnosis for format string vulnerability and double free vulnerability. In Chapter 4, we perform the buffer overflow vulnerability case study. This result can help other researchers to design more effective buffer overflow attack response systems and debugging tools. The implementation detail and experiments are displayed in Chapter 5 and Chapter 6. In Chapter 6, we evaluate the proposed BOVD approach and test the buffer overflow vulnerability case study results. We also perform intensive loop studies of WCET benchmark and some real world applications to evaluate how our loop analysis can handle different loop structures. The Chapter 7 and Chapter 8 are for the discussion and application of the proposed BOVD approach. Chapter 9 concludes the thesis work.

2.0 RELATED WORK

No previous work has addressed the vulnerability diagnosis problem thoroughly. All of the existing attempts[65, 85, 99, 69, 72, 80, 95] can find the vulnerability point using different approaches but none can figure out the vulnerability conditions. Bouncer[34] can only understand vulnerability conditions when the vulnerability happens through unsafe usages of library functions. Each approach and its limitations will be discussed briefly in Section 2.1.

Signature generation is one of the traditional defense schemes for Internet security attacks. The direct application of the BOVD is for automatic vulnerability-based signature generation[24, 26] for buffer overflow attacks. In section 2.2, I will review the related work on this research field.

Even though this thesis work is the first to investigate the vulnerability diagnosis problem, researchers have invented various mitigation techniques to defend against intrusion attacks on buffer overflow vulnerabilities. In Section 2.3, I summarize this research topic briefly which has been very active during the last few decades.

2.1 EXISTING ATTEMPTS

The existing attempts[72, 65, 85, 99, 95, 80] on vulnerability diagnosis all do attack detection using different methods, more or less vulnerability analysis. Among them, [72, 65, 85, 99, 95] apply their attack detection and vulnerability analysis results for signature generation. They all can find the vulnerability point (some can only handle special cases), but none can provide any semantic information of the vulnerable program. COVERS[65], DIRA[85] and Packet Vaccine[95] use input fields' lengths in their signature results. MemSherlock[80] considers the destination buffer size variable as another factor that can cause buffer overflow vulnerability. However, they all can't handle any other buffer overflow vulnerability cases.

COVERS[65] assumes that some part of the exploit will overflow some pointer values. They then do the forensic analysis of the victim memory region surrounding the corrupted pointer value by string matching it with the exploit. They do further context identification to figure out which fields of the input triggered the attack. They obtain the length constraints by analyzing all of the existing benign inputs and using the longest length value as the length constraints of that field. Their results can introduce false positives and they can't handle other buffer overflow cases. Another limitation of this approach is that it can't handle any encoding/decoding of the exploit during the program execution.

Xu *et al.* apply both static program analysis and hardware watch point to find out the corrupting instruction[99]. After that, they do the value propagation to find the propagation history of the corrupted data. Unfortunately, they don't use this information as a diagnosis result to do the signature generation. Their diagnostic result is the instruction that corrupts the sensitive data and the stack trace when memory corrupts. They generate the signature also by simply using the value that overwrites the sensitive data.

DIRA[85] proposes to extend the GCC compiler to instrument the source code. It checks every control-sensitive data. The resulting programs can detect the buffer overflow attacks. After the detection of the attack, it will trace back the memory logging data to the input to identify which packets are responsible for the attack. The signature is simply the malicious packets information including the regular expression of the packets and the length constraints which can protect the attack using the same exploit. It is also limited by requiring source code, modifying GCC compiler, protecting control-sensitive data only.

Newsome *et al.*[72] proposes using the dynamic taint analysis to do the attack detection. They can find the vulnerability point easily by comparing with other approaches as they collect all of the runtime traces. They generate the signature using the input value that is used to overwrite the security critical data which is far from satisfaction. This thesis work is an extension of their work.

Packet Vaccine[95] randomizes address-like strings in packet to do the attack detection. For buffer overflow exploits, they generate alternative inputs by changing the size of the anomalously long fields. So, they consider the length of the input fields as the only factor that can cause a buffer overflow. Also, the generated bound result is coarse-grained bound result.

MemSherlock[80] requires source code and debugging information which may not be available for commodity software. It does the taint analysis for the copied value and the malloc size parameter, so there are buffer overflow cases as discussed in Chapter 4 that they can't handle.

The running example demonstrated in Figure 23 shows how the existing attempts will fail to diagnose it correctly. It has a programmer implemented data copy procedure that has an integer field controls the total number of writes instead of the source buffer size. The destination buffer is on heap, however, it is an array element of a struct. Its size isn't determined by a malloc size variable. None of the above approaches can diagnose this example correctly.

VSEF[69] performs diagnosis at execution level. It can successfully generate the vulnerability point but it can't provide any semantic information for the vulnerable program.

Larochelle's static analysis of source code to detect likely buffer overflow vulnerabilities[63] relies on source code and the semantic annotations.

2.2 SIGNATURE GENERATION

There are two classes of signatures, content-based or exploit-based signatures[60, 70, 40, 84, 61] and vulnerability-based signatures[24, 26, 94]. Autograph[60], EarlyBird[84], Honeycomb[61] and Polygraphs[70] search for invariants from the exploit traffic which are defined as exploit-based signatures[26]. Since they are based on specific exploit instances either by pattern matching or exploit syntax/semantics analysis. They may have both high false-positive rates and high false-negative rates[71, 74].

Brumley et al. are the first to propose generating signatures based on the semantic information of the vulnerable program using forward symbolic execution[24] and backward weakest precondition[26]. However, they assume the vulnerability point and the vulnerability condition are given in both approaches. Their contributions are more on the program path exploration (or the input analysis step in this thesis work). This thesis work concentrates on how to find the vulnerability point and vulnerability condition. This thesis's solution can bridge the gap of Brumley's work by providing the vulnerability point and vulnerability condition for buffer overflow vulnerabilities.

Shield[94] is an attack defense system by content-based filtering. The signature for the filtering is vulnerability-based signatures that are generated manually. This thesis's results can highly benefit the security engineers to find the vulnerability point and to understand the vulnerability condition to generate the patches and signatures accurately and efficiently.

ShieldGen[41] is a black-box approach for generating signatures for unknown vulnerabilities. Their coarse-grained bound results for their buffer overflow vulnerabilities may cause future non-control data attacks[29] by overwriting a local variable. They also can't handle the case when multiple input fields are involved in the BOVC.

Bouncer[34] applies function summary and path slicing to understand the vulnerability conditions when the vulnerability happens through unsafe usage of library functions. However, it can't handle when the vulnerability happens through some programmer implemented code blocks. Also, the bound result is obtained through static analysis of binary programs. The static analysis can't guarantee the local variable recovery.

2.3 BUFFER OVERFLOW DEFENSE SYSTEMS

Various mitigation techniques have been proposed to defend against the buffer overflow attacks. In this subsection, I summarize them briefly. They are grouped by their implementation approaches.

Compiler Patches or Extensions: Cowan *et al.* proposed StackGuard[37] and PointGuard[36] by inserting terminator or random canaries to protect the return address and function pointers, etc. Similar to StackGuard and PointGuard but using a more secure protection system, StackShield[91] enhances the GCC compiler to protect overwriting of the return address and function pointers. ProPolice[47] is a more sophisticated GCC compiler extension for protecting applications from stack-smashing attacks. It reorders the local variables that the `char` buffers always are allocated at the bottom so they can't overflow any other local variables. It also protects the stack frame in a similar way to StackGuard. Return Address Defender[31], RAD, also a GCC patch, protects return address by storing a copy of the return address and compare the return address with the stored one each time when it returns. StackGhost[48] provides similar approaches to both of the RAD and StackGuard to protect the return address. A cryptographic method of the return address is also proposed in StackGhost.

In order to protect stack smashing, researchers proposed to have non-executable stack[42]. But there are many workarounds of this solution. For example, buffer overflows on the heap/BSS/Data targeting the function pointers or longjmp buffers are not protected.

Boundary check may be the only solution that can prevent buffer overflow attacks completely. Jones and Kelly is the first to propose the GCC compiler patch[58] for bound checking at run-time. CRED[78] is then proposed to enhance the approach. Unfortunately, it suffers from the performance loss. Also the approaches in this category requires the access of source code and they can't be used to help the existing legacy systems.

Linking Stage: One possible source of buffer overflow comes from some of the standard string copy C libraries, like `strcpy()`, `sprintf()`, `memset()` etc., which don't do the bound checking. Libsafe[20] is proposed to patch the standard C libraries to do boundary check. Since then, Libverify[21] and Libsafeplus[18] are proposed to enhance Libsafe by providing return address verification etc. However, their bound checking only address the standard C libraries, they will fail to protect the program if the programmer implements the memory handling on his own.

Source Code Level: D. Wagner *et al.* proposed several methods of static analysis of source code to detect buffer overflow vulnerabilities[93, 92]. They proposed doing integer range analysis[93]. They also proposed modeling the program's correct execution behavior through source code analysis and use this model to do the run-time execution monitoring[92].

CodeSonar[11], a commercial tool that does source-code level analysis that identifies complex bugs at compile time. CCured[67, 68] and Cyclone[56] change the C program syntax to enhance type and bounds checking. However, these approaches won't be able to help for legacy systems of commercial software when the source code isn't available.

In summary, each of the above approaches has its own strengths and also has its own limitations. None of them can defend against buffer overflows completely. Due to the dichotomy between arrays and pointers, complete array bound checking is impossible in C. For example, some compilers can provide the array indexing protection like in `buffer[i]`, but fail the protection for `buffer + i`.

Most of the above approaches are analyzed and studied in the comparison studies[97, 83]. These two comparison study are published in 2003. Since then, researchers have explored other methods to defend against buffer overflow attacks including the following Program Randomization, Program Integrity Analysis and Dynamic Taint Analysis.

Program Randomization: Some researchers also proposed to randomize some parts of the operating system, e.g. the location of the stack, the location of the heap, the instruction set etc., to make the attack more difficult[22, 30, 59, 81, 86, 62, 23]. However, they still can't defend the attack completely.

Program Integrity Approaches: In 2006, Castro *et al.* proposed to achieve secure software system through enforcing Data-flow Integrity[28]. They do the static data dependency analysis and instrument the program at runtime to check the data-flow integrity. The same idea was presented by Zhou *et al.* in MICRO-37 2004 using hardware support[102].

Abadi *et al.* proposed to enforcing Control-flow Integrity[15] to defend against software attacks. This work checks the validation of the control target at run-time. The valid destination targets of each control are determined by the Control Flow Graph (CFG) generated through static analysis.

In 2008, Akritidis *et al.* proposed preventing memory error exploits by enforcing write integrity[17]. They use points-to analysis at compile time to compute the set of objects that can be written by each instruction in the program.

Dynamic Taint Analysis: Dynamic taint analysis technique emerges to computer security community in around 2004 to detect overwritten-based attacks. Using the dynamic taint analysis[72, 76, 39, 88, 90], a huge class of exploits can be detected for commodity software where source code and the patched libraries aren't available.

Crandall *et al.* proposed Minos[39], a micro-architecture implementation of Dynamic Taint Analysis. Suh *et al.* proposed similar Dynamic Information Flow Tracking [88] using hardware support (each register has an one-bit tag). In the same year, RIFLE[90], another hardware support computer architecture is proposed by Vachharajani *et al.* from Princeton to enforce information-flow security. Concurrently and independently, Newsome *et.al.* worked on the software implementation of this dynamic taint analysis in binary level[72]. Newsome's work[72] achieved dynamic taint analysis that doesn't need any hardware or OS modifications. In 2006, Qin *et al.* improved the binary level software approach by using a more efficient dynamic binary translator and by applying three optimization techniques[76]. Xu *et al.* also implemented fine-grained taint analysis in source code level[100].

3.0 THIS THESIS APPROACH

In this section, we will discuss about this thesis' approach towards automatic and accurate buffer overflow vulnerability diagnosis in detail.

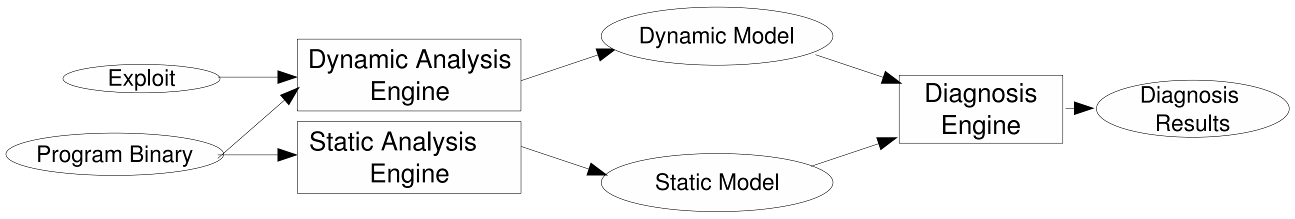


Figure 4: The System Architecture

The high level architecture is demonstrated in Figure 4. It consists of three major components: the dynamic analysis engine, the static analysis engine and the diagnosis engine. The dynamic analysis engine obtains the runtime information about the vulnerable program by executing the vulnerable program on the given exploit. We refer to this runtime information as dynamic model. The static analysis engine extracts the static information from the vulnerable binary program. The static analysis results are referred to as static model. The diagnosis engine correlates the dynamic model and the static model to infer the BOVP and BOVC. The diagnosis engine is the major contribution of this thesis work. Figure 5 illustrates the inner structure of the diagnosis engine which will be discussed in detail in Section 3.3.

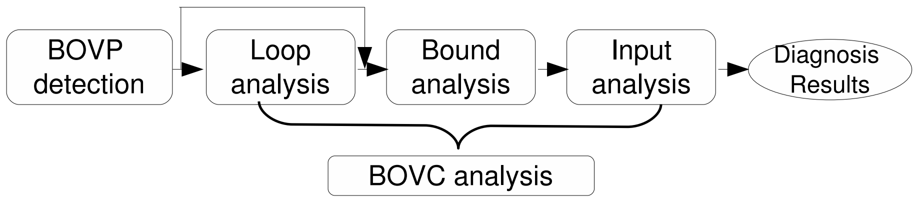


Figure 5: The Inner Structure of Diagnosis Engine

3.1 DYNAMIC ANALYSIS ENGINE

The dynamic analysis engine generates a dynamic model describing how the vulnerable program processes an exploit. At a higher level, the dynamic model consists of a trace of CPU and memory state updates as the vulnerable program executes on the exploit.

More specifically, the dynamic analysis engine runs the vulnerable program in an emulated environment that is capable of dynamic taint analysis. When sending the exploit input to the program, the dynamic analysis engine marks the input as tainted, and tracks the tainted data propagation in the program. It starts to record the dynamic model starting from when the exploit is initially read in. For each user-space instruction executed from the process of that program, the dynamic analysis engine records the current program counter, the current instruction, the values of its operands, the taint status of the operands, current `ESP` value etc. It keeps recording until the tainted exploit input is misused, such as when it is loaded into CPU's instruction pointer register. Similarly, if the exploit is to attack a kernel vulnerability, the dynamic analysis engine records the dynamic states for the instructions in the kernel space.

3.2 STATIC ANALYSIS ENGINE

The static analysis engine creates a static model from the vulnerable program (including any libraries it is linked against). Unlike the dynamic analysis model, the static model contains the information about the whole program, not just the execution path traversed and states touched during the dynamic analysis process. Thus, the static model can provide information that is unavailable in the dynamic model.

To construct a static model, the static analysis engine parses and disassembles the program binary. The static analysis engine raises the binary program to intermediate representation (IR) program and does preprocessing of the IR program. Specifically, it does the constant propagation, deadcode elimination, static single assignment (SSA) generation. The static analysis engine also generates the control flow graph (CFG) of the IR program that is raised from the program binary.

3.3 DIAGNOSIS ENGINE

Figure 5 illustrates the inner structure of the diagnosis engine. It includes the BOVP detection and the BOVC analysis. The BOVC analysis is achieved through loop analysis, bound analysis and input analysis. The loop analysis is skipped if the data copy procedure is known standard C library functions or system functions.

Given the dynamic model, the BOVP is achieved by searching back the taint trace to find out the latest instruction that overwrites the security critical data, which is the BOVP. Please refer to Section 7.6 for how we handle the malicious array index cases when the BOVP doesn't belong to any data copy procedure.

In the remaining of this section, we will concentrate on the discussion of how we combine the dynamic model, the static model, and the BOVP information to understand the BOVC. Specifically, we will discuss about the three steps analysis: the loop analysis, the bound analysis and the input analysis.

3.3.1 Loop Analysis

For buffer overflow vulnerabilities that happen through unsafe usage of standard C library functions, we obtain the total number of writes information through function summary. We apply loop analysis to understand programmer implemented data copy procedure. Our loop analysis step is based on the observation of all known buffer overflow attacks that most of them take place in loop context. We apply loop detection to locate a programmer implemented data copy procedure. We come up with a generalized data copy procedure and apply loop analysis to infer the total number of writes of executing this programmer implemented data copy procedure. In my previous work[44], the loop structure also helps on the switch design to enable predictive multiplexed switching in multiprocessor networks.

```
iv1 = iv1init;
iv2 = iv2init;
while (iv1 < Loop_Bound){
  writeAddress = startAddress + startOffset + iv2;
  [writeAddress] = {some value};
  iv2 += iv2upate;
  iv1 += iv1upate;
}
```

Figure 6: A Data Copy Procedure

Figure 6 demonstrates the pseudo-code for generalized data copy procedure. The $iv1$ and $iv2$ are loop induction variables (iv)[66] that are updated by regular patterns during each loop iteration. A buffer overflow occurs when the destination buffer is written beyond its bound.

In this data copy procedure, the number of loop iterations is defined in Equation 3.1 and the number of writes in each loop iteration to the destination buffer is $iv2_{update}$. So the total number of writes after the loop execution will be as in Equation 3.2. If the Equation 3.3 can be guaranteed, we can say this data copy procedure is safe. Otherwise, this data copy is vulnerable. In this example, if the user can control any variable in Equation 3.3, a buffer overflow attack may occur.

$$numIteration = \frac{Loop\_Bound - iv1_{init}}{iv1_{update}} \tag{3.1}$$

$$totalNumWrites = iv2_{update} \times numIteration \tag{3.2}$$

$$sizeof(destination) > startOffset + iv2_{init} + totalNumWrites \tag{3.3}$$

We assume the loop body won't change the Loop_Bound value to make the above equations accurate. The same loop heuristic is also applied in Larochelle's work[63]. We also apply some other loop heuristics. Here, we list all of the loop heuristics we use:

- The loop body won't update the Loop_Bound value. This loop heuristic is necessary and sufficient to make Equation 3.1, Equation 3.2 and Equation 3.3 accurate.
- If the loop exit condition is `source[j] == c`, we infer the Loop_Bound is the string length of the source buffer until the first encounter if c, we use `strlen(source|c)` to represent this. Specifically, if `c` equals 0, we infer the Loop_Bound is the source buffer size, `strlen(source)`.
- We assume the attacker may only control the Loop_Bound and the vulnerable buffer size variable to trigger the buffer overflow attack to simplify the problem. Even though if the attacker can control any variable in Equation 3.3, a buffer overflow attack may occur.

