The dissertation of Qiang Zeng was reviewed and approved* by the following:

Peng Liu  
Professor of Information Science and Technology  
Dissertation Advisor, Co-Chair of Committee

Thomas La Porta  
Professor of Computer Science and Engineering  
Co-Chair of Committee

Sencun Zhu  
Associate Professor of Computer Science and Engineering

Soundar R.T. Kumara  
Professor of Industrial and Manufacturing Engineering

Raj Acharya  
Professor of Computer Science and Engineering  
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.
Abstract

The society nowadays relies heavily on digitized information and services. Among others, cyber-security is one of the cornerstones of the digital world. The reality is that everyday numerous computer systems are compromised, and that sensitive information is leaked, corrupted or forged. It does not only cause massive loss, but also hurts the confidence of people over digitized information processing, such as electronic commerce, digital hospitals, and online banking.

In order to enhance software security, the status of program execution is usually checked and verified, aiming at detecting anomalies and cyber-attacks in their early stages. Once an intrusion is detected, the service provider needs to diagnose the attack and fix the issue promptly.

Although the procedure of monitoring, diagnosing, and fixing is widely adopted when dealing with software failures as well as security incidents, there exist many unresolved issues in each of the actions. First, security checking and verification interleave with functional code, and thus slow down program execution; in reality, security is frequently sacrificed for the sake of speed. Second, enterprise software is complicated, comprising millions of lines of code and a whole stack of intricate components. Once an anomaly is detected, it is like looking for a noodle in a haystack to diagnose an attack and figure out the root cause. Third, after a software vulnerability is reported, patch generation by the software company is a lengthy process, which leaves the system vulnerable to attacks for a long time.

Our work is devoted to making the procedure of monitoring, diagnosing and fixing more efficient and intelligent. We thus proposed, built, and evaluated techniques towards concurrent monitoring, automated diagnosis, and instant defense generation. First, in order to resolve the tension between security checking and performance optimization, we propose a novel concurrent monitoring technology, named *software cruising*, which separates security checking from program functionality computation and runs them on separate processors or cores. It enforces monitoring in a concurrent and non-blocking fashion, and is featured with high efficiency and scalability. Unlike conventional security techniques, which usually trade effect for efficiency, software
cruising satisfies both the monitoring effect and efficiency needs.

Next, one of the main reasons that diagnosis is time-consuming is the lack of critical information in logs. Among a variety of runtime information, the calling context, i.e., the sequent of functions on the call stack, is especially useful; it provides precise information about which components are connected to the anomalies. While some techniques have been proposed to track calling context efficiently, they lack a reliable and precise decoding capability; or they work only under restricted conditions, that is, small programs without object-oriented programming or dynamic component loading. These shortcomings have limited the application of calling context tracking in practice. We propose an encoding technique, named DeltaPath, without those limitations: it provides precise and reliable decoding, supports large-sized programs, both procedural and objected-oriented ones, and copes with dynamic class/library loading. The technique thus enables calling context tracking in a wide variety of scenarios.

Finally, We present a new form of defense generation for implementing self-shielding software. Given an instance of exploitation of a software vulnerability, a defense can be generated (without resorting to the software company) instantly and automatically. We have applied the technique to dealing with buffer overrun bugs, such as the Heartbleed vulnerability. Our insight is that, given a buffer overrun bug, the buffers that can be overrun share the same calling context when they were allocated. Based on the observation, we creatively utilize the calling context encoding technique to characterize and distinguish heap buffers that can be exploited by attacker, and apply costly enhancement precisely to those problematic buffers. We present HeapTherapy, a heap memory allocator that performs the characterization and installs defenses automatically. Our experiments illustrate that by applying HeapTherapy Nginx server becomes immune to the Heartbleed attack. Moreover, HeapTherapy defeats various other real-world overflow attacks and the slowdown averages only 6% on SPEC CPU2006.

By leveraging rich computation resources in multicore architectures as well as techniques such as virtualization, software cruising performs non-blocking monitoring with minimal performance penalty. Due to the availability of critical runtime information, diagnosis becomes directed and precise. The instant defense generation represents a promising direction for implementing self-shielding software. The evaluation shows that software security can be significantly enhanced through concurrent monitoring, intelligent anomaly diagnosis, and instant defense generation.
# Table of Contents

List of Figures ix

List of Tables xii

Acknowledgments xiii

## Chapter 1

### Introduction 1

1.1 An Insecure and Buggy Digital World 1

1.2 Challenges 2

1.2.1 Tension Between Monitoring and Performance 2

1.2.2 Diagnosis Lacks Critical Information 3

1.2.3 Patching Takes Much Time 3

1.3 Ideas and Techniques 4

1.3.1 Non-blocking Monitoring 4

1.3.1.1 Cruiser 5

1.3.1.2 Kruiser 7

1.3.2 Calling Context Encoding for Diagnosis 9

1.3.3 Context Sensitive and Automated Defense Generation 11

1.4 Summary of Contributions 13

## Chapter 2

### Related Work 15

2.1 Buffer Overflow Countermeasures 15

2.2 Virtual-machine introspection 18

2.3 Calling Context Encoding 19

2.4 Automated Defense Generation 20

## Chapter 3

### Cruiser: A Concurrent User-space Heap Integrity Scanner 22
3.1 Background .......................................................... 22
  3.1.1 Lock-free Synchronization .................................. 22
3.2 Motivations .......................................................... 23
  3.2.1 Why Concurrent Detection and Challenges ................. 23
  3.2.2 Why Lock-free and Challenges .............................. 24
3.3 Design Overview ................................................... 25
  3.3.1 Buffer Structure .............................................. 25
  3.3.2 Cruiser ......................................................... 26
3.4 Design and Implementation ...................................... 27
  3.4.1 Collection of Buffer Information ......................... 27
    3.4.1.1 Express data structure ................................ 27
    3.4.1.2 Warehouse data structure ............................. 29
    3.4.1.3 Applying CIC in Cruiser ............................... 33
  3.4.2 Buffer Release and Node Removal ......................... 34
  3.4.3 Extensions and Optimizations .............................. 36
    3.4.3.1 Extensions .................................................. 36
    3.4.3.2 Optimizations ............................................ 37
3.5 Evaluation ........................................................ 38
  3.5.1 Effectiveness .................................................. 38
  3.5.2 Performance and Memory Overhead ......................... 39
  3.5.3 Scalability .................................................... 40
  3.5.4 Detection Latency ............................................ 41
3.6 Software Cruising ................................................ 43
3.7 Summary .......................................................... 44

Chapter 4
Kruiser: A Concurrent Kernel Heap Integrity Scanner 45
4.1 Challenges .......................................................... 45
4.2 Overview .......................................................... 47
4.3 Kernel Cruising ................................................... 48
  4.3.1 Page Identity Array ......................................... 49
  4.3.2 Race conditions ............................................... 50
  4.3.3 Semi-synchronized Non-blocking Cruising ............... 51
4.4 System Design and Implementation ............................. 55
  4.4.1 Background .................................................... 55
  4.4.2 Architecture ................................................... 55
  4.4.3 Direct Memory Mapping .................................... 56
  4.4.4 In-VM Protection .............................................. 59
  4.4.5 Placing Canaries ............................................. 61
  4.4.6 Locating Canaries ............................................ 63
### Chapter 4

**Evaluation**

- **Effectiveness**
- **Performance and Scalability**
- **Detection Latency**

**Scalable Deployment**

**Summary**

### Chapter 5

**DeltaPath: Precise and Scalable Calling Context Encoding**

- **Motivation and Main Ideas**
- **Background**
- **DeltaPath Encoding**
  - **Encoding for Object-oriented Programs**
  - **Encoding for Large-scale Object-oriented Programs**
- **Practical Issues**
  - **Dynamic Class Loading**
  - **Flexible Encoding**
- **Implementation**
- **Evaluation**
  - **Static Program Characteristics**
  - **Performance Comparison**
  - **Dynamic Program Characteristics**
- **Related Work**
- **Summary**

### Chapter 6

**HeapTherapy: A Self-shielding Heap Memory Allocator**

- **Motivation and Main Ideas**
- **Scenarios of Applying Codeless Patching**
- **Description Method**
  - **Requirements**
  - **A Context-Sensitive Perspective**
  - **Calling-Context Retrieval and Comparison**
- **System Design**
  - **Codeless Patch**
  - **Architecture**
- **Offline Patch Generation**
- **Defense against Zero-day Overflow Attacks**
- **Implementation**
- **Evaluation**
List of Figures

3.1 Buffer structure. .......................................................... 26
3.2 Cruiser architecture. .................................................... 26
3.3 CruiserList. ............................................................... 30
3.4 Deletion of the first genuine node. ................................. 32
3.5 Execution time with Cruiser (normalized by the execution time without Cruiser), compared with DieHarder. The last set of bars is the geometric mean of the execution time of all benchmarks. ......................... 40
3.6 Throughputs of the Apache web server for varying numbers of concurrent requests. ................................................... 42

4.1 Overview of Kruiser. ..................................................... 48
4.2 Kruiser Architecture. .................................................. 56
4.3 The direct memory mapping mechanism. ........................... 57
4.4 Memory protections in the kernel and monitor address space ..... 60
4.5 Address space switching via transition pages in SMP. .......... 60
4.6 Placing canaries into kernel objects. ............................... 62
4.7 PIA entry. ............................................................... 62
4.8 SPEC CPU2006 performance (normalized to the execution time of original Linux). ....................................................... 66
4.9 Throughput of the Apache web server for varying numbers of concurrent requests. ....................................................... 66

5.1 Example for PCCE encoding. Edge annotations are addition values; an addition value “+c” means “ID+=c” is executed before the invocation and “ID-=c” is executed after; the superscript on D (D′) disambiguates two call sites in D both invoking E. ........................................ 73
5.2 Intuition of our algorithm. ............................................. 75
5.3 Intuition of encoding with dynamic dispatch. A virtual function call in $p$ can be dispatched to multiple methods $n_1, n_2, \ldots, n_k$. The candidate addition value of a node $n_i$, denoted by $CAV[n_i]$, is initially 0, and it is updated each time the addition value of an edge leading to $n_i$ is calculated. We process the call in two steps. First, the addition value $a = \max\{CAV[n_1], \ldots, CAV[n_k]\}$ is obtained; second, $CAV[n_i]$ is updated as $ICC[p] + a$, for $i = 1, \ldots, k$. The updated $CAV[n_i]$ is then used to calculate the addition value for the next incoming edge of $n_i$.

5.4 Example for encoding with dynamic dispatch. The superscript on $D$ ($D'$) disambiguates two call sites in $D$ both invoking $E$. $D'E$ and $DF$ are due to a virtual function call in $D$, and $CF$ and $CG$ are due to a virtual function call in $C$. Node annotations are ICC values.

5.5 Example for encoding in large programs. $C$ and $D$ are anchor nodes. The superscript on $D$ ($D'$) disambiguates two call sites in $D$ both invoking $E$. $D'E$ and $DF$ are due to a virtual function call in $D$, and $CF$ and $CG$ are due to a virtual function call in $C$. The annotation of a node denotes the ICC values of the node relative to anchor nodes; for example, $ICC[E][D] = 2$ means the ICC of $E$ relative to anchor $D$ is 2. A stack is used to store the anchor node identifiers and the ID values when invoking them. The encoding of a call path $CFG$, for example, is represented by $c$ on the stack, where $c$ contains anchor $C$’s identifier and the encoding ID value when $C$ is invoked, along with the encoding ID value 2, which is the sum of addition values of $CF$ and $FG$.

5.6 Incomplete call graph. $BD$ and $BX$ are due to the same virtual function call, where $X$ is from a dynamically loaded class unexpected by static analysis.

5.7 JDK library classes are excluded from encoding.

5.8 Execution speeds applying PCC, DeltaPath without call path tracking (wo/CPT), and DeltaPath with call path tracking (w/CPT), respectively. The speed means the throughput (operations per minute) that reflects the rate at which the system was able to complete invocations of the workload of that benchmark, and it is normalized against the native runtime.

6.1 The process of applying Codeless Patching.

6.2 Part of the calling context tree of MySQL 5.5.19. The dashed lines indicate omitted functions.

6.3 An example of PCC encoding. Three lines are added to maintain the encoding in the function.

6.4 The architecture of Codeless Patching.
6.5 The pseudo code of the malloc function. ................................. 102
6.6 A buffer with a guard page attached. ................................. 103
6.7 A buffer with a padding and a guard page attached. ............... 104
6.8 A buffer surrounded by canaries. ................................. 106
6.9 The relationship of the two vulnerable calling contexts identified by the
diagnosis engine. The dashed lines indicate omitted functions. .... 108
6.10 Execution time overhead and RSS memory overhead of CCID encoding. 113
6.11 Execution time overhead of HeapTherapy. Each bar represents one ex-
perimenatal setting. For example, “5 patches, 5-page padding” represents
the setting in which 5 simulated patches are applied and each vulnerable
buffer is padded with 5 extra pages. The guard page is also enabled for
all vulnerable buffers. ........................................... 113
6.12 RSS memory overhead of HeapTherapy. ............................. 114
List of Tables

2.1 Comparison of some widely deployed countermeasures and Cruiser. . . 17

3.1 The effectiveness experiment against real-world vulnerabilities. . . . 38

3.2 Memory overhead and cruise cycle (Results of Eager Cruiser are en-
closed in parentheses; the maximum/average CruiserList lengths are
normalized by the maximum/average heap buffer counts, respectively). . 41

4.1 Different cruising cycle for different applications in the SPEC CPU2006
benchmark (The numbers refer to the numbers of kernel objects that are
scanned in a cruising cycle). . . . . . . . . . . . . . . . . . . . . . . . . . . . 67

5.1 Static program characteristics (SPECjvm2008). The maximum encoding
ID values of sunflow and xml.validation, shown in bold, are larger
than the maximum value of a 64-bit integer (around 1.8e19), so the two
benchmarks need anchor nodes. . . . . . . . . . . . . . . . . . . . . . . . 86

5.2 Dynamic program characteristics (SPECjvm2008). . . . . . . . . . . 89

6.1 A list of tested programs that contain heap buffer overflow bugs. Our
evaluation shows that HeapTherapy can prevent not only heap data
corruption but also program crashes in all cases. . . . . . . . . . . . . . 109

6.2 Buffer allocation and CCID profiling. Benchmark programs in bold text
are allocation-intensive. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112
Acknowledgments

The completion of this dissertation marks the end of my doctoral study at The Pennsylvania State University. Throughout the time, I have got invaluable help from many people, whom I am deeply grateful to.

My advisor, Peng Liu, has given me the greatest guidance and support. His open-mindedness is one of the main reasons I keep exploring new areas and proposing fresh ideas. Once and again, he encourages me to take a “deep dive” into research problems, and not to give up upon failures. He always offers generous help in both my research and life. In November 2011 my father had a stroke, and I hence returned to China to take care of my family until my father recovered. Dr. Liu approved the leave and provided financial help. He also provided great support during my job hunting season. I feel tons of gratitude to him.

I would like to thank my collaborators, Dinghao Wu, Donghai Tian, Jorge Lobo, Poonam Yadav, Seraphin Calo, Junghwan Rhee, Hui Zhang, Nipun Arora, Guofei Jiang, and Mingyi Zhao. All the research achievements I made have been due to extensive and close collaborations. For many times some of them and I have worked together overnights. I appreciate the internship opportunities at the IBM T.J. Watson Research Center and the NEC Research Labs America. I will never forget that on a snowing night of January 2013 after working late in the NEC Lab I drove back home, and unfortunately hit a tree and totaled my car; Jun Wang and Jungwhan Rhee gave me rides, which made it possible that I continued to conquer the most challenging part of the ongoing research in the following days.

I would also like to thank my committee members, Thomas La Porta, Sencun Zhu, and Soundar Kumara, who have provided kind help in my proposal and dissertation preparations.

Peng Liu, Jorge Lobo, Junghwan Rhee, Thomas La Porta, and Ji Yu have written me reference letters, for which I am sincerely grateful.

Finally, I give many thanks to all the fellows of the Cyber Security Lab, where we have followed our passion and chased dreams together.
Dedication

To my inspiring parents.
Chapter 1
Introduction

1.1 An Insecure and Buggy Digital World

Individuals, businesses and governments store and process sensitive data on computers. Their privacy and reputation are dependent on a secure computation, so they expect the information will not be leaked, ruined or forged. Unfortunately, most commodity computer platforms are designed primarily for performance and functionality; as a result, they provide insufficient assurance toward security. Most operating systems and a large number of applications employ efficient but unsafe languages (e.g., C or C++). They are released with an unknown number of security vulnerabilities. For example, the National Vulnerabilities Database collect thousands of newly reported software vulnerabilities each year. Some of them are exploited by attackers for extended time before they are noticed and patched by public. Due to the large number of vulnerabilities, numerous computers are compromised. A study show that over 25% of US computers are infected with malicious software [1].

Personal users and enterprises are faced with organized criminals. Financially-motivated attacks have facilities and resources to launch large-scale online attacks. Thousands of computers can be compromised in a short time and then controlled by botnets, which are used for Distributed Denial of Service (DDoS) attacks precisely targeting specific enterprises and governments or sending massive spam and virus. Thus, the digital world has an adversary and insecure environment.

Another issue that repeatedly frustrates both users and developers is software bugs, which themselves are one of the mains reasons of vulnerable software. Although component testing, system testing, static analysis and improved software engineering practice
can catch or prevent many defects before software release, numerous bugs exist with
the shipping of mainstream commercial software. It is inevitable to release software
containing bugs, as the company cannot afford the resource to fully test the software,
and the market cannot wait until the company delivers “clean” software. Debugging and
patching almost accompany the life cycle of commercial software.

1.2 Challenges

After software is released and deployed, it is under the risk of being attacked. In order to
detect intrusions and attacks in early stages, critical security checking and verification are
usually inserted into functional code. Once an attack is detected, system administrators
and security specialists are supposed to respond by finding out the vulnerable component.
If a misconfiguration, rather than a software bug, is the root cause, the fix does not rely
on the software company. Otherwise, they either patch the system, given the patch is
available; or submit a bug/symptom report to the software company and wait for the
patch to be generated. The procedure of monitoring, diagnosing and fixing is widely
used. However, there are many unresolved challenges.

1.2.1 Tension Between Monitoring and Performance

Secure computation may be achieved with adequate security measures. However, decades
of security research has shown that heavy-weight security techniques are rarely adopted,
especially in production systems, because performance usually has higher priority than
security from the point of view of most personal users and businesses. Security enhance-
ment usually competes computation resources with the original logic; such that, the more
sophisticated security policy is enforced, the more severe slowdown is seen. The tension
between security and performance has been a barrier against a more secure computation,
even though it is well realized that our system should be better monitored against attacks.

A close monitoring over program execution for anomaly detection typically directly
slows down the program execution by frequently blocking them. Thus, an efficient
runtime monitoring is desirable.
1.2.2 Diagnosis Lacks Critical Information

Users expect software systems to have good reliability and availability as power supply. The reality, however, is that when a problem arises, it is not unusual that it takes long time to investigate and fix it.

Although enterprises put great amount of effort on improving the situation, and IT professionals and developers spend days and nights investigating the root cause of a problem, the nature of today’s software, which usually comprising millions of lines of code and complicated interactions with other software, complicates the problem investigation and debugging. One of the common complaints is the lack of logs that illustrate how the problem was caused. For example, a crash message only tells a problem has happened, but seldom helps explain how and why. The lack of logs about critical runtime information is one of the main factors that lead to a time-consuming diagnosis.

Problem investigation can be easier given detailed and meaningful execution trace, which has been widely used for in-house testing. During the software testing, every event and environment change can be recorded for later analysis. However, it is not realistic to conduct such detailing logging in production systems, as it can severely slowdown the service. One strategy is to enable the detailed logging only when reproducing the bug. However, due to the complexity of the software stack and the configuration of the enterprise environment, developers may not be able to reproduce a given problem. Sometimes a lot of diverse reasons can lead to the same symptom, which further obscure the investigation effort of reproducing the problem especially

1.2.3 Patching Takes Much Time

Once a zero-day attack is identified through diagnosis, the software company is supposed to debug and generate a patch. Debugging is notoriously challenging and time-consuming, and hence the process of releasing a patch is usually lengthy. After a problem is reported, the software company first contacts system administrators to get an input that can reproduce the problem. With the input, software developers localize the bug using debugging tools like eFence [2] and Valgrind [3], and then fix the code. The patch has to be carefully tested to ensure that new bugs are not introduced. Studies show that the average time between initial reports of a critical security-sensitive bug and the release of a patch for enterprise applications is nearly one month [4].

After the patch is generated, it is important to apply it. Once a vulnerability is well
known, more and more script kiddies acquire scanning and attacking scripts to launch massive attacks. However, applying patches is not risk-free. Fresh patches may contain new bugs leading to instability or changing program semantics accidentally [5]. The trade-off between urgent and cautious patching is a practical consideration and causes patch management predicaments. Research suggests a delay of at least 10 days to balance the risk of applying defective patches and the threat of being attacked.

Delayed patching leaves numerous systems running with publicly known vulnerabilities. According to Symantec, those known non-patched vulnerabilities are responsible for the widespread success of Internet-based attacks [4].

1.3 Ideas and Techniques

1.3.1 Non-blocking Monitoring

Nowadays multicore and manycore architectures are increasingly popular. Considerable computation resources are left idle. We propose to resolve the conflict and tension between monitoring and performance through software cruising, which attains concurrent and efficient monitoring and only incurs negligible or a little performance overhead [6]. The monitoring computation is conducted on spare cores without competing with software execution, while the original program and the monitor communicate in a non-blocking and light-weight fashion, so that both high performance computation and an effective monitoring can be achieved.

An implementation of the software cruising technology in a specific program (or kernel) consists of three parts. First, the original program is slightly modified, either through instrumentation, function interposition or source code modification, to collect minimal but critical information during execution. Second, the information is stored and maintained in an highly efficient data structure. Third, based on the collected information the monitor conducts an efficient and concurrent monitoring work on spare cores.

Software cruising has two prominent characteristics. First, unlike conventional security enforcement which is inlined into the program execution itself, our monitoring runs concurrently with the program execution. Second, the monitor and the program execution are synchronized to enforce an effective monitoring, but the monitor does not block the program to make progress. The two characteristics ensure our monitoring impose minimal performance overhead over the original program execution.
1.3.1.1 Cruiser

While software cruising is a general monitoring technology and can be used to monitor various aspects of program execution, we first apply it to a type of infamous and heavily exploited vulnerabilities: buffer overflows.

Buffer overflow has been one of the top software vulnerabilities. In 2009, 39% of the security vulnerabilities published by US-CERT (cite) were related to buffer overflows. As of September 2010, 12 of the 20 most severe vulnerabilities ranked by US-CERT were buffer overflow related. Vulnerabilities listed by security websites such as SecurityFocus [7] and Securiteam [8] manifest a similar pattern. The related exploits, such as CodeRed [9] and SQLSlammer [10], have inflicted billions of dollars worth of damages [11].

A buffer overflow occurs when a program, while writing data to a buffer, overruns the buffer’s boundary and overwrites adjacent memory. There are mainly two types of buffer overflows according to the overflowed buffer’s memory region, namely stack-based buffer overflows and heap-based buffer overflows.

Stack-based buffer overflows are the most exploited vulnerability, as the return addresses for function calls are stored together with buffers on stack. By overflowing a local buffer, the return address can be overwritten so that, when the function returns, the control flow is redirected to execute malicious code, which is called “stack smashing” [12]. Other forms of stack-based buffer overflow attacks overwrite frame pointers and local variables, e.g. function pointers, to affect program behaviors [13, 14]. Many countermeasures against stack-based buffer overflow attacks have been devised, such as StackGuard [15], StackShield [16], Non-executable stack [17] and Libsafe [18], some of which have been widely deployed.

Exploitation of a heap-based buffer overflow is similar to that of a stack-based buffer overflow, except for that there are no return addresses or frame pointers on heap. For some widely deployed memory allocators, such as Doug Lea’s malloc [19] for the glibc library, altering memory management information to achieve arbitrary memory overwrites is a general way to exploit a heap-based buffer overflow [20, 21]. More recently non-control data based exploits [10, 22, 23], by means of tampering the content of a memory block adjacent to the overflowed buffer, have been increasing.

As stack-based buffer overflow attacks are better understood and defended, heap-based buffer overflows have gained growing attention of attackers. According to the National Vulnerability Database [24], 177 heap-based buffer overflow vulnerabilities
were published in 2009. As of September 2010, 287 heap-based buffer overflow vulnerabilities had been published in that year. Related exploits have affected widely deployed programs [10, 21]. We thus target the handling of heap buffer overflows.

In addition, heap is the most volatile region during program execution. It is more challenging to monitor heap integrity compared to monitor many other program execution characteristics, such as GOT, static variables and code page corruption. In other words if software cruising can be applied to heap integrity problems, it should be able to deal with a lot of other security problems.

Current buffer overflow detectors can be roughly classified into two categories: static and dynamic approaches. Static analysis tools usually have high false alarm rates; dynamic buffer overflow detectors can provide precise detection and generally there can be no false alarms [25]. However few dynamic heap buffer overflow detectors are widely deployed due to one or more of the following reasons: (1) Most countermeasures result in high performance overhead [2,3,26–31]; (2) Some only protect specific libc functions [26, 32]; (3) A few of them only work with specific memory allocators [32, 33]; (4) Many require source code for recompilation [28–31, 34–36]; (5) Some are incompatible with legacy code [34–36]; and (6) Some require special platforms or hardware supports that are rarely available [29, 30].

We present Cruiser—a novel dynamic heap buffer overflow detector which does not have those limitations [6]. The key ideas are (1) to create a dedicated monitor thread, which runs concurrently with user threads to cruise over, or keep checking, dynamically allocated buffers against buffer overflows; and (2) to utilize lock-free data structures and non-blocking algorithms, through which user threads communicate with the monitor thread with minimum overhead and without being blocked. Buffer addresses are collected in a lock-free data structure efficiently without blocking user threads. By traversing the data structure, buffers on heap are under constant surveillance of the concurrent monitor thread. Each dynamically allocated buffer is surrounded by two canary words; as long as a canary is found corrupted, a buffer overflow is detected.

Different from conventional methods that detect buffer overflows inside user threads, which evidently delays protected programs, we propose to move the detection work out of user threads and enforce it in a separate thread, which we call software cruising. The approach leverages the hardware evolution trend that multicore processors and multiprocessor machines are more and more popular, which allows us to deploy dedicated monitor threads running concurrently with user threads to enhance application security. However
applications may be significantly slowed down due to the overhead of communication and synchronization between the monitor threads and user threads. We address this problem by designing highly efficient lock-free data structures and non-blocking algorithms; their scalability implies the approach can be applied to not only single-threaded programs but also large-scale multithreaded applications. Our experiments show that the software cruising approach is practical—it only imposes an average of 5% performance overhead on SPEC CPU2006, and the throughput slowdown on Apache is negligible on average.

1.3.1.2 Kruiser

We then apply software cruising to enhancing system security. Compared with user-space buffer overflows, kernel-space buffer overflow vulnerabilities are more severe in that once such a vulnerability is exploited, attackers can override any kernel-level protection mechanism. Recently, more and more realistic buffer overflow exploits have been released in modern operating systems including Linux [37], OpenBSD [38] and the latest Windows 7 system [39].

