CERIAS Tech Report 2011-05
Reverse Engineering of Data Structures from Binary
by Zhiqiang Lin
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Zhiqiang Lin

Entitled Reverse Engineering of Data Structures from Binary

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Dr. Dongyan Xu
Chair

Dr. Xuxian Jiang

Dr. Xiangyu Zhang

Dr. Eugene Spafford

Dr. David Brumley

Dr. Charles Killian

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): Dr. Dongyan Xu

Dr. Xiangyu Zhang

Approved by: Dr. Sunil Prabhakar 07/13/2011
Head of the Graduate Program Date

Graduate School Form 20
(Revised 9/10)

PURDUE UNIVERSITY
GRADUATE SCHOOL

Research Integrity and Copyright Disclaimer

Title of Thesis/Dissertation:

Reverse Engineering of Data Structures from Binary

For the degree of Doctor of Philosophy

I certify that in the preparation of this thesis, I have observed the provisions of *Purdue University Executive Memorandum No. C-22,* September 6, 1991, *Policy on Integrity in Research.**

Further, I certify that this work is free of plagiarism and all materials appearing in this thesis/dissertation have been properly quoted and attributed.

I certify that all copyrighted material incorporated into this thesis/dissertation is in compliance with the United States' copyright law and that I have received written permission from the copyright owners for my use of their work, which is beyond the scope of the law. I agree to indemnify and save harmless Purdue University from any and all claims that may be asserted or that may arise from any copyright violation.

Zhiqiang Lin

Printed Name and Signature of Candidate

07/13/2011

Date (month/day/year)

*Located at http://www.purdue.edu/policies/pages/teach_res_outreach/c_22.html

REVERSE ENGINEERING OF DATA STRUCTURES FROM BINARY

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Zhiqiang Lin

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2011

Purdue University

West Lafayette, Indiana

To my parents and my wife.

ACKNOWLEDGMENTS

First and foremost, I owe my deepest gratitude to my advisor, Professor Dongyan Xu. I would like to thank him for his extraordinary guidance, support, and encouragement throughout my entire PhD study. In particular, he provided me the plenty of freedom to do the research I was fascinated, he guided me in evaluating the research for the right direction, he stayed late in the night for paper deadlines, he polished all of my professional writing and presentation, and he encouraged me to pursue the academic career. I am so fortunate of having him as my advisor. I will cherish the time of working with him and guide my students with the methodology he guided me.

Second, I would like to sincerely thank my co-advisor, Professor Xiangyu Zhang. He guided me in the research of binary analysis in which I was highly interested. He was always available, and always had sharp comments, neat insights, and constructive suggestions for my research ideas. I enjoyed all of the brainstorming discussion with him.

I am also extremely grateful to my committee members, Professor Xuxian Jiang. He has been a great mentor and friend for the past several years. Not only did he help me build the necessary technical skills in system research, but also enrich me the mind-set for a good system security researcher. In addition, he also gave me invaluable advice for the career development. And I have had a great time of working with him on a number of interesting research problems.

I would also like to thank my other committee members, Professor David Brumley, Professor Charles Killian, and Professor Eugene Spafford. This dissertation greatly benefitted from their careful reading, insightful comments, and high standards. Also, I have learned invaluable lessons from interacting with them. A special thank goes to Professor Spafford for his shepherding of the scientific and rigorous aspect of this dissertation.

I would like to extend my thanks to many other faculty members in the computer science department, for the courses they offered that help me to lay a foundation for the

work in this dissertation. I also thank Professor Ninghui Li for serving in my prelim exam committee, Professor Cristina Nita-Rotaru for giving me the invaluable advice on the academic life in US, and Professor Patrick Eugster for the sharing of his research experience. Meanwhile, I am deeply indebted to Professor John C.S. Lui for his tremendous help and advice on my career path, and also indebted to Dr. Vinod Yegneswaran and Dr. Phillip Porras for their mentoring during my summer internship at SRI International.

Lab FRIENDS is my academic home. I would like to thank all of my lab mates with whom I had a time intersection, Zhui Deng, Sahan Gamage, Zhongshu Gu, Ardalan Kangarlou, Junghwan Rhee, Ryan Riley, Paul Ruth, Bo Sang, Dannie Stanley, and Cong Xu for their valuable collaborations and assistance on my research, as well as for their intriguing discussions ranging from research problems to real life. Their friendship will be one of my best fortunes.

My life at West Lafayette would have been much less fun without many of my Chinese friends. Tiancheng Li has given me tremendous help since my first day at Purdue. Qihua Wang has always enthusiastically offered his advice and help whenever I asked. I am also very grateful to Feng Chen, Yang Liu, Ziqing Mao, Wenqi Shen, Ling Tong, Zhuangzhuang Xi, Rong Zhang, and Wei Zhang for all the joy they have brought to me in these years.

Finally, this dissertation is dedicated to my parents, Changxiang Lin and Guanghuan Li, and my gorgeous wife, Jing Huang, for their love, continuous support and encouragement. Without them, I would not be able to reach this point and finish my PhD dissertation.

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Lin, Zhiqiang Ph.D., Purdue University, August 2011. Reverse Engineering of Data Structures from Binary. Major Professor: Dongyan Xu, Xiangyu Zhang.

Reversing engineering of data structures involves two aspects: (1) given an application binary, infers the data structure definitions; and (2) given a memory dump, infers the data structure instances. These two capabilities have a number of security and forensics applications that include vulnerability discovery, kernel rootkit detection, and memory forensics.