The above loop heuristics work very well in practice. We make the third loop heuristic to simplify our prototype implementation. In Section 4.2, we generate other buffer overflow cases by removing this assumption.

After this step analysis, we can figure out the vulnerable buffer and the total number of writes of the data copy procedure in symbolic forms. For known Standard C Library functions and known system calls that do the data copy, we can obtain the number of writes and the vulnerable buffer information directly through function summary as we know the definitions of those functions in advance. This loop analysis helps us to analyze the data copy procedures implemented by programmers.

We first do the *loop detection* to find the loop that contains the BOVP. We then do the *loop induction variable* (iv) detection. Especially we want to find the iv_i that is used to control the loop iterations and the iv_w that composes write index. We also do the *loop variant variable* detection which may be composed of a *loop invariant variable*[66] and an `iv` variable. The vulnerable buffer writes are indexed by those variables. We then study the *loop exit condition* to infer Loop_Bound information. If the loop exit condition `source[i] == 0`, we infer the Loop_Bound is the source buffer size. If the loop exit condition is `i < a`, we know the Loop_Bound is the integer a. For loop exit condition is `source[i] == 0`, we further look for the iv_i that composes the read index to calculate the number of loop iterations. The memory write instruction that is indexed through some loop variant variables may indicate the vulnerable buffer.

Multiple Loop Exit Conditions. For the loop constructs that have multiple loop exit conditions, we have multiple Loop_Bounds. We use logical *or* to represent them. For example, if we have loop exit conditions `source[j] == 0` and `i < a`, the Loop_Bound is `a || sizeof(source)`. We then look for the iv_1 that updates i and the iv_2 that composes the read index j. The estimated number of loop iterations is $min(\frac{a-iv1_{init}}{iv1_{update}}, \frac{sizeof(source)}{iv2_{update}})$.

Some critical instruction address information are transferred to the bound analysis step. Those instructions include the conditional jump instruction (loop exit condition), memory write instructions and memory read instructions.

Please refer to Section 7.1 for how we handle other complicated loop situations.

3.3.2 Bound Analysis

In this step, we want to answer the sizeof(dest) information in Equation 3.3. To be more specific, we want to find the location and size of the vulnerable buffer. The vulnerable buffer size can be a fixed number, e.g. the vulnerable buffer is on stack, Data/BSS, or an array member of a `struct`. It can also be a variable, e.g. the `malloc` size.

Our novel fine-grained bound analysis step is achieved by analyzing the memory traces. Our dynamic model contains the runtime traces. Our runtime traces record the current program counter, the current instruction, the values of its operands, the taint status of the

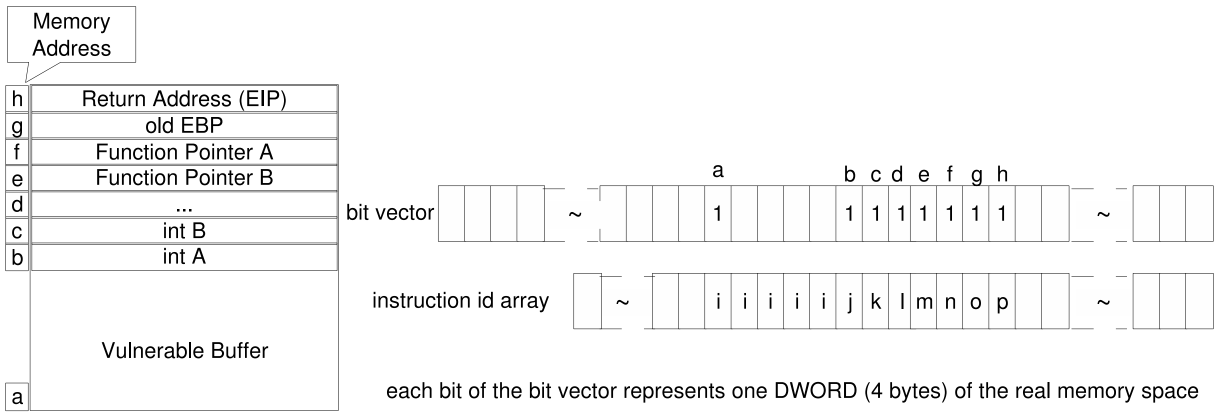


Figure 7: Our bound analysis approach: memory traces analysis

operands, the current ESP value, the function information of the instruction. The ESP is the stack pointer. They include all of the memory access and memory update information. The Figure 7 demonstrates our bound analysis approach. We use a bit vector to represent the memory space. We use one bit to represent one DWORD (4 bytes) in the real memory space. In practice, this works pretty well.

At higher level, we want to use the bit vector states to represent the memory states. Once we have memory states update, we update the bit vector states. We mark the bit to 1 when the corresponding memory region stores data e.g. return address, old EBP , metadata, local variable access etc. The EBP is the base pointer. We reset the bit to 0 when corresponding memory is cleaned, e.g. function return, free function call etc. Specifically, our static model can provide the memory operation functions information. Those memory operation functions include malloc, calloc and free. With this information, we can identify the memory operation function call in the runtime traces.

Each instruction contains the current ESP value. By parsing our traces for the first round, we can estimate the stack range using the ESP values, the heap bound using the malloc returns. Given the results after first round scan, we can distinguish a memory access on stack, heap or Data/BSS. During the second round scan, we will update the bit vector to represent the memory state updates. Specifically, we compare the ESP value of the current instruction with the previous instruction. If the ESP value is increased, it can be a POP instruction or a function return. We reset the bits corresponding to the memory regions between the two ESPs to 0. Otherwise, if the ESP is increased or isn't changed, we mark the bits corresponding to the memory accesses of that instruction to 1. The call instruction is handled separately because the ESP value is increased but there is no memory access of the call instruction.

We also handle the memory operation functions at the same time. Given the memory operation functions information from the static model, we can distinguish the function calls of malloc, calloc and free in our runtime traces. When we meet the malloc function call, we mark the bits for the metadata location to 1. Those metadata address information are also in our runtime traces. The calloc function is handled differently, because besides the metadata, the data region are also accessed in the runtime traces. We reset the data region specifically after the calloc return. We maintain a memory allocation history to record malloc and calloc function calls. Each record contains the instruction address that makes the function call, the size parameter information, metadata addresses information. When we meet a free function call, the function parameter has the value of the starting address of the memory region to free. We then search the memory allocation history for the latest record that matches the free function call to determine the memory region for bit vector resetting. We also maintain a memory deallocation history that contains each free function call information. This can be used to diagnose the double free vulnerability.

With the above implementation, we still can't handle when the vulnerable buffer had been already written by another data copy procedure before the data copying process that causes the buffer overflow attack. We meet this situation in real world application, atphttpd, during our experimental study. To handle this case, we also maintain an instruction-id array to help marking the bound of a memory object instead of each bit. If a write instruction is not within a data copy procedure, we simply use the instruction address as the instruction id. If a write instruction is within a library function, we use that library function's name concatenated with the call address as the instruction id. This is used to distinguish the different calls of the same library function. If a write instruction is within a programmer implemented data copy procedure, we use the function name concatenated with the loop id as the instruction id. This is used to diagnose the data copy implemented through sequential loops. When we reset the the bit vector, we will also reset the instruction id array. When we mark the bit vector, we will compare the current instruction id with the instruction id for the previous bit first. If they have the same id, we will update the instruction-id array only and we ignore the bit vector updates. The continuous memory region that is updated by instructions with the same instruction-id are inferred as data array. The continuous memory region that is updated by instructions with the same same function name concatenated with different loop-id are inferred as data array.

We can now infer the bound information using our bit vector which marks the bound of memory objects. Once we reach the first memory write instruction of the vulnerable data copy procedure (this information is given from the loop analysis), we first check the location of the vulnerable buffer. If it is on stack or Data/BSS, we simply search the next 1 on the bit vector. The vulnerable buffer size is 4×`searchSteps`. If it is on heap, we first do the same search for next 1 on the bit vector. We then compare the size results with the `malloc` size parameter value (this can be obtained from the memory allocation history). If they are equal, then the `malloc` size parameter determines the vulnerable buffer size. In this case, we check the taint information of the `malloc` size variable. Otherwise, the vulnerable buffer is an array member of a `struct` or `object`, we will return a fix number in this case. For example, in Figure 7, the vulnerable buffer is on stack and the size of the vulnerable buffer is $4 \times 5 = 20$ bytes. In our running example in Figure 23, the vulnerable buffer is on heap but the vulnerable buffer size is not determined by the `malloc` size parameter, but a fixed number 36. The variable access sequence matters in this approach. However, we can overwrite those variables that haven't been written.

In bound analysis step, we also check the taint information for Loop_Bound. This Loop_Bound is the corresponding function parameter if the data copy procedure is some standard C library function. If the BOVP belongs to a programmer written data copy procedure, the Loop_Bound information (instruction address) is passed from the loop analysis step. We also check the taint information of the `malloc` size parameter if the vulnerable buffer size is determined by that variable.

3.3.3 Input Analysis

Once we obtain the taint information of the vulnerable buffer size variable (if applicable) and the Loop_Bound, we calculate the dynamic slicing of the runtime traces that contain the tainted variables and perform forward symbolic execution along the trace to figure out how the user inputs are related with them. For example, if the vulnerable buffer size is determined by a `malloc` size parameter and that variable is tainted, we will obtain the slicing contains that variable. We then run symbolic execution to see how the input is related with it. If Loop_Bound is the source buffer size and the source buffer is tainted, we obtain the slicing that contains the source buffer. We perform symbolic execution to see how the input is related with the source buffer variable, or how the source buffer is constructed.

Only one execution path is examined in the above input analysis. It can't generate the condition to guarantee the program execution reach the vulnerability point. The signature generated using the above approach will introduce low false negatives and low false positives. The low false negatives can happen when a new exploit explores a new execution path and how the inputs are related with the vulnerability condition is different on that path. The low false positives happen when changing other input fields makes the vulnerability point unreachable.

We can also return the static slicing of the program that contains those malicious variables to help security engineers to do precise signature generation. This can greatly reduce the workload for security engineers.

As discussed in Section 1.2, the input analysis is an independent research field. This thesis only addresses this problem simply. This isn't a major contribution of this thesis work. In Section 8.2, we will discuss about how to enhance this thesis work towards generating zero false-positive and zero false-negative signatures for buffer overflow vulnerabilities.

3.4 HANDLING OTHER MEMORY CORRUPTION ATTACKS

3.4.1 Format String Vulnerability.

All of the format string vulnerabilities occur in Standard C `printf` family functions. Once we figure out that the BOVP belongs to one of those functions, we check the taint status of the format string. If the format string is tainted, this program is vulnerable for format string vulnerability. If we can determine the number of `args` in the `vararg`, we can restrict the number of directives appear in the format string. If the format string is not tainted, we continue our normal BOVD process.

3.4.2 Double Free.

Double free vulnerability occurs by calling free function twice on the same memory address without allocating the same memory region between the two free function calls. The bound analysis step of our diagnosis engine maintains a memory allocation history and a memory deallocation history. We diagnose the double free vulnerability by extending the current BOVD implementation. When we meet a free function call, we check the current memory deallocation history. If there is a record that frees the same memory region and we can't find a record on the memory allocation history that allocate this memory region between this two free function calls, we can diagnose that there is a double free vulnerability. We return the two free records as the diagnosis results. This result is much better than the current systems which can only indicate which memory address has double free. Our double free diagnosis results can return which two free function calls can cause double free.

4.0 BUFFER OVERFLOW CASE STUDY

From Chapter 2, we can see the existing diagnosis attempts and vulnerability-based signature generation approaches are far from satisfaction. Researchers use their own heuristics on how a buffer overflow attack can happen when they design their defense and debugging systems. There is no previous research on studying the reasons that can cause a buffer overflow or the buffer overflow case study.

In this Chapter, we do buffer overflow vulnerability case study. We only consider the buffer overflow vulnerabilities that can be controllable by the user inputs. Because only in those cases, the attacker can attack the buffer overflow vulnerability by generating zero-day buffer overflow attacks through Internet. We don't consider those buffer overflow vulnerabilities that are built-in the programs. For example, both of the sizeof(dest) and total number of writes are fixed and the sizeof(dest) is less than the total number of writes. There are six basic basic cases that are built on serials assumptions. We provide the diagnosis results, signature suggestions and patch suggestions for each basic case. By removing each assumption of the basic cases, we come up with the other buffer overflow vulnerability cases. We prove that the case study results (basic cases and other cases) are complete. By the end of this Chapter, we study some sample Standard C Library and System functions that have caused buffer overflow vulnerabilities in real world applications. We will state under what conditions those functions may cause buffer overflows.

4.1 SIX BASIC CASES

The six basic cases cover all of the real world buffer overflow vulnerabilities we have studied. The basic bases are based on the following assumptions:

- Assume the buffer overflow attack happens through a data copy procedure. This data copy procedure is a code block that writes contents to a data array. It can be in any format including the generalized data copy procedure (one or more loop exits), data copy procedure implemented through nested loops, or sequential write statements.
- Assume the attacker can only control the sizeof(dest) or/and Loop_Bound.
- Assume the Loop_Bound includes the two major loop exit condition cases for a data copy procedure. One is `source[i] == c` and the other is `j < a`. The Loop_Bound is `strlen(source|c)` and `a` respectively.
- Assume there is one Loop_Bound or one loop exit condition of the data copy procedure. So the generalized data copy procedure with multiple loop exits or data copy procedure implemented through nested loops are excluded.

To simplify the analysis results, we assume the *iv* composes the read index is $i = i + 1$, the *iv* for loop iteration variable is $j = j + 1$ and the write index contains an *iv* that has form $k = k + 1$, to simplify the analysis results.

We use s to represent a fixed value for size, a to represent a fixed value for an integer, us to represent a user controllable value for array size, ua to represent a user controllable value for an integer. We use `U(us)` and `U(ua)` to represent how the user inputs are related with the us and ua.

Loop Exit Condition is source[i] == c. Using one of our loop heuristics, the Loop_Bound is the string length of the source buffer until the first encounter of `c` when the loop exit condition is `source[i]== c`. We use `strlen(source|c)` to represent this. There are three buffer overflow vulnerability cases based on what values the user input can control: the `strlen(source|c)`, the vulnerable buffer size, or both.

- Case I: **The `strlen(source|c)` is related with user input.** Source buffer is from user input and the vulnerable buffer has a fixed value size. We found several real world buffer overflow vulnerable applications belonging to this case, e.g. CSRSS(MS06-040). *The diagnosis results*: vulnerable buffer location, vulnerable buffer size (s), the total number of writes is `strlen(source|c)` (us) and how it is related with the user inputs ($U(us)$). *The signature suggestion*: `s > U(us)`. *The patch suggestion*: add `s > us` condition check before the data copy procedure. The example vulnerable program is demonstrated in Figure 24.

- Case II: **Vulnerable buffer size is related with user input.** In this case, the source buffer is fixed. The vulnerable buffer is on heap and the vulnerable buffer size is determined by the malloc size parameter. This variable is directly from user input or related with user input (e.g. integer overflow happens from the input value to the malloc size). *The diagnosis results*: heap buffer overflow, malloc size parameter (ua), the `strlen(source|c)` (s), how the malloc size variable is related with the user inputs ($U(ua)$). *The signature suggestion*: `U(ua) > s`. *The patch suggestion*: add `ua > s` condition check before the data copy procedure. The example vulnerable program is demonstrated in Figure 25.

- Case III: **Both of the `strlen(source|c)` and vulnerable buffer size are related with user inputs.** *The diagnosis results*: heap buffer overflow, malloc size parameter (ua), the `strlen(source|c)` (us), how the malloc size parameter and source buffer are related with the user inputs ($U(ua)$ and $U(us)$). *The signature suggestion*: `U(ua) > U(us)`. *The patch suggestion*: add `ua > us` condition check before the data copy procedure. The example vulnerable program is demonstrated in Figure 26.

Loop Exit Condition is `j` $<$ `a`. The Loop_Bound is a when the loop exit condition is `j < a`. There are three buffer overflow vulnerability cases based on what values the user inputs can control, the Loop_Bound a, the vulnerable buffer size, or both.

- Case IV: **Integer bound a is related with user input.** The vulnerable buffer has a fixed size value. We find real world vulnerable program, MS07-017, belongs to this case. *The diagnosis results*: vulnerable buffer location, vulnerable buffer size (a), integer bound (ua), how the integer bound ua is related with the user inputs ($U(ua)$). *The signature suggestion*: `a > U(ua)`. *The patch suggestion*: add `a > ua` condition check before the data copy procedure. The example vulnerable program is demonstrated in Figure 27 which is motivated by real world vulnerable application, ANI(MS07017).

- Case V: **Vulnerable buffer size is related with user input.** The vulnerable buffer is on heap and its size is determined by a variable. The integer bound a is a constant. We find real world vulnerable program, `nullhttpd`, belongs to this case. *The diagnosis results*: heap buffer overflow, malloc size parameter (ua), integer bound (a), how the malloc size variable is related with the user input ($U(ua)$). *The signature suggestion*: `U(ua) > a`. *The patch suggestion*: add `ua > a` condition check before the data copy procedure. The example vulnerable program is demonstrated in Figure 28 which is motivated by real world vulnerable application, nullhttpd.
- Case VI: **Both of the vulnerable buffer size variable and the integer bound variable are related to the user inputs.** We find real world vulnerable program, `PNG(MS05-025)`, belongs to this case. *The diagnosis results*: heap buffer overflow, malloc size parameter (ua_1), integer bound (ua_2), how the malloc size parameter and integer bound variable are related with the user inputs ($U(ua_1)$ and $U(ua_2)$). *The signature suggestion*: U(ua_1) > U(ua_2). *The patch suggestion*: add ua_1 > ua_2 condition check before the data copy procedure. The example vulnerable program is demonstrated in Figure 28 which is motivated by real world vulnerable application, PNG(MS05025).

4.2 OTHER CASES

There are many other buffer overflow vulnerability cases beyond the above six basic cases. The following four categories of other cases are generated by removing each of the above assumptions.