Many effective countermeasures against stack-based buffer overflows have been proposed, some of which, such as StackGuard [15] and ProPolice [40], have been widely deployed in compilers and commodity OSes. On the other hand, practical countermeasures against heap-based buffer overflows are few, especially in the kernel space. To our knowledge, there are no practical mechanisms that have been widely deployed detecting kernel space heap buffer overflows. Previous methods suffer from two major limitations: (1) some of them perform detection before each buffer write operation [28,34–36,41], which inevitably introduce considerable performance overhead. This kind of inlined security enforcement can heavily delay the monitored process when the monitored operations become intense; (2) some approaches do not check heap buffer overflows until a buffer is deallocated [42,43], so that the detection occasions entirely depend on the control flow, which may allow a large time window for attackers to compromise the system. Other approaches [44,45] either depend on special hardware or require the operating system to be ported to a new architecture, which are not practical for wide deployment.

We present Kruiser, a concurrent kernel heap overflow monitor [46]. Unlike previous solutions, Kruiser utilizes the commodity hardware to achieve highly efficient monitoring with minimal changes to the existing OS kernel. Our high-level idea is consistent with the canary checking methods, which first place canaries into heap buffers and then check
their integrity. Once a canary is found to be tampered, an overflow is detected.

Different from conventional canary-based methods that are enforced by the kernel inline code, we make use of a separate process, which runs concurrently with the OS kernel to keep checking the canaries. To address the concurrency issues between the monitor process and OS kernel, we design an efficient data structure that is used to collect canary location information. Based on this data structure, we propose a novel semi-synchronized algorithm, by which the heap allocator does not need to be fully synchronized while the monitor process is able to check heap canaries continuously. The monitor process is constantly checking kernel heap buffer overflows in an infinite loop. We call this technique *kernel cruising*. Our semi-synchronized cruising algorithm is non-blocking. The kernel execution is not blocked by monitoring, and monitoring is not blocked by the kernel execution. Thus the performance and other impacts on kernel execution characteristics are very small on a multi-core architecture.

We have explored kernel heap management design properties to collect heap buffer region information at page level instead of individual buffers. A conventional approach is to maintain the collection of canary addresses of live buffers in a dynamic data structure, which requires hooking per buffer allocation and deallocation. Instead of interposing per heap buffer operation, we explore the characteristics of kernel heap management and hook the much less frequent operations that switch pages into and out of the heap page pool, which enables us to use a fix-sized static data structure to store the metadata describing all the canary locations. Compared to using a dynamic data structure, our approach avoids the overhead of data structure growth and shrink; more importantly, it reduces overhead and complexity of the synchronization between the monitor process and the canary collecting code.

To provide performance isolation and prevent the monitor process from being compromised by attackers, we take advantage of virtualization to deploy the monitor process in a trusted execution environment. Kruiser employs the Direct Memory Mapping technique, by which the monitor process can perform frequent memory introspection efficiently. On the other hand, the buffer address information is collected inside the VM to avoid costly hypervisor calls; Secure In-VM (SIM) [47] approach is adapted to protect the metadata from attackers.
1.3.2 Calling Context Encoding for Diagnosis

It is notable that during diagnosis and debugging, the calling context is frequently used to narrow down the faulty components and functions. Therefore, in order to retrieve the calling context when analyzing a logged event, we work on a new technique that provides the calling context information during logging.

A calling context is the sequence of active function/method invocations that lead to a program location. It provides critical information about dynamic program behavior. The usage has been demonstrated in a wide range of applications, such as debugging [3, 48–60], event logging and error reporting [61–63], testing [64–67], anomaly detection [68–70], performance optimization [71], and profiling [72–77]. For example, system call event logging is critical for the analysis and diagnosis of the program execution in many production systems. Simply logging the system call events fails to record how program components interact when a system call is issued, while recording calling contexts would be very informative. Take profiling as another example; context sensitive profiling is powerful as it associates data such as execution frequencies, overhead and object life time with calling contexts, and thus provides precise information for program understanding and optimization [78].

It is straightforward to obtain calling contexts through stack walking, which, however, is expensive [79, 80]. A few encoding techniques, which represent a calling context using one or more integers, have been proposed to track calling contexts continuously with low overhead. Bond and McKinley [81] proposed a technique, probabilistic calling context (PCC), that computes a probabilistically unique integer ID, essentially a hash value, for each calling context. Although PCC encoding is efficient and compact, it does not provide decoding, which is essential to applications that require inspecting and understanding contexts, such as debugging, error reporting and event logging [82].

In order to enable the decoding capability, Breadcrumbs was built on PCC [83]. It collects additional dynamic information to assist decoding. Specifically, it records encoding values at relatively cold call sites. Depending on the threshold defining the hot code, the technique either incurs large overhead or sacrifices decoding accuracy and reliability. Besides, the decoding has to be offline because it involves expensive computation (their evaluation used the limit of 5 seconds) for recovering one context.

It is straightforward to obtain calling contexts through stack walking, which, however, is expensive [79, 80]. A few encoding techniques, which represent a calling context
using one or more integers, have been proposed to track calling contexts continuously with low overhead. Bond and McKinley [81] proposed a technique, *probabilistic calling context* (PCC), that computes a probabilistically unique integer ID, essentially a hash value, for each calling context. Although PCC encoding is efficient and compact, it does not provide decoding, which is essential to applications that require inspecting and understanding contexts, such as debugging, error reporting and event logging [82].

In order to enable the decoding capability, *Breadcrumbs* was built on PCC [83]. It collects additional dynamic information to assist decoding. Specifically, it records encoding values at relatively *cold* call sites. Depending on the threshold defining the hot code, the technique either incurs large overhead or sacrifices decoding accuracy and reliability. Besides, the decoding has to be offline because it involves expensive computation (their evaluation used the limit of 5 seconds) for recovering one context.

Computing calling contexts with low overhead and precise and reliable decoding capability is challenging. Recent progress was made by Sumner et al. [82], who proposed *precise calling context encoding* (PCCE) evolved from path profiling [72]. This technique iterates every possible calling context through static analysis and represents each using a unique encoding during runtime, so the context can be recovered precisely from the encoding.

However, as pointed out in previous research [83], PCCE would not work in the presence of virtual methods and dynamic class loading. Besides, handling large-scale software is a challenge to this technique, as it lacks a scalable solution to the encoding for a large number of calling contexts. More discussion about PCCE is in Section 3.1. The challenges limit the application of the encoding technique in a large range of scenarios, as a lot of software nowadays is written in object-oriented languages with dynamically loaded components and a large number of calling contexts.

We present *DeltaPath* [84], an efficient calling context encoding technique with a precise decoding capability and the support for both procedural and object-oriented programs. Similar to PCCE, the technique leverages the Ball-Larus path profiling algorithm [72]. It obtains a unique encoding for each context at runtime by summing up the *addition values* of the call sites forming the context. Addition values are computed using our encoding algorithm based on static analysis of the target program, then an addition value and the addition operation are assigned to each call site through instrumentation. Unlike PCCE, which assumes small or medium-scale software without object-oriented programming or dynamically loaded components, DeltaPath does not have those limi-
tations. It supports both procedural and object-oriented programming, allows dynamic class loading, and works with large-scale software.

DeltaPath resolves the limitations based on the following insights and ideas. The addition value of a call site is related to the number of calling contexts ending at it, while a big program usually contains a very large number of calling contexts, such that addition values may go beyond, i.e., overflow, the encoding integer. To avoid integer overflows, it is very inefficient to represent and operate on addition values using some class (e.g., BigInteger in Java). The insight is that a long calling context can be divided into several shorter pieces, each can be encoded using an integer. The DeltaPath encoding algorithm automatically finds a small number of functions acting as such dividers for the whole program, avoiding integer overflows systematically with low overhead. In addition, a virtual function call can be dispatched to many possible functions, while PCCE computes an addition value for each target. It is infeasible to insert a bulky switch statement at each virtual function call site and execute it to choose the addition value, for virtual function call sites are ubiquitous and frequently invoked in object-oriented (OO) programs. Our encoding algorithm computes a single addition value for one call site to minimize the code cache pressure and execution slowdown. Moreover, dynamically loaded classes introduce calling contexts that are not considered during static analysis. A call path tracking technique is designed to detect unexpected calling contexts and keep the encoding correct.

We implemented DeltaPath and performed experiments with a variety of Java programs. Our evaluation results show that its performance is comparable with that of PCC [81], which is a highly efficient and the state of the art calling context encoding technique capable of working in the presence of object-oriented programming and dynamic class loading. Compared to PCC, DeltaPath introduces precise and reliable decoding.

1.3.3 Context Sensitive and Automated Defense Generation

We aim at an instant and automated defense generation technique that satisfies the following requirements. (R1) given a problem-reproducing input, it generates a defense at the user’s side, (R2) Patches do not introduce new bugs, (R3) Patches do not change program semantics. While several automated patch generation techniques have been proposed satisfying (R1) [85–87], they do not ensure (R2) and (R3).

We present a new patching technique, named Codeless Patching, and apply it to
addressing heap buffer overrun bugs satisfying R1-R3. It is supposed to work as a provisional patch until a stable and indefective one is available. Our insight is that, given a buffer overrun bug, the buffers that can be overrun, named vulnerable buffers, share the same set of calling contexts for buffer allocations, named vulnerable allocation-time calling contexts or vulnerable calling contexts for short. It motives us to apply context-sensitive defenses. Specifically, after we obtain the set of vulnerable calling contexts through replaying the overrun or other techniques, at program execution whenever a buffer allocation request is hooked, we search the current calling context in the set of vulnerable calling contexts; if there is a match, a vulnerable buffer is identified, and the buffer allocation is redirected to a predefined handling that allocates the buffer with a guard page and/or a padding attaching at its end. Such that a buffer overrun is prevented with a program termination when the overrun touches a guard page, or is safely resolved because of the sufficient padding space. The buffer enhancement incurs low overhead, since it is only applied to vulnerable buffers.

We creatively employ a calling context encoding technique [88] to speed up the calling context comparison operation; it encodes a calling context into a single integer, so that the match operation of a pair of calling contexts is transformed to an integer comparison. The calling context encoding module operates continuously during program execution to encode the current calling context into an integer and its overhead averages only 1.9% on SPEC CPU2006. The encoding value of a calling context is called its calling-context identifier, or CCID; and hence the encoding value of a vulnerable calling context is called a vulnerable CCID (VCCID).

A codeless patch comprises a tuple of integers in the form of <the bug type, a VCCID, other parameters such as the length of padding>. A patch essentially enables the enhancement on a set of buffers, so it does not introduce new bugs (R2) or change program semantics (R3).

When an input reproducing a known buffer overrun is available, the patches can be generated by system administrators (R1). We illustrate the process using the handling of Heartbleed vulnerability as an example. Our experiments illustrate that Codeless Patching prevents information leakage on Nginx service running with OpenSSL under the Heartbleed attack, and the throughput overhead is only 6.4%.

Codeless Patching can also work with existing detection techniques to defend against zero-day heap overrun attacks. While realizing that Codeless Patching is orthogonal with detection techniques, we implement HeapTherapy [89], which handles heap buffer
overflow attacks, as a demonstration of how Codeless Patching can cooperate with an existing overrun detection technique to deliver a more powerful security countermeasure: it turns a tool that detects overflow but does not prevent buffer corruption to one that prevents data corruption due to a bug once it is detected. The experiments show that HeapTherapy can defend against real-world overflow attacks and incurs only 6.2% of slowdown on SPEC CPU2006.

1.4 Summary of Contributions

We made the following contributions in improving software security towards highly efficient monitoring, automated diagnosis, and self-shielding:

- We proposed a novel concurrent program monitoring technique called software cruising, which leverages multicore architectures and utilizes non-blocking data structures and algorithms to achieve high efficiency and scalability. To the best of our knowledge, Cruiser is the first work reported in the open literature that utilizes concurrent threads to detect buffer overflows. Among existing buffer overflow detectors, this is the first work that utilizes and designs lock-free data structures to support large-scale applications. In addition, software cruising is applied to enhancing kernel security. We proposed a novel non-blocking concurrent monitoring algorithm, in which neither the monitor process nor the monitored process needs to be fully synchronized to eliminate concurrency issues such as race conditions; the monitor keeps checking live kernel memory without incurring false positives.

- We proposed a new precise calling context encoding technique that supports both procedural and OO programs. Encoding space pressure is addressed systematically. It thus allows the encoding technique to be applied to large-scale software. Dynamic class loading is handled properly using the call path tracking technique. We implemented DeltaPath and evaluated it on a variety of Java programs. The efficiency of DeltaPath is comparable with that of PCC while it provides precise and reliable decoding. The new calling context encoding technique can be used to record the calling context with minimal overhead whenever an event of interest is to be logged. The log is then used to provide rich runtime information during diagnosis by recovering the calling context.
• We proposed a concrete approach to self-shielding software. As a demonstration we implemented HeapTherapy, which handles failures caused by heap buffer overrun bugs by automated diagnosis and instant defense generation on the user side. It is a creative use of calling context encoding techniques for characterizing and identifying heap buffers being exploited by attackers.
Chapter 2  
Related Work

As both Cruiser, Kruiser, and HeapTherapy are countermeasures against buffer overflows, we first review some of the existing work on buffer overflows in Section 2.1. Next, Kruiser applies a new virtual machine monitoring scheme; thus Section 2.2 covers the related work on virtual machine introspection. Then, we summarize previous work on calling context retrieval in Section 2.3. Finally, Section 2.4 covers some related work on automated defense generation, with a focus on techniques employing calling context information, which is also used in HeapTherapy.

2.1 Buffer Overflow Countermeasures

We divide the existing countermeasures against buffer overflow attacks into the following seven categories. Given extensive research in this area, this is not intended to be exhaustive.

**Bounds checking:** Many static analysis tools fall under this category [90, 91], which detect buffer overflows by examining source code statically and automatically. This approach usually suffers from high false positive or negative rate [25]. Some dynamic approaches [34–36] change the C pointer representation to carry buffer size information with pointers to enable bounds checking, i.e. “fat pointer,” which is incompatible with legacy library code. CRED [28], built on the work of Jones and Kelly [41], does not change pointer representations but associates a buffer bound lookup with each pointer reference. However, the performance overhead is more than 2X. A recent work, baggy bounds checking [33], reduces the cost of bounds lookups by relaxing bounds checking precision, which however may lead to false negative, and it relies on specific memory allocators. Some library-based countermeasures [18, 26, 32] provide bounds checking
only for specific functions in the C standard library.

**Canary checking:** Canary was firstly proposed in StackGuard [15], which tackles stack smashing attacks by placing a canary word before the return address on stack. Attempts to overwrite the return address would corrupt the canary value first. Although there are arguments that canary-based countermeasures can be bypassed [13, 14], the wide deployment and successes of StackGuard and its derivation ProPolice [40] have manifested their effectiveness. Robertson et al. [42] first proposed to use canary to protect heap chunk metadata. A canary is placed at the beginning of each chunk, thus when a buffer on heap is overflowed, the canary of the adjacent chunk is corrupted, which, however, is not detected until the adjacent chunk is coalesced, allocated or deallocated; therefore the detection relies on program execution.

**Return address (RA) shadow stack or stack split:** StackShield [16], RAD [92] and their derivations [93,94] maintain an RA shadow stack, i.e. a copy of the RA is saved on the shadow stack at the prologue of a function call and is compared against the RA on the conventional stack at the epilogue. If the two RAs diverge, a buffer overflow is detected. In [95], the stack is split into an RA stack and a data stack, such that return addresses are protected from buffer overflows.

**Non-executable (NX) memory:** By setting the memory pages as non-executable, NX memory [17, 96] prevents code injected onto stack and heap from being executed. However it can be bypassed by a return-to-libc attack, which overwrites function pointers or return addresses with function addresses in libc, e.g. system().

**Non-accessible memory:** Both Purify [27] and Valgrind [3] insert guard zones, which are marked as inaccessible, surrounding dynamically allocated buffers, and track all memory references. When a guard zone access is detected, e.g. due to buffer overflows, an error is reported. Eletric Fence [2] places an inaccessible memory page immediately after (or before) each dynamically allocated buffer. If a buffer is overflowed (or underflowed), a segmentation fault is signaled. Tools in this category result in significant memory and performance overhead.

**Randomization and obfuscation:** Address Space Layout Randomization (ASLR) [96, 97] randomizes the locations of stack, heap and/or variable locations for each execution, such that a buffer overflow attack, such as return-to-libc, cannot be achieved reliably; that is, probabilistic protection is provided. However as it requires programs to be compiled into position-independent executables, it is incompatible with legacy code. In addition, it may be defeated by brute-force attacks or bypassed by partial overwrite attacks on
the least significant bytes of a pointer [98]. PointGuard [99] encrypts pointers stored in memory and decrypts them before loading them into registers, such that pointers corrupted by attackers will not be decrypted to intended values. This countermeasure is incompatible with legacy code and cannot protect non-pointer data. Instead of randomizing pointers, instruction set randomization [100] keeps instructions encrypted, and decrypts them only before they are fetched into processors, which, however, results in substantial performance overhead. It can be bypassed by return-to-libc attacks.

**Execution monitoring:** Program shepherding [29] monitors control flow transfer in order to enforce a security policy. Buffer overflow attacks that lead to deviant control flow transfer are prevented. Control-flow integrity [30] determines a program’s control flow graph beforehand and ensures that the control flow adheres to it. Castro et al. [31] proposed to compute a data-flow graph using static analysis and monitor whether the program data flow adheres to the graph. Like Cruiser, n-variant execution [44, 101] also takes advantage of multicore and multiprocessor architectures to enhance security. It runs a few variants of a single program simultaneously; behavioral divergences among the variants raise alarms. Execution monitoring usually imposes high performance overhead.

Despite so many countermeasures, only a few of them, such as StackGuard, ASLR, and NX memory, are widely deployed in production systems. Table 2.1 summarizes the properties of these three approaches and compares them with Cruiser. The common properties of these approaches include low performance overhead, easiness to deploy and apply, no false alarms, compatibility with mainstream platforms and program semantics loyalty. In addition to having all the advantages, Cruiser have three other important properties: compatibility with legacy code, no need for recompilation, i.e. working with

---

Table 2.1. Comparison of some widely deployed countermeasures and Cruiser.

<table>
<thead>
<tr>
<th>Property</th>
<th>StackGuard</th>
<th>ASLR</th>
<th>NX</th>
<th>Cruiser</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low performance overhead</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Easy to deploy and apply</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>No false alarms</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Mainstream platform compatible</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Program semantics loyalty</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Legacy code compatible</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>No need for recompilation</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Able to locate corrupted buffers</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

As described in Section 3.4.2, one variant of Cruiser does incur false alarms, however, at an extremely low probability (1/2^{64} in 64-bit OS) and can be safely ignored in practice.
binary executables, and *ability to precisely locate corrupted buffers*, which is critical for testing, debugging, and security monitoring.

Unlike Cruiser that hooks per heap buffer allocation and deallocation, Kruiser explores the characteristics of kernel heap management to interpose the much less frequent operations that switch pages into and out of the heap page pool, such that our system relies on a fix-sized array data structure instead of the lock-free data structures to maintain the metadata. The monitoring algorithms are thus very different. In addition, the hybrid monitoring scheme differs a lot from the user space monitoring.

### 2.2 Virtual-machine introspection

Garfinkel and Rosenblum [102] first proposed the idea of performing intrusion detection from outside of the monitored system. Since then, out-of-VM introspection has been applied to control-flow integrity checking [103, 104], malware prevention, detection, and analysis [105–114], and attack replaying [115]. They monitor static memory areas (e.g. kernel code, Interrupt Description Table), interpose specific events such as page faults, trace system behaviors, or detect violations of invariants between data structures. Considering the volatile properties of heap buffers, these approaches are infeasible for kernel heap buffer overflow monitoring; for example, it is impractical to interpose every memory write on the heap. Some approaches detected buffer overflow attacks as a side effect by detecting corrupted pointers or control flows, but cannot deal with non-pointer and non-control data manipulation on heap buffer objects. Approaches, such as kernel memory mapping and analysis, can be misled by buffer overflow attacks or perform better without heap corruption. Our approach can be complementary to them providing lightweight heap buffer overflow detection.

In contrast to out-of-VM monitoring, SIM [47] puts the monitor back into the VM and enables secure in-VM monitoring by providing discriminative memory views for the monitored system and the monitor. Our approach makes use of this technique to protect the heap metadata, while the monitor process still runs out-of-VM to achieve parallel monitoring, leveraging the multiprocessor architecture. The hybrid scheme enables a secure and efficient monitoring.

OSck [113] also performs kernel space cruising for rootkit detection. As OSck does not synchronize the running kernel and the verification process, it needs to suspend the system when an anomaly is detected to avoid false positives, while our approach does not
need to stop the world for detection. In addition, OSck does not check generic buffers allocated using kmalloc, which are common attack targets, while Kruiser checks the whole kernel heap.

2.3 Calling Context Encoding

**Stack Walking.** Stack walking is commonly used to obtain calling contexts in debugging [116] and error reporting [79, 80]. However, for applications that need continuously capture calling contexts, it usually incurs excessive overhead.

**Dynamic Calling Context Tree.** A dynamic calling context tree (CCT) [78,117] is a summary of calling contexts represented as a tree data structure. Maintaining a complete CCT incurs large space and time overhead, while a CCT obtained using sampling may miss contexts of interest. In contrast, calling context encoding approaches [81–83, 118] including our work provide concise representation of all calling contexts.

**Path Profiling.** Ball and Larus developed an algorithm to encode intraprocedural control flow paths [72]. Melski and Reps extended the work by capturing both inter- and intraprocedural control flow [74]. It encodes the whole control flow transfer history leading to a program point, but their approach does not scale, because there exist too many possible paths for nontrivial programs, and the inserted code is complex. Calling context encoding targets a succinct representation of active methods on the call stack with high efficiency.

**Precise Calling Context Encoding.** Sumner et al. proposed a precise calling context encoding technique that allows decoding [82, 118]. However, it does not work well with object-oriented programming, large-scale software, and dynamic class loading. The challenges are resolved in our work.

**Probabilistic Calling Context.** Probabilistic calling context [81] by Bond et al. provides an encoding technique that works with OO programs. The encoding result is essentially a hash value of the identifiers of the functions in the calling context. Its advantages over DeltaPath are that the encoding result is consistently only one integer and it does not need static analysis. However, PCC does not provide decoding. Their later work addresses this shortcoming by combining call graph analysis and recording of encoding results [83]. Due to the hash based encoding nature, it remains as a probabilistic approach in essence thus leaving the decoding stage inaccurate, unreliable and/or expensive. In addition, its decoding has to be offline. In production systems where precise and timely
response is needed, deterministic and instant decoding, which is supported by DeltaPath, is a highly desired property.

### 2.4 Automated Defense Generation

A few systems generate patches automatically for known attacks [85–87]. They either rely on the availability of the source code to generate a conventional patch or require binary instrumentation. Both have risks of introducing new bugs and changing program semantics for changing the code. HeapTherapy does not change the source or binary code.

The second category encompasses various systems that learn from past faults (e.g., crashes) and enhance themselves to achieve a better handling of recurring faults. Rx system adjusts the program’s execution environment after a failure occurs [119]. It lacks precise diagnosis guiding such adjustments; instead, it follows some intuition-based rules to try various adjustments until one works. The trial-and-error approach may experience many failures before a success. The reactive immune system learns from faults and checks for recurrences of previously seen faults before a real buffer overrun occurs [120]. When the incoming fault is detected, it forces the current function to return with a speculative error code, which may not be the programmer’s intention. Therefore, it also has the risk of modifying program semantics. HeapTherapy bases its defense generation on precise diagnosis information, and guarantees that it does not change program semantics.

Calling contexts have been used in failure handling in various forms. (1) Xu et al. developed an automatic approach to software failure handling [121]. Similar to HeapTherapy, it diagnoses software failures automatically with the corruption time process memory image. Both their work and HeapTherapy can be built on checkpointing/rollback based recovery, though. Their approach filters out malicious user request based on signature match. In order to reduce false positives, they associate signature checking with specific program states, e.g., calling contexts. They perform a signature checking only when the current calling context matches a recorded calling context from diagnosis. They share the same observation with us, that is, a vulnerability can be exploited only at a specific server execution state. Their work exploits calling contexts to guide signature match, while Codeless Patching uses calling contexts to direct enhancement. (2) Newsome et al. proposed a destination-based check by monitoring illegal writes to specific sensitive data, such as function pointers; they also distinguish legitimate and illegal writes based on
calling contexts [122]. They assume a buffer overflow only threatens some specific sensitive data. Their approach is less effective for heap buffer overflows due to the uncertainly of the adjacency of heap buffers. (3) Exterminator [123] is built on DieHard [124], a custom memory management scheme. It leverage the information on call stack, which requires expensive stack walking. The speed overhead averages 81.2% for allocation-intensive programs. Our scheme is independent from allocation algorithms and our speed overhead is 15.8% on average for allocation-intensive programs.

In order to detect control hijacking attacks, Wagner et al. propose to determine whether a sequence of system calls are valid based on their calling contexts [125]. Feng et al. improved their work and propose that the calling contexts of two continual system calls transit following some patterns, which can be learned from training [68]. Our context-sensitive defenses aim at locating buffers vulnerable to buffer overrun attacks.
Chapter 3  
Cruiser: A Concurrent User-space Heap Integrity Scanner

Despite extensive research over the past few decades, buffer overflow remains as one of the top software vulnerabilities. In this chapter we apply software cruising to heap safety monitoring and present a concurrent heap buffer overflow monitor Cruiser. In Section 3.1 we briefly discuss lock-free synchronization. We discuss our motivations in Section 3.2 and present the design overview in Section 3.3. In Section 3.4 we describe the design and implementation. In Section 3.5 we present the evaluation results. We discuss applications of software cruising beyond buffer overflows in Section 3.6 and conclude with Section 3.7.

3.1 Background

3.1.1 Lock-free Synchronization

A conventional approach to multithreaded programming is to use locks to synchronize access to shared resources. However, the lock-based approach causes many problems, one of which is lock contention. No matter whether the thread holding a lock is running or descheduled, other threads waiting for the lock are blocked, which limits concurrency and scalability. Another problem is priority inversion, i.e. a low priority thread holding the lock cannot get scheduled while high priority threads are waiting for the lock. Although fine-grained locking reduces lock contention, it introduces more lock overhead and increases the risk of deadlock.

In contrast to the lock-based approach, lock-free and wait-free algorithms allow high
concurrency and scalability. An algorithm is lock-free if in a finite number of execution steps, at least one of the program threads makes progress, while an algorithm is wait-free if in a finite number of steps, every thread makes progress [126]. All wait-free algorithms are lock-free but the reverse is not necessarily true. Both are non-blocking and, by definition, they are immune to deadlock and priority inversion.

Lock-free algorithms commonly rely on hardware synchronization primitives. A typical primitive is Compare-And-Swap (CAS) [127]; it takes three arguments \((\text{addr, expval, newval})\) and performs the following atomically:

\[
\begin{align*}
\text{if } (*\text{addr} != \text{expval}) & \\
\text{return false;} & \\
*\text{addr} = \text{newval;} & \\
\text{return true;}
\end{align*}
\]

Specifically, if the memory location \(\text{addr}\) does not hold the expected value \(\text{expval}\), the Boolean \(\text{false}\) is returned; otherwise the new value \(\text{newval}\) is written to it and the Boolean \(\text{true}\) is returned, atomically.

### 3.2 Motivations

#### 3.2.1 Why Concurrent Detection and Challenges

There are mainly two categories of dynamic heap-based buffer overflow detectors. One category [42] detects buffer overflows inside memory allocation functions such as malloc and free, while the other [26, 32] enforces detection inside specific libc functions such as strcpy and gets. Both execute detection code inside user threads, which inevitably affects application performance, and the performance overhead is proportionally correlated with the invoke density of related functions. In addition, because detection is enforced in specific functions, they suffer from either severe temporal limitations, i.e. buffer overflows are not detected until one of the malloc function family is called, or spatial limitations, i.e. only a few libc functions are protected. Approaches that enforce bounds checking for each buffer reference do not have such limitations; however, they usually incur high performance overhead [28, 41] or false negative rate [33].