In this dissertation, we present an integrated framework for reverse engineering of data structures from binary. There are three key components in our framework: REWARDS, SigGraph and DIMSUM. REWARDS is a data structure definition reverse engineering component that can automatically uncover both the syntax and semantics of data structures. SigGraph and DIMSUM are two data structure instance reverse engineering components that can recognize the data structure instances in a memory dump. In particular, SigGraph can systematically generate non-isomorphic signatures for data structures in an OS kernel and enable the brute force scanning of kernel memory to find the data structure instances. SigGraph relies on memory mapping information, but DIMSUM, which leverages probabilistic inference techniques, can directly scan memory without memory mapping information.

We have developed a number of enabling techniques in our framework that include (1) bi-directional (i.e., backward and forward) data flow analysis, (2) signature graph generation and comparison, and (3) belief propagation based probabilistic inference. We demonstrate how we integrate these techniques into our reverse engineering framework in this dissertation.

We have obtained the following preliminary experimental results. REWARDS achieved over 80% accuracy in revealing data structure definitions accessed during an execution. SigGraph recognized Linux kernel data structure instances with zero false negative and close-to-zero false positives, and had strong robustness in the presence of malicious pointer manipulations. DIMSUM achieved over 20% accuracy improvement than previous non-probabilistic approaches without memory mapping information.

1. INTRODUCTION

1.1 Dissertation Statement

In computer science, a data structure is a particular way of storing (e.g., using array, tree, or graph) and accessing (e.g., sequential, pre-order, or depth-first) data in a computer so that it can be used efficiently [1]. Typically, a data structure is composed of a number of fields, and each field has a specific type. The organization of the data structure fields forms a layout (i.e., the *syntax* of the data structure). The types, which tell the computer and the programmer information about the values and operations that a specific data type can handle, concern the *semantics* of the data structure. Almost all programs use data structures, namely, software development is essentially "Algorithms + Data Structures = Programs" [2].

A data structure usually has two representations. One is the abstract representation, which is the definition of the data structure and this definition is determined by the programmers and used during software development. The other is the concrete representation, which is the instance of the data structure and this instance is created at run-time during program execution.

A data structure is started from a programmer's definitions and is eventually translated into a binary form when the software is compiled. In the reverse direction, "*can we reverse engineer the data structure definitions (i.e., the abstract representation of data structure) from binary code?*" Also, the data structure is instantiated as data structure instances in memory at run-time. The instances are just the raw bits and bytes. Thus, "*can we recognize the specific data structure instances from raw memory images?*"

Both compiled code and raw memory images are in binary forms. That is why we call this process *reverse engineering of data structures*. In general, "Reverse engineering is the process of analyzing a subject system to create representations of the system at a higher

level of abstraction" [3]. More specifically, we aim to *reverse engineer the data structure definitions from binary code*, and *recognize data structure instances from a memory image*. These two capabilities have many computer security and forensics applications.

1.2 Why Data Structure Reverse Engineering is Important

Knowledge of data structure is valuable in many applications. During software development, compilers use data structure semantics to detect meaningless or probably invalid code [4]. For example, a compiler can identify an expression like an integer divided by a string as invalid because, in the usual sense, one cannot divide an integer by a string. Also, a compiler may use the static type of a value to optimize the storage it needs as well as the operations on the value. For example, according to the IEEE specification for single-precision floating point numbers (the static types) [5], many C compilers represent the float data type in 32 bits though theoretically it should be $(-\infty, +\infty)$, and uses floating-point-specific operations on those values.

In the context of software debugging, programmers often need to know both the semantics and the syntax of the data structure to examine a specific memory cell. For example, if a programmer wants to examine a stack variable, he or she must know in advance the types (e.g., integer, string, or pointer), and then use the specific format needed to display the value.

In addition to traditional applications in software development, data structure knowledge has a wide impact in computer security and forensics, such as in the following examples.

- **Vulnerability Discovery –** Knowledge about data structure layout is often used by attackers. For example, a buffer overflow attack relies on the attacker knowing that a buffer is close to a function pointer or return address [6]. Such a data structure layout pattern can actually guide the vulnerability discovery. For example, if a penetration tester or an attacker knows the layout of a stack frame or network message, he or

she can reduce the fuzz [7–11] space and speed up the vulnerability discovery as demonstrated in Packet Vaccine [12] and ShieldGen [13].

- **Exploit Generation –** An exploit is a particular input that can trigger a vulnerability [14–16]. To compromise a remote machine, attackers often construct exploits based on the program data structures, because what the attacker can manipulate is always the input data of the program. For example, from the data structure syntax (e.g., the size of a buffer), an attacker could directly know the exact distance between an exploitable buffer and a return address or a function pointer, and thereby easily manipulate his input to hijack the control flow.

- **Protocol Format Reverse Engineering –** Protocol format reverse engineering aims to reveal the format of incoming and outgoing network messages [17, 18]. Such messages are usually composed of a number of program-defined data structures. If we can reverse engineer the data structures of a program, then we can correlate the incoming and outgoing messages with the reverse engineered data types. A number of recent protocol reverse engineering techniques (e.g., [19–22]) have followed such methodology.

- **Memory Forensics –** Memory forensics is to identify, extract, and analyze meaningful information from a piece of memory dump in a forensically sound manner [23–25]. Samples of such information are IP addresses to which the application under investigation is talking and files that are being accessed. Data structure definitions play a critical role in the extraction process. For instance, without data structure information, it is challenging to decide if four consecutive bytes represent an IP address or are only a regular integer. As such, if there is a technique that can automatically extract the data structure with both the syntax and semantics, then such a technique can directly help construct a meaningful view of the in-memory data and benefit the memory forensics process.

- **Virtual Machine Introspection (VMI)** – VMI is a technique that observes the state of an entire OS from the virtual machine level [26]. There is often a semantic gap

between the guest OS and the host OS [27]. In VMI, the layout and semantics of the guest kernel data structure directly control the external interpretation of kernel events [28, 29]. For example, without knowing the data structure of a process descriptor, it is impossible to interpret the guest kernel memory with the right semantics at host side.