Multiple loop exit conditions. There are two situations when we have multiple loop exits. One is when there are multiple loop exits on the generalized data copy procedure. The other is when the data copy procedure is implemented through nested loops. Figure 8 and Figure 9 illustrate these two situations.

```
iv1 = iv1init;

iv2 = iv2init;

iv3 = iv3init;

while(iv1 < Loop_Bound1 || iv3 < Loop_Bound2){

  writeAddress = startAddress + startOffset + iv2;

  [writeAddress] = {some value};

  iv2 += iv2upate;

  iv1 += iv1upate;

  iv3 += iv3update;

}
```

Figure 8: A Data Copy Procedure with Multiple Loop_Bounds (I)

For the generalized data copy procedure that have multiple loop exit conditions, we have multiple Loop_Bounds as illustrated in Figure 8. We use logical *or* to represent them. For example, if we have loop exit conditions `source[i] == 0` and `j < a`, the Loop_Bound is `a || sizeof(source)`. We then look for the iv_1 that updates the read index `i` for `source` and the iv_3 that updates `j`. The estimated number of loop iterations is $min(\frac{sizeof(source)}{iv1_{update}}, \frac{a - iv3_{init}}{iv3_{update}})$. Besides the two Loop_Bound variables, the vulnerable buffer size is the third variable that is controllable by user inputs. Equation 4.1 and Equation 4.2 show how to calculate the total number of loop iterations and the total number of writes generally.

$$numIteration = min(\frac{Loop\_Bound_1 - iv1_{init}}{iv1_{update}}, \frac{Loop\_Bound_2 - iv3_{init}}{iv3_{update}}) \tag{4.1}$$

$$totalNumWrites = iv2_{update} \times numIteration \tag{4.2}$$

For example, the vulnerable buffer is on stack with fixed size (s), the integer bound (ua) and the source buffer size (us) are related with user inputs. *The diagnosis results*: vulnerable buffer location, vulnerable buffer size (s), integer bound variable (ua), source buffer size (us),

```
iv1 = iv1init;
iv2 = iv2init;
iv3 = iv3init;
while(iv1 < Loop_Bound1) {
  while(iv3 < Loop_Bound2) {
    writeAddress = startAddress + startOffset + iv2;
    [writeAddress] = {some value};
    iv2 += iv2upate;
    iv3 += iv3update;
  }
  iv1 += iv1upate;
}
```

Figure 9: A Data Copy Procedure implemented through Nested loops

how integer bound (ua) and source buffer (us) are related with the user inputs ($U(ua)$ and $U(us)$).

The signature suggestion: `s > min(U(ua), U(us))`. *The patch suggestion*: add `s > min(ua, us)` condition check before the data copy procedure. The example vulnerable program is demonstrated in Figure 30.

For the data copy procedure that is implemented through nested loops, we have multiple Loop_Bounds as illustrated in Figure 9. The total number of loop iterations is the product of the number of the outer loop iterations and the number of the inner loop iterations. Equation 4.3 and Equation 4.4 show how to calculate the total number of loop iterations and the total number of writes.

$$numIteration = \frac{Loop\_Bound_1 - iv1_{init}}{iv1_{update}} \times \frac{Loop\_Bound_2 - iv3_{init}}{iv3_{update}} \tag{4.3}$$

$$totalNumWrites = iv2_{update} \times numIteration \tag{4.4}$$

Similarly, a buffer overflow attack can happen when the attacker can control the Loop_Bound1 or/and Loop_Bound2 or/and the sizeof(dest).

The *iv* detection problem for nested loops has been studied very well by researchers[98]. We leave this as a future work to implement the *iv* detection for nested loops and then to calculate the total number of writes by calling nested loops using Equation 4.4.

Other variables in Equation 3.3. In the basic case study, we assume the user input can only control the Loop_Bound and the vulnerable buffer size. However, it is possible that the user input can control the other variables in Equation 3.3 including $iv1_{init}$, $iv2_{init}$, $startOffset$, or even $iv2_{update}$ and $iv1_{update}$. The example vulnerable program is demonstrated in Figure 31.

Malicious Array Index. In the basic case study, we assume the buffer overflow attack happens during a data coping process. It can also happen when the index of the vulnerable buffer is controllable by the user input. Among the 377 vulnerabilities for which Microsoft has issued security bulletins between 2003 and 2006, two applications are known using a field from the user input as an array index without checking whether the array index is within the array bound. ShieldGen[41] is a black-box approach for generating signatures for unknown vulnerabilities. They are believed to be the scheme that can generate the highest quality signatures and can cover the most situations comparing with any other existing approaches (e.g. exploit based or execution based). However, ShieldGen[41] can't handle the malicious array index cases described above. They even can't provide any information on this case.

This thesis work can provide more information on this famous notoriously hard problem. In the BOVP detection step, we search back the trace for the latest instruction that does the malicious write. If this instruction doesnt belong to any data copy procedure (programmer written loops, known library or system functions), we will keep searching back until we find one that is part of a data copy procedure as the BOVP or we find nothing. If we cant find any BOVP, our diagnosis result will state that the vulnerability may belong to this unchecked array index case. The example vulnerable program is demonstrated in Figure 32.

Other Loop_Bound. In the basic cases, we assume the Loop_Bound includes the two major loop exit condition cases for a data copy procedure. One is `source[i] == c` and the other is `j < a`. The Loop_Bound is `strlen(source|c)` and `a` respectively. Is there any other Loop_Bound for a data copy procedure? In paper[46], it presents a loop bound analysis approach based on combination of standard program analysis techniques including program slicing, abstract interpretation and invariant analysis. Here we use function $\phi()$ to represent other Loop_Bound cases. In Figure 33, we demonstrate the other Loop_Bound case. We can apply the technique described in paper[46] to analyze the bound results.

4.3 CASE STUDY IS COMPLETE

In this section, we prove that the above six basic cases and four categories of other cases can cover all of the buffer overflow vulnerability cases.

Theorem. The six basic cases and the four category other cases cover all of the buffer overflow vulnerability cases. The buffer overflow vulnerabilities are those that can be controllable by the user inputs.

Proof by enumeration: The six basic cases are based on the following four assumptions:

- Assume the buffer overflow attack happens through a data copy procedure. This data copy procedure is a code block that writes contents to a data array. It can be in any format including the generalized data copy procedure (one or more loop exits), data copy procedure implemented through nested loops, or sequential write statements.
- Assume the attacker can only control the sizeof(dest) or/and Loop_Bound.
- Assume the Loop_Bound includes the two major loop exit condition cases for a data copy procedure. One is `source[i] == c` and the other is `j < a`. The Loop_Bound is `strlen(source|c)` and `a` respectively.
- Assume there is one Loop_Bound or one loop exit condition of the data copy procedure. So the generalized data copy procedure with multiple loop exits or data copy procedure implemented through nested loops are excluded.

First we assume the buffer overflow attack happens through a data copy procedure which can be in any format. If the buffer overflow happens through a data copy procedure, there is a buffer overflow if and only if the sizeof(dest) is less than the total number of writes after the data copy process. So the attacker can attack the buffer overflow vulnerability through controlling the sizeof(dest) or/and the total number of writes. We assume the attacker can only control the Loop_Bound to control the total number of writes. We also assume that there is only one Loop_Bound and the Loop_Bound is either `strlen(source|c)` or `a`. We can then construct the six basic cases as listed in Section 4.1.

By removing the first assumption "Assume the buffer overflow attack happens through a data copy procedure", we get the first category of the other cases that buffer overflow happens through malicous array index.

By removing the second assumption "Assume the attacker can only control the sizeof(dest) or/and Loop_Bound", we get the second category of the other cases that the buffer overflow happens through controlling the other variables in Equation 3.3. From Equation 3.3, we can see the total number of writes is determined by several variables including $iv1_{init}$, $iv2_{init}$, $startOffset$, $iv2_{update}$, $iv1_{update}$ and Loop_Bound. So besides Loop_Bound, the attacker can also control $iv1_{init}$, $iv2_{init}$, $startOffset$, $iv2_{update}$ and $iv1_{update}$ to control the total number of writes.

By removing the third assumption "Assume the Loop_Bound includes the two major Loop_Bounds", we get third category of the other cases that buffer overflow happens through other Loop_Bound.

By removing the fourth assumption "Assume there is one Loop_Bound", we get the fourth category of the other cases that buffer overflow happens through a data copy procedure that have multiple Loop_bounds. This can be one of the two situations. One is that there are multiple Loop_Bounds in the generalized data copy procedure. The other is that the data copy procedure is implemented through nested loops.

Now we cover the whole space of buffer overflow vulnerability cases that can be controllable by an attacker without any assumption. We conclude the buffer overflow case study results, six basic cases and four categories of other cases, are complete.

h

Table 1: The sample unsafe standard C libraries and system data copy functions study: char* s1, s2, format; int n, socket, flags; File* stream;

Func. Name	Dest. Buffer	Loop_Bound	Unsafe Condition
strcpy(s1, s2)	s1	strlen(s2)	sizeof(s1) < strlen(s2)
strncpy(s1, s2, n)	s1	n	sizeof(s1) < n
memcpy(s1, s2, n)	s1	n	sizeof(s1) < n
memset(s1, s2, n)	s1	n	sizeof(s1) < n
recv(socket, s1, n, flags)	s1	n	sizeof(s1) < n

4.4 STANDARD C LIBRARY FUNCTIONS AND SYSTEM FUNCTIONS THAT CAN CAUSE BUFFER OVERFLOWS

There are many unsafe Standard C Library and system data copy functions that can cause buffer overflow vulnerabilities. We studied those functions in advance and got the function summaries. Once we meet a buffer overflow that is caused by any of those functions, we can retrieve the vulnerable buffer and the total number of writes information directly.

In Table 1, we list sample standard C library and system data copy functions that can be unsafe and that have been found causing buffer overflow vulnerabilities in our experiments on real world vulnerable applications.

In Windows Vista, certain APIs are banned including `strcpy`, `strncpy` etc. However `memcpy` is not on the banned APIs list which causes the stack buffer overflow in MS07-017[53]. From the table, we can see `memcpy` is actually as dangerous as `strncpy`. Our study can help security engineers to decide which functions and under what conditions they are unsafe.

5.0 IMPLEMENTATION

In this Chapter, we discuss our implementation details. As the major contribution of this thesis work is on the diagnosis engine, I will concentrate on the implementation details of the diagnosis engine which is independently implemented by me. For the details on dynamic analysis engine and static analysis engine, please refer to BitBlaze [5] project for details.

5.1 DYNAMIC ANALYSIS ENGINE

We have implemented the dynamic analysis engine, TEMU [5], on QEMU [13], a whole-system emulator that uses dynamic translation techniques. At runtime, an instruction in the guest system is translated into several micro operations, which are then executed in the host system. This feature enables us to perform dynamic instrumentation on any instructions in the guest system (including the instructions in the kernel space). To implement dynamic taint analysis, we have extended these micro operations to propagate taint information. We have also modified the virtual network device to taint the network input.

In addition, we instrument the beginning of each instruction. If it is from the examined process, we disassemble it and record the runtime information, including the current program pointer, the current instruction, and the states of its operands etc. If the operands are tainted, we also record the taint information.

TEMU only provides hardware-level information about the guest system, such as CPU and memory state. However, to determine if an instruction is from the examined process or the kernel, we need software-level knowledge from the guest system. To achieve this, we have developed a kernel module and load it into the guest system to obtain the necessary software-level information. This module is aware of the creation and deletion of processes. When a new process is created, the kernel module obtains the value of current `CR3` register. As `CR3` contains the physical address of the current process's page table, it is different (and unique) for each process. All this information is passed on to the dynamic analysis engine through a predefined I/O port. Therefore, when instrumenting an instruction, we simply check the current `CR3` to determine if the examined process is running.

5.2 STATIC ANALYSIS ENGINE

The static analysis engine is implemented on top of our binary analysis framework, Vine[5], which can disassemble an x86 binary, converts it into an intermediate representation (IR), and perform serials of preprocessing on the raised IR program.

We translate the x86 program to the VinE modeling language [5]. This language simplifies the analysis of complex x86 to that of a simple RISC-like language.

After we raise the x86 program to VinE modeling language, our static analysis engine will perform serials of preprocessing of the raised program. Specifically, it will do constant propagation, deadcode elimination, static single assignment (SSA) generation. It can generate the control flow graph (CFG) of the raised IR program. It also provides the functionality of program chopping and the forward symbolic program execution. It also provides the calling information for the memory functions. Those memory functions include malloc, calloc and free.

Symbolic Execution. The symbolic execution is used in our input analysis step. In order to perform sound symbolic execution, we must correctly interpret the semantics and effects of all assembly statements. The x86 instruction set is complex. Many instructions have implicit side effects (e.g., add sets the eflags register on overflow), may have implicit operands (e.g., the memory segment selector), may behave differently for different operands (e.g., shifts by 0 do not set eflags), and there are even single instruction loops (e.g., rep instructions). Thus, to reduce the complexity of the symbolic execution logic, for each instruction that needs to be executed symbolically, we first translate it into a sequence of much simpler intermediate representation (IR) statements (VinE). Our IR resembles a RISC-like assembly language. The translation from an x86 instruction to our IR is designed to correctly model the semantics of the original x86 instruction, including making all the implicit side effects explicit (e.g., setting the eflags register). We then perform symbolic execution on the IR statements, instead of directly with the x86 instruction set.

5.3 DIAGNOSIS ENGINE

Besides detecting the BOVP by searching back the runtime traces to find out the latest instruction that wrote the security critical data, our diagnosis engine performs three steps analysis towards understanding the BOVC. These three steps analysis includes loop analysis, bound analysis and input analysis.

The loop analysis includes loop detection, loop induction variable (*iv*) detection[66], loop invariant and variant variable detection[16, 66], loop exit condition extraction, memory read and memory write extraction. My loop detection can handle both of the reducible graph and irreducible graph[89, 52]. My loop *iv* detection is based on the algorithm provided in[66]. My loop invariant and loop variant variable detection is implemented based on its definition[66]. I return the code slicing of the loop body for each loop exit condition and each memory write index if the write index is a loop variant variable. I also return the code slicing of the memory read index if the loop exit condition is `strlen(source|c)`. I then evaluate the results for the loop exit condition and the memory write index on the sliced program.

Program slicing is a subset of the programs that contains the statements which can affect the results of the interested variable(s)[96]. My slicing works on the IR program. I apply data flow analysis to retrieve the code statements that are related with our interested variables (e.g. loop exit condition, write index).

The bound analysis scans the runtime traces for two rounds. During the first round, it estimates the stack range using the `ESP` values, the heap bound using the malloc returns. The `ESP` is the stack pointer. Our runtime traces contain the current `ESP` in each instruction. During the second round scan, we update the bit vector and instruction id array to represent the memory state updates to mark the memory object bounds. Those memory state updates include the function call, function return, memory access, memory allocation and memory deallocation. Our current memory function handling can only handle the calling conventions we have met in our RedHat73 test bed. More calling conventions[6] can be extended for other applications running on different platforms. For the windows applications we evaluated in Section 6.2, we simply identify the vulnerable buffer location is on heap or on stack.

They calling conventions we can handle include passing arguments by register `eax`, by pushing on stack either explicitly (e.g. `push %eax`) or implicitly (e.g. `movl $0xa (%esp, 1)`) and passing function return results using register `eax`. For example, for function `malloc`, the function argument is the `malloc` size and the function return is the starting address of the allocated memory. The meta data are accessed in the `malloc` function body; for function `free`, the function argument is the starting address of the memory to be deallocated.

The input analysis returns the dynamic slicing of the runtime traces that contains the malicious variables and then perform symbolic execution on the trace to see how the inputs are related with the malicious variables.

5.3.1 Loop Detection

I did thorough study on the loop detection research area[89, 52, 33] and implemented the loop detection algorithm in a comprehensive way given the Control Flow Graph(CFG) of the vulnerable program. The implementation can handle both of the reducible graph and irreducible graph cases[89, 52, 33]. This is much better than the BlackHat[3] community's approach and IDAPro's[7] approach. The irreducible graph happens in practice when we were testing our loop detection implementations.

A basic algorithm to detect loops is proposed in [16, 66], by applying dominator analysis. Node d is the dominator of node n if every path from the entry node of CFG to n goes through d. Briefly speaking, this algorithm works by calculating the dominators $D(n) = d_1, ..., d_i, ...$ for each node n and testing if there is an edge from node n to node d_i. If there is such an edge, then this edge is the back edge, node d_i is the loop head and node n is in the loop body that returns to the loop head.

This algorithm can only detect loops with single entries, reducible graph[89]. But in practice, graphs contain loops with multiple entries, which are defined as irreducible graph, happen. As Figure 10(a) shows, Loop 2 has two entries: Node 5 and 6. In this case, we apply the node collapsing algorithm [52, 89] to determine if a CFG contains a loop with multiple entries, and the node splitting algorithm [52, 89] to break such a loop. We then can make an irreducible graph reducible.

Figure 10(c) shows the CFG after the node splitting algorithm is performed. The node collapsing and node splitting procedures may be performed multiple times until the resulted CFG contains no loops with multiple entries. In this example, after the first-time node splitting procedure, we have already got a CFG containing no loops with multiple entries. The final CFG is shown in Figure 10(d), in which Loop 2 has been broken to have a single entry. Then we can perform the basic algorithm on this CFG to detect loops.

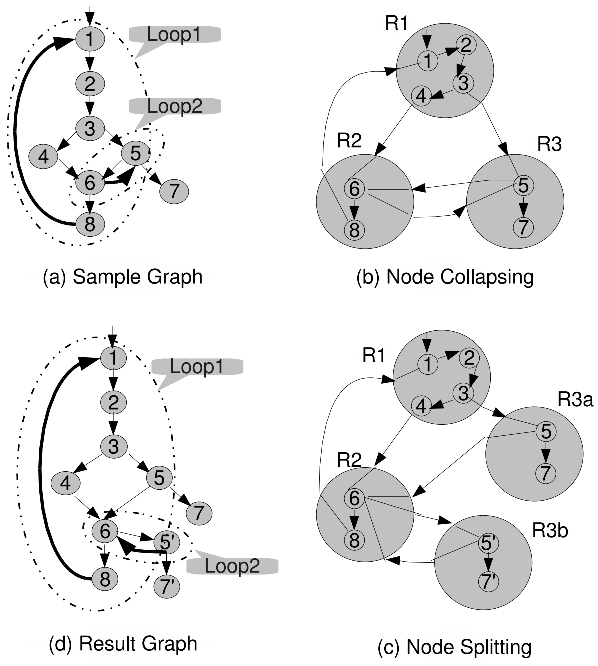


Figure 10: Examples of Transforming Irreducible Graph to Reducible Graph

Figure 11 illustrates the CFG of a real-world program, `passlogd`. Using the above algorithms, we can identify two loops in this CFG. Now given the instruction 0x7801212f that overwrites the returned address, we first locate that this instruction is in the block_135.

Then with the information of identified loops, we can know that it is in Loop 1, which has entry node 122.