We propose to move detection code out of user threads and execute it in a separate monitor thread, which constantly cruises over buffers on heap, such that user threads
are not delayed. Rich computational resources on modern machines, especially widely available multicore and multiprocessor architectures, enable us to run a dedicated monitor thread without competing too much resources with user threads, in other words, applications can potentially gain enhanced security with no pain.

However, synchronization is one of the major challenges. In Cruiser, a collection of heap buffer addresses needs to be maintained, so that the monitor thread surveils live buffers, and in the meanwhile avoids checking deallocated buffers, which would otherwise incur false alarms or segfaults. Therefore, the user threads and monitor thread have to be synchronized when buffers are allocated or deallocated. A conventional approach to achieving synchronization is to use locks; however, it has various limitations, such as severe performance degradation due to lock contention and low scalability, which is manifested by our first attempt.

In our first attempt, a lock-based red-black tree was used to collect buffer addresses. Inside a malloc call, the address of the newly allocated buffer is inserted into the tree with $O(\log n)$ time complexity where $n$ is the number of collected addresses. Similarly inside a free call, the address of the released buffer is removed from the tree with $O(\log n)$ complexity also. Meanwhile a monitor thread traverses the tree to check the buffers. All the buffer address insert, delete and traverse operations are synchronized using locks. Our experiments showed that user threads were significantly delayed. The problem becomes more severe as more user threads contend locks and the tree grows.

### 3.2.2 Why Lock-free and Challenges

The limitations of lock-based approach pushed us towards lock-free synchronization in order to avoid lock contention and improve scalability. However, the difficulty of designing non-blocking algorithms is well recognized, which often thwarts the application of this approach.

We escaped the problems in our second attempt by utilizing the state-of-the-art extensible lock-free hash table algorithm proposed by Shalev and Shavit [128], such that the user threads and monitor thread can operate on the hash table concurrently, and each buffer address can be inserted into or removed from the hash table in $O(1)$ time. Although good scalability is achieved, the operation time is significant compared to malloc and free calls. Specifically, the slowdown of each pair of malloc and free calls observed in our experiment is more than 5X on average. The overhead is unacceptable.
for many applications with massive dynamic memory allocation.

To address these challenges, we have designed our own lock-free data structures and non-blocking algorithms to achieve concurrent detection with low overhead and high scalability, which will be presented in Section 3.3 and 3.4.

3.3 Design Overview

In addition to custom lock-free data structures, two design choices were made. First, as presented above, removing buffer addresses inside `free` calls may significantly delay user threads. In Cruiser the `free` function marks the buffer with a tombstone flag; when the monitor thread checks the buffer and finds it no longer alive, the monitor thread removes the buffer address from the collection of heap buffer addresses, such that the concern of delaying `free`s is resolved and the data structure representing the buffer address collection can be simplified. The details are covered in Section 3.4.2. Second, instead of modifying a specific memory allocator, Cruiser is implemented as a dynamic shared library to interpose the `malloc` function family and it passes the allocation requests to the corresponding memory allocator functions, therefore Cruiser can work with any memory allocator and it can be applied to protecting binary executables without instrumenting them.

3.3.1 Buffer Structure

Our method inserts two canary words around each buffer, namely *head canary* and *tail canary*, as shown in Figure 3.1, so that whenever a buffer is overflowed (underflowed), the tail (head) canary is corrupted. The *size* field, which is the encryption (XOR) result of the buffer size and a secret key, is used to locate the tail canary given a buffer address. As buffer size information is encrypted, it is not leaked to attackers, and it is more difficult for attackers to counterfeit. The head canary is the encryption result of another secret key, the buffer size and the buffer address. If the head canary and the size field cannot be decrypted to consistent size values, a buffer overflow is detected. As the buffer address is used to generate the canary, each buffer has a unique head canary, thus even if the canary of a buffer is leaked, it is difficult for attackers to forge the canary of another buffer without knowing the buffer sizes and addresses. The tail canary is encrypted and verified the same way using a different secret key. All the keys are initialized as random...
numbers when the monitored program is started.

| Size | Head canary | User buffer | Tail canary |

**Figure 3.1.** Buffer structure.

### 3.3.2 Cruiser

Our previous attempts maintain a collection of buffer addresses but lead to high overhead. To efficiently collect memory allocation information, we design the cruiser information collection (CIC) architecture which is composed of (1) a lock-free express data structure onto which user threads put information, (2) a lock-free warehouse data structure that supports multiple threads to concurrently insert, delete and access information, and (3) a non-blocking deliver thread to copy the information from the express to the warehouse data structure. Instead of inserting information into the warehouse directly, user threads put the information onto the express data structure highly efficiently, and the deliver thread takes care of the rest of the information collection work, thus performance impact on user threads is minimized. In addition, as the warehouse structure supports concurrent operations, CIC scales well.

**Figure 3.2.** Cruiser architecture.

Based on CIC, we present a dynamic heap-based buffer overflow detector—Cruiser, which uses CIC to collect memory allocation information, e.g. buffer addresses and sizes. As shown in Figure 3.2, the malloc calls are hooked to place the buffer allocation information onto the express data structure and return promptly; the deliver thread then finishes the information collection.
From the perspective of Cruiser, the life cycle of a dynamically allocated buffer can be divided into three phases: **Pre-checking**: Inside a malloc call, a buffer that is three words larger than what the user thread requests is allocated. The buffer is filled as specified in Section 3.3.1, and the buffer allocation information used for overflow detection, such as the buffer address, is put onto the express data structure. Then the malloc call returns the address of the *user buffer* (see Figure 3.1). **Checking**: The pre-checking phase ends when the deliver thread moves the address from the express to the warehouse data structure, which is traversed by the monitor thread to detect buffer overflows. **Post-checking**: Inside the free call, the buffer is marked with a tombstone flag by encrypting the head canary once again using another key; later when the monitor thread checks the buffer and finds it no longer alive, the dated buffer information is removed from the warehouse by the monitor thread, so it is the monitor thread rather than user threads that tides up dated metadata information.

### 3.4 Design and Implementation

This section describes the design and implementation of Cruiser. Section 3.4.1 describes the data structures in CIC and how CIC is used in Cruiser to maintain buffer information. Section 3.4.2 presents the algorithms to release buffers and delete dated metadata information. We elaborate special issues on extensions and optimizations in Section 3.4.3.

#### 3.4.1 Collection of Buffer Information

**3.4.1.1 Express data structure**

We implement the express data structure based on the single-producer single-consumer FIFO wait-free ring buffer proposed by Lamport [129]. Lamport’s algorithm allows a producer thread and a consumer thread to operate concurrently on a ring. The synchronization overhead between the producer and the consumer is low, as two threads are synchronized via read/write instructions on the two control variables *head* and *tail*. Because of its high efficiency, the data structure has been applied to Gigabit network packet processing systems [130, 131].

To avoid the failure of Enqueue operation when the ring is full, we extend the basic ring to a linked list of rings, called *CruiserRing*, as shown in Listing 3.1. Whenever the ring is full and a new element is produced, instead of returning failure in Enqueue
as in Lamport’s algorithm, the producer creates a new ring with doubled capacity and
links it after the full ring; the producer proceeds to insert elements into the new ring.
Accordingly, in Dequeue when the ring is consumed up and another ring is linked after it,
the consumer destroys the empty ring and proceeds to work on the next one. Because
the ring size grows exponentially, as long as the speed of the consumer matches that of
the producer, CruiserRing will converge to a stable state quickly. (The speed mismatch
problem is addressed in Section 3.4.3.) Unless the new ring creation fails, CruiserRing
ensures the success of the producer, which implies that the producer always moves on
without dropping data.

As each CruiserRing supports one producer thread, a CruiserRing is needed for each
producer thread. The method AddCruiserRing (see Listing 3.1) shows how to construct a
list of CruiserRings in a lock-free manner, such that a single consumer thread can walk
along the list to access all CruiserRings.

### Listing 3.1. CruiserRing (Express data structure).

```c
struct Ring {
  Element *buffer;
  unsigned int size;
  unsigned int head, tail;
  Ring *next; // next Ring
};

struct CruiserRing {
  Ring *pr, *cr; // producer ring and consumer ring
  CruiserRing *next; // next CruiserRing
};

CruiserRing *Head; // head of CruiserRing list

NEXT(index, size) { return (index + 1) % size; }

Enqueue(pr, data) {
  if (NEXT(pr->head, pr->size) == pr->tail) {
    newRing = createRing(2 * (pr->size));
    if (null == newRing)
      return failure;
```
pr->next = newRing;
pr = newRing;
}
pr->buffer[pr->head] = data;
pr->head = NEXT(pr->head, pr->size);
return success;
}

Dequeue(cr, data) {
if (cr->head == cr->tail) {
if (null == cr->next)
    return failure;
temp = cr; cr = cr->next;
destroy(temp);
    return Dequeue(cr, data);
}
data = cr->buffer[cr->tail];
cr->tail = NEXT(cr->tail, cr->size);
return success;
}

AddCruiserRing(cruiserRing) {
do {
cruiserRing->next = oldValue = Head;
} while (!CAS(&Head, oldValue, cruiserRing));
}

3.4.1.2 Warehouse data structure

We implement the warehouse data structure as a custom lock-free list, called CruiserList. CruiserList is a linked list of segments, each of which is a linked list itself with a never-removed dummy node as the segment head, as shown in Figure 3.3 and Listing 3.2.

Listing 3.2. CruiserList (Warehouse data structure).
Node *head; // head of CruiserList
Insert(dummy, node) {

51    node->next = dummy->next;
52    dummy->next = node;
53 }
54
55 Traverse() {
56    Node *prev, *cur, *next;
57    cur = leftBoundary->next;
58    if (cur == null)
59        return;
60
61    /\Process the first genuine node*/
62    if (!IsMarkedDelete(cur))
63        if (CheckNode(cur) returns PLEASE_DELETE_ME)
64            /\Node removal is deferred to avoid contention*/
65            MarkDelete(cur);
66
67    /\ Process the rest genuine nodes */
68    prev = cur; cur = cur->next;
69    while (cur != rightBoundary) {
70        next = cur->next;
71        if (IsMarkedDelete(cur) ||
72            CheckNode(cur) returns PLEASE_DELETE_ME) {
73            prev->next = next;
74            DeleteNode(cur);
75        }
76        else
77            prev = cur;
78            cur = next;
79    }
80 }

Figure 3.3. CruiserList.
AddSegment() {
    Node *newDummy = AllocateDummyNode();
    do {
        newDummy->next = oldValue = head;
    } while (!CAS(&head, oldValue, newDummy));
    return newDummy;
}

The basic form of CruiserList contains one segment, which supports a single insert thread to insert nodes and a single traverse thread to traverse the list concurrently. The method CheckNode is invoked in Traverse to check each node and returns whether the node should be deleted.

New nodes are always inserted between the dummy node and the first genuine node (the node linked immediately after the dummy node). If the first genuine node is determined to be deleted (Line 63), it should not be removed directly, as it may otherwise lead to list corruption or node loss. Specifically, if the first genuine node is removed between the execution of Line 51 and Line 52 when a new node is being inserted, the list is corrupted, because the newly inserted node has been linked to a deleted node. Another situation is when the first genuine node $M$ is determined to be deleted, a new node $N$ is inserted, which is not known by the traverse thread. Consequently, by removing node $M$, the dummy node is linked to the node after node $M$, such that node $N$ is lost. The contention between node insertion and deletion is a common problem in lock-free data structures.

A conventional method to resolve the contention problems in non-blocking algorithms is to use CAS in a loop to insert or delete a node, as in the CruiserRing method AddCruiserRing (see Listing 3.1). However, CAS is relatively expensive and due to contention concurrent operations may experience frequent failure and retry of CAS instructions, which delays the progress of concurrent threads.

In CruiserList, we essentially eliminate the contention and thus CAS is not needed, as shown in the method Traverse. In our algorithm, the first genuine node is never removed until new nodes have been inserted, thus new nodes can always be inserted between the dummy node and the first genuine node safely by the insert thread, while the traverse thread never touches the link between the dummy node and the first genuine node. Therefore, the node insertion and deletion operations essentially play in different
arenas, and thus have no contention.

Specifically, when the first genuine node is determined to be deleted, it is marked as to-be-deleted by calling the method MarkDelete, which fills a special null value in the data field of the node, or sets the least significant bit (LSB) of the next pointer of the node, as the node address is usually word-aligned and the LSB of the next pointer is thus not used. Then the marked node is removed in a future round of traversal when it is no longer the first genuine node. Figure 3.4 shows the process of removing the first genuine node A. It is first marked, but not deleted. After a new node C is inserted, it will be deleted shortly. It is possible that no more new nodes are inserted and the marked node sticks in the list; however, there is only one such node in a segment and normally this occurs only in the residual period of program execution. Note that the user buffer is freed; only the first metadata node may stay alive after the corresponding buffer is released. We can resolve this using CAS if it becomes a serious issue. The Insert method inserts a node, just as in a single-threaded list, between the never-removed dummy node and the first genuine node which is never removed directly.

Figure 3.4. Deletion of the first genuine node.

The technique of marking a node as to-be-deleted was first used in the lock-free FIFO queue algorithm [132] proposed by Prakash et al., then used in Harris’s [133] and Michael’s [134] lock-free lists, respectively. All of them use this technique to prevent new nodes from being linked to a marked node. As insertion and deletion may operate on the same node, contention still exists; and they rely on CAS. Our algorithm allows new nodes to be linked to a marked node. Only simple reads and writes are needed.
The basic CruiserList can be easily extended to multiple segments using the method AddSegment, so that it can support multiple insert and traverse threads. Each insert thread has a thread-private variable pointing to the dummy node of a different segment, into which this thread inserts nodes. On the other hand, the segments can be partitioned into multiple disjoint groups; each traverse thread walks on a different group denoted by two thread-private variables leftBoundary and rightBoundary, which point to the dummy nodes of segments. If there is only one traverse thread, its segment group consists of the whole CruiserList. For the sake of simplicity, the Traverse method in Listing 3.2 is only for one-segment groups. The time for a traverse thread to cruise through its segment group once is called a cruise cycle.

Compared to general-purpose lock-free lists, CruiserList is highly efficient and has the following advantages: (1) Wait-free access and zero-contention: Both insert and traverse threads keep making progress, and node insertion and deletion are executed in different arenas; (2) No ABA problem: The ABA problem [127] is historically associated with CAS. It happens if in a thread a shared location with a value A was read, then CAS comparing the current value of the shared location against A succeeds though it should not, as between the read and the CAS other threads change the value of the shared location from A to B and back to A again. The only CAS in Line 86 has no ABA problem, because it is impossible for head, which was changed from the address of the dummy node A to that of the dummy node B, to change back to A without removing any dummy nodes; and (3) No special memory reclamation needed: For a typical lock-free data structure, when a node is removed by a thread, its memory cannot be released immediately because other threads may be accessing it. So special memory reclamation mechanisms are needed, such as reference counters and hazard pointers [135]. On a given segment of the CruiserList, the traverse thread is the only thread that deletes nodes, so it is not concerned with accessing nodes being released by other threads. It is not a problem for the insert thread either, as it only accesses the content of the dummy node, which is never removed.

3.4.1.3 Applying CIC in Cruiser

Cruiser uses the CIC mechanism to collect memory allocation information (see Figure 3.2). The malloc calls of user threads are hooked and memory allocation information is put on the CruiserRings. The deliver thread moves the information from CruiserRings to CruiserList, while the monitor thread calling Traverse of CruiserList cruises over buffers to detect overflow according to the collected information in CruiserList. The
buffer overflow detection code is executed in CheckNode (see Listing 3.2), which returns PLEASE_DELETE_ME if the buffer is found no longer alive. More details about CheckNode are described in Section 3.4.2.

Cruiser is implemented as a dynamic shared library to interpose the malloc function family. It contains a constructor (initialization) function, which gets executed when the monitored program is started. Inside the constructor function, the keys are initialized with random numbers in /dev/urandom, the CruiserList is created and initialized, and the deliver thread and the monitor thread are created, which share the same address space as user threads. Each user thread creates its CruiserRing when it invokes its first malloc call. All the memory blocks used by Cruiser (the data structures and keys) are allocated using mmap with two inaccessible guard pages [2] surrounding each of them, such that they cannot be overflowed.

### 3.4.2 Buffer Release and Node Removal

Once a buffer is freed, the node in the CruiserList containing the corresponding buffer information becomes dated and should be removed; otherwise, buffer overflow checks over the buffer may incur false alarms or segmentation faults, as the buffer memory may have been reused or unmapped. Removing dated nodes inside free calls may significantly delay user threads. We address the problem with the following two approaches; both enable the monitor thread to tidy-up CruiserList.

The first approach is a lazy two-step memory reclamation algorithm. First, when a free call is intercepted, the target buffer is marked with a tombstone flag by encrypting the head canary with another key, called the *release key*; the free call returns without releasing the buffer, which becomes a *zombie buffer*. Second, when the monitor thread checks the buffer and finds it marked with a tombstone, the method CheckNode in Listing 3.2 releases the buffer and returns PLEASE_DELETE_ME. This approach removes dated information effectively without incurring false alarms or segfaults. The drawback is that buffer release is delayed; however, since all zombie buffers are bound to be released no later than the next cruise cycle, the delay should be reasonably short.

The second approach does not delay memory reclamation; however, it requires some changes about the head and tail canaries and needs the assistance of recovery techniques. The head canary field is filled with a random number (rather than the encrypted size value), while the tail canary is the encryption (XOR) result of the head canary value and a
The random number along with the buffer address is collected in CruiserList. Inside the free call, the target buffer is checked against buffer overflow by decrypting the tail canary and comparing with the head canary.\(^1\) If they are different, a buffer overflow is detected; otherwise the head canary is set as zero, after which the buffer is released and the free call returns. When the monitor thread checks a buffer and finds the number stored in the head canary is not the same as that stored in the CruiserList, it assumes the buffer has been released and then the corresponding dated node is removed from the CruiserList. After a buffer is released and reused, the memory location of the original head canary may happen to be written with the same value as that before the buffer was released, so that when the monitor thread checks the node using the dated buffer information, it would incorrectly determine this buffer is still alive and thus false alarms are possible; however, the probability is extremely low (in 64-bit OS, it is \(1/2^{64}\)), which can be safely ignored in practice.

Three scenarios need to be considered for the second approach. First, when a buffer is tampered due to an overflow occurred in its preceding adjacent buffer, it would be incorrectly determined as a released buffer by the monitor thread; however, this does no harm and as the adjacent buffer is under surveillance, the overflow can still be detected. Second, when a buffer is underflowed, the monitor thread would also treat it as a released buffer. Underflows are rare compared to overflows; moreover we can address the problem by saving the buffer size in the CruiserList node, against which the size field of each buffer is checked by the monitor thread in CheckNode to detect underflows.

Third, when the monitor thread checks a buffer that has been released and the corresponding memory page(s) has been unmapped, a segfault is triggered. The problem can be addressed using some recovery techniques [136]. In Linux, a SIGSEGV signal handler can be installed firstly. Each time before the monitor thread accesses a buffer, it calls sigsetjmp to save the calling environment. Once a SIGSEGV signal is triggered due to an invalid access, the monitor thread is trapped to the SIGSEGV handler, and the calling environment can be recovered by calling siglongjmp. Windows also has similar recovery mechanism called Structured Exception Handling [137].

Cruisers with the two approaches are called *Lazy Cruiser* and *Eager Cruiser*, respectively. We have implemented both in Linux.

\(^1\)If the size field has been corrupted, the tail canary cannot be located correctly. As a result, the read of the tail canary may incur segfault, which, however, essentially exposes buffer overflows. If necessary, the same recovery technique described here can be used to deal with segfault.
3.4.3 Extensions and Optimizations

3.4.3.1 Extensions

Flexible deployment options: The deployment of Cruiser is flexible. One method is to implement Cruiser as a dynamic shared library. By setting the LD_PRELOAD environment variable to the path of the Cruiser library, an administrator can selectively enable Cruiser for certain applications. With this method the malloc function family are interposed by Cruiser, which invokes the memory allocation functions in the system library to enforce dynamic memory allocation, thus no system library is altered. This is the deploying method we adopt in the experiments.

A second method is to integrate Cruiser with the system dynamic library for dynamic memory allocation. The advantage is that memory and performance overhead can be reduced. For example, the overhead due to malloc function family interposition can be avoided; some memory allocators, such as dlmalloc [19], place the buffer size information in the beginning of each chunk, so Cruiser does not need to maintain that information additionally.

A third method is to implement Cruiser as a static library and integrate it into the compiler. Considering the two methods above cannot be applied to statically linked applications, the third method is complementary to them. Regardless of the deploying method, Cruiser has no effect on applications that perform their own memory management, neither can it detect buffer overflows inside a structure currently, which is a limitation shared by other techniques that detect buffer overflows at the level of memory blocks [2, 3, 26, 27, 32, 33, 42]. However, Cruiser can be extended to monitor buffers inside a struct by inserting canary words.

More Cruiser threads: Although our experiments show that the Cruiser configuration with one deliver thread and one monitor thread is sufficient for common applications, it may be desirable to extend Cruiser with multiple deliver and monitor threads. For example, there may be many user threads requesting dynamic memory intensively that a single deliver thread cannot match the speed of buffer allocation; or the CruiserList is so long that it takes much time for a monitor thread to traverse through the CruiserList once. As both CruiserList and the list of CruiserRings support multiple threads, Cruiser can be easily extended with more cruiser threads to protect various applications.
3.4.3.2 Optimizations

**Memory reuse:** To mitigate memory allocation intensity and speed up node insert and delete in CruiserList, a ring buffer is adopted to store the addresses of removed nodes with the monitor thread as the producer and the deliver thread as the consumer. Nodes removed from CruiserList are not deleted but stored in the ring unless it is full. Accordingly when the deliver thread needs a node, it first tries to retrieve a node from the ring; only when the ring is empty, a new node is allocated. The simple ring buffer can be replaced with other advanced wait-free queues, such as a CruiserRing, to support more efficient nodes buffering strategy.

On the other hand, each user thread requesting dynamic memory owns a CruiserRing. Considering some applications fork and kill threads frequently, instead of allocating and releasing CruiserRings intensively, we reuse CruiserRings. A Boolean flag indicating whether the CruiserRing is available for reuse is added into the CruiserRing structure. As the deliver thread traverses along the list of CruiserRings, if it detects a user thread has exited, it marks the related CruiserRing as available for reuse. When a user thread needs a CruiserRing, it will try to reuse an available CruiserRing before allocating a new one.

**Backoff strategies:** Although our experiments show that Cruiser imposes low overhead, the performance can be further improved with backoff strategies, for example, by inserting NOP instructions or sleep calls inside the monitor thread. Reduced monitor intensity leads to less memory access interference, thus decreases performance impact. Cruiser can also switch to monitor buffers selectively, for example, buffers involved in data flows stemming from networks or user inputs. For Lazy Cruiser, it is a good choice to ignore large buffers and hence release them inside free calls directly under intense memory pressure. More advanced monitor strategies based on computational resource dynamics can be adopted as well.

**Variants of the deliver thread:** In Cruiser, the deliver thread is busy polling CruiserRings. Actually it can go to sleep when there is no information to deliver and be waken up by user threads via signals. To avoid sending a signal per malloc call, a global status flag indicating whether the deliver thread is asleep can be used. The flag is set as awake or asleep by the deliver thread; a wake-up signal is sent to the deliver thread only when it is asleep.

Another variant is to combine the deliver thread and the monitor thread; we can have the hybrid thread delivering information and monitoring nodes alternatively, such that only one busy thread is needed and the data structures can be further simplified. The
<table>
<thead>
<tr>
<th>Program</th>
<th>Vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>wu-ftpd 2.6.1</td>
<td>Free calls on uninitialized pointers</td>
</tr>
<tr>
<td>Sudo 1.6.4</td>
<td>Heap-based buffer overflow</td>
</tr>
<tr>
<td>CVS 1.11.4</td>
<td>Duplicate free calls</td>
</tr>
<tr>
<td>LibHX 3.5</td>
<td>Heap-based buffer overflow</td>
</tr>
<tr>
<td>Lynx 2.8.8 dev.1</td>
<td>Heap-based buffer overflow</td>
</tr>
<tr>
<td>Firefox 3.0.1</td>
<td>Heap-based buffer overflow</td>
</tr>
</tbody>
</table>

Table 3.1. The effectiveness experiment against real-world vulnerabilities.

drawback is that the monitoring may be interrupted frequently.

**Better ring algorithms:** Based on Lamport’s ring [129], some other ring algorithms have been proposed [130, 131]. We used the ring algorithm proposed by Lee et al. [131] in our experiments to mitigate the false sharing problem in Lamport’s ring.

### 3.5 Evaluation

We evaluated the effectiveness of Cruiser, its performance and memory overhead, and analyzed the detection latency issue. This section presents our results.

#### 3.5.1 Effectiveness

We evaluated the effectiveness of Cruiser with two experiments. Our first experiment was carried out using the SAMATE Reference Dataset (SRD) [138] maintained by NIST. The dataset contains 12 test cases on heap-based buffer overflows due to contiguous writes, which are caused by assignments, memcpy, strcpy, snprintf, etc., and another 10 test cases that fix the overflows. Cruiser detects all the overflows in the 12 test cases and there is no false positive for the 10 sound test cases.

The second experiment tested the effectiveness against both well-known historic exploits (wu-ftpd [139], Sudo [140], CVS [141]) and recently published vulnerabilities (LibHX [142], Lynx [143], Firefox [144]), shown in Table 3.1. Each vulnerable program was run with Cruiser and attacks were launched on them. Each attack was executed 50 times and Cruiser detected all the overflows, duplicate and invalid frees.

These experiments demonstrate that our technique is effective in detecting not only heap-based buffer overflows but also memory leakage and heap corruption including duplicate frees and frees on invalid pointers. Therefore, Cruiser is a good candidate
for finding heap corruption and memory leakage defects during development as well as monitoring production systems.

### 3.5.2 Performance and Memory Overhead

We evaluated the performance and memory overhead of Cruiser using the SPEC CPU2006 Integer benchmark suite. The experiments were performed on a Dell Precision Workstation T5500 with two 2.26GHz Intel Xeon E5507 quad-core processors and 4GB of RAM running 32-bit Linux 2.6.24. Cruiser was implemented as a dynamically linked library and loaded by setting the LD_PRELOAD environment variable. In all the experiments, Cruiser created one deliver thread and one monitor thread with the optimizations memory reuse and better ring algorithms enabled. We ran each experiment three times and present the average result. The variance was negligible.

We evaluated both Lazy Cruiser and Eager Cruiser, and compared with DieHarder [145], which also provides probabilistic heap safety. Figure 3.5 shows the execution time of the two implementations normalized by the execution time of original programs. The average performance overhead is 12.5% for Lazy Cruiser and 5% for Eager Cruiser, while DieHarder imposes 20% penalty on average. For the majority of the benchmark programs, the overhead imposed by Cruiser is negligible. The perlbench has the highest overhead due to its significantly dense dynamic memory allocation. Eager Cruiser performs generally better than the lazy version, mainly because the former allows immediate memory reclamation and reuse. The experiments show that Cruiser can be deployed in field practically.