- **Kernel Rootkit Detection –** A kernel rootkit is kernel level malware that hides important kernel objects such as process descriptors or kernel modules. To hide the kernel objects, a rootkit attacker usually has knowledge of the corresponding kernel data structure definitions. For example, to hide a malicious process, the attacker could manipulate the previous and next pointer of the running process list, and then hide it. As a result, a rootkit detector could use the signature of the corresponding data structure and scan the memory in order to recognize the hidden object [30–32].

- **Malware Classification –** To classify malware, anti-virus software primarily relies on the signatures in the malware code. As data structure is one of the important aspects of a program, if the data structure has some unique patterns to a particular program, it is thus possible to use the data structures as malware signatures. In particular, as demonstrated in Laika [33], we can automatically derive the syntax of the data structure from malware code through machine learning techniques and use such syntax as malware signatures.

- **OS Kernel Fingerprinting –** Similar to malware classification using data structures, we could also use the kernel data structures to fingerprint an OS kernel, which is particularly desirable in cloud computing. For example, a public cloud computing platform usually hosts virtual machines (VMs) with various OS kernels. In order to perform VMI [26, 28, 34] on these guest VMs, a prerequisite is to know the specific version of a guest's OS kernel [35–37]. However, such information is not always available to the cloud provider. As will be shown in our dissertation, the unique signatures of kernel data structures can sometimes serve as the fingerprint of a specific OS.

1.3 Why Data Structure Reverse Engineering is Challenging

There are a number of new challenges in (1) reverse engineering data structure definitions from an application binary, and (2) recognizing data structure instances from a memory image.

First, to reverse engineer the data structure syntax and semantics, we have to first disassemble [38] and then analyze the binary code.

- **Static analysis** to reveal the data structure is difficult because of the lack of symbolic information. Also, alias analysis is particularly hard while it is essential to deciding data flow and hence variable semantics. For example, variable discovery [39] is a static analysis technique that recovers the syntactic characteristics of variables, such as a variable's offset in its activation record, size, and hierarchical structure. This technique requires alias analysis and abstract interpretation at binary level, and it does not recover any semantic information about data structures.

- **Dynamic analysis** is challenging as well because many variables are dynamically created and de-allocated at runtime, making it complicated to track and resolve the variable types based on memory locations, which may be re-used during runtime. The *dynamic variable lifetime* may also affect the coverage of data structures as some may be de-allocated before their types are resolved.

Second, to recognize the data structure instances, the state-of-the-art solution is to derive the value-invariant data structure signatures, and use them to scan memory. However, the following are new challenges.

- **The value-invariant is not always available**. Many existing solutions rely on the *field value invariant* exhibited by a data structure (i.e., a field with either a constant value or a value in a fixed range) as its signature [32, 40–43]. Unfortunately, many kernel data structures cannot be covered by the value-invariant scheme. For example, some data structures do not have fields with invariant values or value ranges espe-

cially for pointers. It is also possible that an invariant-value field may be corrupted, thereby making the corresponding data structure instance un-recognizable.

- **Avoiding signature isomorphism when leveraging points-to relation.** Complementary to the value-invariant, we could explore the structural invariant (i.e., the points-to shape of the data structure) as signatures. However, it is possible that two distinct data structures may lead to *isomorphic* signatures that cannot be used to distinguish instances of the two data structures. Hence there is a new challenge to identify the sufficient and necessary conditions to avoid signature isomorphism between data structures.

Third, to scan memory and traverse the points-to relation between data structures, we have to resolve memory mapping [30, 31], namely, given a virtual address, we need to resolve its destination's physical address. However, existing techniques to do so are only suitable for live data instances. As such, we have to recognize the dead data instances (i.e., the data instances that has been deallocated or belong to a dead process). The new challenges include the following in particular.

- **Absence of memory mapping information**. Given a set of memory pages, very little information is available about which pages belong to which process, let alone the sequencing of the physical pages in the virtual address space of a process. Even if we can identify some pointers in a page, it is very hard to follow those pointers without the address mapping information.

- **Absence of type/symbolic information for dead memory**. To map the raw bits and bytes of a memory page to meaningful data structure instances, type information is necessary. For example, if the content at a memory location is 0, its type could be integer, floating point, or even pointer. If these bits and bytes belong to the live memory and the symbolic information is available, then they can be typed through the reference path (as in [30]). However, such information is not available.

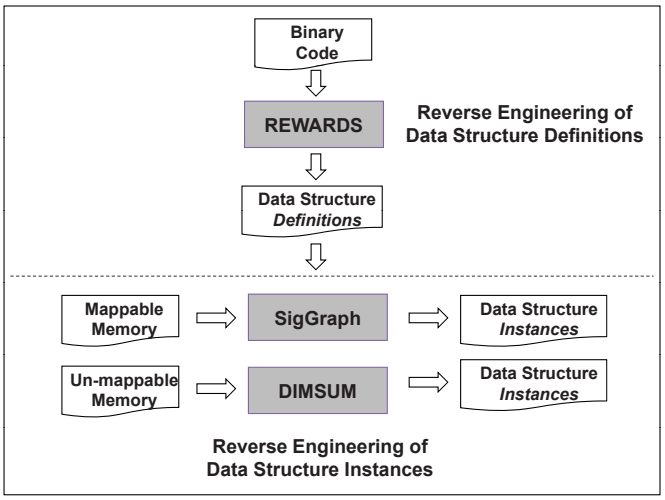


Fig. 1.1.: A framework for reverse engineering of data structure from binary

1.4 Contributions

Our dissertation will address these challenges, develop new techniques to automatically reverse engineer data structure definitions, and recognize data structure instances from not only live memory, but also dead memory.

Our contributions can be summarized as follows.