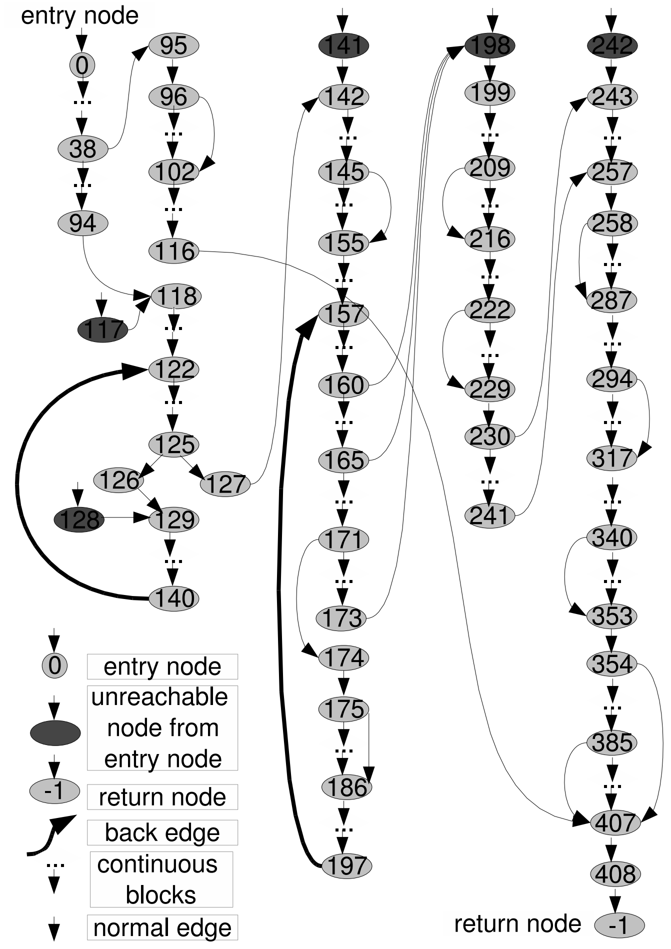


Figure 11: The CFG of passlogd

5.3.2 Loop Induction Variable Detection

Our goal of loop analysis is to find out how the number of writes is determined by different variables. Based on the observation that the loop memory accesses in a data copy procedure are usually composed and the loop exit condition are usually controlled by loop induction variables, I implemented loop induction variable detection.

Induction variables are discussed in detail in [66]. Basically they are variables that are updated through regular mathematics patterns during each loop iteration. The *basic* or *fundamental induction variables* are those that are modified by some constant amount each time, e.g. `i = i + 4`. The *dependent induction variables* are those that are linear equations of the *basic induction variable*, e.g. `j = a` $\times$ `i + b`, where `a` and `b` are constants or loop invariants.

The input of the induction variable detection is a loop body in Static Single Assignment(SSA)[66]. There are ϕ-statements, $lhs = \phi(a_1, a_2, ..., a_n)$ and single assignment statements, $x = y$ when both of the right hand side and left hand side are variables in our Vine IL. The existence of these statements makes the induction variable detection extremely difficult. I modified the algorithm given on the text book[66] to be able to handle this by creating a hashtable. The key of the hashtable is a variable and the value is a list of variables that are the same through the program propagation of that key variable. The algorithm finds the *basic induction variables* that are satisfied with pattern `i = i` $\pm$ `a` first, then recursively look for the *dependent inductive variables* that match pattern `k = m` $\times$ `a` where a is a constant variable and m is one of the new found inductive variables in the previous round. During the first iteration, m is one of the *basic induction variables*. This iteration will continue until no new *dependent inductive variables* can be found. The algorithm of the induction variable detection is given in Table 2.

Using the implemented algorithm to handle the sample program block in $x = x + 1; z = y \times 5; y = x + 6; q = z \times 2;$, it will find x as the *basic induction variable*. During the following iterations, it will find y as a *dependent induction variable* in the first round, z in the second round, q in the third round. During the fourth iteration, the result is empty, so the program stops.

5.3.3 Loop Invariant and Variant Variables Analysis

In some situation, some variable is not an induction variable but we still want to know if it is a loop variant or loop invariant variable. If a computation produces the same value in every loop iteration, it is a loop invariant variable, otherwise, it is a loop variant variable. This concept is defined well in Chapter 13.2[66].

The definition of loop invariant variable is defined as below: A definition t = xy in a loop is loop invariant if x and y are constants, or all reaching definitions of x and y are outside the loop, only one definition reaches x (or y), and that definition is loop-invariant.

Based on this definition, I designed and implemented the algorithm to find loop invariants and loop variants variables as below. During the first iteration of the loop body, for each statement, if the right hand side variable(s) are constants or no-reaching definitions in the loop body, they are saved in a list, say loop_invariant1. During the next round of iteration, we check the right hand value(s) that either both are members of loop_invariant1, or at least one is a member of loop_invariant1, the other is either a constant or no-reaching definition in the loop body. The lvalue of this statement is a loop invariant, we save them in loop_invariant2. This process will continue. For iteration i, we check the right hand value(s) of each statement that either both are members of loop_invariant(i-1) or at least one is a member of loop_invariant(i-1) and the other is a constant or no-reaching definition in the loop body or a member of $\bigcup_{j=1}^{i-2}(loop\_invariant(j))$. This process will continue until no more loop invariants can be found. By merging all of the lists, we obtain the final result of loop invariants of the left hand values (lvalues).

The remaining left hand values are loop variants, we save them in a list named loop_variants. The algorithm of calculating the loop variant variables is given in Table 3. Now given a variable, either a left hand value (lvalue) or a right hand value (rvalue), if it exits in loop_variants list, we say it is a loop variant variable, otherwise, we say it is a loop invariant variable. As the same as in loop induction variable detection algorithm, I build a hashtable to handle the ϕ-statement and single assignment statement when both of the right hand side and left hand side are variables. Using this loop invariant and loop variant variables analysis, we can find those variables that are not induction variables but are updating itself during each loop iteration. These information is useful in the loop memory access pattern analysis.

Table 2: The induction variable detection algorithm

Algorithm: Induction Variable Detection

```
1: Input: Loop body in SSA (single static assignment) format
2: Output: The induction variables of the loop body
3: i = 0;
4: while ((a = nextStatment(LoopBody)) != null)
5:     if (patten(a) = patten(x = x ± i))
6:         if (i = constant)
7:             $loop\_iv_i \cup = x$
8: while ($loop\_iv_i$ != null)
9:     i++;
10:    Init(LoopBody);
11:    while ((a = nextStatment(LoopBody)) != null)
12:        if (patten(a) = patten(y = z ± i))
13:            if (i = constant and z ∈ $loop\_iv_{i-1}$)
14:                $loop\_iv_i \cup = y$
15:        if (patten(a) = patten(y = z × j))
16:            if (j = constand and z ∈ $loop\_iv_{i-1}$)
17:                $loop\_iv_i \cup = y$
18: for (j = 0; j ¡= i; j++)
19:    $loop\_iv \cup = loop\_iv_i$
20: return $loop\_iv$
```

Table 3: The loop variant and loop invariant variable detection algorithm

Algorithm: loop variant and loop invariant variable detection

```
1: Input: loop body in SSA (single static assignment) format
2: Output: loop variant variables of the loop body
3: i = 0;
4: while ((a = nextStatment(LoopBody)) != null)
5:     if (patten(a) = patten(x = i op j))
6:         if (i,j = constant or have no reaching def)
7:             $loop\_invar_i \cup = x$
8: while ($loop\_invar_i$ != null)
9:     i++;
10:    Init(LoopBody);
11:    while ((a = nextStatment(LoopBody)) != null)
12:        if (patten(a) = patten(x = i op j))
13:            if (i,j $\in loop\_invar_{i-1}$)
14:                $loop\_invar_i \cup = x$
15:            if (i $\in loop\_invar_{i-1}$ and j $\in constant \cup noreachingdef \cup loop\_invar_{0,1,i-2}$)
16:                $loop\_invar_i \cup = x$
17:            if (j $\in loop\_invar_{i-1}$ and i $\in constant \cup noreachingdef \cup loop\_invar_{0,1,i-2}$)
18:                $loop\_invar_i \cup = x$
19: for (j = 0; j $\leq$ i; j++)
20:    $loop\_invar\_lhs \cup = loop\_invar_i$
21: while (v = nextVariable(LoopBody) != null)
22:    if (v $\notin loop\_invar\_lhs$)
23:        $loop\_var \cup = x$
24: return $loop\_var$
```

6.0 EXPERIMENTS

In this Chapter, we evaluate our approach through several aspects. First, we evaluate how our generalized loop structures can handle the loops in the real programs. The generalized loop structures include the Generalized-data-copy-procedure, the Generalized-data-copy-procedure with multiple loop exists and the data copy procedure implemented through nested loops. They are illustrated in Figure 6, Figure 8 and Figure 9 respectively. The real programs we studied include the WCET (Worst Case Execution time) benchmark[14] and several real world applications that have multiple known vulnerabilities. These several real world applications include 3proxy, atphttpd, ghttpd, nullhttpd.

We also have performed a series of study on real world buffer overflow vulnerable applications to evaluate the effectiveness of our BOVD approach and to confirm our buffer overflow vulnerability case study results. We use six representative examples to demonstrate the effectiveness of our approach. Among the six examples, the `PNG` and `RPC` vulnerable applications have data copy procedures implemented by programmers. Our loop heuristic and loop analysis approach can handle them very well. We can correctly infer the result of the total number of writes in Equation 3.3 for both of them. The `cfingerd` has a format string vulnerability and the `3proxy` has a double free vulnerability. We use them to demonstrate the effectiveness of our approach to handle other memory corruption attacks. The `nullhttpd` has a heap buffer overflow vulnerability that the user can control the vulnerable buffer size and the `SQL` has a stack buffer overflow vulnerability. We use them to demonstrate the effusiveness of our bound analysis approach. In Section 6.3, we list all of the applications we have studied and classify them using our buffer overflow vulnerability case study results.

6.1 LOOP STRUCTURE STUDY

We studied the loops of real programs to see how well they can be evaluated through our generalized loop procedures. For those loops that have write statements that are indexed through loop variant variables, we classify them as data copy loops. Otherwise, they are non-data copy loops. For data copy loops, we evaluated them through the accuracy of the calculated total number of loop iterations and the total number of writes. For non-data copy loops, we evaluated them through the accuracy of the calculated total number of loop iterations using the Equation 3.1, Equation 3.2, Equation 4.3, Equation 4.4, Equation 4.1 and Equation 4.2.

The real programs we studied include the WCET benchmark[46] and some real world applications. The WCET benchmark programs are maintained by the Mlardalen WCET research group. The benchmarks are collected from several different research groups and tool vendors around the world. Each benchmark is provided as a C source file (file.c). In Table 4, we list the description and number of lines of C code of each program of the WCET benchmark.

The real world applications we studied include `3proxy`, `atphttpd`, `ghttpd` and `nullhttpd`. In Table 5, we list the brief descriptions, the programs and the number of lines of C code for each program.

Table 6, Table 7 and Table 8 list all of the evaluation results. They all use the same notation as described in Table 6. Among those notations, (a)(b)(c) are for non-data copy loops and (d)(e)(f) are for data copy loops. The three categories of each group are used to distinguish loop with one exit, loop with multiple loop exits (non-nested loop) and nested loops. The (1)(2)(3)(4) are for the four situations that we can accurately estimate the number of loop iterations; we can't accurately estimate the number of loop iterations; we can accurately estimate the total number of writes; we can't accurately estimate the total number of writes. For non-data copy loops, we evaluate them through the number of loop iterations only, so only (1)(2) are evaluated.

We summarize the loop structure study results for real world applications on Figure 12 and for WCET benchmark on Figure 13. We list the results of both in Figure 14 to compare

Table 4: The WCET benchmark programs (loop structure study source 1)

Program	Description	number of lines of C Code
adpcm	Adaptive pulse code modulation algorithm	875
bs	Binary Search in an array of 15 integer elements	113
bsort100	bubble sort benchmark program	127
cnt	Counts non-negative numbers in a matrix	128
compress	data compression program, adopted from SPEC95	506
cover	Program for testing many paths	238
crc	Cyclic redundancy check computation on 40 data bytes	125
duff	Using Duff's device to copy 43 bytes array	83
edn	Finite Impulse Response (FIR) filter calculations	284
expint	Series expansion computing an exponential integral	156
fac	Recursive program to calculate factorials	25
fdct	Fast Discrete Cosine Transform	238
fft1	Fast Fourier Transform using Cooly-Turkey algorithm	218
fibcall	Iterative Fibonacci, used to calculate fib(30)	71
fir	Finite impulse response filter (signal processing)	274
inssort	Insertion sort on a reversed array of size 10	89
jcomplex	Nested loop program	60
jfdctint	Discrete-cosine transformation on 8*8 pixel block	374
lcdnum	Read ten values, output half to LCD	62
ludcmp	LU decomposition algorithm	149
matmult	Matrix multiplication of two 20*20 matrices	162
minver	Matrix inversion for 3x3 floating point matrix	197
ndes	Embedded code with many complex bit operations	230
ns	Search in a multi-dimensional array	531
nsichneu	Simulates an extended Petri net	4246
prime	Search in a multi-dimensional array	45
qsort-exam	Linear equations by LU decomposition	118
qurt	Root computation of quadratic equations	164
select	Select the n:th largest number in floating point array	111
sqrt	Square root function implemented by Taylor series	73
st	This program computes for two arrays of numbers the sum, the mean, the variance, and standard deviation, the correlation coefficient between the two arrays	157
statement	Automatic generated code	1273
ud	Linear equations by LU decomposition	160

Table 5: Some real world applications (loop structure study source 2)

Application	Description	Program	number of lines of C Code
3proxy	tiny free proxy server	common.c	608
		datatypes.c	798
		dighosts.c	137
		dnspr.c	196
	total number of lines of C Code		1739
atphttpd	simplistic, memory caching web server	http_handler.c	247
		main.c	343
		mime.c	73
		sockhelp.c	343
	total number of lines of C Code		1006
ghttpd	a fast and efficient HTTP server that has CGI support	main.c	235
		protocol.c	329
		util.c	241
	total number of lines of C Code		805
nullhttpd	a free, lightweight HTTP server that has CGI support, not a production server	cgi.c	315
		config.c	165
		files.c	151
		format.c	181
		http.c	289
		main.c	91
		server.c	603
		win32.c	231
	total number of lines of C Code		2026

Table 6: The loop analysis results for the real world applications: (1) can handle the number of loop iterations; (2) can't handle the number of loop iterations; (3) can handle the total number of writes; (4) can't handle the total number of writes; (a) Non-Data Copy Loop: one loop exit; (b) Non-Data Copy Loop: multiple loop exits; (c) Non-Data Copy Loop: nested loops; (d) Data Copy Loop: one loop exit; (e) Data Copy Loop: multiple loop exits; (f) Data Copy Loop: nested loops.

Application		Non-Data Copy Loops			Data Copy Loops		
		(a)	(b)	(c)	(d)	(e)	(f)
3proxy	(1)	2	1	0	3	1	0
	(2)	6	3	0	0	0	0
	(3)	-	-	-	1	1	0
	(4)	-	-	-	2	0	0
atphttpd	(1)	6	0	0	3	0	0
	(2)	3	1	0	0	0	0
	(3)	-	-	-	2	0	0
	(4)	-	-	-	1	0	0
ghttpd	(1)	0	2	0	2	2	0
	(2)	1	5	0	0	1	0
	(3)	-	-	-	2	2	0
	(4)	-	-	-	0	1	0
nullhttpd	(1)	17	0	0	4	1	0
	(2)	12	2	0	1	0	0
	(3)	-	-	-	4	1	0
	(4)	-	-	-	1	0	0
total	(1)	25	3	0	12	4	0
	(2)	22	11	0	1	1	0
	(3)	-	-	-	9	4	0
	(4)	-	-	-	4	1	0

Table 7: The loop analysis results for WCET benchmark (I).The same notation as in Table 6.

Application		(a)	(b)	(c)	(d)	(e)	(f)	Application		(a)	(b)	(c)	(d)	(e)	(f)
	(1)	9	0	0	9	0	0		(1)	0	0	0	0	0	0
adpcm	(2)	1	1	0	0	0	0	bs	(2)	1	0	0	0	0	0
	(3)	-	-	-	9	0	0		(3)	-	-	-	0	0	0
	(4)	-	-	-	0	0	0		(4)	-	-	-	0	0	0
	(1)	0	0	0	1	0	1		(1)	0	0	1	0	0	0
bsort100	(2)	0	0	0	0	0	0	cnt	(2)	0	0	0	0	0	0
	(3)	-	-	-	1	0	1		(3)	-	-	-	0	0	0
	(4)	-	-	-	0	0	0		(4)	-	-	-	0	0	0
	(1)	0	0	0	3	2	0		(1)	2	0	0	0	0	0
compress	(2)	2	0	0	0	0	0	cover	(2)	0	0	0	0	0	0
	(3)	-	-	-	3	2	0		(3)	-	-	-	0	0	0
	(4)	-	-	-	0	0	0		(4)	-	-	-	0	0	0
	(1)	2	0	0	1	0	0		(1)	0	0	0	1	0	0
crc	(2)	0	0	0	0	0	0	duff	(2)	0	0	0	0	0	0
	(3)	-	-	-	1	0	0		(3)	-	-	-	1	0	0
	(4)	-	-	-	0	0	0		(4)	-	-	-	0	0	0
	(1)	3	0	0	2	0	3		(1)	2	0	2	0	0	0
edn	(2)	0	0	0	0	0	0	expint	(2)	0	0	0	0	0	0
	(3)	-	-	-	2	0	2		(3)	-	-	-	0	0	0
	(4)	-	-	-	0	0	1		(4)	-	-	-	0	0	0
	(1)	1	0	0	0	0	0		(1)	3	0	0	0	0	0
fac	(2)	0	0	0	0	0	0	fdct	(2)	0	0	0	0	0	0
	(3)	-	-	-	0	0	0		(3)	-	-	-	0	0	0
	(4)	-	-	-	0	0	0		(4)	-	-	-	0	0	0
	(1)	3	0	0	3	0	1		(1)	0	1	0	0	0	0
fft1	(2)	1	0	0	0	0	0	fibcall	(2)	0	0	0	0	0	0
	(3)	-	-	-	2	0	0		(3)	-	-	-	0	0	0
	(4)	-	-	-	1	0	1		(4)	-	-	-	0	0	0
	(1)	0	0	1	0	0	0		(1)	0	0	0	0	0	0
fir	(2)	0	1	0	0	0	0	inssort	(2)	0	0	1	0	0	0
	(3)	-	-	-	0	0	0		(3)	-	-	-	0	0	0
	(4)	-	-	-	0	0	0		(4)	-	-	-	0	0	0
	(1)	0	0	0	0	0	0		(1)	1	0	0	2	0	0
jcomplex	(2)	0	0	1	0	0	0	jfdctint	(2)	0	0	0	0	0	0
	(3)	-	-	-	0	0	0		(3)	-	-	-	1	0	0
	(4)	-	-	-	0	0	0		(4)	-	-	-	1	0	0

Table 8: The loop analysis results for WCET benchmark (II).The same notation as in Table 6.