In addition to the three-word tag associated with each buffer, the memory overhead of Cruiser is mainly due to its data structures CruiserRings and CruiserList. For Lazy Cruiser, zombie buffers are another source. To precisely analyze the memory overhead, as shown in Table 3.2, we measured the maximum size of CruiserRing and the maximum and average lengths of CruiserList normalized by the live buffer counts at sample time (we sampled at the end of each cruise cycle), respectively, from which we can get the percentage of dated nodes and zombie buffers. As the CruiserList length is normalized, the maximum length can be less than the average one, as in the case of xalancbmk. The initial CruiserRing has 1024 elements; the size of each element is one word in Lazy Cruiser and two words in Eager Cruiser, respectively. For the majority of benchmarks, the CruiserRing does not grow, while the maximum CruiserRing in perlbench test with
Figure 3.5. Execution time with Cruiser (normalized by the execution time without Cruiser), compared with DieHarder. The last set of bars is the geometric mean of the execution time of all benchmarks.

Lazy Cruiser experienced 3 times of growth (recall that the ring grows exponentially). The length of CruiserList is close to the count of live buffers on heap. In other words, the percentage of dated nodes and zombie buffers is low, and on average it is negligible.

3.5.3 Scalability

To evaluate the scalability of our approach on multithreaded programs, we compared the throughputs of the Apache web server with and without Cruiser. We ran Apache 2.2.8 on the same workstation specified in Section 3.5.2, and used ApacheBench 2.3, which ran on a machine with a 2.4GHz Intel Core 2 Duo processor, 4GB of RAM, and Mac OS X 10.6.4, to measure the Apache throughput over a Gbit LAN network. We issued repeated requests for a 5KB HTML page with various numbers of concurrent clients. We observed that Apache allocated two heap buffers per request; ApacheBench issued one million requests for each concurrency number. Figure 3.6 shows the throughputs of Apache (labeled as baseline) and Apache with Lazy Cruiser and Eager Cruiser, respectively. The throughputs of Apache with the two versions of Cruiser are almost the same. The maximum 3% slowdown appears around concurrency number 7, while the average slowdown is negligible. For concurrency numbers greater than 11, the throughputs with
Table 3.2. Memory overhead and cruise cycle (Results of Eager Cruiser are enclosed in parentheses; the maximum/average CruiserList lengths are normalized by the maximum/average heap buffer counts, respectively).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Maximum CruiserRing size</th>
<th>Maximum CruiserList length</th>
<th>Average CruiserList length</th>
<th>Average cruise cycle (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>perlbench</td>
<td>8192 (1024)</td>
<td>1.06 (1.16)</td>
<td>1.02 (1.06)</td>
<td>4.3e4 (1.2e5)</td>
</tr>
<tr>
<td>bzip2</td>
<td>1024 (1024)</td>
<td>1.00 (1.00)</td>
<td>1.00 (1.00)</td>
<td>.43 (1.2)</td>
</tr>
<tr>
<td>gcc</td>
<td>2048 (2048)</td>
<td>1.02 (1.05)</td>
<td>1.00 (1.01)</td>
<td>1.3e3 (3.5e3)</td>
</tr>
<tr>
<td>mcf</td>
<td>1024 (1024)</td>
<td>1.00 (1.00)</td>
<td>1.00 (1.00)</td>
<td>.35 (.59)</td>
</tr>
<tr>
<td>gobmk</td>
<td>1024 (1024)</td>
<td>1.00 (1.00)</td>
<td>1.00 (1.00)</td>
<td>.37 (1.6)</td>
</tr>
<tr>
<td>hmmmer</td>
<td>1024 (1024)</td>
<td>1.11 (1.29)</td>
<td>1.00 (1.00)</td>
<td>.26 (1.6e2)</td>
</tr>
<tr>
<td>sjeng</td>
<td>1024 (1024)</td>
<td>1.11 (1.00)</td>
<td>1.00 (1.00)</td>
<td>.36 (0.82)</td>
</tr>
<tr>
<td>libquantum</td>
<td>1024 (1024)</td>
<td>1.00 (1.00)</td>
<td>1.00 (1.00)</td>
<td>.16 (0.49)</td>
</tr>
<tr>
<td>h264ref</td>
<td>1024 (1024)</td>
<td>1.00 (1.00)</td>
<td>1.00 (1.00)</td>
<td>3.3e2 (1.7e3)</td>
</tr>
<tr>
<td>omnetpp</td>
<td>2048 (1024)</td>
<td>1.08 (1.08)</td>
<td>1.02 (1.08)</td>
<td>9.8e4 (3.2e5)</td>
</tr>
<tr>
<td>astar</td>
<td>4096 (2048)</td>
<td>1.04 (1.06)</td>
<td>1.00 (1.00)</td>
<td>19 (88)</td>
</tr>
<tr>
<td>xalancbmk</td>
<td>2048 (1024)</td>
<td>1.00 (1.00)</td>
<td>1.02 (1.07)</td>
<td>7.3e4 (1.5e5)</td>
</tr>
</tbody>
</table>

and without Cruiser are almost identical. We measured until concurrency number 110 when the client’s CPU was saturated, and the throughput slowdown remained negligible. The machine running Apache has 8 cores in processors, thus the Cruiser threads compete processor time since concurrency number 6; however as the working threads in Apache increase, the percentage of processor time used by the Cruiser threads decreases, and thus the slowdown declines.

### 3.5.4 Detection Latency

Heap-based buffer overflows can be divided into two classes [42]. One class of attacks alter memory allocators’ metadata. Exploits in this class are often achieved by releasing a corrupted buffer [20]. Cruiser defeats this kind of exploits completely, as all buffers are checked before release.

The other class comprises attacks that overflow a buffer to alter the content of its adjacent memory block. This kind of exploits are not achieved until the corrupted content is used; the attacks are not detected until corrupted buffers are checked by a monitor thread. Like DieHarder, Cruiser provides probabilistic heap safety for this kind of attacks. Assume an attack takes time $E$ to achieve the exploit after canary smashing and a cruise cycle takes time $C$, if $E \geq C$, Cruiser can detect the overflow before the exploit is achieved;
Figure 3.6. Throughputs of the Apache web server for varying numbers of concurrent requests.

otherwise, assume the detection latency (elapsed time since the overflow until detection) is uniformly distributed on the interval \((0, C]\), the probability \(P\) for Cruiser to detect an exploit before it completes is \(E/C\). As shown in Table 3.2, with Lazy Cruiser, 5 of the 12 benchmarks’ average cruise cycles are shorter than 0.5\(\mu s\), and 8 of them are not longer than 0.5ms; with Eager Cruiser, 5 benchmarks’ average cruise cycles are not longer than 1.6\(\mu s\), and 7 of them are shorter than 0.2ms. The average cruise cycles for Apache test are 16\(\mu s\) and 78\(\mu s\) for Lazy and Eager Cruisers, respectively. Considering \(C = NT\), where \(N\) is the number of nodes in the CruiserList and \(T\) is the average time for a monitor thread to check a node, we can expect a high prevention probability by keeping \(N\) small. One way is to divide the CruiserList into several segment groups and create the same number of monitor threads, each of which cruises over a shorter part of CruiserList. For example, by dividing the CruiserList into two parts and running one more monitor thread on either part in omnetpp test, the average cruise cycle decreased by 42% and 47% for Lazy Cruiser and Eager Cruiser, respectively. Another way is to only monitor suspicious buffers, for example, those buffers involved in data flows that stem from networks or user inputs.

Considering Cruiser shares the same address space as user programs, an attacker with arbitrary memory access privileges of the compromised program can bypass Cruiser theoretically. Otherwise a reliable and precise attack against Cruiser is hard to build. In Cruiser, the canaries of a buffer are the XOR result of the buffer’s address, size and the
keys which were initialized using random numbers, so it is difficult to predict or restore a canary. For example, an elaborate attack that exploits other vulnerabilities, such as format string [14], to obtain the keys still needs the target buffer size and address information to calculate its canaries. Blind access for Cruiser’s data structures will normally incur segfault, as they are surrounded by inaccessible guard pages. The function pthread_kill can be interposed by Cruiser to prevent Cruiser threads from being killed, and Cruiser threads can detect the liveness of each other when running, thus it is very unlikely to subvert Cruiser. Detection can be evaded by terminating the process, which, however, explicitly exposes the attack and is not a usual way of real attacks.

3.6 Software Cruising

We have applied the software cruising technique to efficient heap overflow detection by moving the detection work out of user threads. Potentially all the inlined verification, monitoring and resource reclamation work can be migrated from user threads to one or more monitor threads for concurrent execution in background. This section discusses other applications of software cruising, some of which we are currently working on as an extension of this work.

**Background Software Monitoring:** Depending on the security policies enforced, inlined security enforcement may incur high performance overhead. Software cruising takes a very different way by moving inlined security enforcement out of user threads and executing them in concurrent monitor threads, which can reduce considerable performance overhead. Although synchronization and race conditions between user and monitor threads are potential challenges, they can be solved using lock-free data structures as in Cruiser. For example, we can implement a call trace collector by instrumenting the call instruction in the binary and placing the target address in a CruiserRing. A concurrent monitor thread analyzes the call trace to evaluate whether specific control-flow policies are followed. As simple examples, some control transfers are suspicious; or a control-flow policy may require that a certain function is called no more often than another function (such restrictions may be desirable to prevent some “confused deputy” attacks [146]). The monitor thread can detect abnormalities based on the call trace and a finite automaton or simple counting. Other straightforward applications include integrity-checking of important program structures such as the Global Offset Table.

**OS Kernel Cruising:** It is desirable to adopt software cruising to monitor OS kernel
memory integrity and other safety and liveness properties. We plan to develop a prototype that can monitor integrity of OS kernel memory. One of the challenges of cruising OS kernels is how to minimize the impact on the kernel memory layout since kernel code contains many low-level programming idioms that rely on certain memory layouts. One way to solve this problem is to selectively monitor some buffers and make manual transformation.

**Concurrent Resource Reclamation**: Software Cruising can also be applied to implement efficient resource reclamation, for example safe memory reclamation for lock-free data structures. For lock-free dynamic objects, when a thread removes a node, it is possible that some other thread has earlier read a reference to that node, and is about to access its contents, therefore the memory occupied by the node should not be released or reused directly. When designing lock-free data structures, safe memory reclamation is a major concern. Recent progress was made by Michael [135]. The core idea is to associate a number of pointers, called *hazard pointers*, with each thread. A hazard pointer points to a node that may be accessed later by that thread; whenever a thread frees a retired node, it has to scan hazard pointers of other threads to make sure the node is not pointed to by any of the hazard pointers. To achieve a low amortized overhead, a thread does not free retired nodes until it accumulates a certain number of retired nodes. The inlined batch processing of retired nodes inevitably delays user threads and memory reclamation. The problem can be solved elegantly by deploying a concurrent thread, which takes over the work of memory reclamation by scanning hazard pointers to determine which retired nodes can be released safely. The original methodology is complementary to this solution in case too many retired nodes are accumulated.

### 3.7 Summary

We have applied software cruising to Cruiser, a dynamic heap-based buffer overflow detector on the Linux platform. It is straightforward to adapt Cruiser to other platforms such as Windows and Mac. We have evaluated Cruiser on a variety of programs to show its effectiveness. The performance overhead of monitoring SPEC CPU2006 benchmark is about 5% on average, and negligible in the majority of cases. Cruiser also scales well on multithreaded programs; the slowdown on the Apache throughput with different numbers of concurrency is negligible on average and 3% maximal. The experiments show that Cruiser is feasible to be applied.
Software cruising can also be applied to enhancing kernel memory safety. Kernel-space buffer overflow vulnerabilities are more dangerous than the user-space buffer ones, in that once such a vulnerability is exploited, attackers may have the full control of the whole system by, e.g., installing rootkits. We first describe the challenges that we encountered in Section 4.1. The solutions and an overview of our system are presented in Section 4.2. We then present the kernel cruising algorithm in Section 4.3 and the design and implementation in Section 4.4. We evaluate kruiser in 4.5 and discuss the deployment issue in Section 4.6.

4.1 Challenges

In this section, we present the challenges we have encountered during the design and implementation of this work. Their solutions are presented in the next section.

C1. Synchronization. Since the monitor process checks heap memory which is shared and modified by other processes, synchronization is vital to ensure the monitor process locate and check live buffers reliably without incurring false positives.

Lock-based approach: A straightforward approach is to walk along the existing kernel data structures used to manage heap memory, which is usually accessed in a lock-based manner. This requires the monitor process to follow the locking discipline. When the lock is held by the monitor process, other processes may be blocked. On the other hand, the monitor process needs to acquire the lock to proceed. Both the kernel performance
and monitoring effect will be affected using the lock-based approach. Another approach is to collect canary addresses in a separate dynamic data structure such as a hash table. By hooking per buffer allocation and deallocation, the canary address is inserted into and removed from the hash table, respectively. Nevertheless, it still does not reduce but migrate the lock contention, since the monitor process and other processes updating the hash table are synchronized using locks.

Lock-free approach: Scanning volatile memory regions without acquiring locks is hazardous [113], which usually needs to suspend the system to double check when an anomaly is detected. The whole system pause is not desirable and sometimes unacceptable. Another approach is to maintain the collection of canary addresses in a lock-free data structure. All processes update and access the data structure in a non-blocking manner. However, the contention between accessing processes may still lead to high overhead.

C2. Self-protection and canary counterfeit. As a countermeasure against buffer overflow attacks, our component can become an attack target itself. We rely on a monitor process that keeps checking—that is, cruising—the kernel heap integrity. After the system is compromised by exploiting the buffer overflow vulnerabilities, attackers may try to kill the monitor process to disable the detection completely. Attackers can also tamper or manipulate the data structure needed by our component to mislead or evade the detection. Moreover, attackers may try to recover the canary after corrupting it. In short, we assume a threat model that attackers control the system completely once they have compromised the system by exploiting heap buffer overflows, that is, they can read/write memory and alter the control flow arbitrarily. We also assume attackers do not know the canary of the buffer to be overflowed before attacks, which is a reasonable common assumption as in all canary-based security schemes [15, 42].

C3. Compatibility. Kernel heap management is among the most important components in OS kernels, whose data structures and algorithms are generally well designed and implemented for efficiency. Thus, the concurrent heap monitoring should not introduce much modification for heap management. Moreover, the solution should be compatible with mainstream systems as well as hardware.
4.2 Overview

Kruiser attaches one canary word at the end of each heap buffer and generates a separate monitor process, which keeps scanning, or *cruising*, the canaries to detect buffer overflows and runs concurrently with the monitored system. In this section we present an overview of the Kruiser architecture and the design choices addressing the challenges presented in the previous section. As shown in Figure 4.1, the monitor process is run in a different VM from the monitored OS to strengthen self-protection. The heap buffer metadata and hooking code are kept in the monitored VM to achieve efficient buffer information collection. The monitor accesses the inter-VM heap metadata via an efficient technique called direct memory mapping. To achieve a concurrent monitoring, the monitor process needs to locate and access the canaries reliably and efficiently, while the monitored system allocates and deallocates the buffers and heap pages continuously.

**To address the synchronization challenge (C1)**, We explore the characteristics of kernel heap management, and propose to interpose heap page allocation and deallocation, by which we maintain concise metadata describing canary locations in a separate efficient data structure. Compared with interposing per buffer allocation and deallocation, the interposition is lightweight and the resultant overhead is much lower. The per page metadata is concise, which enables us to use a fix-sized static data structure to store it. Compared with using a concurrent dynamic data structure to collect canary addresses, the contention due to synchronizing data structure growth and shrink and the overhead due to data structure maintenance (node allocation and deallocation) are completely eliminated. More importantly, as the monitor process traverses our own data structure rather than relying on existing kernel data structures, it is more flexible to design the synchronization algorithm, i.e. the monitor process do not need to follow the synchronization discipline imposed by the kernel data structure. Therefore, we are able to design a highly efficient semi-synchronized non-blocking algorithm, which enables the monitor process to constantly check the live memory of the monitored kernel without incurring false positives.

**To address the self-protection and canary counterfeit challenge (C2),** we apply the virtualization technology to deploy the monitor process into a trusted environment (Figure 4.1). To ensure the same efficiency as in-the-box monitoring, we introduce the Direct Memory Mapping (DMM) technique, which allows the monitor process to access the monitored OS memory efficiently. To protect the heap metadata and interposition
code from being compromised by attackers, we apply the SIM [47] framework, which enables the data and code to be protected safely and efficiently inside the monitored VM. As shown in Figure 4.1, by utilizing the VMM, we introduce two separate address spaces in the monitored VM, and address space 2 is used to place the heap metadata and interposition code. The entry code and exit code are the only ways to transfer execution between the two address spaces so that the metadata can be updated. Canaries are generated applying efficient cryptography, such that once a canary is corrupted, it is difficult for attackers to infer and then recover the canary value.

To address the compatibility challenges (C3), we made minimal changes to the existing kernel heap management based on the commodity hardware. Specifically, we hook the allocation/deallocation that adds/removes pages into/from the heap page pool to update the corresponding heap metadata in our data structure, so that kernel heap buffer allocation algorithms are not changed. On the other hand, the major monitor component is located out of the monitored kernel leveraging the popular VMM platform, which is widely used in cloud computing nowadays.

4.3 Kernel Cruising

In this section, we present the semi-synchronized non-blocking kernel cruising algorithm. We introduce the data structure used in the algorithm in Section 4.3.1. We discuss
potential race conditions in Section 4.3.2 and describe our algorithm in Section 4.3.3.

### 4.3.1 Page Identity Array

Kernels usually maintain heap metadata in dynamic data structures. For example, Linux kernel uses a set of lock-based lists to describe the heap page pool. It is tempting to walk along the existing data structures to check heap buffers. This way the concurrent monitor process has to follow the locking discipline, which would introduce intense lock contention. Another concurrent approach, as used in kernel memory mapping and data analysis for kernel integrity checking [113], is to check without acquiring locks and freeze the monitored VM for double-check to avoid false positives, which may require suspending the VM frequently in our case.

Instead of relying on kernel-specific data structures, we maintain a separate structure called Page Identity Array (PIA). Its basic form is a static array data structure with each entry recording the identity of a page frame. A variety of page identity information can be of interest, such as per page signature, access control, accounting and auditing data. With regard to concurrent heap monitoring, a PIA entry records whether a page frame is used for heap memory, and if so, the metadata that is used to locate canaries within the page. The first entry corresponds the first page frame, and so forth. Since the kernel memory address space is fixed, the size of PIA structure can be predetermined. This way we only need to hook functions that add pages into the heap page pool and that remove pages from it, updating metadata in the corresponding entries. The monitor traverses the PIA structure and check canaries according to the stored metadata. Compared to interposing per buffer allocation and deallocation and collecting canary addresses in a dynamic data structure, the overhead due to function hooking and data structure maintenance is largely reduced. We postpone details about metadata and memory overhead analysis in Section 4.4.

The idea of using a fixed-size data structure is due to the insight into kernel heap management. We assume that a kernel page, if used for heap memory, is divided into buffer objects of equal size and that all the buffers in this page are arranged as an array, which is true in most commodity systems. Given a heap page and its initial buffer object address and size, the monitor process can locate all the buffers within this page, such that the metadata stored in each PIA entry can be small. Before a process (or a kernel
adds a page into the heap page pool, the canaries within the page are initialized and the corresponding PIA entry is updated. By scanning the canaries within each page, the monitor process detects buffer overflows. Although some buffer objects are not allocated and some canary checking may be not necessary, the simple read operations do not introduce much overhead. For 64-bit systems, multi-level PIA structure similar to page tables and Compare-And-Swap instructions are used to achieve scalability and ensure the nonblocking characteristics.

### 4.3.2 Race conditions

Exploring the characteristics of kernel heap management, we proposed the static PIA structure, which avoids heap monitoring from relying on kernel-specific heap data structure and supports highly efficient random access. Nevertheless, synchronization between the monitor process and processes updating page identities is still an issue. For example, when the monitor process reads an entry, another process may be updating it. Without synchronization, the consistency of PIA entries cannot be ensured, which implies the monitor process cannot retrieve heap buffers reliably.

Before we present the kernel heap cruising algorithm, we first discuss the potential race conditions for sharing the PIA structure, which motivate our semi-synchronized design in Section 4.3.3. Three categories of processes need to access the PIA structure: the monitor process, processes updating PIA entries when pages are added into and removed from the pool, respectively. When multiple processes access the PIA structure, a variety of race conditions can occur, some of which are subtle.

**Non-atomic entry write:** As updating a PIA entry is not atomic, a race condition occurs if we allow multiple processes to modify the same entry simultaneously, which would corrupt the entry. Lock-based synchronization is simple, but it incurs high performance overhead and blocks heap operations.

**Non-atomic entry read:** When the monitor process is reading a PIA entry, another process may be updating it. However, as the read and update of an entry are not atomic, the monitor process may read inconsistent entry values.

**Time of check to time of use (TOCTTOU):** For a given entry if the corresponding page is in the heap pool, the monitor process checks canaries within that page, during which,

---

1In this paper we will use the two terms interchangeably.
however, the page may be removed from the pool and used for other purposes, such that false alarms may be issued.

To avoid false alarms, it is tempting to double check whether the page has been removed from the heap page pool when a canary is detected tampered. Specifically, a flag field indicating whether the page is in the pool is contained in each entry. A process removing the page out of the heap page pool resets the flag; when a heap buffer corruption is detected, the monitor process double checks the flag to make sure the page is still in the pool. A buffer overflow is reported only when a canary is tampered and the flag in the PIA entry is not reset. However, it cannot avoid the ABA hazard as discussed below.

**ABA hazard:** An ABA hazard occurs when one process reads a value \( A \) from some position, and then needs to make sure the position is not updated since last access by reading it again and comparing the second read value with \( A \). However, between the two reads, other processes may have updated the position from value \( A \) to \( B \) then back to \( A \). In our case, it may lead to an ABA hazard if the monitor process intends to determine whether the entry has been updated by reading the flag twice, considering that other processes may have removed the page from the heap page pool and then added it back between the two reads, such that the idea of double-checking the flag can still lead to false alarms due to ABA hazards.

Compared to the idea of walking along existing kernel data structures, we apparently have conquered nothing except migrating the synchronization problems to the PIA structure. However, as presented below, we propose a semi-synchronized algorithm based on PIA to resolve all the problems without incurring false positives or high overhead.

### 4.3.3 Semi-synchronized Non-blocking Cruising

We propose an efficient semi-synchronized non-blocking kernel cruising algorithm, as shown in Figure 4.1, that works with the PIA structure. It resolves the concerns of race conditions without introducing complex synchronization mechanisms, such as fine-grained locks and intricate lock-free data structures.

We add an unsigned integer field version in each entry, which records the “version” of the corresponding page. It is initialized to be an even number when the corresponding page is not in the heap page pool. Whenever a page is added into or removed from the pool, its corresponding version number is incremented by one, so that an odd version number indicates a heap page, and an even number indicates a non-heap page. Because
the size of the version field is one word, the read and write of a version value is atomic, which is critical for the correctness of our algorithm.


```c
// Add a page into the heap page pool
AddPage(page) {
  ...
  /* Inside critical section */
  Initialize all the canaries within the page
  Update the metadata in PIA[page];
  smp_wmb(); // This write memory barrier enforces a store ordering
  PIA[page].version++;
  ...
}

// Remove a page out of the heap page pool
RemovePage(page) {
  ...
  /* Inside critical section */
  for (each canary within the page)
    if (the canary is tampered)
      alarm(); // A Buffer overflow is detected
  PIA[page].version++;
  ...
}

DoubleCheckOnTamper(page, ver) {
  uint ver_recheck = PIA[page].version;
  if (ver_recheck != ver)
    return; // The page was already removed/reused
  alarm(); // A buffer overflow is detected
}

Monitor() {
  uint ver1, ver2;
```
for (int page = 0; page < ENTRY_NUMBER; page++) {
  ver1 = PIA[page].version;
  if (!((ver1 % 2))
    continue; // Bypass non−heap page

  smp_rmb(); // This read memory barrier enforces a load ordering
  Read the metadata stored in PIA[page];
  smp_rmb();
  ver2 = PIA[page].version;
  if (ver1 != ver2)
    continue; // Metadata was updated during the read

  for (each canary within the page){
    if (the canary is tampered)
      DoubleCheckOnTamper(page, ver1);
  }
}

Avoid Concurrent Entry Updates: The kernel commonly has its own synchronization mechanisms to prevent one page frame from being manipulated for inconsistent purposes at the same time. For example, Linux functions kmem_getpages and kmem_freepages, which add page frames into and remove them from the heap page pool, respectively, operate on page frame in a critical section with lock protection. These two functions correspond to AddPage and RemovePage in Figure 4.1, respectively. The PIA entry update operations can be put into the critical section of these two functions; it is thus ensured that two processes cannot update the same entry simultaneously. By leveraging the existing synchronization mechanisms in kernel to maintain the PIA entries, the additional overhead is minimal since updating metadata in a PIA entry is fast. As long as the kernel prevents one page frame from being manipulated by two processes simultaneously, there should be synchronization mechanisms serving for this purpose, so the “free-ride” is widely available.

Avoid Using Inconsistent Entry Value: Instead of preventing the monitor process from reading inconsistent entry value, we allow it to occur. However, we use a double-check algorithm to detect potential inconsistency and avoid using inconsistent values. We read the version field in an entry first (Line 34), and then retrieve other entry fields followed
by another read of the version field (Line 41). The page is to be scanned if and only if the two reads of the version field retrieve identical odd version numbers. Here we assume the wraparound of the version value does not occur between the two reads. Considering that page frame switch in and out of the kernel heap pool is infrequent, it very unlikely that the version number wraps around a 32-bit unsigned integer between the two reads.

Specifically, assume there is a non-heap page frame and the AddPage function adds it into the heap page pool. In its critical section it first updates the metadata and then the version number (Line 9) in the corresponding page entry, such that if the monitor process reads the version number of the entry being updated and the read is before the version number update (Line 9), it will retrieve an even number, which indicate a non-heap page. The monitor process will bypass this page (Line 35) according to our algorithm. A write memory barrier (Line 8) is inserted before the version number update, which preserves an observable update order. It is a convention to assume a sequential consistency memory model in the parallel computing literature when describing a concurrent algorithm; however, the observable update sequence [147] is vital to the correctness of our algorithm, so we point it out explicitly.

The version number is not incremented until RemovePage removes the page from the pool. It does not need write memory barriers around the version update because the enter and exit of a critical section imply a full memory barrier, respectively. Therefore, as long as the two reads of the version field retrieves identical odd values, the retrieved metadata values are consistent. Two read memory barriers (Line 38 and 40) are inserted into the Monitor function, such that an observable load ordering is enforced among the reads of the version number and metadata. But note that the read and write memory barriers are not needed on x86 and AMD64 platforms [148], as they already preserve the loads and stores orders we need.

**Identify TOCTTOU and ABA Hazards:** Without locks or other synchronization primitives, it is difficult to avoid TOCTTOU and ABA hazards. Rather than avoiding the hazards, the algorithm takes a different approach to recognizing potential hazards to avoid false alarms. When a canary is found changed, the monitor process does not report an overflow immediately. Instead, it makes sure the page being checked has not ever been removed out, which is indicated by the version number again. As long as the version number does not change compared to the last read (Line 26), it can be determined that the page has persisted as a heap page; in this situation, if a canary is found corrupted, a buffer overflow is reported without concerns of false positives.
The non-blocking algorithm is constructed using simple reads, writes, and memory barriers without introducing complicated and expensive synchronization mechanisms. The monitoring is wait-free as it guarantees progress in a finite steps of its own execution; i.e., it is non-blocking. The monitor process reads version numbers to determine its control flow, so it is lightly synchronized, while other processes manipulating heap pages make progress without being synchronized or blocked by the monitor process. In other words, the synchronization is one-way. That is why we call it a semi-synchronized non-blocking cruising. On PIA entries, write-write is synchronized with a free-ride from the existing kernel functions, while read-write is not synchronized. It resolves the concern of a variety of subtle race conditions without the need of freezing the entire system for recheck. It does not have false positives and enables efficient concurrent heap monitoring.