- We present a systematic framework for reverse engineering of data structures from a binary. As illustrated in Figure 1.1, this framework includes three key components: REWARDS[1] (reverse engineering data structure definitions), SigGraph[2], and DIMSUM[3] (reverse engineering data structure instances), which will provide a complete solution for data structure and data instance reverse engineering.

- REWARDS is a first-step, dynamic analysis based data structure reverse engineering technique. Not only can it reveal the syntax (i.e., the layout) of the data structure, but the semantics (i.e., the meaningful use) of the data structures as well.

[1]REWARDS is the acronym for Reverse Engineering Work for Automatic Revelation of Data Structures.
[2]SigGraph stands for *Graph* based *Sig*natures.
[3]DIMSUM is the acronym for Discovering InforMation with Semantics from Un-mappable Memory.

- SigGraph proposes that points-to relations between data structures can be leveraged to generate graph-based *structural* invariant signatures. SigGraph is a technique that can systematically generate *non-isomorphic* signatures for data structures in an OS kernel and enable the brute-force scanning.

- DIMSUM is the first technique that can uncover semantic data of interest in memory pages without memory mapping information. It is a *probabilistic inference*-based approach, and is able to automatically build graphical models based on boolean constraints generated from the data structure and the memory page contents.

- We have implemented our reverse engineering framework, and our experimental results show that this framework is highly efficient and practical. In particular, REWARDS achieves high accuracy in revealing the data structures accessed during an execution; SigGraph can recognize Linux kernel data structure instances with zero false negative and close-to-zero false positives; and DIMSUM achieves higher effectiveness than previous non-probabilistic approaches without memory mapping information.

1.5 Scope of this Dissertation

This dissertation presents a suite of new techniques for reverse engineering of data structures from a binary, and there are a number of assumptions.

- **Architecture** – We tested our techniques on X86 platform. There are some modifications when applying our techniques to other platforms. For instance, when examining memory content, as X86 is little endian but PowerPC is big endian, we have to accordingly adjust this difference when scanning memory.

- **Operating System** – We also assume the operating system is UNIX/Linux. These are the testing operating system (OS) we used during our prototype development.

- **Programming Languages** – We assume the program is written in C/C++ (the system programming language). For other programs written in such as Java (byte code), or

Python, or Perl, they are out of scope of this dissertation, as they have different runtime mechanisms than that of the native binary code compiled from C/C++.

- **Compilers** – Correspondingly we assume the programs are compiled by standard compilers such as `gcc`. This is because different compilers will have slightly different ways in organizing data structure layout and passing the function parameters.

- **Binary Code** – We also assume no obfuscation [44] against the binary code, and no layout randomization [45] against the data structure.

- **Memory Mapping** – Our technique relies on pointer navigation. We assume the virtual address translation mechanism (e.g., the page mapping) still exists for our SigGraph approach.

- **Unencryped Memory Pages** – Though the input to our DIMSUM is a set of pages, from which we can identify the data instances of interest with confidence, we assume such pages are not encrypted.

1.6 Dissertation Overview

This dissertation presents a data reverse engineering framework that contains a number of new techniques (i.e., REWARDS, SigGraph, and DIMSUM). Each technique has its own distinct goals, challenges, and input, but all are geared towards uncovering the data structures and are naturally integrated into our framework. In particular, as illustrated in Figure 1.1, REWARDS lays the foundation of this framework as SigGraph and DIMSUM both rely on the data structure definitions. SigGraph and DIMSUM complement each other as SigGraph focuses on mappable memory while DIMSUM focuses on unmappable memory.

An outline of this dissertation is as follows.

- **Chapter 1** explains the need for our reverse engineering framework based on a number of security and forensics applications, for which it would be highly useful.

We also examine a number of challenges we have to address in order to realize our framework.

- **Chapter 2** provides an overview of the component of our data structure reverse engineering framework. For each component, we present the fundamental principles behind our techniques and identify the capabilities from the user perspective.

- **Chapter 3** presents the design of our first component, REWARDS, a reverse engineering technique that can automatically reveal program data structures from a binary, based on dynamic analysis. REWARDS leverages the data flow and type revealing execution point to resolve data structure types. We give greater details on how we designed and evaluated REWARDS in this chapter.

- **Chapter 4** presents the design of our second component, SigGraph, a brute-force scanning technique to identify data structure instances through graph-based signatures. We show in this chapter that the points-to relations between data structures can be leveraged to generate graph-based *structural* invariant signatures. Our experiments with a range of Linux kernels show that SigGraph-based signatures achieve high accuracy in recognizing kernel data structure instances via brute force scanning. We further show that SigGraph achieves better robustness against pointer value anomalies and corruptions, without requiring global memory mapping and object reachability.

- **Chapter 5** describes the design of our third component, DIMSUM, a *probabilistic inference*-based approach to uncovering semantic data of interest in memory pages without memory mapping information. Given a set of memory pages and the specification of a target data structure, DIMSUM can identify instances of the data structure in those pages with quantifiable confidence. Our experiments with the applications on Linux platform show that DIMSUM achieves higher effectiveness than previous non-probabilistic approaches without memory mapping information.

- **Chapter 6** demonstrates the applications of our framework, specifically, in memory forensics, vulnerability discovery, kernel rootkit detection, and kernel version inference.
- **Chapter 7** examines the limitations of our framework and outlines our future work.
- **Chapter 8** reviews and compares our techniques with related work.
- **Chapter 9** concludes this dissertation. We end with a discussion of a number of open research problems in this area.

2. A DATA STRUCTURE REVERSE ENGINEERING FRAMEWORK

Motivated by the needs from security applications and the limitations of existing solutions, we present a new framework for data structure reverse engineering. The goal of our framework is to provide a higher level of abstractions of data structures instead of bits and bytes only from either an application binary code or a memory image.