Application		(a)	(b)	(c)	(d)	(e)	(f)	Application		(a)	(b)	(c)	(d)	(e)	(f)
	(1)	1	0	0	0	0	0		(1)	5	0	0	3	0	0
lcdnum	(2)	0	0	0	0	0	0	lms	(2)	2	0	0	0	0	0
	(3)	-	-	-	0	0	0		(3)	-	-	-	3	0	0
	(4)	-	-	-	0	0	0		(4)	-	-	-	0	0	0
	(1)	0	0	0	2	0	2		(1)	0	0	0	0	0	2
ludcmp	(2)	0	0	0	0	0	0	matmult	(2)	0	0	0	0	0	0
	(3)	-	-	-	2	0	2		(3)	-	-	-	0	0	2
	(4)	-	-	-	0	0	0		(4)	-	-	-	0	0	0
	(1)	0	0	1	1	0	3		(1)	9	0	0	3	0	0
minver	(2)	0	0	0	0	0	0	ndes	(2)	0	0	0	0	0	0
	(3)	-	-	-	1	0	3		(3)	-	-	-	2	0	0
	(4)	-	-	-	0	0	0		(4)	-	-	-	1	0	0
	(1)	0	0	1	0	0	0		(1)	1	0	0	0	0	0
ns	(2)	0	0	0	0	0	0	nsichneu	(2)	0	0	0	0	0	0
	(3)	-	-	-	0	0	0		(3)	-	-	-	0	0	0
	(4)	-	-	-	0	0	0		(4)	-	-	-	0	0	0
	(1)	1	0	0	0	0	0		(1)	0	0	0	0	0	1
prime	(2)	0	0	0	0	0	0	qsort-exam	(2)	0	0	1	0	0	0
	(3)	-	-	-	0	0	0		(3)	-	-	-	0	0	1
	(4)	-	-	-	0	0	0		(4)	-	-	-	0	0	0
	(1)	1	0	0	0	0	0		(1)	0	0	0	0	0	0
qurt	(2)	0	0	0	0	0	0	select	(2)	0	0	0	0	0	1
	(3)	-	-	-	0	0	0		(3)	-	-	-	0	0	0
	(4)	-	-	-	0	0	0		(4)	-	-	-	0	0	1
	(1)	1	0	0	0	0	0		(1)	3	0	0	1	0	0
sqrt	(2)	0	0	0	0	0	0	st	(2)	0	0	0	0	0	0
	(3)	-	-	-	0	0	0		(3)	-	-	-	1	0	0
	(4)	-	-	-	0	0	0		(4)	-	-	-	0	0	0
	(1)	0	0	0	0	0	0		(1)	0	0	2	0	0	2
statement	(2)	1	0	0	0	0	0	ud	(2)	0	0	0	0	0	0
	(3)	-	-	-	0	0	0		(3)	-	-	-	0	0	2
	(4)	-	-	-	0	0	0		(4)	-	-	-	0	0	0
	(1)	49	1	8	32	2	16			0	0	0	0	0	0
total	(2)	7	2	3	0	0	1	-		0	0	0	0	0	0
	(3)	-	-	-	29	2	14			0	0	0	0	0	0
	(4)	-	-	-	3	0	3			0	0	0	0	0	0

the result differences. On Figure 12, we can see our generalized data copy procedure can handle data copy loops with analysis accuracy around 80%. This is much better than when they are applied for handling non-data copy loops. The same trend can be seen on Figure 13. Overall, our generalized data copy procedures can handle data copy loops with very high analysis accuracy. This result is very encouraging.

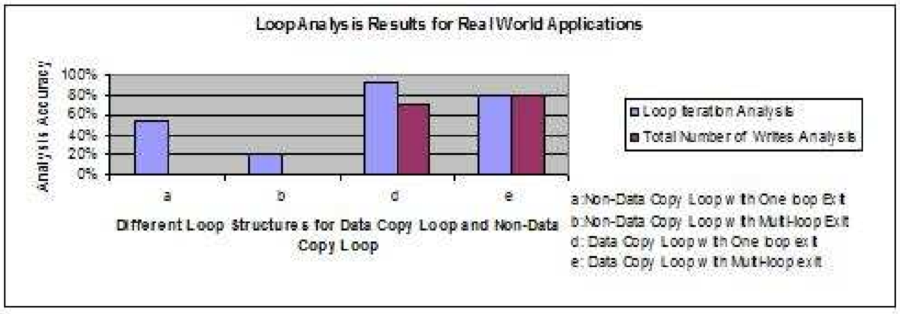


Figure 12: The loop structure study result for real world applications

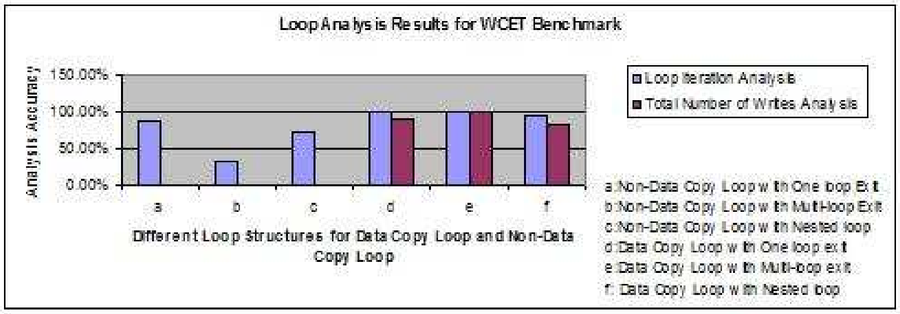


Figure 13: The loop structure study result for WCET Benchmark

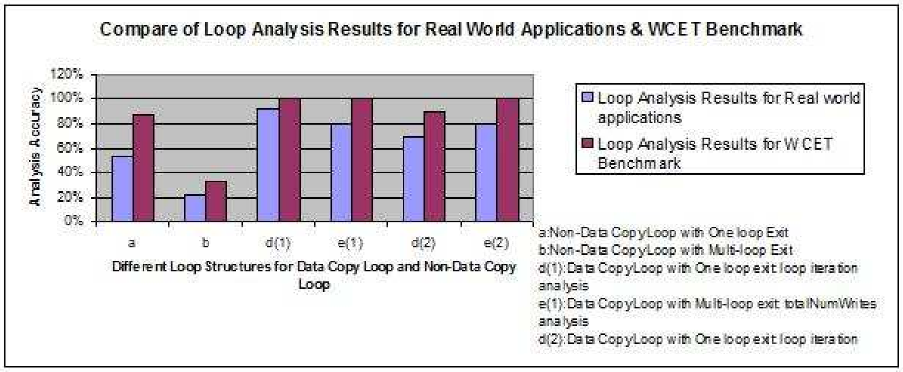


Figure 14: Compare the loop structure study results for WCET benchmark and real world applications

6.2 REPRESENTATIVE EXAMPLES

PNG heap buffer Overflow (Bugtrap ID#1394) PNG (Portable Network Graphics) is a file format for images utilized by many programs. Each PNG image contains a series of required or optional chunks, e.g. PLTE, tRNS etc. Each chunk contains a length field. If the length of the tRNS chunk and the length of the PLTE entries don't match, a buffer overflow may happen due to the lack of the length field validation check in the vulnerable application.

In our approach, we first find the BOVP using our diagnosis engine. It is a write instruction inside of function processTRNS. We then do the loop analysis of that function. This is a programmer written data copy procedure. There is one loop exit condition, `iv < a`. The `iv` has `init` value 0 and pattern `iv += 1`.

Two of the total four memory writes are indexed by loop variant variables and these two variables are the same. They are interesting to us. They write one byte each time. They are actually two write instructions on the two branches inside of the loop body. Since they have the same index, this won't affect our analysis results. The loop variant variable equals `iv + b` where the `iv` is the same `iv` in the loop exit condition and the `b` is a loop invariant. This loop invariant variable actually states the starting address of the vulnerable buffer. From this loop analysis, we can infer the total number of writes after the loop execution is `a` using Equation 3.3. We return the instruction addresses of the loop exit condition, the memory write instructions that we extracted to the bound analysis step.

In the bound analysis step, we get the taint information for `a` and the value of `b`. The variable `a` is tainted and the value of `b` shows that the vulnerable buffer is located on heap. In the input analysis step, we obtained the dynamic slicing contains this `a` variable and then performed the symbolic execution on the traces. With the data format information (we assume it is given here), this variable `a` is actually the length field of tRNS chunk.

The above diagnosis results can help the security engineers understand the vulnerability pretty well. They can realize that the length field of tRNS chunk needs to be validated because that determines the total number of bytes written to the vulnerable buffer. The PNG format specification[8] states that the tRNS chunk must not contain more alpha values than there are palette entries. The real patch actually checks the number of alpha values with the palette entries before the data copy procedure.

Windows DCOM RPC (MS03-026) The Windows DCOM RPC vulnerability is a stack buffer overflow vulnerability in Microsoft's DCOM RPC service. This is the other example that have a data copy procedure implemented by programmer.

In our approach, we first find the BOVP. It is a write instruction inside of function GetMachineName. We then do the loop analysis of that function. This is a programmer written data copy procedure. The loop exit condition is `source[i] = '\'`. The read index is composed by an `iv` variable, which has `init` value 0 and pattern `iv+=2`. There is only one memory write instruction in the loop body that is indexed by a loop variant variable. This loop variant variable equals `iv + b`. The `iv` is the same `iv` variable for the read index `i` and the `b` is a loop invariant variable which states the starting address of the vulnerable buffer. From this loop analysis, we can infer the total number of writes of this data copy procedure is `strlen(source|\)`. We return the instruction addresses of the loop exit condition, the memory write statement to the bound analysis step.

In the bound analysis step, we get the tainted information of `source` and the value of `b`.

The `source` is tainted and the value `b` indicates that the vulnerable buffer is on stack. In this example, there are no local variables allocated between the vulnerable buffer and the `return address`, `Old EBP`. Our bound analysis approach can find the bound results efficiently and accurately, which is 32 bytes.

In the input analysis step, we get the dynamic slicing of the traces that contains the `source` variable and then perform the forward symbolic execution on the traces. With the format information (we assume it is given), this string region corresponds to a Name field.

We can generate the signature directly for this example: `strlen(Name)` $\leq$ `32`.

nullhttpd heap buffer overflow (Bugtrap ID#5774) The nullhttpd[9] is a small multithreaded web server. It has a remote heap buffer overflow vulnerability. This is a very good example belong to buffer overflow vulnerability Case V.

In our approach, we first find the BOVP. It is a write instruction belong to `recv` function call. We know the functionality of this function in advance. We omit the loop analysis step.

In the bound analysis step, we get the tainted information of the third parameter, the length field `n`, and the value of the second parameter, the starting address of the destination buffer. The `n` is untainted which has constant value 1024. The starting address of the vulnerable buffer states that it is located on heap. In our memory allocation history, we successfully find the malloc record that does this memory allocation. The bound result is the same as the malloc size, so we infer the parameter of the malloc determines the vulnerable buffer size. Further investigation indicates that this parameter is tainted.

In the input analysis step, we get the dynamic slicing of the traces that contains the malloc parameter variable and then perform the forward symbolic execution on the trace. The malloc parameter equals `a + 1024` where `a` is a length field.

We can generate the signature directly for this example: `lengthField > 0`.

Microsoft SQL Server (MS02-039) This SQL vulnerability is a stack buffer overflow vulnerability in SQL server 2000 Resolution service (SSRS). The SQL Slammer worm attacks this vulnerability.

In our approach, we first find the BOVP. It is a write instruction belong to `sprintf` function call. We know the functionality of this function in advance. We omit the loop analysis step.

In the bound analysis step, we get the tainted information of the format string and the value of the first parameter, the starting address of the vulnerable buffer. The format string is untainted, so we can confirm that this is not a format string vulnerability. The starting address of the vulnerable buffer states that the vulnerable buffer is located on stack. Our bound analysis can accurately find the bound information for this vulnerable buffer which is 132 bytes. The format string has length 40 bytes.

In the input analysis step, we get the dynamic slicing of the traces that contains the `vararg` parameter for the `sprintf` function and then perform symbolic execution on the trace. It is a name field.

We can generate the signature directly for this example: `strlen(nameField)` $\leq$ `92`.

cfingerd format string vulnerability (Bugtrap ID#2576) cfingerd is a finger daemon that contains a format string vulnerability in older versions. We test this vulnerability on version 1.4.1.

In our approach, we first find the BOVP. It is a write instruction belong to `SYSLOG` function call. We know the functionality of this function in advance. We omit the loop analysis step.

In the bound analysis step, we get the tainted information of the format string. The format string is tainted. Further investigation indicates that there is no `vararg` in this function call. We can then restrict the number of directives of the format string as 0.

3proxy double free vulnerability (Bugtrap ID#26180) 3proxy is a tiny proxy server which is prone to a double-free memory-corruption vulnerability in older versions. We test this vulnerability on 3proxy-0.5.3i. The double free vulnerability crashed the program. We analyzed the traces collected before the program crash. In our free function call history, two free function calls have the same parameter value, the starting address of the memory region to free. And there isn't a new malloc record between the two function calls corresponding to this memory region. We confirm it is a double free vulnerability and we return the two free function call records as the diagnosis results.

We summarize our signature results for buffer overflow vulnerability applications in Table 9.

Table 9: Signature results for buffer overflow vulnerable applications

Vulnerable Application	Signature Result
Windows DCOM RPC	strlen(nameField) ≤ 32
Microsoft SQL Server	strlen(nameField) ≤ 92
nullhttpd	lengthField > 0
PNG(MS05025)	lengthField(tRNS) $\leq \frac{lengthField(PLTE)}{3}$

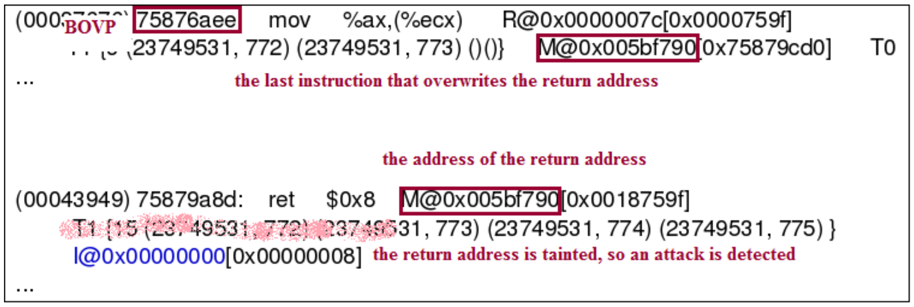
In Table 10, we list our performance results. All the numbers are in seconds. The BOVP detection is mainly consumed by the dynamic-taint analysis and the BOVC is mainly consumed by the disassembling process. For windows applications, the traces are very huge which take longer time for the BOVP detection.

Table 10: The performance evaluation

Vulnerable Application	BOVP Detection (sec)	BOVC Analysis (sec)
Windows DCOM RPC	100.83	56.78
Microsoft SQL Server	77.51	35.16
nullhttpd	1.33	20.05
PNG(MS05025)	637.82	27.37
cfingerd	44.14	24.79
3proxy	1.59	18.52

Now, I use the Windows DCOM RPC vulnerability to demonstrate the diagnosing process. The Windows DCOM RPC is attacked by the record-setting blaster worm.

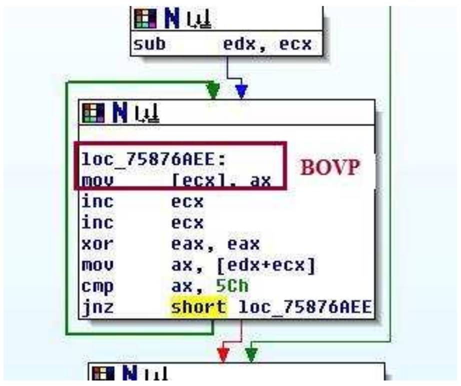
Figure 15 lists partial of the runtime traces. We can see the return address is tainted. An attack detection alarm is raised by the dynamic analysis engine. We search back the trace to find the latest instruction that overwrote the return address. This instruction is the BOVP. Using loop detection, we locate the loop that contains this BOVP. This loop is illustrated in Figure 16. In Figure 17, we can see how our loop analysis retrieve the program slicing of the loop exit condition and memory read and memory write indexes and conclude the total number of writes information using Equation 3.2. The loop analysis result is illustrated in Figure 18. The read instruction address and write instruction address are inputs to the bound analysis step. Figure 19 illustrates this process. We check the taint status of the read-in value which is tainted. We obtain the starting address of the vulnerable buffer and calculate the bound result. The bound result is listed in Figure 20.



```
(000[illegible]) 75876aee    mov    %ax,(%ecx)      R@0x0000007c[0x0000759f]
      ... [illegible] 23749531, 772) (23749531, 773) ()()}     M@0x005bf790[0x75879cd0]    T0
...

(00043949) 75879a8d:   ret    $0x8    M@0x005bf790[0x0018759f]
      T1 {15 (23749531, 772) (23749531, 773) (23749531, 774) (23749531, 775) }
      I@0x00000000[0x00000008]
...
```