4.4 System Design and Implementation

4.4.1 Background

Linux adopts the slab allocator\(^2\) for kernel heap management. It uses caches to organize heap buffer objects. There are two types of caches in kernel heap, namely general caches and specific caches. General caches are mainly used to serve kmalloc calls requesting heap buffers of various sizes, while each specific cache is used to allocate objects of a specific kernel data structure, such as task_{struct}. A cache consists of one or more slabs, each of which occupies one or more physically contiguous pages and contains objects of the same type. When a slab is created to serve a buffer request, additional objects are created in the slab’s memory pages to serve further buffer requests.

4.4.2 Architecture

The architecture, as shown in Figure 4.2, can be divided into three parts: VMM, Dom0 VM, and DomU VM. The Monitor Process in Dom0 VM executing Monitor (Listing 4.1) in an infinite loop to monitor the kernel of DomU VM. A tiny component, namely Memory Mapper, inside the VMM is used to map the kernel memory of the monitored VM to the monitor process, which is detailed in Section 4.4.3. The custom driver in Dom0 VM is used to assist the monitor process to release extra memory during the

\(^2\)Similar schemes are widely used in other commodity systems, such as Solaris and FreeBSD.
memory mapping. The Page Identity Array and the interposition code inside AddPage and RemovePage (Listing 4.1) reside in the kernel space of DomU VM, whose protection is presented in Section 4.4.4.

The out-of-VM monitoring ensures performance isolation and secureness, but usually leads to high overhead. The in-VM information collection provides native code execution and memory access environments, but may be vulnerable to attacks. By addressing the problems, we combine the two schemes as a hybrid solution to provide a secure and efficient monitoring.

### 4.4.3 Direct Memory Mapping

To achieve an out-of-the-box monitoring, a conventional method is to run a monitor process in a trusted VM and perform virtual machine introspection (VMI) via the underlying VMM. However, frequent memory introspection would incur high performance overhead. Each such operation requires VMM to walk the monitored VM’s page table and map the target machine frames to be accessible from the monitor process. To avoid this problem, we introduce Direct Memory Mapping (DMM), by which the monitor process can perform frequent memory introspection with only one-time involvement of
the VMM. The basic idea is that the VMM manipulates the page table of the monitor process such that the monitor process can access the kernel memory of the monitored OS directly, as illustrated in Figure 4.3. Note that the custom driver is implemented as a loadable kernel module, such that the Dom0’s kernel code is not modified. The procedure of DMM can be divided into three stages.

First, the Monitor Process allocates a chunk of memory whose size is determined by the maximum number of memory pages used for DomU VM’s kernel heap (\(0\) in Figure 4.2). As Linux kernel heap only resides in physically contiguous memory areas, its maximum size is less than 896MB in 32-bit kernels even if the physical memory size is larger than 896MB. The goal of this stage is to create a contiguous range of virtual addresses. By properly manipulating the page table entries (PTEs), the VMM enables the monitor process to access the memory of the target OS kernel within the monitor’s virtual address space. However, due to the demand paging mechanism, actually the memory for PTEs are not allocated when the virtual addresses are created. Therefore, we need to access the created memory chunk to trigger the creation of PTEs before operating on them.

Second, the Monitor Process notifies the Custom Driver to reclaim the newly allocated pages (\(1\)) with the PTEs retained. This is necessary because the Monitor Process only needs the new virtual addresses and the corresponding PTEs but does not use the allocated pages; returning these pages back can save a lot of memory. Specifically, this stage
consists of four steps. 1) The Custom Driver first walks the page table of the Monitor Process to identify the PTEs for the memory chunk allocated in the first stage (①). 2) Then, with these identified PTEs, the Custom Driver searches for the corresponding page descriptors used by the page frame management. 3) After that, the Custom Driver clears the relevant flags in these page descriptors (e.g., active flag), and resets their reference counters, map counters as well as other related information. 4) Finally, the Custom Driver invokes the API of the buddy system (i.e., \texttt{free-page()}) to release the page frames.

Third, after the Custom Driver finishes reclaiming pages, it informs the Memory Mapper to perform DMM for the Monitor Process (③). By looking up the DomU’s physical-to-machine (P2M) table (④), the Memory collects all the MFNs of the DomU. With the mapping information, the Memory Mapper updates the PTEs of the Monitor Process accordingly. Specifically, given the newly allocated virtual address range, the Memory Mapper walks the User Page Table to find the corresponding PTEs (⑤), whose page frame numbers are then changed to the MFNs that are collected from the P2M table. In this way, the Monitor Process can access the entire kernel of the target OS with its own page table.

Once the Page Identity Array is allocated and initialized in DomU VM, it invokes a hypercall to notify the underlying VMM (⑥), which then informs the monitor process to begin cruising over the kernel heap (⑦)(⑧).

**Reducing TLB Pressure.** As the memory area that the Monitor Process accesses may be large when a lot of kernel slabs are produced, the kernel cruising may incur high TLB pressure. To address this problem, we exploit the extended paging mechanism that is supported by commodity microprocessors. Specifically, we set the Page Size flag in the page directory entries, enabling the size of page frames to be 2MB instead of 4KB (the page frame will be 4MB in size if it is in None-PAE mode). Note that to this end we also need the hypervisor to support the extended paging. Fortunately, Xen (with PAE enabled) mainly uses 2MB super pages to allocate memory for guest VMs. On the other hand, to ensure the extended paging to work properly, we require the starting virtual address allocated for the monitor process should be 2MB-aligned. To meet this requirement, the Monitor Process needs to allocate 2MB extra memory during the first stage, and then adjust the starting virtual address to be 2MB-aligned before performing DMM.
4.4.4 In-VM Protection

Since the PIA data structure (metadata) and the interposition code reside in the kernel space of DomU VM, attackers may manipulate them directly after exploiting buffer overflow vulnerabilities. To solve this problem, a conventional method is to move the data structure and code to be protected into the hypervisor or another trusted VM. However, it will incur significant performance overhead when the world switches between the hypervisor and the VM become frequent, especially for such fine-grained monitoring as in our case. Instead, we employ the SIM [47] framework, which enables a secure and efficient in-VM monitoring. Specifically, the hypervisor creates a separate protected address space inside DomU VM and puts the code and data to be protected in it, such that those memory regions are protected from the DomU VM kernel by the hypervisor, and the separate address space can only be entered and exited through specially constructed protected gates.

In our case, we need to move the interposition code added in the critical section of AddPage and RemovePage as well as the PIA data structure in Figure 4.1 to the protected memory regions. To this end, we construct two shadow page tables (SPTs) specifying different access permissions for the kernel and the In-VM monitor part. As shown in Figure 4.4, within a kernel address space, a process is not allowed to access the monitor code and data regions, while the kernel code cannot be executed after a process switches to the monitor address space. To invoke the monitor’s code in the kernel address space, the transition code is used to switch address spaces and is executable in both address spaces. The transition code modifies the CR3 register, which contains the physical address of the root of the target shadow page table. By default, any change of CR3 will result in a VMExit. Fortunately, a recent hardware feature allows us to change the CR3 without being trapped to the hypervisor if its value is in the CR3_TARGET_LIST, which is maintained by the hypervisor.

Address Space Maintaining and Switching in SMP. Maintaining and switching shadow page tables in Symmetric MultiProcessing (SMP) involves two challenges: 1) The SPTs for the kernel address space and the monitor address space should get synchronized for correctness. 2) The transition code should determine the correct CR3 target when switching back to the kernel address space.

3 Note that the In-VM monitor part only includes the PIA and the interposition code and will be referred to as the monitor in this section for short, while the monitor process still runs out of the VM.
Figure 4.4. Memory protections in the kernel and monitor address space.

Figure 4.5. Address space switching via transition pages in SMP.
To address the first challenge, our approach explores the observation that the monitor only needs to access the kernel heap (for placing canaries) and the kernel stack (for accessing the arguments and storing local variables), which only reside in non-paged contiguous memory areas. Hence, by looking up the P2M table, we can build the memory mapping in the SPT used by the monitor with one-time effort, and then no synchronization is needed.

As to the second challenge, although one common transition page for the entry code is sufficient, one transition page for the exit code is needed for per processor, considering that different processor may have entered the monitor address space from different process address spaces. In each transition page, the \textit{CR3} address to be assigned has to be equal to the address of the shadow page directory that was used by the current processor prior to entering the monitor address space, as shown in Figure 4.5. To this end, we modify hypervisor to update the \textit{CR3} target used in the associated exit code when a processor performs process switches. The hypervisor should also update the \textit{CR3\_TARGET\_LIST} accordingly.

\textbf{Security Check.} By invoking duplicate AddPage for the corrupted page, attackers can recover the canaries. To avoid this problem, we add one more check in the protected code to prevent pages with odd PIA version numbers from being added again. On the other hand, if attackers invoke RemovePage maliciously, the final round of canary checking in the function can detect overflows.

\subsection*{4.4.5 Placing Canaries}

To detect underflows as well as overflows, it is straightforward to place two canaries surrounding each buffer. Actually, Linux with slab debug-enabled version has adopted this scheme to place canaries. Unfortunately, this method does not make kernel objects aligned in the first-level hardware cache, which may result in more cache misses. To overcome this limitation, we only use one canary instead of two canaries to surveil each kernel object. Since the same type of kernel objects are grouped together inside a slab, our approach can still detect the heap underflow attack occurred in one object (but not the first one in a slab) by checking the canary attached by the previous object.

As shown in Figure 4.6(a), we apply two different ways to place the canary. For the specific caches, we first pad the objects to be word-aligned in size. Then, we add one word canary following the object. Finally, to ensure the object get L1 cache line
aligned, we put some additional padding at the end of this object. On the other hand, as the objects in general caches have already got L1 cache line aligned in size, there is no need to change the form of these objects. Instead, we place a canary in the last word of each object. In addition, we hook the general object allocation function (i.e., kmalloc), and increase the original requested size by one word to hold the canary.

Although the scheme above works well to detect underflows (and overflows), it cannot deal with underflows occurred in the first object, as there is no canary preceding it. To tackle this issue, as shown in Figure 4.6(b), we exploit the existing infrastructure to add a canary before the first object. Specifically, if the slab descriptor is located rightly before the first object, the canary is placed at the end of this slab descriptor; or if there is a slab color, we put a canary in the last word of this color.

**Secure Canary Generation.** To set canary values for kernel objects, a practical solution

---

A slab color is a padding put in the beginning of each slab to optimize the hardware cache performance.
should meet the two requirements: R1) after attackers have compromised the monitored kernel via buffer overflows, they cannot recover the corrupted canaries; R2) The canary generation and verification algorithms should be efficient so that they will not affect the system performance and detection latency. To satisfy these requirements, we employ a stream cipher (RC4 [149]) to generate canary values. For each slab, we first extract a random number from the entropy pool in Linux. Then, this random number is used as the key “stretched” by RC4 into a stream of bytes, the length of which is decided by the number of objects inside the slab. Finally, each 4 bytes of this stream is selected as a canary value for each object. On the other hand, regarding canary checking, we store the key (i.e., the random number) into the corresponding PIA entry for each slab.

**Guaranteed Detection.** With the In-VM protection and secure canary generation, attackers can not hide their attacks in that 1) The In-VM protection prevent attackers from manipulating the PIA entries; 2) The canary generation based on the stream cipher guarantees the difficulty for attackers to recover the corrupted canaries within one cruising cycle. Therefore, the attacks are bound to be detected within one cruising cycle after compromising the system, unless the attackers know the exact canary value to be corrupted beforehand, which usually implies the overread and overrun vulnerabilities overlap for exactly the same buffer area and which is very rare.

### 4.4.6 Locating Canaries

To locate and verify canaries in the Monitor Process, we hook the slab allocations and deallocations to store the metadata into the PIA entries, one of which is shown in Figure 4.7. The `mem` field record the starting address of the first object within the slab. As each PIA entry corresponds to one physical page, we only need to remember the last 12 bit of the address, which equals the offset within one page. For the `obj_size` field, we store the actual object size, including the size of padding for word alignment.

By adding the start address of one object and its actual object size, we can get the canary address. To acquire the start address of the next object, the PIA entry contains the `buffer_size` field, which refers to the whole object size after adding the canary as well as the padding for cache line alignment. The `num` field indicates the number of objects within a slab. To locate the canary that resides in the slab descriptor, we record the slab descriptor size in the `slab_size` field, which additionally includes the size of the object descriptor and the following padding. With the starting address of the first object
subtracting the slab descriptor size, we get the starting address of the slab descriptor and
then locate the canary, whose offset within the slab descriptor is predetermined. On the
other hand, if the slab descriptor is kept off the slab, we set the value of the slab_size
to zero. Accordingly, we employ a different method to locate the canary before the
first object. In particular, we check whether the starting address of the first object is
page-aligned, if not, it indicates there is a color placed in the front. Then, we can check
the canary safely.

As introduced previously, kernel heap are managed in different slabs, one of which
consists of one or more physically contiguous pages. Therefore, the slab that contains
several pages should correspond to several entries in the PIA. In order to facilitate
recording the slab canary information into PIA entries, we just use the first associated
entry to store the whole information, and keep other associated entries empty.

It is worth mentioning that we utilize the page allocator to dynamically allocate kernel
memory for the PIA data structure during the kernel’s initialization. Basically, the total
memory occupied by the PIA is determined by the number of pages in the heap. However,
the proportion is unchanged even if all the physical memory are used by the kernel heap.
Since each PIA entry has only 24 bytes in our implementation, the memory overhead
is as low as 24/4096. Furthermore, it is possible to reduce the size of the PIA entry by
packing its fields.

4.5 Evaluation

To evaluate Kruiser, we developed a prototype of Kruiser based on 32-bit Linux and the
Xen hypervisor (with PAE enabled), and conducted effectiveness tests and measured
performance overhead. All the experiments were run on a Dell Precision Workstation with
two 2.26GHz Intel Xeon quad-core processors and 6GB memory. The Xen hypervisor
(with PAE enabled) version is 3.4.2. We used Ubuntu 8.04 (linux-2.6.24 with PAE
enabled) as Dom0 system and Ubuntu 8.04 (linux-2.6.24 with PAE disabled) as DomU
system (with HVM mode). Moreover, we allocated 1 GB memory and 4 VCPU for this
DomU system.
4.5.1 Effectiveness

To test whether Kruiser can detect heap buffer overflows, we deliberately introduced three explicit vulnerabilities [37, 150] in the Linux kernel, and then exploited these bugs. In our first test, we modified the kernel function cmsghdr_from_user_compat_to_kern, making it process some user-land data without sanitization, such that malicious users launch heap-based buffer overflow attacks via the sendmsg system call. For the second test, we loaded a vulnerable kernel module that is developed by ourselves. The function of this module is to use a dynamic general buffer to store certain data transferred from the user-land. However, the module does not perform boundary check when it stores the user data. In the third test, we also employed a loadable kernel module to export a bug in kernel space. Unlike the second test, we constructed a specific slab in this module, and allocated the last object in this slab to store certain user-land information [37]. As a result, this vulnerability enables attackers to overwrite a page next to the slab by transferring large size data into the kernel object. We then launched three types of heap-based buffer overflow attacks, respectively. Each attack was executed 10 times and Kruiser detected all these overflows successfully. The experimental results indicate that Kruiser is effective in defending against kernel heap buffer overflow attacks.

4.5.2 Performance and Scalability

To evaluate the performance of our monitoring mechanism, we carried out a set of experiments. Each of these experiments was conducted in three different environments, including original Linux, Kruiser with SIM protection (referred as SIM-Kruiser subsequently), and Kruiser without SIM protection.

In the first experiment, we executed the SPEC CPU2006 Integer benchmark suite. Figure 4.8 shows that the average performance overhead for both Kruiser and SIM-Kruiser are negligible. When the slab allocation is frequent, the performance overhead is a little bit higher, such as in gcc; however, the maximal performance overhead is less than 3%.

For the scalability measurement, we tested the throughput of the Apache web server with concurrent requests. Specifically, we ran Apache 2.2.8 to serve a 3.7KB html web page. We used ApacheBench 2.3 running on another machine—a Dell PowerEdge T300 Server with a 1.86G Intel E6305 CPU, 4 GB memory and Ubuntu 8.04 (linux-2.6.24)—to measure the Apache throughput over a GB LAN network. Each time we issued 10k http...
requests with various numbers of concurrent clients, and we observed that the number of the kernel heap buffer object allocation increases along with the concurrency level. As shown in Figure 4.9, the performance overhead imposed by Kruiser and SIM-Kruiser are both relatively stable. On average, Kruiser only incurs about 3.8% performance degradation and SIM-Kruiser about 7.9%.
Benchmark | Maximum number | Minimum number | Average number | Average cycle(µs) |
--- | --- | --- | --- | --- |
perlbench | 107,824 | 105,145 | 106,378 | 39,259 |
cc | 79,085 | 76,325 | 76,682 | 27,662 |
gcc | 78,460 | 76,810 | 77,413 | 27,774 |
mcf | 82,885 | 79,328 | 79,540 | 28,156 |
gobmk | 80,761 | 80,345 | 80,519 | 28,606 |
hmmer | 81,278 | 80,435 | 80,591 | 28,635 |
sjeng | 81,437 | 80,259 | 80,535 | 28,610 |
libquantum | 80,911 | 80,317 | 80,407 | 28,493 |
h264ref | 80,756 | 80,337 | 80,480 | 28,572 |
omnetpp | 82,109 | 80,796 | 81,088 | 28,836 |
astar | 81,592 | 81,022 | 81,097 | 28,897 |
xalancbmk | 99,436 | 82,747 | 88,454 | 30,190 |

Table 4.1. Different cruising cycle for different applications in the SPEC CPU2006 benchmark (The numbers refer to the numbers of kernel objects that are scanned in a cruising cycle).

4.5.3 Detection Latency

We recorded the average cruising cycles (i.e., the average time for scanning all the PIA entries) for different applications in SPEC CPU2006, in order to evaluate the detection latency, which is less than or equal to the cruising cycle at the attack time. As shown in Table 4.1, 10 of 12 applications’ average cruising cycles are shorter than 29 ms, and the other two applications’ are below 40 ms. We also recorded the number of scanned kernel objects in each cruising cycle. The results indicate that the average cruising cycle is mainly determined by the average number of scanned kernel objects. Let \( N \) be the number of scanned kernel objects and \( T \) the average time for the monitor process to check a kernel object. We have \( C = NT \), where \( C \) is the cruising cycle. We can reduce the cruising cycle by keeping \( N \) small. One approach is to divide the PIA entries into different parts, and for each part, we create a separate monitor process. Another approach is to only monitor objects in general caches. This is practical because attackers mainly exploit this category of buffers in the real world.

4.6 Scalable Deployment

Large data centers using shipping-containers packed with thousands of servers each are common nowadays. Therefore, scalable deployment is a critical requirement for intrusion
detection measures in data centers. Unlike traditional interposition-based monitors, which may intervene normal functionalities frequently, Kruiser imposes minimal interference and performs monitoring in parallel with the monitored VM. Moreover, one Kruiser instance is able to monitor multiple VMs given an acceptable detection latency much longer than the cruising cycle, without affecting the guaranteed detection property. In addition, the performance isolation provided by the underlying VMM ensures the monitor process and the monitored VM do not abuse computing resources to interfere with each other, which is a desirable property for users.

With the popularity of multi-core architectures, servers built with many cores are more and more common. The hardware evolution trend embraces the concurrent monitoring fashion, as the cost for a unit core running a monitor instance decreases sharply, and the extra energy consumption by one core is relatively low for machines with hundreds of cores. Therefore, the scalability and low cost properties imply that Kruiser can be practically applied to large data centers and server farms.

4.7 Summary

We have presented KRUISER, a semi-synchronized concurrent kernel heap monitor that cruises over heap buffers to detect overflows in a non-blocking manner. Unlike traditional techniques that monitor volatile memory regions with security enforcement inlined into normal functionality (interposition) or by analyzing memory snapshots, we perform constant monitoring in parallel with the monitored VM on its live memory without incurring false positives. The hybrid VM monitoring scheme provides high efficiency without sacrificing the security guarantees. Attacks are bound to be detected within one cruising cycle. Our evaluation has shown that Kruiser imposes negligible performance overhead on the system running SPEC CPU2006 and 7.9% throughput reduction on Apache. The concurrent kernel cruising approach leverages increasingly popular multi-core architectures; its efficiency and scalability manifest that it can be deployed in practice.
Chapter 5  |  DeltaPath: Precise and Scalable Calling Context Encoding

A calling context is the sequence of active function/method invocations that lead to a program location. Calling contexts provide important information for a large range of applications, such as event logging, profiling, debugging, anomaly detection, and performance optimization. While some techniques have been proposed to track calling context efficiently, they lack a reliable and precise decoding capability; or they work only under restricted conditions, that is, small programs without object-oriented programming or dynamic component loading. These shortcomings have limited the application of calling context tracking in practice. We propose an encoding technique, named DeltaPath, without those limitations: it provides precise and reliable decoding, supports large-sized programs, both procedural and objected-oriented ones, and can cope with dynamic class/library loading. The technique thus enables calling context tracking in a wide variety of scenarios.

The rest of this chapter is organized as follows. Section 5.1 describes the motivation and main ideas behind the technique. Section 5.2 presents the encoding background. Section 5.3 presents DeltaPath. Section 5.4 discusses several practical issues. The implementation details and evaluation results are presented in Section 5.5 and Section 5.6, respectively. The related work is discussed in Section 5.7. The chapter is summarized in Section 5.8.
5.1 Motivation and Main Ideas

It provides critical information about dynamic program behavior. The usage has been demonstrated in a wide range of applications, such as debugging [3, 48–60], event logging and error reporting [61–63], testing [64–67], anomaly detection [68–70], performance optimization [71], and profiling [72–77]. For example, system call event logging is critical for the analysis and diagnosis of the program execution in many production systems. Simply logging the system call events fails to record how program components interact when a system call is issued, while recording calling contexts would be very informative. Take profiling as another example; context sensitive profiling is powerful as it associates data such as execution frequencies, overhead and object life time with calling contexts, and thus provides precise information for program understanding and optimization [78].

It is straightforward to obtain calling contexts through stack walking, which, however, is expensive [79, 80]. A few encoding techniques, which represent a calling context using one or more integers, have been proposed to track calling contexts continuously with low overhead. Bond and McKinley [81] proposed a technique, probabilistic calling context (PCC), that computes a probabilistically unique integer ID, essentially a hash value, for each calling context. Although PCC encoding is efficient and compact, it does not provide decoding, which is essential to applications that require inspecting and understanding contexts, such as debugging, error reporting and event logging [82].

In order to enable the decoding capability, Breadcrumbs was built on PCC [83]. It collects additional dynamic information to assist decoding. Specifically, it records encoding values at relatively cold call sites. Depending on the threshold defining the hot code, the technique either incurs large overhead or sacrifices decoding accuracy and reliability. Besides, the decoding has to be offline because it involves expensive computation (their evaluation used the limit of 5 seconds) for recovering one context.

Computing calling contexts with low overhead and precise and reliable decoding capability is challenging. Recent progress was made by Sumner et al. [82], who proposed precise calling context encoding (PCCE) evolved from path profiling [72]. This technique iterates every possible calling context through static analysis and represents each using a unique encoding during runtime, so the context can be recovered precisely from the encoding.

However, as pointed out in previous research [83], PCCE would not work in the
presence of virtual methods and dynamic class loading. Besides, handling large-scale software is a challenge to this technique, as it lacks a scalable solution to the encoding for a large number of calling contexts. More discussion about PCCE is in Section 5.2. The challenges limit the application of the encoding technique in a large range of scenarios, as a lot of software nowadays is written in object-oriented languages with dynamically loaded components and a large number of calling contexts.

We present DeltaPath, an efficient calling context encoding technique with a precise decoding capability and the support for both procedural and object-oriented programs. Similar to PCCE, the technique leverages the Ball-Larus path profiling algorithm [72]. It obtains a unique encoding for each context at runtime by summing up the addition values of the call sites forming the context. Addition values are computed using our encoding algorithm based on static analysis of the target program, then an addition value and the addition operation are assigned to each call site through instrumentation. Unlike PCCE, which assumes small or medium-scale software without object-oriented programming or dynamically loaded components, DeltaPath does not have those limitations. It supports both procedural and object-oriented programming, allows dynamic class loading, and works with large-scale software.

DeltaPath resolves the limitations based on the following insights and ideas. The addition value of a call site is related to the number of calling contexts ending at it, while a big program usually contains a very large number of calling contexts, such that addition values may go beyond, i.e., overflow, the encoding integer. To avoid integer overflows, it is very inefficient to represent and operate on addition values using some class (e.g., BigInteger in Java). The insight is that a long calling context can be divided into several shorter pieces, each can be encoded using an integer. The DeltaPath encoding algorithm automatically finds a small number of functions acting as such dividers for the whole program, avoiding integer overflows systematically with low overhead. In addition, a virtual function call can be dispatched to many possible functions, while PCCE computes an addition value for each target. It is infeasible to insert a bulky switch statement at each virtual function call site and execute it to choose the addition value, for virtual function call sites are ubiquitous and frequently invoked in object-oriented (OO) programs. Our encoding algorithm computes a single addition value for one call site to minimize the code cache pressure and execution slowdown. Moreover, dynamically loaded classes introduce calling contexts that are not considered during static analysis. A call path tracking technique is designed to detect unexpected calling contexts and keep
the encoding correct.

We implemented DeltaPath and performed experiments with a variety of Java programs. Our evaluation results show that its performance is comparable with that of PCC [81], which is a highly efficient and the state of the art calling context encoding technique capable of working in the presence of object-oriented programming and dynamic class loading. Compared to PCC, DeltaPath introduces precise and reliable decoding.

We made the following contributions.

- We propose a new precise calling context encoding technique that supports both procedural and OO programs.
- Encoding space pressure is addressed systematically. It thus allows the encoding technique to be applied to large-scale software.
- Dynamic class loading is handled properly using the call path tracking technique.
- We implemented DeltaPath and evaluated it on a variety of Java programs. The efficiency of DeltaPath is comparable with that of PCC while it provides precise and reliable decoding.

5.2 Background

Ball and Larus proposed an efficient algorithm (referred to as the BL algorithm) to encode intra-procedural control flow paths [72]. Each of the paths leading from the entry of a function to the end of it obtains a unique encoding. The algorithm has become canonical in control flow encoding and path profiling.

PCCE [82, 118] leverages the BL algorithm and adapts it to encoding calling contexts, which are essentially inter-procedural paths in a call graph. PCCE encodes the calling context using a small number of integer identifiers (IDs), ideally one. It instruments function calls with additions to an integer ID such that the calling context ending at any program point can be uniquely represented by the ID along with the program counter of the point.

The algorithm calculating addition values consists of two steps of analysis of the target program’s call graph. First, it computes the number of calling-contexts (NC) of each node in the call graph by summing the NC of each of the nodes’ predecessors (the NC of main is 1). Second, with respect to each node, the addition value for the first
edge is 0, and for each of the rest edges the addition value is the sum of the NCs of the predecessors appeared in the previously processed edges.