As shown in Figure 2.1, there are three key components in our framework: (1) REWARDS, (2) SigGraph, and (3) DIMSUM. The input to our framework is either an application binary code or a memory image, and the output is either the data structure definitions or the data structure instances.

In this chapter, we provide an overview of each component of our framework. We first present REWARDS in Section 2.1, then SigGraph in Section 2.2, and finally DIMSUM in Section 2.3.

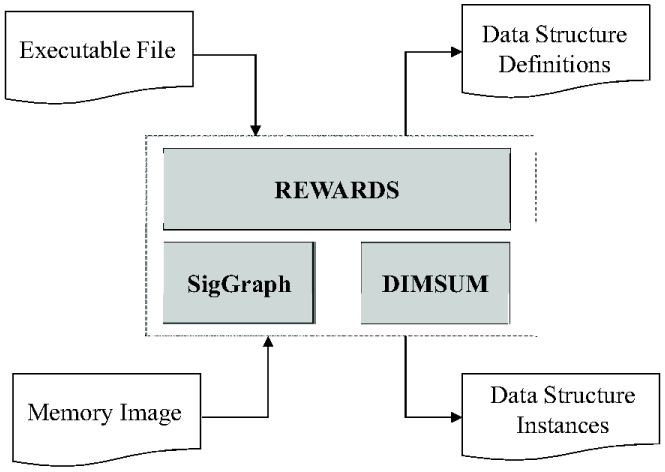


Fig. 2.1.: An overview of our data structure reverse engineering framework

2.1 REWARDS

A desirable capability in many security and forensics applications is automatic reverse engineering of data structures given only a binary. Such capability is expected to identify a program's data structures and reveal their syntax (e.g., size, structure, offset, and layout) and semantics (e.g., "this integer variable represents a process ID"). Such knowledge about program data structures is highly valuable. For example, in memory-based forensics, this knowledge will help locate specific information of interest (e.g., IP addresses) in a memory core dump without symbolic information; and in binary vulnerability discovery, this knowledge will help construct a meaningful view of the in-memory data structure layout and identify those semantically associated with external input for guided fuzz testing.

Despite the usefulness of automatic data structure reverse engineering, existing solutions that suit our targeted application scenarios fall short. First, many works on type inference [46–51] require program source code. Second, in the binary-only scenario, the variables are mapped to low-level entities such as registers and memory locations with no syntactic information, which makes static analysis difficult. In particular, alias analysis is difficult at the binary level while it is essential to type inference – especially semantics inference – because precise data flow cannot be decided without accurate alias information. Variable discovery [39] is a static, binary level technique that recovers the syntactic characteristics of variables, such as a variable's offset in its activation record, size, and hierarchical structure. This technique relies on alias analysis and abstract interpretation at binary level. Moreover, due to the conservative nature of binary alias analysis, the technique does not infer variable semantics. More recently, Laika [33] aims at dynamically discovering the syntax of observable data structures through unsupervised machine learning on program execution. The accuracy of this technique, however, may fall below the expectation of our applications. It also does not consider data structure semantics.

We thus developed the first technique in our framework, an automatic data structure definition reverse engineering system, REWARDS. Given a binary executable file, REWARDS executes the binary, monitors the execution, aggregates and analyzes runtime

information, and finally recovers both the syntax and semantics of data structures observed in the execution at a high level.

More specifically, each memory location accessed by the program is tagged with a *timestamped type attribute*. Following the program's runtime data flows, this attribute will be propagated to other memory addresses and registers that share the same type. During the propagation, a variable's type can be resolved if it is involved in a type-revealing execution point or "type sink" (e.g., a system call, a standard library call, or a type-revealing instruction).

REWARDS infers both the syntax and the semantics of data structures from binary execution. More precisely, we aim at reverse engineering the following information:

- **Data types.** We first aim to infer the primitive data types of variables, such as `char`, `short`, `float`, and `int`. In a binary, the variables are located in various segments of the virtual address space, such as `.stack`, `.heap`, `.data`, `.bss`, `.got`, `.rodata`, `.ctors`, and `.dtors` sections. Although we focus on ELF binary on Linux platform, REWARDS can be easily ported to handle PE binary on Windows. Hence, our goal is essentially to annotate memory locations in these data sections with types and sizes, following program execution. For our targeted applications, REWARDS also infers composite types such as socket address structures and FILE structures.

- **Semantic meanings.** Moreover, we aim to infer the semantic meanings of program variables, which is critical to applications such as computer forensics. For example, an IP address is represented by 4 bytes memory at the binary level, and it may be classified as an integer. In a memory dump, we want to decide if a 4-byte integer denotes an IP address.

- **Abstract representation.** Although we type memory locations, it is undesirable to simply present typed memory locations to the user. During program execution, a memory location may be used by multiple variables at different times; and a variable may have multiple instances. Hence, we derive an abstract representation

for a variable by aggregating the type information at multiple memory locations instantiated based on the same variable. For example, we use the offset of a local variable in its activation record as its abstract representation. Type information collected in all activation records of the same function is aggregated to derive the type of the variable.

Given only the binary, we can observe the following at runtime from each instruction: (1) the addresses accessed and the width of the accesses, (2) the semantics of the instruction, and (3) the execution context such as the program counter and the call stack. In some cases, data types can be partially inferred from the instructions. For example, a floating point instruction (e.g., FADD) implies that the accessed locations must have floating point numbers. We also observe that the parameters and return values of standard library calls and system calls often have their syntax and semantics well defined and publicly known. We define the type revealing instructions, system calls, and library calls as *type sinks*. Furthermore, the execution of an instruction creates a dependency between the variables involved. For instance, if a variable with a resolved type (from a type sink) is copied to another variable, the destination variable should have a compatible type. As such, we model our problem as a type information flow problem.