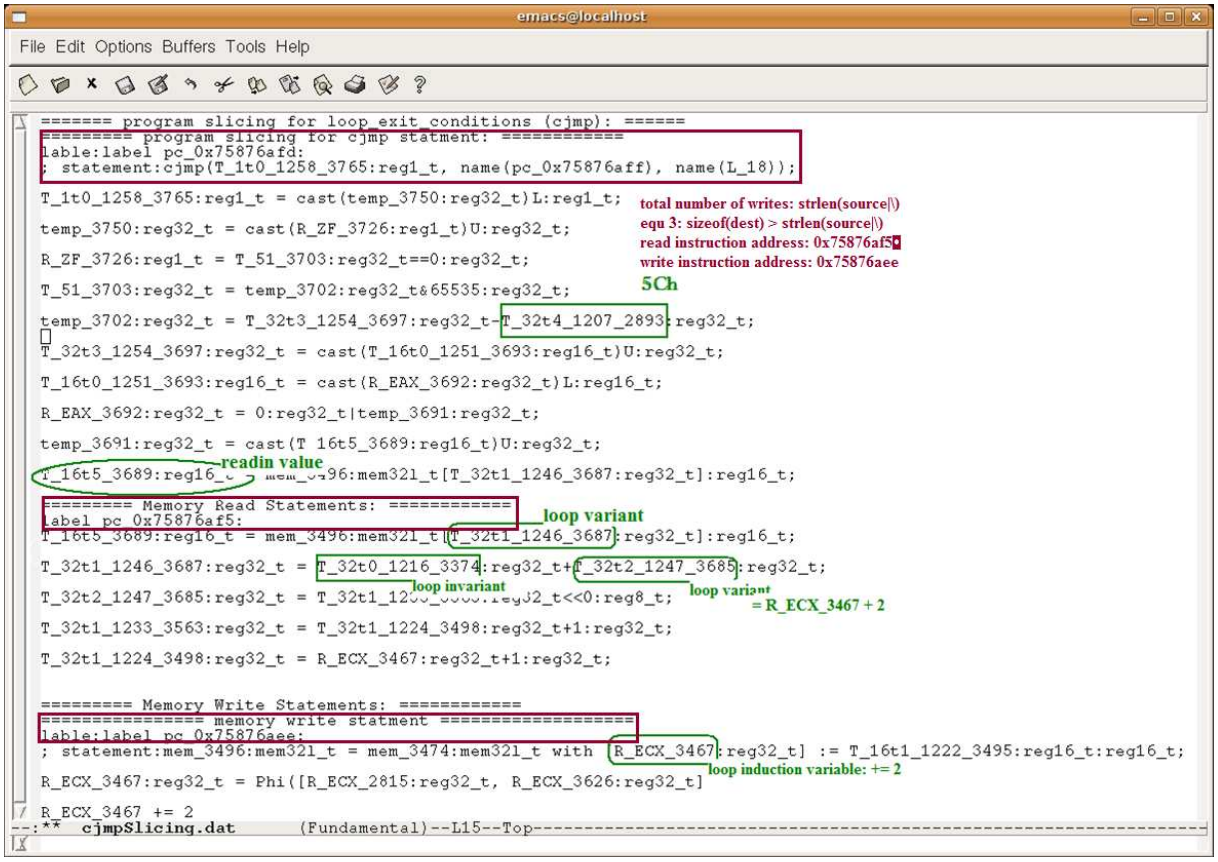
BOVP
the last instruction that overwrites the return address
the address of the return address
the return address is tainted, so an attack is detected

Figure 15: The BOVP detection for Windows DCOM RPC



```
sub     edx, ecx

loc_75876AEE:
mov     [ecx], ax
inc     ecx
inc     ecx
xor     eax, eax
mov     ax, [edx+ecx]
cmp     ax, 5Ch
jnz     short loc_75876AEE
```

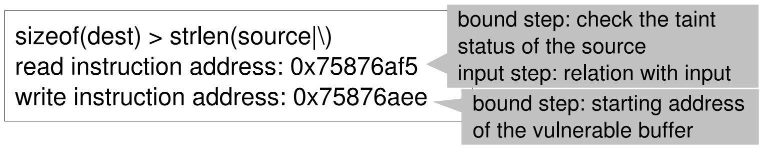
Figure 16: The Loop detection for Windows DCOM RPC



```
======= program slicing for loop_exit_conditions (cjmp): ======
========= program slicing for cjmp statment: ============
lable:label pc_0x75876afd:
; statement:cjmp(T_1t0_1258_3765:reg1_t, name(pc_0x75876aff), name(L_18));

T_1t0_1258_3765:reg1_t = cast(temp_3750:reg32_t)L:reg1_t;

temp_3750:reg32_t = cast(R_ZF_3726:reg1_t)U:reg32_t;

R_ZF_3726:reg1_t = T_51_3703:reg32_t==0:reg32_t;

T_51_3703:reg32_t = temp_3702:reg32_t&65535:reg32_t;

temp_3702:reg32_t = T_32t3_1254_3697:reg32_t-T_32t4_1207_2893:reg32_t;

T_32t3_1254_3697:reg32_t = cast(T_16t0_1251_3693:reg16_t)U:reg32_t;

T_16t0_1251_3693:reg16_t = cast(R_EAX_3692:reg32_t)L:reg16_t;

R_EAX_3692:reg32_t = 0:reg32_t|temp_3691:reg32_t;

temp_3691:reg32_t = cast(T 16t5_3689:reg16_t)U:reg32_t;

T_16t5_3689:reg16_t = mem_3496:mem32l_t[T_32t1_1246_3687:reg32_t]:reg16_t;

========= Memory Read Statements: ============
label pc_0x75876af5:
T_16t5_3689:reg16_t = mem_3496:mem32l_t[T_32t1_1246_3687:reg32_t]:reg16_t;

T_32t1_1246_3687:reg32_t = T_32t0_1216_3374:reg32_t+T_32t2_1247_3685:reg32_t;

T_32t2_1247_3685:reg32_t = T_32t1_12[illegible]:reg32_t<<0:reg8_t;

T_32t1_1233_3563:reg32_t = T_32t1_1224_3498:reg32_t+1:reg32_t;

T_32t1_1224_3498:reg32_t = R_ECX_3467:reg32_t+1:reg32_t;


========= Memory Write Statements: ============
================ memory write statment ===================
lable:label pc_0x75876aee:
; statement:mem_3496:mem32l_t = mem_3474:mem32l_t with [R_ECX_3467:reg32_t] := T_16t1_1222_3495:reg16_t:reg16_t;

R_ECX_3467:reg32_t = Phi([R_ECX_2815:reg32_t, R_ECX_3626:reg32_t]

R_ECX_3467 += 2
--:**  cjmpSlicing.dat      (Fundamental)--L15--Top-----------------------------------
```

emacs@localhost
File Edit Options Buffers Tools Help
total number of writes: strlen(source|\)
equ 3: sizeof(dest) > strlen(source|\)
read instruction address: 0x75876af5
write instruction address: 0x75876aee
5Ch
readin value
loop variant
loop invariant
loop variant
= R_ECX_3467 + 2
loop induction variable: += 2

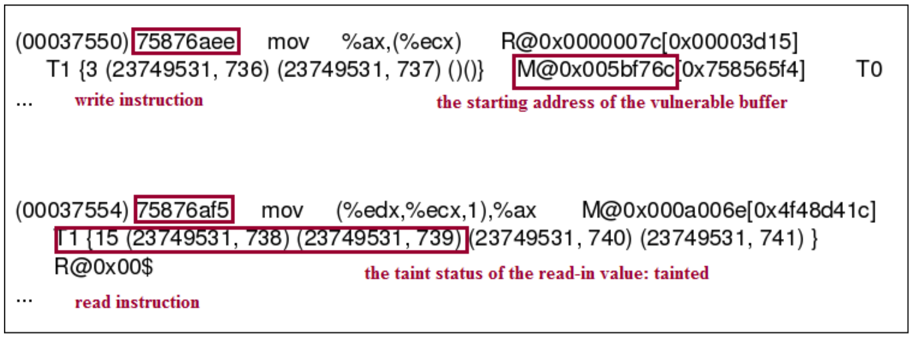
Figure 17: The Loop Analysis for Windows DCOM RPC



```
sizeof(dest) > strlen(source|\)
read instruction address: 0x75876af5
write instruction address: 0x75876aee
```

bound step: check the taint status of the source
input step: relation with input
bound step: starting address of the vulnerable buffer

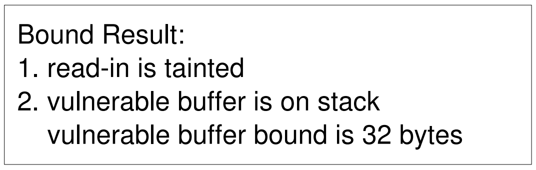
Figure 18: The Loop Result for Windows DCOM RPC



```
(00037550) 75876aee    mov    %ax,(%ecx)      R@0x0000007c[0x00003d15]
    T1 {3 (23749531, 736) (23749531, 737) ()()}    M@0x005bf76c[0x758565f4]      T0
...

(00037554) 75876af5    mov    (%edx,%ecx,1),%ax      M@0x000a006e[0x4f48d41c]
     T1 {15 (23749531, 738) (23749531, 739) (23749531, 740) (23749531, 741) }
     R@0x00$
...
```

write instruction
the starting address of the vulnerable buffer
the taint status of the read-in value: tainted
read instruction

Figure 19: The Bound Analysis for Windows DCOM RPC



Bound Result:
1. read-in is tainted
2. vulnerable buffer is on stack
   vulnerable buffer bound is 32 bytes

Figure 20: The Bound Result for Windows DCOM RPC

Table 11: The buffer overflow case study on real world applications

Vulnerable Application	Vulnerability Function	Data Copy Procedure	Buffer Overflow Cases
CSRSS(MS05018)	DoFontEnum	wcscpy	Case I
CSRSS(MS06040)	CanonicalizePathname	wcscpy	Case I
atphttpd	http_send_error	sprintf	Case I
ghttpd	Log	sprintf	Case I
Windows DCOM RPC	GetMachineName	programmer written	Case I
Microsoft SQL Server	sub42cfbf57	sprintf	Case I
ANI(MS07017)	LoadAnilIcon	memcpy	Case IV
Samba	call_trans2open	strncpy	Case IV
nullhttpd	ReadPOSTData	recv	Case V
PNG(MS05025)	processTRNS	programmer written	Case VI

6.3 BUFFER OVERFLOW VULNERABILITY CASE STUDY

In Table 11, we list the real world buffer overflow vulnerable applications we studied. We classify them based on our buffer overflow vulnerability case study results. The six basic cases cover all of the real world buffer overflow vulnerable applications we studied. We found real world buffer overflow vulnerabilities belonging to Case I, Case IV, Case V and Case VI.

7.0 DISCUSSION

In this Chapter, we discuss our approach through various aspects. We discuss the complex loop cases that will affect our analysis results. We analyze the advantages and the potential limitations of our bound analysis approach. We discuss how our BOVD approach can't be affected by the memory aliasing and indirect jumps in binary program analysis. We do some quantitative analysis of data copy procedure vs. buffer overflow. We perform the false negative analysis and false positive analysis of the signature results using our current input analysis approach. After that, we discuss about the situations that we can't handle.

7.1 COMPLEX LOOP CASES

There are several complex loop cases that will affect our analysis results. They include: 1). There are conditional branch in the loop body. And the written index variable of the vulnerable buffer or the loop iteration variable are updated differently in each conditional branch. 2). The data copy procedure is implemented through sequential loops. 3). The loop exits and the number of writes can't be handled using our loop structures. The next few Sections discuss each of them in detail.

7.1.1 Conditional Branch in loop.

When there is a conditional branch in the loop body, especially when vulnerable buffer write index variable and loop iteration variable are updated differently in each conditional branch, we can't obtain the accurate analysis results of the total number of writes after the loop execution. We do some approximation to conservatively estimate the total number of writes. When the write index variable is updated differently on each conditional branch, we use the one that is updated (incremented or decremented) larger to estimate the number of writes in each loop iteration, $max(iv2update_1, iv2update_2)$. When the loop iteration variable is updated differently on each conditional branch, we use the one that is updated (incremented or decremented) smaller to estimate the number of loop iterations, $min(iv1update_1, iv1update_2$. The final estimated total number of writes is greater than or equal to the real total number of writes.

```
iv1 = iv1init;
iv2 = iv2init;
while (iv1 < Loop_Bound){
  if (condition){
    writeAddress = startAddress + startOffset + iv2;
    [writeAddress] = {some value};
    iv2 += iv2upate1;
    iv1 += iv1upate1;
  }else{
    writeAddress = startAddress + startOffset + iv2;
    [writeAddress] = {some value};
    iv2 += iv2upate2;
    iv1 += iv1upate2;
  }
}
```

Figure 21: The data copy procedure that has the memory write index variable and loop iteration variable updated differently in each conditional branch.

7.1.2 Sequential Loops.

We met sequential loops when we were evaluating our loop analysis approach using some Standard C Library functions including `memset`, `memcpy`, `strncpy`. This happens during the compiler optimization process. The first loop writes four bytes at a time. The following loops finish the remaining. If it happens in a programmer written data copy procedure, we use memory traces information to handle this situation. In our loop analysis step, we actually find all of the loops of the vulnerable function. We assign each memory write instruction inside of loops with an instruction id composed by the function name and the loop id. This loop id distinguishes the loop that contains this memory write instruction. In the bound analysis process, when we mark the bit vector for each memory write, we compare the instruction id with the one on the previous position. If they contain the same function name, we ignore the bit vector update. At the same time, we update the instruction id array for the corresponding memory write. With this information, we can know if a block of memory writes are happened in loops within the same function. We then look for the memory space before the first memory write of the vulnerable loop that contains the BOVP. If that memory space is written by an instruction with an id that has the same function name but different loop id, we will infer these two memory space belong to the same buffer. This process will continue until we find the first memory write in the first loop of the vulnerable function which is the starting address of the vulnerable buffer. The Loop_Bound is still obtained from the loop that contains the BOVP. In Figure 22, we demonstrate how the sequential loops are handled in bound analysis step.

7.1.3 Loop exits and number of writes we can't handle.

In Section 3.3.1, we come up with the Generalized data copy procedure as illustrated in Figure 6 and we answer the total number of writes information using Equation 3.1 and Equation 3.2. In Section 4.2, we discussed the two situations that have multiple loop exits. One is in nested loops. The other is in non-nested loop. The two situations are illustrated in Figure 8 and Figure 9. We solve them through Equation 4.3, Equation 4.4, Equation 4.1, Equation 4.2 respectively.

In Section 6.1, we evaluate the above solutions using loops in real programs. We can handle most of the loops especially data-copy loops. However, there are loops that we can't provide the accurate number of loop iterations or we can answer the number of loop iterations correctly but we can't answer the total number of writes correctly. Here, I list the cases that we can't handle using our loop structures:

- Loop bound is a result of a function.
- The iteration variable is updated irregularly. For example, updated by the result of a function.
- The existence of break statement is a big challenge.
- The write statement does not write continuously to an array, but write to a member of a `struct` array.

7.2 BOUND ANALYSIS DISCUSSION.

Our bound analysis approach can guarantee coarse-grained bound analysis results on stack and heap. It can provide fine-grained bound results for majority of the cases. However, there are situations that it can't provide the fine-grained bound results:

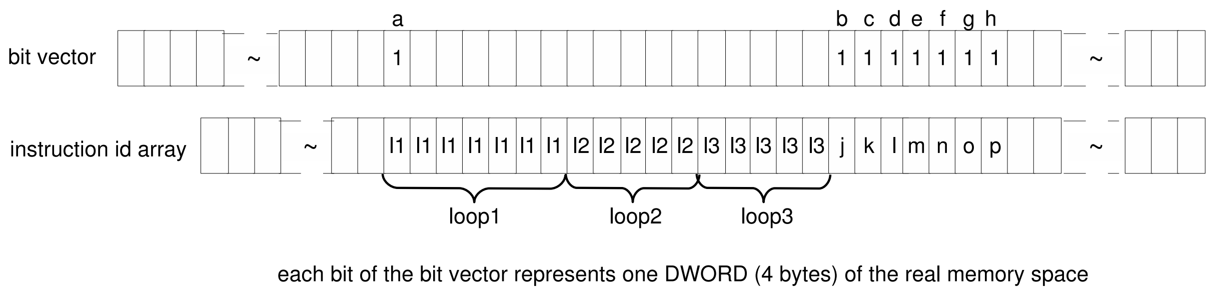

a
b c d e f g h
bit vector
1
1 1 1 1 1 1 1
instruction id array
I1 I1 I1 I1 I1 I1 I1 I2 I2 I2 I2 I2 I3 I3 I3 I3 I3 j k l m n o p
loop1
loop2
loop3
each bit of the bit vector represents one DWORD (4 bytes) of the real memory space

Figure 22: The Memory Map for Sequential Loops

- Variable access sequence matters. Even though we believe we can overwrite those variables that haven't been written, other exploitable path may have different variable access sequence.

- There may be some security critical data on Data region that has been initialized by the program. We can't detect its bound and it can be overwritten by rewriting some other global variables located before it.

However, our bound analysis approach has a very good side effect that it can be used to diagnose double free vulnerabilities as discussed in Section 3.4. The double free vulnerability diagnosis result is much more helpful than the results returned by most of the current systems. The current systems can detect that there is a double free on which memory address. We can return which two `free` calls can cause a double free.

Our current implementation can only handle the calling conventions of the `malloc` and `free` functions on our testbed RedHat73. We will extend our implementation to handle more calling conventions in future.

7.3 THE EFFECT OF MEMORY ALIASING AND INDIRECT JUMPS.

As there isn't a perfect solution for memory aliasing analysis for x86 binary programs, we can't guarantee the detection of the *iv* when it is accessed and maintained through memory load and store. However, as an *iv* is updated through each loop iteration, it is more reasonable to store it in a register by compiler optimization. The real world programs, which have programmer written data copy procedures and that had been discussed in Section 6.2, use registers to store the *iv* variables.

We use program control flow graph (CFG) as an input to do the loop detection. Due to the existence of indirect jumps, our CFG can't represent the program flow precisely when indirect jumps exist. In that case, we may not be able to find the loop that does the data copy. However, we look for loops implemented by programmers that do the data copy which rarely have case switch statements, virtual functions etc. to cause indirect jumps.

7.4 QUANTITATIVE ANALYSIS.

We do this analysis based on the data we collected from ShieldGen[41] since most of the vulnerability information is not publicly available. Even in ShieldGen[41], they can't find enough information for 89 vulnerabilities.

Quantitative analysis of input triggered vulnerabilities. In Table 12, we list the total number of vulnerabilities collected that have been issued on Microsoft security bulletins between 2003 and 2006 and the number of vulnerabilities that can be exploited by inputs. Our diagnosis approach is designed to diagnose the vulnerabilities triggered by inputs.

Table 12: The Quantitative analysis of input triggered vulnerabilities

Total Number of Vulnerabilities	377
Not Enough Information	89
With Enough Information to Classify	288
Vulnerabilities Triggered by Inputs	157
Denial of Service	55
Access Control	25
Scripting Problem	17
Miscellaneous Problem	34
Percentage of Vulnerabilities Triggered by Inputs	54.5%

Quantitative analysis of data copy procedure vs. buffer overflow

In ShieldGen[41], 25 vulnerabilities are selected to do the pencil-and-paper detail analysis. Among the 25 vulnerabilities, two cases are applications using a field from the input as an index into an array without checking whether the index falls within the bounds of the array. These are the buffer overflow cases that the buffer overflow doesn't happen through the data copy process. This unchecked array index is one of the cases that ShieldGen[41] can't handle and this is the case our diagnosis results can't provide precise results. Our diagnosis result can state that the vulnerability may belong to this case, but we can't provide bound information since we can't locate the data copy procedure easily without any extra specification. In Table 13, we display the data information.

Table 13: The Quantitative analysis of data copy procedure vs. buffer overflow

Sample Size	25
Through Unchecked Array Index	2
Through Data Copy Procedure	23
Percentage of Data Copy Procedure triggers buffer overflow	92%

Besides the unchecked array indices's cases, there are some other situations that the ShieldGen[41] can't handle which includes 1) combined length of two separate strings in the input exceeds a certain limit 2) the value of one integer field in the input is larger than the value of another integer field 3) the application uses a collection of old buffers. All of them are not a problem to us. Our `PNG` example belongs to case 2. For case 3, it is also not a problem to us since we can find the buffer bound. In Table 14, we compare the precision results of our approach with ShieldGen[41].