Consider the call graph in Figure 5.1. The annotation of each node indicates its NC. D’s NC = 2, for example, is the sum of the NCs of B and C, denoting there are two possible contexts when D is invoked. The number along each edge is the addition value. Some edges do not have such numbers, meaning the addition values are 0. CG’s addition value is calculated after EG and FG, thus it is the sum (7) of the NC of E (4) and that of F (3).

Take the context ACFG as an example for encoding, the ID increases along CF and FG, respectively, so the result is 6. The table in Figure 5.1 shows the encodings of other calling contexts. Some contexts have the same ID values, for example, AB and AC. It is fine because an encoding is represented by both the ID and the ending node.

Given an encoding, we can precisely recover the calling context from bottom to top. Consider the ID 6 obtained at node G, for example, the edge whose addition value is the greatest but not greater than the ID value is taken, that is, FG. We then jump to node F meanwhile decrease the ID by the addition value, 4, for FG. The decoding continues with F and ID value 2, from which we can recover edges CF and AC the same way.

If cycles exist in the call graph, which implies there are recursions in the program, a

<table>
<thead>
<tr>
<th>context</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>AB</td>
<td>0</td>
</tr>
<tr>
<td>AC</td>
<td>0</td>
</tr>
<tr>
<td>ABD</td>
<td>0</td>
</tr>
<tr>
<td>ACD</td>
<td>1</td>
</tr>
<tr>
<td>ABDE</td>
<td>0</td>
</tr>
<tr>
<td>ACDE</td>
<td>1</td>
</tr>
<tr>
<td>ABD'E</td>
<td>2</td>
</tr>
<tr>
<td>ACD'E</td>
<td>3</td>
</tr>
<tr>
<td>ABDF</td>
<td>0</td>
</tr>
<tr>
<td>ACDF</td>
<td>1</td>
</tr>
<tr>
<td>ACF</td>
<td>2</td>
</tr>
</tbody>
</table>

**Figure 5.1.** Example for PCCE encoding. Edge annotations are addition values; an addition value “+c” means “ID+=c” is executed before the invocation and “ID-=c” is executed after; the superscript on D (D') disambiguates two call sites in D both invoking E.
recursive call path is divided into acyclic sub-paths, each of which is encoded separately, such that a stack of IDs is used to represent a call path of recursions. We refer the readers to [118] for more details. Since recursions are handled in the same way by our technique, we omit its discussion and assume acyclic call graphs in the rest of the chapter.

As a first step towards calling context encoding that allows precise decoding, PCCE works under restrictive conditions: programs in procedural languages without dynamically loaded components or high encoding space pressure. This has limited the application of PCCE in a variety of scenarios. Our goal is to deliver a calling context encoding technique without those limitations.

5.3 DeltaPath Encoding

This section presents DeltaPath. Section 5.3.1 covers encoding in the presence of virtual function calls, while Section 5.3.2 considers encoding space pressure along with object-oriented programming and describes a systematic solution.

5.3.1 Encoding for Object-oriented Programs

With object-oriented programming a function call can be dispatched to multiple targets. Such polymorphism complicates encoding, as a call site may have conflicted addition values due to the multiple dispatch targets. For example, assume edges \( D'E \) and \( DF \) in Figure 5.1 are due to the same virtual function call site. According to the PCCE algorithm, the addition values for edge \( D'E \) and \( DF \) are 2 and 0, respectively.

It is straightforward and tempting to choose the addition value based on the dynamic dispatch result using a `switch` statement, which, however, will significantly increase the code to be inserted and slow down the program execution, as virtual function call sites are massive and frequently invoked in OO programs.

In order to minimize the encoding overhead, our goal is to calculate a single addition value for each call site. However, addition values generated by the PCCE algorithm do not work for the purpose. For example, if 2 is used as the single addition value in the case above, the calling contexts \( ABDF \) and \( ACF \) will both be encoded to 2, which violates the principle of a unique encoding for each different context. The computation of the addition values thus requires a new encoding algorithm.

The upper bound of the encoding space representing the calling contexts ending at a
Figure 5.2. Intuition of our algorithm.

A node is called its inflated calling-context count (ICC). That is, the calling contexts of a node \( n \) are encoded using the integers in \([0, \text{ICC}[n]]\). The basic idea of our algorithm is to ensure the invariant that for any given node, its encoding space is divided into disjoint sub-ranges, with each sub-range encoding calling contexts along one incoming edge of the node. Figure 5.2 illustrates the intuition behind the idea. Assume for a given node \( n \), it has totally \( m \) incoming edges. The addition value along \( p_i n \) is denoted by \( \text{AV}[p_i] \). The invariant is kept if the following conditions are satisfied: (1) \( \text{AV}[p_1] \geq 0 \) and \( \text{AV}[p_i] \geq \text{ICC}[p_{i-1}] + \text{AV}[p_{i-1}] \) for \( i = 2, \ldots, m \), and (2) \( \text{ICC}[\text{main}] = 1 \) and \( \text{ICC}[n] \geq \text{ICC}[p_m] + \text{AV}[p_m] \) if \( n \neq \text{main} \).

It can be proved using induction. The encoding ID is initialized as 0 at the main function, and \( \text{ICC}[\text{main}] = 1 \); the encoding ID is in \([0, \text{ICC}[\text{main}]]\), thus the invariant is satisfied at \text{main}. Assume all the predecessors of \( n \) satisfy the invariant. The calling contexts ending at \( n \) are encoded into \( m \) disjoint sub-ranges: \( [\text{AV}[p_1], \text{ICC}[p_1] + \text{AV}[p_1]] \) encodes the contexts along edge \( p_1 n \), and generally, \( [\text{AV}[p_i], \text{ICC}[p_i] + \text{AV}[p_i]] \), for \( i = 2, \ldots, m \), encodes the contexts along \( p_i n \), where \( \text{AV}[p_i] \geq \text{ICC}[p_{i-1}] + \text{AV}[p_{i-1}] \), that is, \( \text{AV}[p_i] \geq \text{the upper bound of the sub-range encoding the contexts along } p_{i-1} n \). Plus, according to condition (2), all the sub-ranges fall in the range \([0, \text{ICC}[n]]\).

Figure 5.3 illustrates how ICCs and addition values are computed satisfying the conditions above. The virtual function call inside \( p \) can be dispatched to \( n_1, n_2, \ldots, n_k \).

---

\(^1\)An addition value is associated with a call site, so more precisely it should be denoted as \( \text{AV}[p_i n] \); we use \( \text{AV}[p_i] \) instead for the sake of conciseness.
Each node $n_i$ is associated with a variable $CAV[n_i]$, denoting the candidate addition value that should be considered when computing the addition value for the next incoming edge of $n_i$. $CAV[n_i]$ is initially 0 and keeps updating along with the calculation of the addition value. In this case, after the addition value for the call site is assigned as $a = \max\{CAV[n_1], \ldots, CAV[n_k]\}$, the candidate addition values are updated as $CAV[n_i] = ICC[p] + a$. Later the updated $CAV[n_i]$ is used to calculate the addition value for the next incoming edge of $n_i$. By computing the addition value of a call site as the maximum value among the candidate addition values of the nodes that the call site can be dispatched to and updating the CAVs using the chosen addition value, condition (1) described above is satisfied. After the addition value for the last incoming edge of a node $n$ is calculated and $CAV[n]$ is updated for the last time, $ICC[n]$ is assigned as $CAV[n]$, satisfying condition (2).

Algorithm 1 shows the encoding algorithm. The input of the algorithm is the call graph of the target program, $CG = \langle N, E \rangle$, where $N$ is a set of nodes with each representing a function and $E$ a set of directed edges. Each call edge $e \in E$ is a triple $\langle n, m, l \rangle$ where $n, m \in N$, are the caller and callee, respectively, and $\langle n, l \rangle$ is a call site potentially invoking $m$. In Java, for example, $l$ is the byte code index of the call site in $n$. A call edge in our algorithm is modeled as a triple instead of a caller and callee pair in order to distinguish multiple call sites in the caller that may invoke the same callee. In the
Algorithm 1 Encoding with dynamic dispatch

```plaintext
function ENCODING(N, E)

ICC[main] ← 1
for n ∈ N do
    CAV[n] ← 0
for n ∈ N in topological order do
    for e = ⟨p, n, l⟩ of the incoming edges of n do
        cs = ⟨p, l⟩
        if cs ∈ processedSites then
            continue
        processedSites ← processedSites ∪ {cs}
        AV[cs] ← CalculateIncrement(cs)
    if n ≠ main then
        ICC[n] ← CAV[n]
function CALCULATEINCREMENT(cs = ⟨p, l⟩)
a ← 0
for each e = ⟨p, n, l⟩ dispatched from cs do
    if CAV[n] > a then
        a ← CAV[n]
for each e = ⟨p, n, l⟩ dispatched from cs do
    CAV[n] ← ICC[p] + a
return a
```

encoding examples below we omit l and simply use nm to denote an edge for the sake of simplicity, though.

In the beginning, ICC[main] is set to 1 (Line 2), and the CAV of each node is initialized to be 0 (Line 3–4). Then the nodes are visited in a topological order; a node is visited after all its predecessors have been visited. For each non-main node n, ICC[n] is assigned (Line 13) after all its incoming edges are processed (Line 6–11). For each incoming edge of the node being visited, the algorithm identifies its call site (Line 7), and determines whether it has already been processed by searching the call site in processedSites, which stores all the call sites that have been processed; as multiple edges can be due to one call site, this ensures that the algorithm processes each call site only once. Function CALCULATEINCREMENT calculates the addition value for the call site and then updates the CAV of each of the nodes that the call site can be dispatched to. Due to the topological traversal order, when node n is processed, the ICC values of
Figure 5.4. Example for encoding with dynamic dispatch. The superscript on D (D') disambiguates two call sites in D both invoking E. D'E and DF are due to a virtual function call in D, and CF and CG are due to a virtual function call in C. Node annotations are ICC values.

its predecessor nodes are already assigned, so the update of $CAV[n]$ can refer to $ICC[p]$ without problems (Line 20).

Consider the example in Figure 5.4. $ICC[A]$ is set to be 1 (Line 2), and all the CAVs are initialized as 0 (Line 3–4). Upon the visit of B, the addition value for AB is $CAV[B] = 0$ (Line 18), then $CAV[B] = ICC[A] + 0 = 1$ (Line 20) and $ICC[B] = CAV[B] = 1$ (Line 13). Node C is processed similarly. Then the topological traversal reaches D. BD is first processed, so $CAV[D] = 0$ is used as the addition value. Next, $CAV[D]$ is updated as $1(= ICC[D] + 0)$, which is then used as the addition value for the next incoming edge CD, and $CAV[D]$ is updated as $2(= ICC[C] + 1)$ (Line 20). As CD is the last incoming edge of D, $ICC[D]$ is assigned as $2(= CAV[D])$ (Line 13).

The traversal then visits node E. After DE is processed, $CAV[E] = ICC[D] + 0 = 2$ (Line 20). Next, the last incoming edge of E, D'E, is processed. Note that the two edges D'E and DF are due to the same call site, and $CAV[F] = 0$. The addition value for this call site is hence calculated as $max\{CAV[E], CAV[F]\} = 2$ (Lines 16–18). Then $CAV[E]$ and $CAV[F]$ are updated as $4(= ICC[D] + 2)$ (Lines 19–20). After that, $ICC[E]$ is updated as $CAV[E] = 4$ (Line 13). Other nodes are similarly processed following the algorithm. In the end, each call site in Figure 5.4 obtains a single addition value, while each calling context can be encoded uniquely.
In order to accommodate virtual function calls, the algorithm inflates the number of calling contexts (NC) of a node and uses it as the upper bound of its encoding space, which is thus termed as the inflated calling-context count (ICC). For example, \( NC[F] = 3 \), while \( ICC[F] = 5 \); the gap between the two enables a uniform addition value 2 for the virtual call site leading to edges \( D/E \) and \( DF \). It is interesting to note that when there is no virtual function in a program, \( ICC[n] = NC[n] \) for any given node \( n \), which is the case in the PCCE encoding for procedural programs.

The decoding remains the same (Section 5.2), so it is omitted.

### 5.3.2 Encoding for Large-scale Object-oriented Programs

In both PCCE and Algorithm 1, the addition value at a call site reflects the number of calling contexts ending at the call site, while the number of calling contexts grows exponentially with the size of a call graph. Thus the computation of addition values during static analysis can incur integer overflows. Moreover, runtime integer overflows may also occur when computing the encoding ID by summing up addition values.

The runtime integer overflows can be easily resolved despite some cost: before each addition operation, the encoding judges whether an integer overflow can occur; if so, the current ID value is pushed onto a stack, and the ID is reset to 0 before the encoding continues. The encoding result is then represented as a stack of IDs along with the current ID value. The integer overflow problem during static analysis is more challenging. Implementation using some big integer classes, for example, Java library provides the BigInteger class supporting representation of huge integers, is straightforward but would be very inefficient, as addition values are represented as objects and during runtime each addition operation becomes a function call. Our goal is to completely avoid the runtime integer overflow and resolve the integer overflow during static analysis without using big integer classes.

While PCCE achieves this goal by pruning edges during static analysis to ensure that the resultant call graph can be encoded by a single integer, the technique is not scalable for encoding large-sized programs, as massive edges at the deep portion of the call graph would be pruned and the pruned edges are handled at a relatively high runtime cost the same way a runtime integer overflow is processed as aforementioned. We aim at a scalable and efficient solution to the integer overflow problem.

The observation is that so far all calling contexts begin at the main function, such
that the encoding space keeps growing when encoding those calling contexts ending at the deep portion of a call graph. So instead of encoding a calling context as a whole, we divide it into pieces, each of which can be encoded using an integer without overflows; it implies that each addition value can be represented by an integer, since the encoding value of each piece is the sum of the addition values within the piece.

We thus design an advanced version of the encoding algorithm, which chooses a set of anchor nodes dividing all long calling contexts in a program into shorter pieces. Each piece begins at an anchor node and is encoded relative to it. During runtime the encoding maintains a stack. When the program invokes an anchor node \( p \), the current encoding ID value along with the anchor node’s identifier is pushed onto the stack, then the ID is reset to 0 and the encoding continues. When the invocation of \( p \) returns, the ID is recovered with the popped ID value. In this way the previously global encoding space pressure is distributed along anchor nodes, and each calling context piece can be encoded separately and locally. The anchor nodes virtually act as barriers that keep the encoding space pressure from flowing downstream along the call graph.

There are two challenges to be resolved when designing the algorithm. First, it needs to find the anchor nodes. Second, given a call site, there exist multiple calling context pieces all reaching the call site while starting from different anchor nodes; hence, multiple addition values for the call site may be obtained due to the encoding relative to different anchor nodes.

We revise the static analysis in Algorithm 1 to resolve the two challenges, as shown in Algorithm 2. It automatically picks anchor nodes. Initially only the main node is in the anchor node set \( An \) (Line 2). Whenever an integer overflow occurs while processing an edge \( \langle p, n, l \rangle \), \( p \) is added into the set of anchor nodes \( An \) (Line 15) and the static analysis is rerun (Line 16).

In order to cutting calling contexts correctly, function IDENTIFYTERRITORIES first walks the “territory” of each anchor node. Specifically for each anchor it finds out the nodes and edges that can be reached through a bounded depth-first search, which starts the traversal from the anchor node and retreats at other anchor nodes; those anchor nodes form the boundary of the territory. From the nodes and edges in each territory we derive \( n\text{anchors}[n] \) and \( e\text{anchors}[e] \), which represent the anchor nodes that can reach node \( n \) and edge \( e \), respectively.

The territories of anchor nodes overlap, which explains the reason behind the second challenge. In order to resolve it, the candidate addition values (CAV) and the inflated
calling-context counts (ICC) of a node are extended to two-dimensional arrays with the first index still representing the node and the second an anchor node, taking into account of multiple anchor nodes that can reach the node. For example, \( \text{CAV}[n][r] \) denotes the candidate addition value that should be considered when processing the next incoming edge \( e \) of \( n \) if and only if \( r \in \text{eanchors}[e] \).

The addition value for a given call site is determined by the maximum of the candidate addition values of all its possible dispatch target nodes to address the dynamic dispatch (Line 31–35). All the anchor nodes that can reach each of the possible dispatch edges are taken into account, so that the territory conflict is resolved.

After a node’s incoming edges are processed (Line 8–16), the ICC values are updated. For a non-anchor node \( n \), the ICC relative to each anchor in \( \text{nanchors}[n] \) is updated (Lines 18–19), while the ICC of each anchor node is set to 1 (Line 21), which is equivalent with \( \text{ICC}[^{\text{main}}] = 1 \) in Algorithm 1. The encoding algorithm is correct because the invariant described in Section 5.3.1 is satisfied. Specifically, the encoded ID obtained at \( n \) is in the range of \( [0, \text{ICC}[n][\text{top}]) \), where \( \text{top} \) is the top anchor node saved on the stack. It is notable that the runtime overflow checks are not needed, as the algorithm ensures that there is no integer overflow for ICC values (Line 19), which are the upper bounds of the encoding spaces.

Figure 5.5 shows an example of encoding involving two anchor nodes \( \text{C} \) and \( \text{D} \). Consider the encoding of the calling context \( \text{CFG} \). Edge \( \text{CF} \) is first processed. Edges \( \text{CF} \) and \( \text{CG} \) are due to the same virtual call site. As \( \text{CAV}[\text{F}][\text{C}] \) and \( \text{CAV}[\text{G}][\text{C}] \) are initially both 0 (Line 4–6), the addition value for the call site is thus calculated as \( \max(\text{CAV}[\text{F}][\text{C}], \text{CAV}[\text{G}][\text{C}]) = 0 \) (Line 33). Then \( \text{CAV}[\text{F}][\text{C}] \) and \( \text{CAV}[\text{G}][\text{C}] \) are updated as \( \text{ICC}[\text{C}][\text{C}] + a = 1 \) (Line 38). The addition value for the virtual function call in \( \text{D} \) are calculated similarly. After \( \text{DE}, \text{DE}' \) and \( \text{DF} \) are processed, now focus on node \( \text{G} \). EG is processed next with addition value 0. \( \text{CAV}[\text{G}][\text{D}] = \text{ICC}[\text{E}][\text{D}] + 0 = 2 \) (Line 38), and \( \text{CAV}[\text{G}][\text{C}] \) is still 1, so \( \max(\text{CAV}[\text{G}][\text{D}], \text{CAV}[\text{G}][\text{C}]) = 2 \) is used as the addition value for \( \text{FG} \) (Line 33).

For decoding, we first recover the deepest piece of the calling context according to the current ID and the anchor node on the stack top. Then pop the anchor node and the ID to continue decoding the next piece of the calling context. The process repeats until the stack is empty. Given the ID 2 obtained at \( \text{G} \) and the anchor node \( \text{C} \) on stack top, we recover a call path \( \text{CFG} \). Then anchor node \( \text{C} \) and the saved ID are popped from the stack.
Figure 5.5. Example for encoding in large programs. C and D are anchor nodes. The superscript on D (D') disambiguates two call sites in D both invoking E. D'E and DF are due to a virtual function call in D, and CF and CG are due to a virtual function call in C. The annotation of a node denotes the ICC values of the node relative to anchor nodes; for example, ICC[E][D] = 2 means the ICC of E relative to anchor D is 2. A stack is used to store the anchor node identifiers and the ID values when invoking them. The encoding of a call path CFG, for example, is represented by c on the stack, where c contains anchor C’s identifier and the encoding ID value when C is invoked, along with the encoding ID value 2, which is the sum of addition values of CF and FG.

5.4 Practical Issues

5.4.1 Dynamic Class Loading

So far we assume a complete call graph for static analysis. However, in reality the dynamic characteristics of a program can render this assumption invalid. Dynamic class loading, which is common in Java for example, makes the generation of a complete call graph ahead of runtime unresolvable.

As a result, relative to a call graph generated for static analysis, dynamically loaded classes introduce unexpected call paths (UCPs). Figure 5.6 shows such an example, where node X and its incoming and outgoing edges are missing during static analysis stage due to dynamic class loading. The context ABXE contains a UCP B → X → E; its encoding ID value is 0. However, if we decode it, an incorrect calling context ACE is obtained. This kind of UCPs is hazardous as they lead to incorrect encoding and
Figure 5.6. Incomplete call graph. BD and BX are due to the same virtual function call, where X is from a dynamically loaded class unexpected by static analysis.

decoding. Consider another context ABXD, which contains a UCP B → X → D, and BD and BX are due to the same virtual call site. Its encoding ID 0 can be decoded to ABD. Although the decoded result does not contain the dynamically loaded node X, it includes all the other nodes in the right order. We thus consider the encoding correct and this kind of UCPs benign.

In order to keep the encoding correct in the presence of dynamic class loading, we need to detect hazardous UCPs and respond accordingly. Inspired by the control flow integrity (CFI) technique [30], we propose a call path tracking technique that checks call transfers, and apply it to detect hazardous UCPs.

The technique consists of static analysis and runtime enforcement. In the beginning of the static analysis each node in the call graph is in a separate set. The analysis then traverses the call graph. For each call site, it finds out the dispatch target nodes, and merge the sets that contain those nodes. In the end each of the sets left is assigned with a unique set identifier (SID), and nodes in the same set share the SID. In runtime before a call is issued, the expected callee node’s SID along with the call site and the current encoding ID value is saved. At the entry point of each statically loaded function, the expected SID is compared against the function’s SID. If they are not equal, a hazardous UCP is detected. Consider the UCP B → X → E for example. E is able to detect it as hazardous since the expected SID set by B is not equal to E’s SID.

Once a hazardous UCP is detected, the encoding responds as follows. First, the expected SID, the call site, the encoding ID, and the current function’s identifier (E, in this case) are pushed onto stack. The encoding ID is reset to zero, and the encoding
continues. At the exit point of E, the pushed information is popped to balance the stack and recover the encoding variables. It is easy to see that the technique allows benign UCPs; for example, in the case of $B \rightarrow X \rightarrow D$, the expected SID set by B is equal to D’s SID.

During decoding, whenever the ID is 0 and the current function’s identifier is equal to the one on the stack top, it pops the information from the stack to continue the decoding. Recall that the information pushed on the stack can be due to invocation to an anchor node, a hazardous UCP or a recursion. An extra integer can be used to indicate the information type in each stack element.\(^2\)

Alternative solutions exist. For example, we can maintain a variable representing the depth of invocations of dynamically loaded functions by incrementing and decrementing the variable at each dynamically loaded function’s entry and exit points, respectively. Such that each statically loaded function detects a UCP if the depth is not zero. The reset and recover of the depth variable using a stack are required when statically and dynamically loaded function calls interleave. The main engineering advantage of the call path tracking solution is that instrumentation of dynamically loaded classes is completely avoided, while in practice it is sometimes difficult or infeasible to instrument them. For example, if they are loaded by custom class loaders, the instrumentation component usually needs to modify the custom loaders; and the security policy of some third-party components does not allow instrumentation.

### 5.4.2 Flexible Encoding

It is desirable that we can perform a selective encoding for the components of interest in order to reduce encoding overhead. For example, the JVM and library methods are usually of less interest compared to application ones, as JVM implementation and Java libraries are often considered “black boxes” [83]. When recovering calling contexts, users may want to obtain all application methods, while JVM and library methods can be ignored. PCC, for example, redefines that a calling context consists of application functions only and hence encodes application functions solely [81, 83].

Leveraging the call path tracking technique we can skip encoding components of no interest the same way we handle dynamically loaded classes to achieve a more efficient and flexible encoding. As illustrated in Figure 5.7, JDK methods and the associated

\(^2\)Our implementation borrows two bits from the method identifier integer to identify the type of a push, so it does not need an extra integer.
edges (denoted by dashed circles and lines) are of no interest and excluded from the call graph when running Algorithm 2, and the encoding is performed only on application methods. During runtime we rely on the call path tracking to detect unexpected UCPs and encode correctly.

Consider the calling context ABDFG for example. only edge AB is encoded, while edges BD, DF and FG are skipped and hence no overhead is incurred. G detects the hazardous UCP at its entry point, so it responds as discussed in Section 5.4.1 and continues the encoding. Finally, ABG, which consists of application methods only, can be recovered from the encoding result.

It illustrates another advantage of the call path tracking solution over the depth tracking one: no encoding or UCP detection code is executed inside the excluded components, such that the more components are excluded from encoding, the less overhead is incurred by encoding.

5.5 Implementation

The implementation of DeltaPath consists of static analysis and a runtime component. The input of the static analysis is the bytecode of the target program. Besides, the user can optionally specify classes of interest for selective encoding. We use WALA [151] to generate the call graph based on a context insensitive control flow analysis, 0-CFA [152].
Algorithm 2 is then run to compute addition values.

The runtime component is implemented as a Java agent [153]. During runtime it hooks the loading of each class and instruments the call sites based on the static analysis results using Javassist [154].

Our implementation does not require the source code. It is not dependent on a specific Java Virtual Machine (JVM); it can work with any JVM compatible with JDK 5.0.

### 5.6 Evaluation

In this section we present the effectiveness and efficiency our encoding technique. We use the SPECjvm2008 benchmark suite [155], which contains a variety of Java programs including compilers (compiler.*), cryptography applications (crypto.*), scientific computation (scimark.*), and xml applications (xml.*). All experiments were performed on a machine with an Intel Core i7 CPU and 8GB RAM. We used Ubuntu 10.04 and Sun JDK 1.6.0.24 on this machine.

#### 5.6.1 Static Program Characteristics

In SPECjvm a common dispatcher is used in all benchmarks to configure the workload and collect results. It is excluded from the encoding so that the statistics properly represent the characteristics of individual benchmark programs.

<table>
<thead>
<tr>
<th>program</th>
<th>size (bytes)</th>
<th>encoding-all</th>
<th>encoding application</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>nodes</td>
<td>edges</td>
<td>CS</td>
</tr>
<tr>
<td>compiler.compiler</td>
<td>114K</td>
<td>2308</td>
<td>7329</td>
</tr>
<tr>
<td>compiler.sunflow</td>
<td>85K</td>
<td>1846</td>
<td>4185</td>
</tr>
<tr>
<td>compress</td>
<td>59K</td>
<td>1298</td>
<td>2675</td>
</tr>
<tr>
<td>crypto.aes</td>
<td>133K</td>
<td>2656</td>
<td>8201</td>
</tr>
<tr>
<td>crypto.rsa</td>
<td>133K</td>
<td>2656</td>
<td>8204</td>
</tr>
<tr>
<td>crypto.signverify</td>
<td>135K</td>
<td>2694</td>
<td>8290</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>261K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>scimark.fft.large</td>
<td>57K</td>
<td>1273</td>
<td>2616</td>
</tr>
<tr>
<td>scimark.lu.large</td>
<td>57K</td>
<td>1260</td>
<td>2590</td>
</tr>
<tr>
<td>scimark.monte_carlo</td>
<td>56K</td>
<td>1269</td>
<td>2614</td>
</tr>
<tr>
<td>scimark.sor.large</td>
<td>57K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>scimark.sparse.large</td>
<td>57K</td>
<td>1265</td>
<td>2605</td>
</tr>
<tr>
<td>sunflow</td>
<td>458K</td>
<td>7727</td>
<td>25485</td>
</tr>
<tr>
<td>xml.transform</td>
<td>752K</td>
<td>9766</td>
<td>38010</td>
</tr>
<tr>
<td>xml.validation</td>
<td>478K</td>
<td>6703</td>
<td>23092</td>
</tr>
</tbody>
</table>

Table 5.1. Static program characteristics (SPECjvm2008). The maximum encoding ID values of sunflow and xml.validation, shown in bold, are larger than the maximum value of a 64-bit integer (around 1.8e19), so the two benchmarks need anchor nodes.
The static analysis is performed in two encoding settings. One is to encode functions of both JDK and application classes (encoding-all) and the other encodes applications functions only (encoding-application). The encoding-application setting corresponds to the scenario that the user is only interested in application functions in a calling context. Based on the call path tracking technique, DeltaPath provides the flexibility of encoding only the components of interest.