We have developed a prototype of REWARDS and have used it to analyze a number of binaries. Our evaluation results show that REWARDS is able to correctly reveal the types of the variables observed during a program's execution. Furthermore, we have demonstrated the unique benefits of REWARDS to a variety of application scenarios. In *memory image forensics*, REWARDS helps recovering semantic information from the memory dump of a binary program. In *binary fuzzing for vulnerability discovery*, REWARDS helps in the identification of vulnerability "suspects" in a binary for guided fuzzing and confirmation.

2.2 SigGraph

Given a data structure definition (which can be acquired by REWARDS), identifying instances of that data structure in a memory image is an important capability in memory im-

age forensics [25, 52–55], kernel integrity checking [30, 32, 43, 56, 57], and virtual machine introspection [26, 28, 34]. Many state-of-the-art solutions rely on the *field value-invariant* exhibited by a data structure (i.e., a field with either a constant value or a value in a fixed range) as its signature [32, 40–43]. Unfortunately, many data structures cannot be covered by the value-invariant scheme.

We thus present a complementary scheme for data structure signatures, and instantiate this problem to Linux kernel data structures. Different from the value-invariant-based signatures, our approach, called SigGraph, uses a *graph structure rooted at a data structure* as its signature. More specifically, for a data structure with pointer field(s), each pointer field – identified by its offset from the start of the data structure – points to another data structure. Transitively, such points-to relations entail a graph structure rooted at the original data structure. We observe that data structures with pointer fields widely exist in OS kernels. For example, when compiling the whole package of Linux kernel 2.6.18-1, we found that over 40% of all data structures have pointer field(s). Compared with the field values of data structures, the "topology" of kernel data structures (formed by "points-to" relations) is more stable. As such, SigGraph can uniquely identify kernel data structures with pointers.

The basic idea behind SigGraph is to explore the inter-data structure points-to relations to generate non-isomorphic data structure signatures. A salient feature of SigGraph-based signatures is that they can be used for *brute force scanning*: Given an *arbitrary* kernel memory address x, a signature (more precisely, a memory scanner based on it) can decide if an instance of the corresponding data structure exists in the memory region starting at x.

As such, SigGraph is different from the global "top-down" scanning employed by many memory mapping techniques (e.g., those for software debugging [58] and kernel integrity checking [30, 56]). Global "top-down" scanning is enabled by building a global points-to graph for a subject program – rooted at its global variables and expanding to its entire address space. Instances of the program's data structures can then be identified by traversing the global graph starting from the root. On the other hand, brute force scanning is based on multiple, context-free points-to graphs – each rooted at a distinct data structure.

Unlike global scanning, brute force scanning does not require that a data structure instance be "reachable" from a global variable in order to be recognized; therefore achieving a higher level of robustness against attacks that tamper with such global reachability.

To enable brute force scanning, SigGraph faces the new issues of *data structure isomorphism*: the signatures of different data structures, if not judiciously determined, may be *isomorphic*, leading to false positives in data structure instance recognition. To address this problem, we formally define data structure isomorphism and develop an algorithm to compute unique, non-isomorphic signatures for kernel data structures. From the signatures, data structure-specific *kernel memory scanners* are automatically generated using context-free grammars. To improve the practicality of our solution, we propose a number of heuristics to handle practical issues (e.g., some pointers being `null`).

Meanwhile, we obtain two important observations when developing SigGraph: (1) The wealth of points-to relations between kernel data structures allows us to generate *multiple* signatures for the same data structure. This is particularly powerful when operating under malicious pointer mutation attacks, thus raising the bar to evade SigGraph. (2) The rich points-to relations also allow us to avoid complex, expensive points-to analysis of kernel source code for `void` pointer handling (e.g., as proposed in [30]). Distinct data structure signatures can be generated *without* involving the generic pointers.

SigGraph has the following key features:

- It models the topological invariants between a subject data structure and those directly or transitively reachable via points-to relations, and is able to generate *multiple* signatures for the same data structure. This is particularly powerful when operating under malicious pointer mutation attacks, and significantly raises the bar to avoid detection.

- It recognizes and formulates the challenge that different data structures may share isomorphic structural patterns such that false positives are induced if the invariants are not properly chosen; proposes a theoretically sound solution identifying signatures that are guaranteed not to cause false positives in ideal scenarios (e.g. pointers

are always not `null`); and develops a number of practical extensions to adapt the algorithm to real-world scenarios (e.g. some pointers may be `null`).

- It avoids complex, expensive points-to analysis for `void` pointer handling (e.g., in KOP [30]) as it can generate distinct signatures without involving those pointers. The graph-based signatures can often be described by *context-free-grammars* such that the scanners can be automatically generated to recognize data structure instances.

- The graph-based signatures can often be applied at any memory address x, and end users only need to perform pattern matching starting at x using the scanner for data structure T. This brute force scanning avoids the construction of (and dependence on) a *global* memory graph starting from the global variables and stack variables of a program/OS which is different from memory graph-based approaches such as KOP [30].

2.3 DIMSUM

As we have discussed, there are dead data structure instances in the memory such as the deallocated objects or those belonging to a dead process. It is necessary to have techniques to identify these data structure instances. However, the existing solutions critically depend on *memory mapping information*. For example, KOP [30], REWARDS [59] and SigGraph [31] all require that the pointers between data structures be resolvable (and thus trackable) in the memory image. KOP and REWARDS further require that each target data structure instance be reachable (via pointers) from global variables or variables on stack frames.