Table 14: Compare the coverage of our approach with ShieldGen's

	ShieldGen Approach	Our Diagnosis Approach
Precise Filter	19	23
Imprecise Filter	6	2
Total	25	25
Precision Coverage	76%	92%

7.5 FALSE NEGATIVE AND FALSE POSITIVE ANALYSIS FOR SIGNATURE GENERATION.

We will analyze the false negatives and false positives of the generated signature results using our current input analysis approach.

Our bound analysis approach can generate the fine-grained bound analysis results through the memory access analysis of the runtime traces. Our results won't lead to the false negatives generated in ShieldGen[41] due to the coarse-grained bound results. However, in our input analysis step, we only examine the execution path executed by the given exploit and we only examine how the user inputs are related with the malicious variables. We can't generate the input condition to guarantee reaching the vulnerability point. If by changing some input variables, the vulnerability point isn't reachable but the input variables that are related with the malicious variables (e.g. source buffer, destination buffer size or iteration variable) are still satisfy the vulnerability condition, our signature results will lead to false positives. Also, if there is another exploitable execution path and if the relationship of the user inputs to the vulnerability condition is different, our signature results will lead to false negatives.

Please see Section 8.2 for further discussion on how to enhance the current input analysis approach to achieve generating signatures with zero false positives and zero false negatives.

7.6 SITUATIONS WE CAN'T HANDLE PRECISELY.

Our approach assumes the buffer overflow attack happens through some data copy procedure (either programmer implemented, or some library or system functions). If it is a programmer implemented data copy procedure, it is implemented through loop structures instead of fix number of write statements.

In ShieldGen[41], two of the 377 vulnerabilities collected are known using some input field as an index to access the array, our approach won't be able to diagnose them precisely. So far, in our BOVP analysis step, we search back the trace for the latest instruction that does the malicious write. If this instruction isn't belong to any data copy procedure (programmer written loops, known library or system functions), we will keep searching back until we find one that is part of a data copy procedure as the BOVP or find nothing. If we can't find the BOVP, our diagnosis result will state that the vulnerability may be belong to this `unchecked array index` case. However, we can't provide precise bound information since we can't locate the data copy procedure easily without any extra specification. Also, if the data copy procedure is implemented using fixed number of write statements, we can't locate it using current loop detection approach. We won't be able to diagnose the total number of writes information.

Our current input analysis approach returns the dynamic slicing of the run-time traces and then performs the symbolic execution to figure out how the user inputs are related with those malicious variables. So, only one execution path is examined. The signatures generated through this approach can lead to low false negatives and low false positives. Those low false positives happen when other input fields can change the execution path and make the vulnerability point unreachable. However, we can also return the static slicing of the program that contains the malicious variables which can greatly reduce the workload of the security engineers to diagnose the vulnerability and to generate the accurate signatures more efficiently. Please see Section 8.2 for detail discussion on how to enhance the current input analysis approach to achieve generating signatures with zero false positives and zero false negatives.

8.0 APPLICATION

The results of the automatic buffer overflow vulnerability diagnosis can have many applications. First, it can help the security engineers to understand why the program is vulnerable. With the diagnosis information, the security engineers can fix the buggy program and generate the accurate patches more efficiently. Second, the results of the loop analysis and bound analysis can be applied for automatic patch generation. Third, the results of all the three steps analysis can be applied for signature generation and exploit generation. In this section, I discuss the application of the vulnerability diagnosis on patch generation, signature generation and exploit generation.

8.1 PATCH GENERATION

Sidiroglou `et al.` propose automatic patch generation to defend against network worms[82]. They use ProPolice[47] to detect the buffer overflow vulnerability. They then apply some heuristics to do the patch generation. Those heuristics include: 1) moving the offending buffer to heap; 2) new version of malloc that allocates two additional pages. Their approach can mitigate the vulnerability but can't provide a satisfiable patch due to lack of semantic information of the vulnerability.

With the loop analysis and bound analysis results, we can generate more effective patches by restricting the number of writes of the data copy procedure less than the destination buffer size. This can be achieved through binary rewriting.

There are plenty of binary rewriting tools[87, 64, 77, 12, 32, 54] available in academia. ATOM[87] and EEL[64] are designed for RISC computer architecture. Etch[77] is primarily an optimization tool for rewriting Win32/Intel PE executable. LEEL[12] works on inux/x86 binaries. However, it can't handle the indirect control and arbitrary code/data mixing. UQBT[32] is a static binary translation framework which is architecture independent. Detours[54] is for run-time binary interception of Win32 functions.

Researchers have actually proposed defense against attacks through binary rewriting approach[75]. However, their approach achieves the same level of protection as compiler-based approach for stack buffer overflow defense by protecting return address.

In summary, we can generate more effective patches automatically using our diagnosis results.

8.2 SIGNATURE GENERATION

Compared with software patches, signature-based defense can respond faster and is less intrusive to a system since applying and removing a signature does not affect the normal operation of a running system. As a result, signature-based defense becomes an important initial response to zero-day attacks. A key requirement to signature-based defense against zero-day attacks is to automatically generate signatures that have low false positives and low false negatives given an exploit to the vulnerability. Signatures with high false negatives can be evaded by polymorphic mutations of exploits. Signatures with high false positives will block legitimate inputs. Due to the importance of it, researchers have addressed this problem through many different approaches[60, 70, 40, 84, 61, 24, 26, 35, 34]. However, none of them is satisfiable.

Here, we discuss how to achieve the zero-false positive and zero-false negative signature generation by enhancing this thesis work.

Our bound analysis approach can generate the fine-grained bound analysis results through the memory access analysis of the runtime traces. Our results won't lead to the false negatives that ShieldGen[41] generates due to ShieldGen generates coarse-grained bound results. However, in our input analysis step, we only examine the execution path executed by the given exploit. Using symbolic execution and path slicing[55] as proposed in both Vigilante[35] and Bouncer[34], we can generate the input condition to reach the vulnerability point on that execution path. Combining the input condition to reach the vulnerability point, the vulnerability condition (how the user inputs are related with the buffer overflow condition) and the fine-grained bound results, we can generate accurate signature results for the execution path executed by the given exploit. This result is pretty good comparing with any of the existing approaches. However, it can't handle the situation when there is a different execution path and on the other execution path, the input condition to reach the vulnerability point and the vulnerability condition are different from the previous results. We can handle this by generating signatures cumulatively. After generating the first signature, if we detected any new exploit that is not blocked by the signature, we do the same vulnerability diagnosis. If they reach the same vulnerability point, we know they are exploring a different execution path. We combine the new signature results with the previous signature results as a new signature. Using the above input analysis approach, we can achieve generating signatures with zero-false positives and zero-false negatives.

8.3 EXPLOIT GENERATION

Automatic patch-based exploit generation (APEG) is first proposed in[25]. I contribute on the statistic criteria for the binary differences of two given binary programs and I also contribute on some of the experiments on that APEG work. I summarize this APEG work briefly here. For details, please refer to paper[25]. Similarly by checking the safety of the data copy procedures in the updated functions of the two binary programs, we can use the BOVD results to help generating exploits for buffer overflow vulnerabilities. This is discussed in Section 8.3.3 briefly.

Automatic exploit generation can be used by attackers. It is also useful for security practitioners, e.g. prioritizing the bug fixes.

In paper[25], we show that given a program P and a patched version of the program P', we can automatically generate an exploit for input validation vulnerabilities present in P but fixed in P'. We call this *automatic patch-based exploit generation*(APEG). We present techniques which work on binary programs and libraries, thus are applicable to binary-only patch distribution schemes.

8.3.1 Binary Diffs

To isolate what changes have occurred between P and P', security practitioners have developed tools, such as bindiff[79] and EBDS[45], which first disassemble both P and P', and then identify which assembly instructions have changed.

When the difference between a given vulnerable binary program and the patched binary program is small, we can generate the exploits efficiently. So we need a criterion to define the distance between the given pair of binary programs. For those that have small distance, we will apply our following approaches to generate exploits. Others, we simply disregard them.

Using EBDS[45], we can obtain the number of blocks and the match rate for each function of the experimented two binary programs. Given these information, we can apply the Kullback-Leibler Divergence (K-L Divergence)[10] to define the distance of the given two binaries. A similar entropy based approach is used to evaluate the communication predictability in parallel applications[57].

We will explain the K-L Divergence concept briefly and how it can be applied to our problem in the following.

How to Apply K-L Divergence to Solve Our Problem

K-L Divergence is a measure of the difference between two distributions P and Q in information theory[10]. The formula below defines the K-L distance of Q from P and i represents each item enumeration in the distribution.

$$D_{KL}(P||Q) = \sum_i P(i) log \frac{P(i)}{Q(i)} = -\sum_i P(i) log \frac{Q(i)}{P(i)} \tag{8.1}$$

Applying K-L Divergence to solve our problem is straight-forward. The i is for each function in the binaries. For each function, we have the match rate of it from EBDS[45], which is exactly the $\frac{Q(i)}{P(i)}$. The match rate represents the degree of the similarity of that function in the two binaries. It is between 0 and 1. The higher the value is, the less changes

Table 15: Binary Distance Example I

Function Name	Number of Blocks	Match Rate
f_1	10	0.8
f_2	60	0.9
f_3	15	1
f_4	5	1
f_5	10	1

the function has been done. For functions that are the same in the two binaries, the match rate will be 1. The $P(i)$ can be calculated using the number of blocks information. The equation to calculate $P(i)$ is given below where the Block(i) represents the number of blocks in function i and j is an enumeration of all the functions in the binary programs:

$$P(i) = \frac{Block(i)}{\sum_j Block(j)} \tag{8.2}$$

Combining equations (8.1, 8.2) and the match rate information, we can get our equation to evaluate the distance of given two binaries:

$$D_{binary\_pair}(B_1||B_2) = -\sum_i \frac{Block(i)}{\sum_j Block(j)} log(MatchRate(i)) \tag{8.3}$$

Experiments

We are going to show you some examples to apply equation 8.3 to calculate the distance of two binaries.

By applying equation 8.3, we can calculate the distance between these two binaries as

$$D_{binary\_pair}(B_1||B_2) = -(\frac{10}{10+60+15+5+10}log(0.8)+\frac{60}{10+60+15+5+10}log(0.9)) = 0.0372 \tag{8.4}$$

By applying equation 8.3, we can calculate the distance between these two binaries as

$$D_{binary\_pair}(B_1||B_2) = -(\frac{10}{10+60+15+5+10}log(0.3)+\frac{60}{10+60+15+5+10}log(0.2)) = 0.4717 \tag{8.5}$$

Table 16: Binary Distance Example II

Function Name	Number of Blocks	Match Rate
f_1	10	0.3
f_2	60	0.2
f_3	15	1
f_4	5	1
f_5	10	1

From the above simple examples we can conclude that the larger the distance is, the more different the two binaries are. when the given two binaries are the same, the distance will be zero.

In Section 8.3.2 and Section 8.3.3, we will describe two approaches towards automatic patch generation. The Approach I, APEG presented in paper[25], is designed especially for input validation vulnerabilities. The Approach II is for buffer overflow vulnerabilities using our vulnerability diagnosis results.

8.3.2 Approach I

In this approach, we focus on input validation vulnerabilities where user input is not sufficiently sanitized in P, but is sanitized via new checks in P'. Many common vulnerabilities are input validation vulnerabilities which are fixed by adding input sanitization logic.

This APEG approach towards automatic patch-based exploit generation is based on the observation that the new sanitization checks added to P' often 1) identify the vulnerability point where the vulnerability occurs, and 2) indicate the conditions under which we can exploit P.

Thus, the steps to this approach are:

1. Identify the new sanitization checks added in P'. The remaining steps are performed for each new check individually.

2. Generate a candidate exploit x which fails the new check in P' by:
   a. Calculating the weakest precondition to fail the new check in P'. The result is the constraint formula $\mathcal{F}$. We present three approaches for generating the constraint formula target this problem: dynamic approach, static approach and combination of dynamic and static approach.
   b. Use a solver to find x such that $\mathcal{F}(x) = true$. x is the candidate exploit.
3. Verify a candidate exploit is a real exploit by running $\phi(P(x))$.
4. If desired, we can generate polymorphic variants. Let x be a known exploit. Let $\mathcal{F}'(X) = \mathcal{F}(X) \wedge (X <> x)$. Then x' such that $\mathcal{F}'(x') = true$ is a polymorphic variant exploit candidate. This process can be repeated to enumerate polymorphic variants.

In paper[25], we demonstrate that automatic patch-based exploit generation is practical for real-life patched vulnerabilities.

8.3.3 Approach II

The APEG proposed in Section 8.3.2 is targeted on input validation vulnerabilities. The results are also limited by incorrect loop unrolling and indeterminacy of recursive function call during the step of calculating the weakest precondition. Also, when the constraint formula is too complicated, the constraint solver may take forever to get the solution. For example, MS05018 addresses a stack buffer overflow vulnerability. The patched version of the program updated the `wcscpy` to `wcsncpy`. It can't be handled using Approach I as it is not an input validation vulnerability.

However, using this thesis vulnerability diagnosis results, we can do the APEG for some cases that can't be handled by approach I. The steps to Approach II are similar to Approach I. However, it changes identifying the new sanitization checks using the weakest precondition to identifying each data copy procedure in the updated functions using the vulnerability diagnosis approach. The steps to this approach are:

1. Identify each data copy procedure in P that appears in the updated functions between P and P'. The data copy procedure includes known standard C library functions or

system functions that perform data coping functionality and results by applying the loop analysis step. The remaining steps are performed for each data copy procedure.

2. Generate some random inputs. The remaining steps are performed for each random input.
   a. Collect the runtime traces by running the input.
   b. Check the data copy procedure. If it appears in the run time traces, perform the bound analysis to see the bound results of the destination buffer and to see if some input variables can control the data copy procedure or the destination buffer size.
   c. If some input variables can control the data copy procedure or the destination buffer size, perform input analysis step to see how the inputs are related with those variables.
   d. Generate exploits based on the bound analysis results and input analysis results. Because we know the bound results of the destination buffer and how the user inputs control the data copying procedure, we can generate exploits that can cause hi-jack control attacks more easily than Approach I.
   e. Verify a candidate exploit is a real exploit by running $\phi(P(x))$.

9.0 CONCLUSIONS

A buffer overflow occurs when a store instruction writes outside the allocated buffer bounds. In the best case, a buffer overflow results in the program crashing due to an inappropriate memory dereference. In the worst case, a buffer overflow can be exploited to hijack control of a program. The existence of buffer overflow vulnerabilities makes the system susceptible to Internet worms and denial of service (DDoS) attacks which can cause huge social and financial impacts.

Due to its importance, buffer overflow problem has been intensively studied since its emergence. However, automatic BOVD is still an open problem. It is a big gap of software security research. Currently, the vulnerability diagnosis is done manually by security engineers. It is an error-prone activity requiring heavy investment in time and manpower which often delays the protection time, e.g. patch generation time.

This thesis defines the automatic BOVD problem and provides solutions towards automatic BOVD for commodity software. It targets on commodity software when source code and symbol table are not available because the binary code is ubiquitous and the binary code is high fidelity important to the security problem even though it is much more difficult to analyze binary code.

The automatic BOVD problem is: given a buffer overflow vulnerable program P and a working exploit E, we want to automatically and accurately find out where the program is vulnerable, *buffer overflow vulnerability point* (BOVP) and figure out why the program is vulnerable, *buffer overflow vulnerability condition* (BOVC). Since buffer overflow attack occurs by overwriting the security critical data, for example, return address, function pointer etc. We use the instruction that does the malicious write to pinpoint the vulnerability location of the given vulnerable program. The BOVC contains the destination buffer size information, the total number of writes information and how the user inputs are related with them. The above BOVC results assume the buffer overflow happens through a data copy procedure. This thesis can also identify the case when the buffer overflow attack happens through an unchecked array index instead of a data copy procedure.

The solutions combine both of the dynamic analysis techniques and static analysis techniques to achieve the goal. It extends the existing dynamic-taint-analysis attack detection tool to do the BOVP detection and performs three steps analysis towards understanding BOVC. The BOVP is achieved by searching back the trace for the latest instruction that overwrites the security critical data after the attack detection.

Based on the observation that buffer overflow attack happens when the size of the destination buffer is smaller than the total number of writes after the data copying process if the buffer overflow attack happens through a data copy procedure, the three steps analysis towards understanding the BOVC include loop analysis, bound analysis and input analysis. The bound analysis answers the size of the destination buffer information. The loop analysis answers the total number of writes information of programmer implemented data copy procedure. For buffer overflow attack happens through standard C library functions (library data copy procedures), we answer the total number of writes information trough function summaries. The input analysis answers how the user inputs are related with the size of the destination buffer and the total number of writes.

Our loop analysis is based on the observation that most of the real world buffer overflow attacks happen through loop context. We use loop detection to locate a programmer implemented data copy procedure and we propose a generalized data copy procedure and apply loop analysis on it to answer the total number of writes question.

Our bound analysis is achieved through memory traces analysis. This is based on the observation that our runtime traces contain all of the memory update information. Because we can overwrite those variables that have not been written, we can guarantee the fine-grained bound result for the given execution path. Also, because our bound analysis is independent of the location of the vulnerable buffer, we can handle the buffer overflow that happens anywhere including stack, heap, Data/BSS.

Our input analysis generates the dynamic slicing of the traces contains the malicious variable(s) and performs symbolic execution on it to answer how the user inputs are related with those malicious variable(s).

The automatic BOVD has many applications including automatic patch generation, automatic signature generation, automatic exploit generation, and the assistance for the security engineers to understand the vulnerability condition easier and faster. This thesis also discusses how to generate signatures with zero-false positives and zero-false negatives by combining with other research work. Real world vulnerable applications including the buffer overflow vulnerabilities targeted by the record-setting Slammer and Blaster worms are studied. The automatic diagnosis results on those programs demonstrate the effective of this thesis work's approach.

Furthermore, this thesis also does the buffer overflow vulnerability case study. This case study result can have independent interests to researchers. This is because of the lack of previous research on this problem. Researchers use their own heuristic when they design attack response systems or debugging tools. This makes the current research solutions far from satisfaction. Our buffer overflow case study results can help the other researchers to design more effective response systems and debugging tools against buffer overflow attacks.

In summary, this thesis makes the following contributions to software security research:

- It combines both of the static binary program analysis techniques and dynamic taint analysis techniques to provide solutions towards automatic BOVD. It extends the existing dynamic-taint-analysis attack detection tool to do the BOVP detection and performs three steps analysis towards understanding BOVC. The three steps are loop analysis, bound analysis and input analysis.
- It uses loop analysis to understand programmer written data copy procedures under reasonable loop heuristic assumptions. This is based on the observation of all known buffer overflow attacks that most of them take place in loop context.
- It achieves fine-grained bound analysis result of the vulnerable buffer using novel memory trace analysis approach. This can be achieved based on the fact that the runtime traces contain all of the memory operation history. This memory operation includes function call, function return, variable push and pop, variable load and store, `malloc`, `free` etc.