Table 5.1 presents the static characteristics of the benchmark programs with the two settings: encoding-all and encoding-application. For each program, the table details the program size in bytes (size), and for each encoding setting, the number of nodes (nodes) and edges (edges) in the call graph, the number of call sites to be instrumented (CS), the number of virtual function call sites (VCS), and the static maximum encoding ID value (max. ID) which represents the encoding space needed.

With the encoding-all setting, the majority of the benchmarks (13 out of 15) need an encoding space larger than a million. Two benchmarks (sunflow and xml.validation) in particular require huge encoding spaces with numbers shown in bold, such that even a 64-bit integer is insufficient for encoding. Algorithm 2 resolves the integer overflow problem automatically by adding 6 and 7 anchor nodes for sunflow and xml.validation, respectively.

With the encoding-application setting, all programs need much smaller encoding spaces. xml.transform benchmark needs a 64-bit integer for encoding, while the encoding space of each other benchmarks can fit into a 32-bit integer. In addition we observe that with the encoding-application setting the call graph and the number of instrumented call sites of each program are much smaller than with the encoding-all setting, which implies efficiency benefit due to the encoding flexibility.

### 5.6.2 Performance Comparison

We compare our work with the state of the art encoding technique, Probabilistic Calling Context (PCC) encoding [81], which is a purely runtime mechanism representing each calling context as an integer hash value. Similar to DeltaPath, PCC works with object-oriented programs. The original PCC was implemented as a module inside Jikes RVM, which is a research JVM allowing internal optimization options. For example, Jikes RVM’s inlining information enables PCC to optimize the instrumentation by combining multiple encoding computations within inlined functions into a single one. Its performance
overhead is very low on Jikes RVM, 3% on average and 9% at most.

In order to have a head-to-head comparison on our platform, we implemented PCC as a Java agent as well. Therefore, such low-level optimizations are not performed. Note that this is not a limitation of our algorithm but rather due to the implementation choice. If our scheme is implemented in Jikes RVM as the original PCC, both techniques can equally benefit from such low-level optimizations. In fact the composition of our encoding computations (simply adding up multiple additions into one) is as easy as in PCC, and the call path tracking for the inlined functions can also be done at compile time. The main focus of our experiments is to find out whether our technique is comparable with PCC in terms of efficiency and effectiveness using the same implementation technology.

As the original PCC work encodes application functions only, we adopt the encoding-application setting for DeltaPath to instrument the same set of functions. Figure 5.8 illustrates the normalized execution speed when our encoding technique and PCC are applied, respectively. We use the geometric mean to calculate the average slowdown.

Overall DeltaPath (without call path tracking) incurs 32.51% slowdown on average with call path tracking introducing extra 6.79% slowdown, while PCC incurs 0.5% higher overhead than DeltaPath (without call path tracking). For the majority of the benchmarks (11 out of 15 benchmarks), DeltaPath with call path tracking incurs less than 6.5% overhead. In addition, both incur high overhead in a few benchmarks (compress, mpegaudio, scimark.monte_carlo and sunflow), as they contain a few small hot functions; the overhead can be largely reduced if the optimization of combining instrumentations is performed for inlined functions. The results show that DeltaPath is comparable with PCC in all benchmarks, which reflects the similarity of their runtime implementation: both instrument the same set of call sites with simple arithmetic operations.

Figure 5.8. Execution speeds applying PCC, DeltaPath without call path tracking (wo/CPT), and DeltaPath with call path tracking (w/CPT), respectively. The speed means the throughput (operations per minute) that reflects the rate at which the system was able to complete invocations of the workload of that benchmark, and it is normalized against the native runtime.
At a low cost, **DeltaPath provides a reliable and precise decoding capability, which is the most critical difference between DeltaPath and PCC.** In contrast, Breadcrumbs [83], which is built on PCC with the purpose of providing decoding capability, either incurs almost 100% overhead for a “very accurate” decoding, or sacrifices reliability and accuracy significantly even at a moderate cost (extra 20% overhead over PCC).

### 5.6.3 Dynamic Program Characteristics

In order to evaluate the effectiveness of DeltaPath and PCC, we collect the encoded calling contexts at the entry of the instrumented application functions. Table 5.2 presents the statistics. It details the total number of calling contexts collected (total contexts), the maximum/average number of functions in a calling context (max. depth and avg. depth), the number of unique calling context encodings (unique contexts) collected by both techniques. For DeltaPath we present additional information about the encoding stack. The maximum/average depth of the stack (max. depth and avg. depth), the maximum/average number of hazardous UCPs detected per calling context (max. UCP and avg. UCP), and the maximum dynamic encoding ID (max. ID) are presented.

DeltaPath represents each calling context using a stack. Ideally, the stack only contains one element recording the entry node; when the calling context contains recursions, anchor nodes or hazardous UCPs, the depth increases. The maximum/average number of hazardous UCPs is not high, showing that hazardous UCPs are detected in all benchmarks although they are infrequent. The average stack depth is 1∼4.4 varying among the 15

<table>
<thead>
<tr>
<th>program</th>
<th>collected contexts</th>
<th>PCC</th>
<th>DeltaPath</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total contexts</td>
<td>max. depth</td>
<td>avg. depth</td>
</tr>
<tr>
<td>compiler.compiler</td>
<td>92634</td>
<td>15</td>
<td>5.1</td>
</tr>
<tr>
<td>compiler.sunflow</td>
<td>63705</td>
<td>12</td>
<td>5.4</td>
</tr>
<tr>
<td>compress</td>
<td>3243640985</td>
<td>12</td>
<td>10.0</td>
</tr>
<tr>
<td>crypto.aes</td>
<td>14431</td>
<td>9</td>
<td>5.6</td>
</tr>
<tr>
<td>crypto.rsa</td>
<td>538625</td>
<td>9</td>
<td>6.0</td>
</tr>
<tr>
<td>crypto.signverify</td>
<td>541682</td>
<td>9</td>
<td>6.0</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>2489700943</td>
<td>17</td>
<td>13.4</td>
</tr>
<tr>
<td>scismark.fft.large</td>
<td>566237360</td>
<td>12</td>
<td>10.0</td>
</tr>
<tr>
<td>scismark.lu.large</td>
<td>188838329</td>
<td>10</td>
<td>10.0</td>
</tr>
<tr>
<td>scismark.monte_carlo</td>
<td>5033167760</td>
<td>11</td>
<td>10.0</td>
</tr>
<tr>
<td>scismark.sor.large</td>
<td>293603875</td>
<td>10</td>
<td>10.0</td>
</tr>
<tr>
<td>scismark.sparse.large</td>
<td>252002429</td>
<td>11</td>
<td>10.0</td>
</tr>
<tr>
<td>sunflow</td>
<td>2840077292</td>
<td>39</td>
<td>21.8</td>
</tr>
<tr>
<td>xml.transform</td>
<td>92333406</td>
<td>55</td>
<td>15.5</td>
</tr>
<tr>
<td>xml.validation</td>
<td>12900727</td>
<td>11</td>
<td>9.0</td>
</tr>
</tbody>
</table>

Table 5.2. Dynamic program characteristics (SPECjvm2008).
benchmarks, compared to the original calling context depth (5.1~21.8). PCC uses only one integer to represent a calling context, which is more concise than DeltaPath. However, PCC collects fewer unique calling context encodings due to hash collisions, implying that some calling contexts obtain identical encoding results. Therefore, for applications that do not need decoding and allow occasional encoding collisions, such as test coverage and profiling, PCC is a better choice; however, for applications that require precise encoding or decoding, DeltaPath is generally superior to PCC and Breadcrumbs.

5.7 Related Work

Stack Walking. Stack walking is commonly used to obtain calling contexts in debugging [116] and error reporting [79, 80]. However, for applications that need continuously capture calling contexts, it usually incurs excessive overhead.

Dynamic Calling Context Tree. A dynamic calling context tree (CCT) [78,117] is a summary of calling contexts represented as a tree data structure. Maintaining a complete CCT incurs large space and time overhead, while a CCT obtained using sampling may miss contexts of interest. In contrast, calling context encoding approaches [81–83, 118] including our work provide concise representation of all calling contexts.

Path Profiling. Ball and Larus developed an algorithm to encode intraprocedural control flow paths [72]. Melski and Reps extended the work by capturing both inter- and intraprocedural control flow [74]. It encodes the whole control flow transfer history leading to a program point, but their approach does not scale, because there exist too many possible paths for nontrivial programs, and the inserted code is complex. Calling context encoding targets a succinct representation of active methods on the call stack with high efficiency.

Precise Calling Context Encoding. Sumner et al. proposed a precise calling context encoding technique that allows decoding [82,118]. However, it does not work well with object-oriented programming, large-scale software, and dynamic class loading. The challenges are resolved in our work.

Probabilistic Calling Context. Probabilistic calling context [81] by Bond et al. provides an encoding technique that works with OO programs. The encoding result is essentially a hash value of the identifiers of the functions in the calling context. Its advantages over DeltaPath are that the encoding result is consistently only one integer and it does not need static analysis. However, PCC does not provide decoding. Their later work
addresses this shortcoming by combining call graph analysis and recording of encoding results [83]. Due to the hash based encoding nature, it remains as a probabilistic approach in essence thus leaving the decoding stage inaccurate, unreliable and/or expensive. In addition, its decoding has to be offline. In production systems where precise and timely response is needed, deterministic and instant decoding, which is supported by DeltaPath, is a highly desired property.

5.8 Summary

We present DeltaPath, a precise calling context encoding technique that works with both procedural and object-oriented programs. It encodes virtual function calls correctly, and resolves the encoding space explosion problem in large-scale programs. Call path tracking is proposed to deal with dynamic class loading and is applied to achieve the flexibility of encoding only the program components of interest. Compared to probabilistic calling context encoding, DeltaPath has similarly high efficiency with the advantage of precise decoding.
Algorithm 2 Encoding resolving encoding space explosion.

1: function ENCODING($N, E$)
2: $A_n \leftarrow \{\text{main}\}$
3: again: IdentifyTerritories($N, E, A_n$)
4: \textbf{for} $n \in N$ \textbf{do}
5: \hspace{1em} \textbf{for} $r \in \text{nanchors} [n]$ \textbf{do}
6: \hspace{2em} $\text{CAV}[n][r] \leftarrow 0$
7: \hspace{1em} \textbf{for} $n \in N$ in topological order \textbf{do}
8: \hspace{2em} \textbf{for} $e = \langle p, n, l \rangle$ of the incoming edges of $n$ \textbf{do}
9: \hspace{3em} $cs = \langle p, l \rangle$
10: \hspace{3em} \textbf{if} $cs \in \text{processedSites}$ \textbf{then}
11: \hspace{4em} \textbf{continue}
12: \hspace{3em} \text{processedSites} $\leftarrow$ processedSites $\cup \{cs\}$
13: \hspace{3em} $AV[cs] \leftarrow \text{CalculateIncrement}(cs)$
14: \hspace{3em} \textbf{if} $AV[cs] = -1$ \textbf{then} // Overflow detected.
15: \hspace{4em} $An \leftarrow An \cup \{p\}$
16: \hspace{3em} \textbf{goto} again // Restart the encoding.
17: \hspace{1em} \textbf{if} $n \notin An$ \textbf{then}
18: \hspace{2em} \textbf{for} $r \in \text{nanchors} [n]$ \textbf{do}
19: \hspace{3em} $\text{ICC}[n][r] \leftarrow \text{CAV}[n][r]$
20: \hspace{1em} \textbf{else}
21: \hspace{2em} $\text{ICC}[n][n] \leftarrow 1$
22: function IDENTIFYTERRITORIES($N, E, A_n$)
23: \textbf{for} $r \in An$ \textbf{do}
24: \hspace{1em} $\langle \text{visited}N, \text{visited}E \rangle \leftarrow$ BoundedDFS($r$)
25: \hspace{1em} \textbf{for} $n \in \text{visited}N$ \textbf{do}
26: \hspace{2em} $\text{nanchors} [n] \leftarrow \text{nanchors} [n] \cup \{r\}$
27: \hspace{1em} \textbf{for} $e \in \text{visited}E$ \textbf{do}
28: \hspace{2em} $\text{eanchors} [e] \leftarrow \text{eanchors} [e] \cup \{r\}$
29: function CALCULATEINCREMENT($cs = \langle p, l \rangle$)
30: $a \leftarrow 0$
31: \textbf{for each} $e = \langle p, n, l \rangle$ dispatched from $cs$ \textbf{do}
32: \hspace{1em} Assume $\text{eanchors}[e] = \{r_1, \ldots, r_k\}$
33: \hspace{1em} $a' \leftarrow \max \{\text{CAV}[n][r_1], \ldots, \text{CAV}[n][r_k]\}$
34: \hspace{1em} \textbf{if} $a' > a$ \textbf{then}
35: \hspace{2em} $a \leftarrow a'$
36: \textbf{for each} $e = \langle p, n, l \rangle$ dispatched from $cs$ \textbf{do}
37: \hspace{1em} \textbf{for} $r \in \text{eanchors}[e]$ \textbf{do}
38: \hspace{2em} $\text{CAV}[n][r] \leftarrow \text{ICC}[p][r] + a$
39: \hspace{2em} \textbf{if} $\text{CAV}[n][r]$ incurs an integer overflow \textbf{then}
40: \hspace{3em} \text{return} -1
41: \text{return} $a$
Chapter 6  
HeapTherapy: A Self-shielding Heap Memory Allocator

Heap buffer overflow bugs, where reads or writes go beyond heap buffer boundaries, can lead to crashes, erroneous execution, and security vulnerabilities. In this work, we propose HeapTherapy, a comprehensive defense against both heap buffer over-read and over-write. HeapTherapy applies several light-weight detection and shielding methods, including a novel technique for probabilistic over-read detection. A key insight behind our approach is the leveraging of calling context information to locate vulnerable buffers and generate temporary patches at user side. The temporary patches are used to prevent recurring buffer overflows exploiting from the same vulnerability and enable the normal execution of the program.

The remainder of the chapter is organized as follows. Section 6.1 describes the motivation and main ideas of HeapTherapy. Section 6.2 describes the scenarios of applying Codeless Patching. Section 6.3 introduces the method we use to identify vulnerable buffers. Section 6.4 describes the architecture and design of Codeless Patching. Section 6.5 presents how to generate patches offline, and Section 6.6 describes how Codeless Patching can be applied to dealing with attacks in an online realtime style. Section 6.7 summarizes some implementation details. The evaluation results are presented in Section 6.8. We discuss some other potential applications of the technique in Section 6.9, and we conclude in Section 6.10.
6.1 Motivation and Main Ideas

Programs written in C and C++ contain a large number of heap buffer overrun bugs, which involve writes or reads going beyond buffer boundaries. In addition to crashes and erroneous execution, heap buffer overrun bugs can lead to various security threats, including data corruption, control-flow hijack, privilege escalation, and information leakage. The recently published Heartbleed vulnerability, which has affected millions of servers, is due to a heap buffer over-read bug that leads to the threat of leaking sensitive information [156]. Even programs in memory-safe programming languages like Java and JavaScript are not immune to buffer overrun attacks, because the language implementations are usually written in C or C++. For example, privilege escalation attacks launched from an Android app are able to exploit heap buffer overflow bugs contained in the native library through the JNI (Java Native Interface) [157].

Debugging heap overrun bugs is notoriously challenging and time-consuming, and hence the process of releasing a patch is usually lengthy. Once a problem is reported, the software company first contacts system administrators to get an input that can reproduce the problem. With the input, software developers localize the bug using debugging tools like eFence [2] and Valgrind [3], and then fix the code. The patch has to be carefully tested to ensure that new bugs are not introduced. Studies show that the average time between initial reports of a critical security-sensitive bug and the release of a patch for enterprise applications is nearly one month [4].

Once a vulnerability is well known, more and more script kiddies acquire scanning and attacking scripts to launch massive attacks. However, applying patches is not risk-free. Fresh patches may contain new bugs leading to instability or changing program semantics accidentally [5]. The trade-off between urgent and cautious patching is a practical consideration and causes patch management predicaments. Research suggests a delay of at least 10 days to balance the risk of applying defective patches and the threat of being attacked.

Finally, applying patches usually requires restarting programs or systems, which makes many users unwilling to take a timely patching. On the other hand, delayed patching leaves numerous systems running with publicly known vulnerabilities. According to Symantec, those known non-patched vulnerabilities are responsible for the widespread success of Internet-based attacks [4].

A patching technique that satisfies the following requirements can facilitate the
generation and application of patches, and hence can resolve many issues aforementioned: the technique, (R1) given a problem-reproducing input, generates a patch at the user’s side, (R2) does not introduce new bugs, (R3) does not change program semantics, and (R4) allows hot-patching, that is, no need of pausing services for applying patches. While several automated patch generation techniques have been proposed satisfying (R1) [85–87], they do not ensure (R2), (R3), and/or (R4), for they require changing the code through conventional patches or binary instrumentation.

We present a new patching technique, named Codeless Patching, and apply it to addressing heap buffer overrun bugs satisfying R1-R4. It is supposed to work as a provisional patch until a stable and indefective one is available. Our insight is that, given a buffer overrun bug, the buffers that can be overrun, named vulnerable buffers, share the same set of calling contexts for buffer allocations, named vulnerable allocation-time calling contexts or vulnerable calling contexts for short. It motives us to apply context-sensitive defenses. Specifically, after we obtain the set of vulnerable calling contexts through replaying the overrun or other techniques, at program execution whenever a buffer allocation request is hooked, we search the current calling context in the set of vulnerable calling contexts; if there is a match, a vulnerable buffer is identified, and the buffer allocation is redirected to a predefined handling that allocates the buffer with a guard page and/or a padding attaching at its end. Such that a buffer overrun is prevented with a program termination when the overrun touches a guard page, or is safely resolved because of the sufficient padding space. The buffer enhancement incurs low overhead, since it is only applied to vulnerable buffers.

We employ a calling context encoding technique [88] to speed up the calling context comparison operation; it encodes a calling context into a single integer, so that the match operation of a pair of calling contexts is transformed to an integer comparison. The calling context encoding module operates continuously during program execution to encode the current calling context into an integer and its overhead averages only 1.9% on SPEC CPU2006. The encoding value of a calling context is called its calling-context identifier, or CCID; and hence the encoding value of a vulnerable calling context is called a vulnerable CCID (VCCID).

A codeless patch comprises a tuple of integers in the form of <the bug type, a VCCID, other parameters such as the length of padding>. A patch essentially enables the enhancement on a set of buffers, so it does not introduce new bugs (R2) or change program semantics (R3). A new patch can be added into the data structures storing all
the patches to take effect at runtime, so it supports hot-patching easily (R4).

When an input reproducing a known buffer overrun is available, the patches can be
generated by system administrators (R1). We illustrate the process using the handling
of Heartbleed vulnerability as an example. Our experiments illustrate that Codeless
Patching prevents information leakage on Nginx service running with OpenSSL under
the Heartbleed attack, and the throughput overhead is only 6.4%.

Codeless Patching can also work with existing detection techniques to defend against
zero-day heap overrun attacks (this goal is detailed in Section 6.2). While realizing that
Codeless Patching is orthogonal with detection techniques, we implement HeapTherapy,
which handles heap buffer overflow attacks, as a demonstration of how Codeless Patching
can cooperate with an existing overrun detection technique to deliver a more powerful
security countermeasure: it turns a tool that detects overflow but does not prevent buffer
corruption to one that prevents data corruption due to a bug once it is detected. The
experiments show that HeapTherapy can defend against real-world overflow attacks and
incurs only 6.2% of slowdown on SPEC CPU2006.

6.2 Scenarios of Applying Codeless Patching

There are numerous existing approaches that deal with heap buffer overrun attacks. We
discuss some of the approaches closely related to our technique and divide them into the
following two categories.

The first category includes systems that generate patches automatically [85–87]. As
described in Section 6.1, they either generate a conventional patch or require binary
instrumentation. The code change raises the risk of introducing new bugs, changing
program semantics, and usually needs a restart.

The second category encompasses various systems that learn from past faults (e.g.,
crashes) and enhance themselves to achieve a better handling of recurring faults. Rx
system adjusts the program’s execution environment after a failure occurs [119]. It lacks
precise diagnosis guiding such adjustments; instead, it follows some intuition-based
rules to try various adjustments until one works. The trial-and-error approach may
experience many failures before a success. The reactive immune system learns from
faults and checks for recurrences of previously seen faults before a real buffer overrun
occurs [120]. When the incoming fault is detected, it forces the current function to return
with a speculative error code, which may not be the programmer’s intention. Therefore,
it also has the risk of modifying program semantics.

We focus on two scenarios of applying the Codeless Patching technique, which fall into the two categories, respectively.

**Scenario 1: attacking input is available.** Given an input that reproduces an attack, Codeless Patching generates a patch automatically, thus it falls into the first category naturally in this application scenario. The goals of Codeless Patching in this scenario are to satisfy R1-R4 (Section 6.1)

**Scenario 2: attacking input is unavailable and cannot be captured.** Another scenario of applying Codeless Patching is to combine it with existing heap buffer overrun detection techniques. Codeless Patching enhances the system automatically by diagnosing an attack that has occurred and generating patches for preventing recurring attacks. So it falls into the second category as a failure diagnosis and program hardening technique complementary to detection systems. The goals of Codeless Patching in this role is to enhance the program execution in a precisely-directed manner with loyalty to program semantics, once a buffer overrun is detected.

![Figure 6.1. The process of applying Codeless Patching.](image)

Figure 6.1 shows the process of applying Codeless Patching. First, an attack that exploits some program bug is detected. Upon the detection, Codeless Patching performs a diagnosis automatically. The output of the diagnosis is a codeless patch, which is applied to the new program execution to fix the bug. The self-learning process is iterative and cumulative to generate patches for multiple bugs.
6.3 Description Method

6.3.1 Requirements

Given a set of vulnerable buffers that can be overrun due to a bug, we aim at a description method that can summarize the common characteristics of the buffers. Such that once an attack exploiting an overrun bug is detected and the victim buffer is located, the description method is applied to describing the victim buffer.

The description result, a key part of a codeless patch, then can be used in the program execution as a signature to identify other vulnerable buffers due to the same bug. Specifically, the same description method is run over each new buffer; if the description result of a buffer matches that of the victim buffer, we regard it as a potential vulnerable buffer and protect it in a proactive way, so that the buffer cannot be overrun by attacks exploiting the bug. If all the vulnerable buffers are protected, the bug cannot be exploited anymore and is deemed fixed.

The performance advantage of the approach is obvious. Given a precise description method that identifies exactly the vulnerable buffers, the protection is tailored to these buffers, which are usually a small portion of the complete set of buffers. The performance advantage is determined by the preciseness of the description method, though. An extreme case is that all the buffers are regarded vulnerable due to an imprecise description method, then the performance will be largely degraded.

While false positives in identifying vulnerable buffers lead to extra performance overhead, false negatives, that is, some vulnerable buffers are missed from protection, can be exploited by attackers. So the first challenge is to propose a precise method with few to no false negatives and minimal false positives.

Moreover, the description method is invoked for each buffer allocation, followed by a comparison of the description results. So the second challenge is to ensure that the description method and the result comparison operation is efficient.

6.3.2 A Context-Sensitive Perspective

We propose to construct a description method based on the calling context. A calling context is a sequence of active/unreturned method invocations that lead to a runtime operation of interest. It is also loosely referred to as call stack and back trace.
Given an attack that exploits a bug to overrun a buffer, assume the victim buffer’s allocation-time calling context is AC. Our observation is that AC is reproducible when the attack recurs. The calling context is called a vulnerable calling context of the overrun bug, as they are shared by the vulnerable buffers.

Figure 6.2. Part of the calling context tree of MySQL 5.5.19. The dashed lines indicate omitted functions.

Figure 6.2 illustrates such an example. It contains part of the calling context tree of MySQL 5.5.19 with a heap buffer overflow (CVE-2012-5612). To trigger an overflow by exploiting the bug, the pathway of the attack is to reach the `stpcpy` call through the allocation calling context as illustrated in the figure. The victim buffer is always allocated under the vulnerable calling context indicated in the figure. That is, the vulnerable calling context is reproducible when the attack is launched repetitively. In our other tested programs, vulnerable calling contexts are all reproducible also.

The insight is consistent with existing work that employs calling context for security purposes. For example, Xu et al. developed a signature-based approach to identifying malicious requests [121]. In order to reduce false positives, they perform a signature checking only when the calling context upon the process of a user request matches a recorded calling context learned from historical attacks. Newsome et al. proposed a scheme to detect illegal writes to specific sensitive data, such as function pointers;
they only perform the expensive verification when the calling context of the write operation matches that of a malicious write [122]. As in the existing work, the assumption that calling contexts are reproducible under repetitive attacks is regarded as true. The description method of a buffer is, therefore, its allocation-time calling context.

Given a program bug, there exist only a few specific pathways that can exploit the bug [121, 122]. In other words, the number of unique vulnerable calling contexts due to a bug is small, and all of them can be learned cumulatively based on the iterative self-learning Codeless Patching process.

### 6.3.3 Calling-Context Retrieval and Comparison

Retrieval and comparison of calling contexts are frequent operations in an allocation intensive program. It is notable that if they incur high overhead, the overall performance degradation will be significant.

Stack-walking based approach is straightforward for calling context retrieval, but it is too expensive for continuous calling context retrieval. For allocation-intensive programs, it incurs times of overhead [88].

An alternative approach to stack walking is to build a dynamic calling context tree (CCT), where each node in the CCT represents a unique context [158]. The program execution updates a cursor node in the CCT, which represents the current calling context. Experiments show that it incurs up to six times of slowdown and huge space overhead for maintaining a precise CCT.

We propose to employ calling context encoding techniques to speed up the calling context retrieval and comparison. A few encoding techniques, which represent a calling context using one or very few integers, have been proposed to track calling contexts continuously with very low overhead [82, 84, 88]. We use the approach called probabilistic calling context (PCC) encoding [88], for it does not need static analysis and encodes each calling context into only one word. It uses a very simple hash scheme to update the calling context encoding value right before each call site: \( V \leftarrow 3 \times V + cs \), where \( V \) is a thread-local integer variable storing the current calling context encoding value and \( cs \) is an integer hash value of the call site, which is calculated at compilation time based on the file name and line number.

Figure 6.3 shows an example of PCC encoding. It has very high efficiency. The implementation in Jikes RVM averages a 3% slowdown for a global encoding, which
1 foo() {
2     int tmp = V; // Added: load PCC value
3     ...
4     V = 3 * tmp + cs_1; // Added: compute new value
5     cs_1: bar1();
6     V = 3 * tmp + cs_2; // Added: compute new value
7     cs_2: bar2();
8     ...
9 }

Figure 6.3. An example of PCC encoding. Three lines are added to maintain the encoding in the function.

is confirmed by our implementation in LLVM with native code; our implementation of PCC incurs 1.9% average slowdown on SPEC CPU2006.

The encoding value of a calling context is called its calling-context identifier, or CCID; and hence the encoding value of a vulnerable calling context is called a vulnerable CCID (VCCID). Since each calling context is encoded into a single integer, the comparison operation of a pair of calling contexts is transformed to an integer comparison.