Unfortunately, such memory mapping information is not always available. Yet it is desirable for a computer forensics investigator to have the capability of uncovering meaningful forensics information from a set of memory pages *without* memory mapping information. For instance, imagine a cyber crime suspect runs and then terminates an application (e.g., a web browser), and even cleans up the privacy/history data in the disk in order not to leave any evidence. At that moment, however, some of the memory pages previously

belonging to the terminated application process may still exist for a non-trivial period of time – with intact content but without the corresponding page table or system symbol table. While these "dead" memory pages may contain data of forensic interest, existing memory mapping-based forensics techniques (e.g., [30, 31, 59]) will not be able to uncover them because, without memory mapping information, they will not be able to resolve and navigate through pointers in the dead pages.

In addition to the above scenario of "dead pages left by a terminated process," there are other computer forensics scenarios that require analyzing a partial memory image without memory mapping information. For example, after a sudden power-off, a subset of the memory pages belonging to a running process may still exist in the disk due to page swapping. But the memory mapping information maintained by the OS kernel for that process is lost. As another possibility, due to the fact that most existing memory forensics techniques depend on memory mapping information and on the completeness of a process' memory image, counter-measures may be taken by adversaries to inflict digital or even physical damages to the memory image of a computer. For example, it has been shown that advanced kernel-level attacks can be launched to disable the recovery of critical kernel objects from a memory image [31]. And some of those kernel objects contain memory mapping information for application processes. We envision that similar attacks can destroy the mapping information for application processes, disabling most existing techniques.

Therefore, we developed a new system, called DIMSUM, which is capable of uncovering semantic data instances of forensics interest from a set of memory pages without memory mapping information. In particular, DIMSUM remains effective even with an incomplete subset of memory pages of an application process. As such, DIMSUM differs from, and complements most existing approaches to memory forensics (e.g., [25, 30–32, 41, 42, 52, 53, 55, 59, 60]), where the primary focus is on extracting semantic information from *live* memory (either on-line or off-line). Many of these efforts (e.g., [25, 30, 31, 41, 53, 55, 59, 60]) rely on certain memory mapping information – such as the system symbol and page table – to search for variables and data structure instances in the memory that can be

reached directly or indirectly (e.g., by following the pointers between variables such as in KOP [30] and SigGraph [31]).

We also note that some of the existing approaches (e.g., [25, 32, 41, 42]) also leverage value-invariant signatures of data structures (e.g., "data structure field x having a special value or value range"). These techniques are effective if unique signatures can be generated for the subject data structures. Unfortunately, such a signature may not always exist for a data structure.

DIMSUM is based on *probabilistic inference*, which is widely used in computer vision (e.g., [61]), specification extraction (e.g., [62–64]), and software debugging (e.g., [62, 63, 65–68]). Given a set of memory pages and the definitions of the data structures of interest, DIMSUM is able to identify instances of the data structures in those pages. More specifically, by leveraging a probabilistic inference engine, our system automatically builds graphical models from the data structure specification and input page contents, and translates them into *factor graphs* [67], on which probabilistic inference will be carried out to extract target data structure instances quantified with probabilities.

The salient features of DIMSUM are as follows: (1) It recognizes data structure instances of interest with high confidence. Compared to brute force pattern matching methods, it consistently achieves a lower false positive rate. (2) It is robust in highly hostile memory forensics scenarios, where there is no memory mapping information and only an incomplete subset of memory pages are available. We evaluated DIMSUM using a number of real-world applications on Linux platform, and consistently demonstrated its effectiveness.

3. REWARDS: AUTOMATIC REVERSE ENGINEERING OF DATA STRUCTURE DEFINITIONS

In this chapter, we present the detailed design of the first component of our framework, REWARDS, which is a dynamic analysis based scheme to automatically reveal program data structures from binaries.

3.1 Overview

REWARDS aims to infer the data structure definitions defined in the binary code. It is an information flow based approach. Basically, for each memory location accessed by the program, it is tagged with a *timestamped type attribute*. At runtime, this attribute is propagated to other memory addresses and registers that share the same type in a forward fashion by following the program's runtime data flow. During the propagation, a variable's type gets resolved if it is involved in a type-revealing execution point.

3.1.1 Key Techniques

Besides leveraging the forward type propagation technique, to expand the coverage of program data structures, REWARDS involves the following key techniques.

- An on-line *backward type resolution* procedure where the types of some previously accessed variables are *recursively* resolved starting from a type sink. Since many variables are dynamically created and de-allocated at runtime, and the same memory location may be re-used by different variables, it is complicated to track and resolve variable types based on memory locations alone. Therefore, we constrain the resolution process by the timestamps of relevant memory locations such that

variables sharing the same memory location in different execution phases can be disambiguated.

- An off-line resolution procedure that complements the on-line procedure. Some variables cannot be resolved during their lifetime by our on-line algorithm. However, they may be resolved later when other variables having the same type are resolved. Hence, we propose an off-line backward resolution procedure to resolve the types of some "dead" variables.