- It performs buffer overflow vulnerability case study which may have independent interests to security engineers. There are six basic buffer overflow vulnerability cases that cover all of the real world vulnerable applications I studied. Other theoretically possible cases are also discussed. This result can help other researchers to design more effective defense systems.

APPENDIX

RUNNING EXAMPLES

```
struct Data{char data[36]; int tag;}
void process(int count, char* source){
  struct Data *chunk;
  int i, tmp;
  tmp = sizeof(struct Data);
  chunk = (struct Data *)malloc(tmp);
  chunk -> tag = 1;
  for (i=0; i<count; i++)
    chunk -> data[i++] = *source++;
}
void main() {
  char name[65], pass[65], buffer[1024];
  int count, socket;
  ...
  n = read(socket, name, 64);
  n = read(socket, pass, 64);
  n = read(socket, &count, 4);
  ...
  sprintf(buffer, "\%s \%s", name, pass);
  process(count, buffer);
}
```

Figure 23: The running example of buffer overflow vulnerable program.

```
struct Data{char data[36]; int tag;}
void process(char* source){
  struct Data *chunk;
  int i, tmp;
  tmp = sizeof(struct Data);
  chunk = (struct Data *)malloc(tmp);
  chunk -> tag = 1;
  strcpy(chunk->data, source);
}
void main() {
  char name[65], pass[65], buffer[1024];
  int socket;
  ...
  n = read(socket, name, 64);
  n = read(socket, pass, 64);
  ...
  sprintf(buffer, "\%s \%s", name, pass);
  process(buffer);
}
```

Figure 24: Buffer Overflow Cases: Case I

```
void process(int size, char* source){
  char *chunk;
  chunk = (char *)malloc(size);
  strcpy(chunk, source);
}
void main() {
  char name[65], pass[65], buffer[128];
  int size, socket;
  ...
  n = read(socket, name, 64);
  n = read(socket, pass, 64);
  n = read(socket, &size, 4);
  ...
  snprintf(buffer, 127, "\%s \%s", name, pass);
  process(size, buffer);
}
```

Figure 25: Buffer Overflow Cases: Case II

```
void process(int size, char* source){
  char *chunk;
  chunk = (char *)malloc(size);
  strcpy(chunk, source);
}
void main() {
  char name[65], pass[65], buffer[1024];
  int count, socket;
  ...
  n = read(socket, name, 64);
  n = read(socket, pass, 64);
  n = read(socket, &count, 4);
  ...
  sprintf(buffer, "\%s \%s", name, pass);
  process(count, buffer);
}
```

Figure 26: Buffer Overflow Cases: Case III

```
struct ANIChunk{
  char tag[4];
  DWORD size;
  char data[size];
}
struct ANIHeader{
  char data[36];
}
int LoadAniIcon(struct MappedFile* file, ...){
  struct ANIChunk chunk;
  struct ANIHeader header;
  ...
  while(1){
    // read the chunk data from file.
    ReadTag(file, &chunk);
    switch(chunk.tag){
      case 'seq':
      ...
      case 'anih':
       memcpy(&header, chunk.data, chunk.size);
    }
  }
}
```

Figure 27: Buffer Overflow Cases: Case IV, motivated by ANI(MS07017)

```
int read_header(int sid){
  char line[2048];
  ...
  sgets(line, sizeof(line)-1, conn[sid].socket);
  ...
  conn[sid].dat->in_ContenctLength = atoi((char *)&line+16);
  ...
  conn[sid].PostData = calloc(conn[sid].dat->in_ContenctLength + 1024, sizeof(char));
  ...
  rc = recv(conn[sid].socket, conn[sid].PostData, 1024, 0);
  ...
}
```

Figure 28: Buffer Overflow Cases: Case V, motivated by nullhttpd

```
struct PLTEChunk{
  DWORD PLTEentry;
  char PLTEdata[PLTEentry];
}
struct TRNSChunk{
  DWORD TRNSsize;
  char TRNSdata[TRNSsize];
}
void processTRNS(struct MappedFile* file, ...){
  struct PLTEChunk pChunk;
  struct TRNSChunk tChunk;
  ...
  // read the PLTE chunk data from file.
  ReadPLTEData(file, &pChunk);
  ...
  // read the TRNS chunk data from file.
  ReadTRNSData(file, &tChunk);

  for (i = 0; i < tChunk.TRNSsize; i++){
     updatePLTEentry(pChunk.PLTEdata[i], tChunk.TRNSdata[i]);
  }
}
```

Figure 29: Buffer Overflow Cases: Case VI, motivated by PNG(MS05025)

```
void process(int size, char* source){
  char chunk[36];
  for (int i = 0; i < size && source[i] != ’0’; i++){
    chunk[i] = *source++;
  }
}
void main() {
  char name[65], pass[65], buffer[1024];
  int count, socket;
  ...
  n = read(socket, name, 64);
  n = read(socket, pass, 64);
  n = read(socket, &count, 4);
  ...
  sprintf(buffer, "\%s \%s", name, pass);
  process(count, buffer);
}
```

Figure 30: Buffer Overflow Cases: Other Cases (Multiple Loop Exit Conditions)

```
void process(int offset, char* source){
  char chunk[36];
  for(int i = 0; i < 36; i++){
    chunk[offset + i] = source[i];
  }
}
void main() {
  char name[65], pass[65], buffer[1024];
  int offset, socket;
  ...
  n = read(socket, name, 64);
  n = read(socket, pass, 64);
  n = read(socket, &offset, 4);
  ...
  sprintf(buffer, "\%s \%s", name, pass);
  process(offset, buffer);
}
```

Figure 31: Buffer Overflow Cases: Other Cases (startOffset variable in Equation 3.3 can be controlled by user input)

```
void process(int index){
  char chunk[36];
  chunk[index] = 'H';
}
void main() {
  int index, socket;
  ...
  n = read(socket, &index, 4);
  ...
  process(index);
}
```

Figure 32: Buffer Overflow Cases: Other Cases (Malicious Array Index)

```
$iv_1 = iv_{1update}$
$iv_2 = iv_{2update}$
while($iv_1 < \varphi()$){
  $writeAddress = startAddress + startOffset + iv_2$;
  [writeAddress] = {some value};
  $iv_1 += iv_{2update}$;
  $iv_2 += iv_{2update}$;
}
```

Figure 33: Buffer Overflow Cases: Other Cases (Other Loop_Bound)

BIBLIOGRAPHY

[1] http://www.owasp.org/index.php/Testing_for_Stack_Overflow/.

[2] http://www.blackhat.com/.

[3] http://www.blackhat.com/presentations/win-usa-04/bh-win-04-flake.pdf/.

[4] http://www.uninformed.org/?v=1&a=2&t=pdf/.

[5] http://bitblaze.cs.berkeley.edu/.

[6] http://en.wikipedia.org/wiki/Calling_convention/.

[7] http://hex-rays.com/idapro/.

[8] http://www.w3.org/TR/PNG-Chunks.html#C.tRNS/.

[9] http://www.securityfocus.com/bid/5774/.

[10] http://en.wikipedia.org/wiki/Kullback-Leibler_divergence.

[11] Codesonar. http://www.grammatech.com/products/codesonar/overview.html?gclid=CO7ssufP_Y8CFQltZQod6wYdLQ.

[12] Leel. http://www.geocities.com/fasterlu/leel.htm.

[13] Qemu. http://fabrice.bellard.free.fr/qemu/.

[14] Wcet project homepage, 2007. http://www.mrtc.mdh.se/projects/wcet.

[15] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations and applications. In *In 12th ACM Conference on Computer and Communications Security*, November 2005.

[16] A. Aho, R. Sethi, and J.Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1996.

[17] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with wit. In *In Proceedings of the IEEE Symposium on Security and Privacy*, May 2008.

[18] K. Avijit, P. Gupta, and D. Gupta. Tied, libsafeplus: Tools for runtime buffer overflow protection. In *In Proceedings of the 13th USENIX Security Symposium*, August 2004.

[19] G. Balakrishnan. Wysinwyx:what you see is not what you execute. Ph.D. Dissertation, CS, University of Wisconsin-Madison.

[20] A. Baratloo, N. Singh, and T. Tsai. Libsafe: Protecting critical elements of stacks. http://www.research.avayalabs.com/project/libsafe/, December 1999.

[21] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *In Proceedings of the 2000 USENIX Technical Conference*, June 2000.

[22] S. Bhatkar, D. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *In Proceedings of the 12th USENIX Security Symposium*, 2003.

[23] S. Bhatkar, R. Sekar, and D. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *In Proceedings of the 12th USENIX Security*, 2005.

[24] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *In Proceedings of the IEEE Symposium on Security and Privacy*, May 2006.

[25] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *In Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

[26] D. Brumley, H. Wang, S. Jha, and D.Song. Creating vulnerability signatures using weakest preconditions. In *In Proceedings of the 2007 Computer Security Foundations Symposium*, July 2007.

[27] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. Exe: Automatically generating inputs of death. In *In Proceedings of the 13th ACM Conference on Computer and Communication Security*, October 2006.

[28] M. Castro, M. Costa, and T. Harris. Securing software by enforcing dataflow integrity. In *In Proceedings of the 7th Usenix Symposium on Operating Systems Design and Implementation*, November 2006.

[29] S. Chen, J. Xu, and E. Sezer. Non-control-data attacks are realistic threats. In *In Proceedings of the 14th USENIX Security Symposium*, August 2005.

[30] M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. Technical report, Carnegie Mellon University, 2002.

[31] T. Chiueh and F. Hsu. Rad: A compile-time solution buffer overflow attacks. In *In Proceedings of the 21st International Conference on Distributed Computing Systems*, April 2001.

[32] C. Cifuetes and M. V. Emmerik. Uqbt: Adaptable binary translation at low cost. In *IEEE Computer*, March 2000.

[33] J. Cocke and R. Miller. Some analysis techniques for optimizing computer programs. In *In Proceedings of Second International Conference on System Sciences*, 1969.

[34] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: Securing software by blocking bad input. In *In Proceedings of the IEEE Symposium on Security and Privacy*, May 2007.

[35] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *In 20th ACM Symposium on Operating Systems Principles*, October 2005.

[36] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *In Proceedings of the 12th USENIX Security Symposium*, 2003.

[37] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *In Proceedings of the 7th USENIX Security Conference*, 1998.

[38] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *In Proceedings of the DARPA Information Survivability Conference and Exposition*, 2000.

[39] J. Crandall and F. Chong. Minos: Architectural support for software security through control data integrity. In *In 37th International Symposium on Microarchitecture*, 2004.

[40] J. Crandall, Z. Su, S. Wu, and F. Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *In 12th ACM Conference on Computer and Communications Security*, November 2005.

[41] W. Cui, M. Peinado, H. Wang, and M. Locasto. Shieldgen: Automatic data patch generating for unknown vulnerabilities with informed probing. In *In Proceedings of the IEEE Symposium on Security and Privacy*, May 2007.

[42] S. Designer. Linux kernel patch from the openwall project. http://www.openwall.com/linux/README.

[43] D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for c with very low overhead. In *In 28th International Conference on Software Engineering*, 2006.

[44] Z. Ding, R. Hoare, A. Jones, D. Li, S. Shao, S. Tung, J. Zheng, and R. Melhem. Switch design to enable predictive multiplexed switching in multiprocessor netowrks. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[45] eEyE Security. eEye binary diffing suite (EBDS). http://research.eeye.com/html/tools/RT20060801-1.html. Version 1.0.5.

[46] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *WCET*, 2007.

[47] H. Etoh and K. Yoda. Gcc extension for protecting applications from stack-smashing attacks. http://www.trl.ibm.com/projects/security/ssp/.

[48] M. Frantzen. Stackghost: Hardware facilitated stack protection. In *In Proceedings of the 10th USENIX Security Symposium*, August 2001.

[49] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *In ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2005.

[50] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *In Proceedings of the 15th Annual Network and Distributed System Security Symposium*, February 2008.

[51] N. Gupta, A. Mathur, and M. Soffa. Generating test data for branch coverage. In *In Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, September 2000.

[52] M. Hecht and J. Ullman. Flow graph reducibility. *SIAM J. Computing*, 1(2), June 1972.

[53] M. Howard. Lessons learned from the animated cursor security bug, 07. http://blogs.msdn.com/sdl/archive/2007/04/26/lessons-learned-from-the-animated-cursor-security-bug.aspx/, 2007.

[54] G. Hunt and D. Brubacher. Detours: Binary interception of win32 functions. In *In the Proceedings of the third Usenix NT Symposium*, July 1999.

[55] R. Jhala and R. Majumdar. Path slicing. In *In ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2005.

[56] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of c. In *In Proceedings of the USENIX Annual Techniqual Conference*, June 2002.

[57] A. Jones, J. Zheng, and A. Amer. Entropy based evaluation of communication predictability in parallel applications. In *IEICE Transactions Special Section on Parallel/Distributed Computing and Networking*, 2005.

[58] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *In Proceedings of the Third International Workshop on Automated Debugging*, 1995.

[59] G. Kc, A. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *In Proceedings of the 10th ACM Conference on Computer and Communication Security*, 2003.

[60] H. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *In Proceedings of the 13th USENIX Security Symposium*, August 2004.

[61] C. Kreibich and J. Crowcroft. Honeycomb - creating intrusion detection signatures using honeypots. In *In Proceedings of the 2nd Workshop on Hot Topics in Networks*, November 2003.

[62] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *In Proceedings of the 13th USENIX Security Symposium*, 2004.

[63] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *In Proceedings of the 10th USENIX Security Symposium*, August 2001.

[64] J. Larus and E. Schnarr. Eel: Machine-independent executable editing. In *In ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1995.

[65] Z. Liang and R. Sekar. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *In 12th ACM Conference on Computer and Communications Security*, November 2005.

[66] S. Muchnick. *Advanced Compiler Design and Implementation*. Academic Press, 1997.

[67] G. Necula, S. McPeak, and W. Weimer. Ccured: Typesafe retrofitting of legacy code. In *In Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, January 2002.

[68] G. Necula, S. McPeak, and W. Weimer. Taming c pointers. In *In Proceedings of ACM Conference on Programming Language Design and Implementation*, January 2002.

[69] J. Newsome, D. Brumley, and D. Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *In Proceedings of the 13th Annual Network and Distributed System Security Symposium*, February 2006.

[70] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *In Proceedings of the IEEE Symposium on Security and Privacy*, May 2005.

[71] J. Newsome, B. Karp, and D. Song. Paragraph: Thwarting signature learning by training maliciously. In *In Rapid Advances in Intrusion Detection*, 2006.

[72] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *In Proceedings of the 12th Annual Network and Distributed System Security Symposium*, 2005.

[73] A. One. Smashing the stack for fun and profit. *Phrack*, 7(49), Nov 1996.

[74] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif. Misleading worm signature generators using deliberate noise injection. In *In Proceedings of the IEEE Symposium on Security and Privacy*, May 2006.

[75] M. Prasad and T. Chiueh. A binary rewriting defense against stack-based buffer overflow attacks. In *In Proceedings of the USENIX Annual Technical Conference*, June 2003.

[76] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *In 39th International Symposium on Microarchitecture*, 2006.

[77] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, and B. Bershad. Instrumentation and optimizaiton of win32/intel executables using etch. In *In USENIX Windows NT Workshop*, 1997.

[78] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. In *In Proceedings of the 11th Annual Network and Distributed System Security Symposium*, 2004.

[79] F. Schneider. Enforceable security policies. In *ACM Transactions on Information and System Security*, Feb 2000.

[80] E. Sezer, P. Ning, C. Kil, and J. Xu. Memsherlock: An automated debugger for unknown memory corruption vulnerabilities. In *In 14th ACM Conference on Computer and Communications Security*, 2007.

[81] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *In 11th ACM Conference on Computer and Communications Security*, 2004.

[82] S. Sidiroglou and A. D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security and Privacy*, 3(6):41–49, 2005.

[83] I. Simon. A comparative analysis of methods of defense against buffer overflow attacks. http://www.mcs.csuhayward.edu/~simon/security/boflo.html.

[84] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *In Proceedings of the 6th ACM USENIX Symposium on Operating System Design and Implementation*, December 2004.

[85] A. Smirnov and T. Chiueh. Dira: Automatic detection, identification, and repair of control-hijacking attacks. In *In Proceedings of the 12th Annual Network and Distributed System Security Symposium*, February 2005.

[86] A. Sovarel, D. Evans, and N. Paul. Where's the feeb? the effectiveness of instruction set randomization. In *In Proceedings of the 14th USENIX Security Symposium*, 2005.

[87] A. Srivastava and A. Eustace. Atom:a system for building customized program analysis tools. In *In ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1994.

[88] G. Suh, J. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *In Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[89] R. Tarjan. Testing flow graph reducibility. In *In Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, 1973.

[90] N. Vachharajani, M. Bridges, J. Chang, R. Rangan, G. Ottoni, J. Blome, G. Reis, M. Vachharajni, and D. August. Rifle: An architecture framework for user-centric information-flow security. In *In 37th International Symposium on Microarchitecture*, 2004.

[91] Vendicator. Stack shield technical info file v0.7. http://www.angelfire.com/sk/stackshield/, January 2001.

[92] D. Wagner and D. Dean. Intrusion detection via static analysis. In *In Proceedings of the IEEE Symposium on Security and Privacy*, 2001.

[93] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *In Proceedings of Network and Distributed System Security Symposium*, 2000.

[94] H. Wang, C. Guo, D. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *In ACM SIGCOMM*, August 2004.

[95] X. Wang, Z. Li, J. Xu, M. Reiter, C. Kil, and J. Choi. Packet vaccine: black-box exploit detection and signature generation. In *In 14th ACM Conference on Computer and Communications Security*, 2006.

[96] M. Weiser. Program slicing. In *IEEE Transactions on Software Engineering*, July 1984.

[97] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *In Proceedings of the 10th Annual Network and Distributed System Security Symposium*, 2003.

[98] M. Wolfe. Beyond induction variables. In *ACM SIGPLAN'92 PLDI*, 1992.

[99] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt. Automatic diagnosis and response to memory corruption vulnerabilities. In *In 12th ACM Conference on Computer and Communications Security*, November 2005.

[100] W. XU, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *In Proceedings of the 15th USENIX Security Symposium*, 2006.

[101] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically generating malicious disks using symbolic execution. In *In Proceedings of the IEEE Symposium on Security and Privacy*, May 2006.

[102] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. Accmon: Automatically detecting memory-related bugs via program counter-based invariants. In *In 37th International Symposium on Microarchitecture*, December 2004.