6.4 System Design

6.4.1 Codeless Patch

A codeless patch \( C \) comprises a tuple of integers in the form of \( \langle T, VCCID, L, G \rangle \), where \( T \in \{ \text{OVERREAD}, \text{OVERWRITE} \} \) indicating the bug type, \( L \geq 0 \) is the number of bytes in the padding, \( G \in \{ \text{YES}, \text{NO} \} \) indicating whether a guard page is needed.

Based on the efficient retrieval and comparison of calling contexts, vulnerable buffers are identified at the entry of malloc (as well as calloc, realloc and memalign) with high efficiency. Let \( S \) be the set of buffers allocated during the life cycle of a program execution. \( S \) is divided into disjoint subsets, each of which contains buffers with a unique CCID. If the CCID of a subset of buffers is equal to a VCCID in any patch \( C \), the subset of buffers are protected. Therefore, the installation of a patch essentially enables the protection on a subset of buffers.

Each vulnerable buffer is enhanced with \( C.L \) bytes of padding and, if \( C.G = \text{YES} \), a guard page is attached at the end of the buffer; plus, if the bug type is \text{OVERREAD}, the padding is zero-filled to fix information leakage due to the bug.
More details about the buffer structure layout are described in Section 6.5 and Section 6.6.

Due to the hash nature of PCC encoding, if there are collisions such that the encoding value of a different calling context is equal to \( C.VCCID \), then buffers of that calling context are identified as vulnerable buffers also, which is a false positive. The unnecessary protection applied to the buffers only leads to additional performance overhead, but does not cause a security hole. It has been shown in theory and practice that PCC can encode millions of contexts with very few hash collisions [88].

**Hot Patching.** During program execution, the installed patches are stored in a hash table with \( VCCID \) of each patch as the key. The identification of a vulnerable buffer, the installation and the deletion of a patch take \( O(1) \) expected time.

### 6.4.2 Architecture

![Figure 6.4. The architecture of Codeless Patching.](image)

```plaintext
1 malloc(size_t size) {
2    int t = V; // The current CCID is maintained in V
3    Patch *c = hashtable.search(t); // Search patch
4    if(c == NULL) {
5        // The current buffer is not a vulnerable one
6        perform a normal allocation;
7    } else {
8        // The current buffer is a vulnerable one
9        dispatch the request to a handling function according to the bug type in c
10    }
11 }
```

*Figure 6.5. The pseudo code of the malloc function.*
The architecture of Codeless Patching is shown in Figure 6.4. Once an overrun is detected, the diagnosis engine starts to generate a patch, which is then installed by inserting it into the hash table. The calling context encoding module maintains the encoding value of the current calling context continuously.

As illustrated by the code in Figure 6.5, for each buffer allocation request, the memory allocator identifies whether it is a vulnerable buffer by searching the current CCID in the hash table; if so, the request is dispatched to a handling function determined by the bug type in the matched patch and the buffer is protected as aforementioned.

Codeless Patching enables collaborative patch generation, which means the patch can be generated by other users and then shared. This can significantly enhance the software security beyond a single machine.

The patch generation for Scenario 1 and Scenario 2 is elaborated in Section 6.5 and Section 6.6, respectively.

### 6.5 Offline Patch Generation

We assume that an input reproducing the overrun is available. The system for patch generation comprises a custom memory allocator, a custom segmentation fault handler and a calling context encoding module. The patch generation is divided into two steps.

![Figure 6.6. A buffer with a guard page attached.](image)

First, the bug type and the VCCID are identified. The allocator allocates each buffer as shown in Figure 6.6. When a buffer overrun occurs, the guard page is touched and a segmentation fault is delivered by the kernel. A magic word is stored in the beginning of each guard page; the custom signal handler first tries to turn the page being accessed to be accessible (e.g., using mprotect in Linux) and then compares the first word in the page against the magic word to determine whether the signal was due to an access to a guard
If the signal was indeed due to touching a guard page, we then determine the bug type. In Linux, we can determine whether the segmentation fault was due to a read or write based on the information saved in the context variable passed to the signal handler. Then the CCID saved in the guard page is used as the VCCID of the patch to be generated.

The Buffer size field at the head of the buffer is used to calculate the initial address of the guard page of the buffer. When a free is invoked, it first turns the guard page to be accessible and resets the magic word, then deallocates the buffer. Note that the address of a guard page can be easily inferred from the address of the buffer and its size. The buffer size is stored in memory management metadata, the organization of which depends on the allocator implementation. It is worth mentioning that our implementation is independent from the allocation algorithm and implementation. Actually, our custom memory allocator is implemented as a wrapper that hooks the memory management requests. So the additional Buffer size field is critical for our implementation to keep independent from the underlying allocator.

Second, we determine the length of the padding using a binary search. A buffer with both a padding and a guard is as shown in Figure 6.7. We assume the overrun size if finite. For example, Web servers typically set the limit on length for genuine URLs up to 4096 characters [159]; for a DNS server to handle the resolution, the full domain name may not exceed the length of 253 characters [160]; the maximum length of a Heartbleed over-read is 64K bytes. The initial padding length is 4K bytes. If the overrun still occurs, the padding size doubles, and the binary search continues. If, for example, the overrun does not occur with 8K padding, it implies that the actual overrun size ∈ (4k, 8k]. The range can be used as the new search bounds if a more precise overrun size is of interest. Or the search can stops once the overrun stops, which leads to a waste of < 4k padding

1False positives can occur, when the first word of a non-guard inaccessible page matches the magic word and the page happens to be accessed, which has triggered the signal. The probability is extremely low. In addition, we can alter the magic word value in another run to verify it.
space, in this example. We omit the algorithm and more rigorous analysis, since the binary search is straightforward. It is very likely that by manipulating the attacking script, the overrun size changes. In this case, the largest overrun size is taken. If the possible overrun size is infinite or too large, the patch sets a reasonable Max padding size. In this case, a heap overrun vulnerability that can lead to privilege escalation or information leakage is tamed to be a denial-of-service bug.

6.6 Defense against Zero-day Overflow Attacks

This section presents the HeapTherapy system, which combine Codeless Patching with an existing canary-based heap buffer overflow detection technique \cite{42}. It demonstrates how Codeless Patching can be applied to dealing with zero-day overflow attacks, once such an attack is detected.

Limitations. (1) HeapTherapy cannot deal with heap over-read, because the canary-based detection can only detect overflow. This can be resolved by replacing it with some other techniques, e.g., DieHard \cite{124} and Archipelago \cite{161}, that can detect both over-read and overflow. (2) While canaries have been proven to be an effective approach, there have been techniques that can reveal the canary, for example, the format string attack and repetitive probing-based attacks that guess the canary value. While the format string bugs have been largely reduced recently, the probing-based attack has very low probability to succeed, because Codeless Patching can defeat the repetitive probings, once a single probing is detected. Plus, each canary in our system is an XOR of the canary address and a value randomly assigned at the program start. (3) The canary-based technique used in HeapTherapy only checks a canary when a buffer is allocated and deallocated. An advanced attack may have hijacked the control flow through a ROP attack by overwriting a function pointer before a canary check is conducted. However, due to the wide deployment of ASLR \cite{97}, the attacker usually needs a large number of tries before a successful control flow hijack. Again, Codeless Patching is reactive to one single failed try by generating a patch that defeats the bug being exploited. In this sense, Codeless Patching and ASLR are complementary to systematically defeat attacks.

While realizing the canary-based detection technique has limitations, due to the use of random canaries and ASLR, we assume that, for any given heap buffer overflow attack, it either triggers signals or it is detected by the technique. We define both cases as a successful detection of the heap buffer overflow. Upon a detection a core dump file
is generated. The diagnosis engine then analyzes the file to generate a patch. Before describing the diagnosis details, we first introduce the buffer structures in the system.

<table>
<thead>
<tr>
<th>Head canary</th>
<th>CCID</th>
<th>Buffer size</th>
<th>User buffer</th>
<th>Tail canary</th>
</tr>
</thead>
</table>

Figure 6.8. A buffer surrounded by canaries.

There are two types of buffer structures in HeapTherapy. Type 1, as shown in Figure 6.7, is for buffers that have been identified as vulnerable ones, and Type 2, as shown in Figure 6.8, for other buffers. We can determine whether the overflow has touched the guard page of a Type 1 buffer as described in Section 6.5. If so, the padding size for the corresponding patch should be increased. Otherwise, the diagnosis scans the heap buffers until a buffer is found with an intact head canary but a corrupted tail canary. A patch $c$ is generated with $c.VCCID$ equal to the buffer’s CCID.

In order to find the overflow size the diagnosis continues scanning the heap after the victim buffer until it reaches the boundary of the current virtual memory area, or a heap buffer with an intact head canary is found. In both cases, we have identified the upper bound of the overflowed region. The padding size is then set as the size of the overflowed region.

### 6.7 Implementation

We have implemented HeapTherapy in Linux. The PCC encoding is implemented as a compilation pass in LLVM 3.4. A 32-bit thread local variable is added to store the encoding result, so it can work with multithreaded programs. The memory allocation wrapper is implemented as a shared library built on the ptmalloc allocator, which is the default heap memory allocator in Linux.

The software vendors do not need to change their source code. They only need to compile it using the modified compiler and link it with the shared library of the memory allocation wrapper. The software release package includes the shared library and the automatic diagnosis tool. The user interface is not changed and no learning effort is needed. So HeapTherapy is very easy to deploy.

---

2The buffer size is always rounded up to a multiple of a word, so we borrow the last two bits of the Buffer size field to indicate the buffer type.
6.8 Evaluation

We first evaluate the effectiveness of Codeless Patching based on offline patch generation (Scenario 1) and online patch generation (Scenario 2), respectively. For Scenario 1, we use the recent Heartbleed attack as an example and generate patches to fix the bug. For Scenario 2, we use HeapTherapy to both detect and defend against heap overflow attacks in totally 15 programs. We then measure the efficiency of HeapTherapy on SPEC CPU2006 Integer benchmark suite.

6.8.1 Case Study - Defense against Heartbleed Attack

Heartbleed Attack. The recently exposed Heartbleed vulnerability in OpenSSL threatens millions of Web services on the Internet [156]. By sending an ill-formed heartbeat request, the attacker can over-read a buffer on the heap and steals up to 64KB data from the memory. While a Heartbleed attack is widely classified as an over-read attack, our investigation shows that the attack can actually exploit two heap-based vulnerabilities: an uninitialized read bug and an over-read bug. Specifically, the victim buffer has 34KB, while the attacker can manipulate the length $l$ of the read over the buffer. If $l \leq 34$KB, it is just an uninitialized read attack. Otherwise, the attack is a mixed of uninitialized read and an over-read. In this case study, we focus on over-read, and avoid the uninitialized read simply by zero-filling the buffer. There have been effective network-based defenses against Heartbleed attacks, while we propose a new host-based defense using Codeless Patching.

Patch Generation. We use Nginx 1.3.9 and OpenSSL 1.0.1f to create a vulnerable HTTPS service. Based on the technique described in Section 6.5, we first run the Nginx service under the patch generation mode, which enhances each buffer with a guard page. Next, we launch an over-read attack with a random $l$ from 35KB to 64kB. Every time the over-read hits the guard page, it triggers our signal handler, which retrieves the CCID from the beginning of the guard page (Figure 6.6) and stores them as the VCCID of the patch. The worker process then crashes and the Nginx master process automatically starts a new one. We repeated the above process 1000 times.

There are totally two unique VCCIDs generated. The first VCCID (1942045092) was generated only once after the first worker process crashed. The rest 999 ones have the
Figure 6.9. The relationship of the two vulnerable calling contexts identified by the diagnosis engine. The dashed lines indicate omitted functions.

same value (3937440682). It shows that the total unique VCCIDs under repetitive attacks are very few. The reason there are two VCCIDs is that the initial worker process is created by function ngx_start_worker_processes, while all other worker processes are created by a different function ngx_reap_children. Figure 6.9 shows the relationship between the two calling contexts. The case demonstrates that calling contexts are highly reproducible between program runs.

Apply Patches. After identifying the padding size which is $64\text{KB} - 34\text{KB} = 30\text{KB}$, the two patches are generated as follows:

$$\langle \text{OVERREAD, 1942045092, 30KB, YES} \rangle$$
$$\langle \text{OVERREAD, 3937440682, 30KB, YES} \rangle$$

After installing the two patches, the over-read will only return data in padding space, which has been zero-filled. Specifically, we launched 1000 Heartbleed attacks against the patched Nginx service. To better simulate real-world scenarios, we use ApacheBench to generate benign HTTPS connections at the same time. The result shows that Codeless
Patching successfully prevented information leakage in all attempts *without crashing the service*.

*Other Attack Scripts and Other Services.* To further test the effectiveness of the patches, we tried four different Heartbleed attack scripts collected from the Internet. Codeless Patching prevents information leak in all the cases. We have performed similar experiments with multiple FTP services, including pureftpd, vsftpd and proftpd, that run with OpenSSL. The result shows that Codeless Patching can defend against Heartbleed attacks on all these services as well.

*Efficiency.* To evaluate the efficiency of system being protected, we measure the throughput of the patched Nginx and the native one. In both cases, we use the same machine and setting: ApacheBench simulates 100 concurrent clients, which send totally 10,000 benign HTTPS requests; during the process, 100 Heartbleed attack requests are sent to the server. The results show that Codeless Patching incurs 6.4% throughput overhead on Nginx.

### 6.8.2 Effectiveness of HeapTherapy

<table>
<thead>
<tr>
<th>Program</th>
<th>Vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAMATE Dataset</td>
<td>12 heap buffer overflow cases</td>
</tr>
<tr>
<td>MySQL 5.5.19</td>
<td>CVE-2012-5612</td>
</tr>
<tr>
<td>Lynx 2.8.8dev.1</td>
<td>CVE-2010-2810</td>
</tr>
<tr>
<td>libtiff 4.02</td>
<td>CVE-2013-4243</td>
</tr>
</tbody>
</table>

*Table 6.1.* A list of tested programs that contain heap buffer overflow bugs. Our evaluation shows that HeapTherapy can prevent not only heap data corruption but also program crashes in all cases.

We evaluate the effectiveness of HeapTherapy using 12 heap buffer overflow cases by NIST and 3 real-world attacks. The test programs are listed in Table 6.1.

*SAMATE Reference Dataset.* The SAMATE dataset [138] maintained by NIST contains 12 programs with heap buffer overflow vulnerabilities caused by contiguous writes through assignments, `memcpy`, `strcpy`, `snprintf`, etc. For each case, at first overrun, our diagnosis engine generated the VCCID automatically. Then for any new run, the
heap buffer overflow was prevented and all the programs exited without crashes.

**MySQL.** MySQL 5.5.19 contains a heap buffer overflow vulnerability that allows a remote attacker to launch denial of service attacks using crafted database commands, which overflow a heap buffer and corrupt the heap meta data. The heap corruption leads to a segmentation fault and crashes the MySQL service when closing the connection. We applied HeapTherapy to the service and ran the attack script [162] 20 times against it. In the first run, HeapTherapy detected the attack when the segmentation fault is signaled. The signal triggers the diagnosis engine, which retrieves the VCCID of overflowed buffer in the core dump file, generates a codeless patch and installs the patch to the service. Then, in following runs, HeapTherapy successfully prevents heap corruption and avoids MySQL to close the connection normally.

**Libtiff.** Libtiff is a popular library for processing TIFF images. A heap buffer overflow vulnerability was found in the gif2tiff tool in libtiff 3.4-4.03. By manipulating height and width of a GIF image, a remote attacker can exploit this vulnerability to overwrite a heap buffer. We first reproduce the attack using an attacking GIF image input [163]. Upon detection, the diagnosis engine automatically finds the VCCID in the core dump. When we use gif2tiff to open the crafted image again, the tool is able to avoid crash and reports “illegal GIF block type”.

**Lynx.** Lynx is Web browser actively in use today. Lynx 2.8.8dev.1 has a heap buffer overflow vulnerability in its URL decoding code for handling links containing a hostname with a % character in the last two bytes. A remote attacker can put the malformed link in a web page to crash Lynx browsers parsing that page. We first reproduce the attack using exploits provided here [164]. When Lynx crashed, HeapTherapy is able to capture the VCCID of the overflowed buffer. We then use Lynx to revisit the crafted paged 20 times, and Lynx successfully displays the crafted page without any error in each trial.

In all these cases, HeapTherapy is able to accurately detect the calling contexts for vulnerable buffer allocation, use padding to prevent heap writes, and avoid crashes.
6.8.3 Efficiency of HeapTherapy

6.8.3.1 Methodology

Benchmarks and platform. We use SPEC CPU2006 Integer benchmark suite to measure the execution time and memory overhead of programs with HeapTherapy. The reference workload is used as benchmark programs’ input. All results presented are normalized by the execution time of original benchmark programs without HeapTherapy.

The experiments were performed on a Dell Precision Workstation T5500 with 2.26GHz Intel Xeon E5507 processor and 16GB RAM. The operating system is Ubuntu 12.04 with Linux kernel 3.2.0.

Memory overhead measurement. We also measure the memory overhead in terms of average resident set size (RSS) for all benchmark programs. We write a script to read the VmRSS value of /proc/[pid]/status 50 times per second and then calculated the average.

Vulnerability simulation. To simulate the performance of HeapTherapy when handling one or more vulnerabilities, we sample allocation-time CCIDs and create several sets of VCCIDs for each benchmark program. To do so, we first develop a profiler to count the number of buffer allocations, the number of unique allocation-time CCID values and obtain the number of buffer allocations with each unique CCID. Table 6.2 lists our profiling result. From the table we can see that these benchmark programs have a diverse profile of buffer allocation. Three programs, perlbench, omnnetpp and xalancbmk, have intensive memory allocations, and the later two programs also have a large number of unique allocation-time CCIDs.

Then to fairly choose VCCIDs, we sort the CCIDs of each benchmark program according to their allocation count in descending order. Next, for each benchmark program, we sample 3 sets of VCCIDs containing 1, 5 and 10 elements, respectively. The sampling is at fixed interval in the sorted length to simulate vulnerabilities corresponding to different number of buffer allocations. So for 1-VCCID set, the median one was selected. For 5-VCCID set, the interval is 0.2 and CCIDs at positions 0.2, 0.4, 0.6, 0.8 and 1.0 are chosen. 10-VCCID set is created in a similar fashion with interval 0.1. Since benchmark programs mcf and sjeng do not have 10 allocation-intensive CCIDs, we use all their allocation-intensive CCIDs for their 10-VCCID sets.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Buffer allocation count</th>
<th>Unique allocation-time CCID count</th>
</tr>
</thead>
<tbody>
<tr>
<td>400.perlbench</td>
<td>360,728,131</td>
<td>13,909</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>168</td>
<td>10</td>
</tr>
<tr>
<td>403.gcc</td>
<td>28,458,470</td>
<td>913,747</td>
</tr>
<tr>
<td>429.mcf</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>658,034</td>
<td>5,404</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>2,474,268</td>
<td>191</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>179</td>
<td>10</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>177,779</td>
<td>258</td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>267,064,936</td>
<td>162,332,040</td>
</tr>
<tr>
<td>473.astar</td>
<td>4,799,955</td>
<td>184</td>
</tr>
<tr>
<td>483.xalancbmk</td>
<td>135,155,557</td>
<td>131,848,405</td>
</tr>
</tbody>
</table>

Table 6.2. Buffer allocation and CCID profiling. Benchmark programs in bold text are allocation-intensive.

### 6.8.3.2 CCID Encoding Overhead

Our first evaluation was focus on the execution time and memory overhead of CCID encoding. All benchmark programs are instrumented with CCID encoding code but not linked with the customer memory allocator. The result is shown in Figure 6.10. The average execution time overhead is 1.9% and the average memory overhead is 0.2%.

### 6.8.3.3 HeapTherapy Overhead

Next, we evaluate the execution time and memory overhead of HeapTherapy using 3 sets of simulated patches and different page padding. We simulate the overflow by overwriting the padding page. For execution time overhead, the result in Figure 6.11 shows that HeapTherapy caused 4.3% average overhead in 0 patch case. This is the overhead of HeapTherapy when no unfixed vulnerability exists. And the average overhead for cases with more patches and padding pages only increased from 4.3% to 6.2%, which demonstrates the high efficiency of HeapTherapy even when simultaneously handling multiple unfixed vulnerabilities.

As for memory overhead, Figure 6.12 shows that the average overhead of HeapTherapy when no patch is applied is 5.9%. And the overhead for the case of 10 patches and 5 padding pages is increased to 7.7%. For majority of the benchmark programs, the
Figure 6.10. Execution time overhead and RSS memory overhead of CCID encoding.

Figure 6.11. Execution time overhead of HeapTherapy. Each bar represents one experimental setting. For example, “5 patches, 5-page padding” represents the setting in which 5 simulated patches are applied and each vulnerable buffer is padded with 5 extra pages. The guard page is also enabled for all vulnerable buffers.

memory overhead is negligible and the overhead does not change when you increase the patch number and the padding page number.

We also compare HeapTherapy with DieHarder, a memory allocator that can defend heap-based attacks [145]. From the result in Figure 6.11 we could see that DieHarder incurs much higher execution time overhead for allocation-intensive benchmark programs (e.g. 100% for perlbench). And the average speed performance penalty of DieHarder is 20.3%, which is also significantly higher than HeapTherapy. For allocation-intensive programs, the average overhead for DieHarder is 86.1% while HeapTherapy only has a 15.8% overhead on average. In addition, DieHarder only provides probabilistic defense
such that it cannot guarantee the protection against detected vulnerabilities.

In summary, we conclude that HeapTherapy incurs a low overhead for both execution time and memory.

6.9 Other Potential Applications

While our focus is on heap buffer overruns, Codeless Patching can generate patches to deal with many other heap errors, such as double frees, dangling pointers and uninitialized heap buffer read. We can extend the specification of the codeless patch to support more bug types, and add bug-specific predefined handling to extend the memory allocation wrapper. Assume a patch is already generated for a bug, the coding of the bug-specific handling is straightforward, which is an advantage of Codeless Patching. We discuss some examples in the section.

A type of dangling pointer bugs is due to a premature deallocation of heap buffers. Assume the patch $c$ that fixes a dangling pointer bug is generated. We only need to code a simple handling, which is invoked when a `free` call is hooked: the handling identifies whether the buffer’s CCID matches the $c.VCCID$; if so, the buffer’s deallocation is delayed. Such that the pointer variable previously containing a dangling pointer can be dereferenced safely.

To address an uninitialized heap buffer read bug, we write a simple handling that zero-fills the newly allocated buffer. `malloc` calls the handling only when the buffer’s CCID matches the VCCID of a patch that handles an uninitialized read bug. So that the zero-filling is tailed to buffers where an uninitialized read may occur.

Therefore, the `malloc` and `free` wrappers work as dispatchers that invoke bug-specific handling functions, while the search of CCID in the hash table storing the patches determines which handling to invoke. It is extensible to write new handling
functions and install corresponding patches to deal with new types of heap bugs.

### 6.10 Summary

We introduce a novel patching technique, Codeless Patching. The uniqueness is that it fixes heap overruns automatically without changing the program code, thus avoids the concerns of introducing new bugs. Plus, instead of tracking and monitoring overrun operations, it chooses to enhance buffers and leverage hardware protection to detect illegal access. It is efficient because it adopts context-sensitive defenses, and the enhancement is tailored to potential victim buffers. Moreover, it supports hot-patching and collaborative patch generation. In addition to generating patches, Codeless Patching can work with existing detection techniques to get reactive to zero-day attacks and conducts self-evolving memory management to dynamically adjust the set of buffers under protection. Our evaluation shows that Codeless Patching based on a context-sensitive approach is effective and efficient.
Chapter 7
Conclusion

Software is increasingly complicated. It is not surprising that the released software contains known or unknown vulnerabilities; some of them are being exploited even without being noticed, while others are intricate to debug and fix. With the existence of vulnerabilities and various cyber-attacks, software systems need post-release security approaches.

The procedure of monitoring, diagnosing, and fixing is widely used in production systems. Runtime software monitoring has become a critical approach to program comprehension, debugging, testing, performance tracking, optimization, intrusion detection, and security enhancement. When anomalies are detected, diagnosis is an inevitable step. It describes the symptom of a problem being investigated, and, if necessary, it tries to reproduce the problem. Diagnosis helps narrow down the faulty component, and aims at figuring out the root cause, in order that an appropriate solution can be proposed accordingly. Finally, if the problem is due to program bugs, patching is launched to fix the problem. After that, the cyclic procedure continues with monitoring.

When the procedure is applied in practice, a lot of challenges arise. Overhead is an obstacle for applying extensive monitoring, while automation is a barrier for prompt diagnosis and patching. To resolve the challenges, we have presented several ideas and techniques.

First, we have described a generic software monitoring technique, software monitoring, where the monitoring is run concurrently with the application or kernel execution. The progress being monitored is never blocked due to monitoring, so the monitoring imposes negligible or a little overhead on program execution. We have applied it to heap memory integrity monitoring at both user space and kernel space. Depending on the runtime information collected, software cruising can be applied to solving other problems.
For example, runtime information collection and logging can benefit from the concurrent execution idea.

Second, informative logging can significantly speed up the diagnosis. Calling context covers critical information of program execution. To enable informative logging of calling context information in both procedural and object-oriented programs, we have proposed DeltaPath, a precise and scalable calling context encoding technique. In addition to assisting diagnosis, it can be used to a large variety of other applications, such as debugging, error reporting, testing, anomaly detection, and profiling.

Last, HeapTherapy does not only demonstrate instant defense generation, but is also a case study showing the use of calling context information for automated diagnosis. It fixes out-of-bounds heap vulnerabilities through self-shielding. By integrating runtime monitoring and automated defense generation, the software system can respond to and deal with zero-attacks, which is a promising direction of software security research.
Bibliography


[21] CERT ADVISORY, CA-2002-33 HEAP OVERFLOW VULNERABILITY IN MICROSOFT DATA ACCESS COMPONENTS.

[22] CONOVER, M. (1999), “w00w00 on heap overflows,” Www.w00w00.org/ articles/heaptut.txt.


[34] AUSTIN, T. M., S. E. BREACH, and G. S. SOHI “Efficient detection of all pointer and array access errors,” in *PLDI ’04*, pp. 290–301.


Vita
Qiang Zeng

EDUCATION
The Pennsylvania State University, University Park, PA. Ph.D in Computer Science and Engineering, 2014.
Beihang University, Beijing, M.E. in Computer Science and Engineering, 2008.
Beihang University, Beijing, B.E. in Computer Science and Engineering, 2005.

PROFESSIONAL EXPERIENCE
Cyber Security Lab, PSU research assistant (Doctoral study) 08/2009 – 12/2014
Built the first concurrent non-blocking heap integrity scanner, and the first self-evolving safe and efficient memory allocator. Published work in Oakland’14, NDSS’12 and PLDI’11.
NEC Laboratories America, research intern 01/2013 – 04/2013
Improved the technique below to work with dynamic class loading in Java. Published the work in CGO’14, and filed two patents.
NEC Laboratories America, research intern 05/2012 – 08/2012
Built the first scalable call stack information encoding technique for OO programs.
IBM Thomas J. Watson Research Center, research intern 05/2011 – 08/2011
Invented the first safe query processor for autonomous information sharing in multi-party distributed databases. Published the work in TKDE.
State Key Lab of Software Dev. Envir., research assistant (Master study) 08/2005 – 03/2008
Built a scalable distributed system for a video conferencing system. One patent was awarded.

HONORS AND AWARDS
SIGPLAN Professional Activities Committee grant, 2011.
Master’s degree with distinction, Beihang University, 2008.
Exploration Scholarship (the highest honor in the College of CSE), Beihang University, 2007.
Outstanding Freshman Scholarship, Beihang University, 2001.