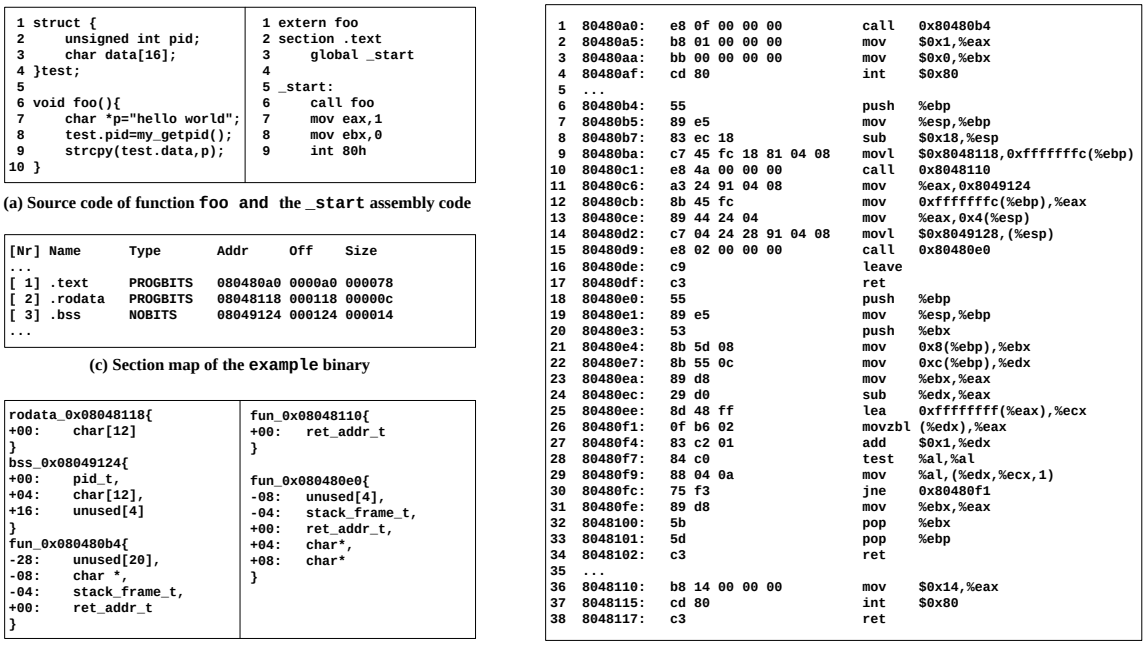
- A method for typed variable abstraction that maps multiple typed variable instances to the same static abstraction. For example, all N nodes in a linked list actually share the same type, instead of having N distinct types.

- A method that reconstructs the structural and semantic view of in-memory data, driven by the derived type definitions. Once a program's data structures are identified, it is still not clear exactly how the data structures would be laid out in memory, which would be a useful piece of knowledge in many application scenarios such as memory forensics. Our method creates an "organization chart" that illustrates the hierarchical layout of those data structures.

3.1.2 A Working Example

To illustrate how REWARDS works, we use a simple program compiled from the source code shown in Figure 3.1(a). According to the code snippet, the program has a global variable `test` (line 1-4) that consists of an `int` and a `char` array. It contains a function `foo` (line 6-10) that calls `my_getpid` and `strcpy` to initialize the global variable. The full disassembled code of the example is shown in Figure 3.1(b) (a dotted line indicates a "`NOP`" instruction). The address mapping of code and data is shown in Figure 3.1(c).

When `foo` is called during execution, it first saves `ebp` and then allocates `0x18` bytes of memory for the local variables (line 8 in Figure 3.1(b)), and then initializes one local variable (at address `0xfffffffc(%ebp)=ebp-4`) with an immediate value



```
 1 struct {
 2     unsigned int pid;
 3     char data[16];
 4 }test;
 5
 6 void foo(){
 7     char *p="hello world";
 8     test.pid=my_getpid();
 9     strcpy(test.data,p);
10 }
```

(a) Source code of function `foo` and the `_start` assembly code

```
 1 extern foo
 2 section .text
 3     global _start
 4
 5 _start:
 6     call foo
 7     mov eax,1
 8     mov ebx,0
 9     int 80h
```

```
[Nr] Name      Type       Addr     Off    Size
...
[ 1] .text     PROGBITS   080480a0 0000a0 000078
[ 2] .rodata   PROGBITS   08048118 000118 00000c
[ 3] .bss      NOBITS     08049124 000124 000014
...
```

(c) Section map of the `example` binary

```
rodata_0x08048118{
+00:    char[12]
}
bss_0x08049124{
+00:    pid_t,
+04:    char[12],
+16:    unused[4]
}
fun_0x080480b4{
-28:    unused[20],
-08:    char *,
-04:    stack_frame_t,
+00:    ret_addr_t
}
```

```
fun_0x08048110{
+00:   ret_addr_t
}

fun_0x080480e0{
-08:   unused[4],
-04:   stack_frame_t,
+00:   ret_addr_t,
+04:   char*,
+08:   char*
}
```

(d) Output of REWARDS

```
 1  80480a0:   e8 0f 00 00 00          call   0x80480b4
 2  80480a5:   b8 01 00 00 00          mov    $0x1,%eax
 3  80480aa:   bb 00 00 00 00          mov    $0x0,%ebx
 4  80480af:   cd 80                   int    $0x80
 5  ...
 6  80480b4:   55                      push   %ebp
 7  80480b5:   89 e5                   mov    %esp,%ebp
 8  80480b7:   83 ec 18                sub    $0x18,%esp
 9  80480ba:   c7 45 fc 18 81 04 08    movl   $0x8048118,0xfffffffc(%ebp)
10  80480c1:   e8 4a 00 00 00          call   0x8048110
11  80480c6:   a3 24 91 04 08          mov    %eax,0x8049124
12  80480cb:   8b 45 fc                mov    0xfffffffc(%ebp),%eax
13  80480ce:   89 44 24 04             mov    %eax,0x4(%esp)
14  80480d2:   c7 04 24 28 91 04 08    movl   $0x8049128,(%esp)
15  80480d9:   e8 02 00 00 00          call   0x80480e0
16  80480de:   c9                      leave
17  80480df:   c3                      ret
18  80480e0:   55                      push   %ebp
19  80480e1:   89 e5                   mov    %esp,%ebp
20  80480e3:   53                      push   %ebx
21  80480e4:   8b 5d 08                mov    0x8(%ebp),%ebx
22  80480e7:   8b 55 0c                mov    0xc(%ebp),%edx
23  80480ea:   89 d8                   mov    %ebx,%eax
24  80480ec:   29 d0                   sub    %edx,%eax
25  80480ee:   8d 48 ff                lea    0xffffffff(%eax),%ecx
26  80480f1:   0f b6 02                movzbl (%edx),%eax
27  80480f4:   83 c2 01                add    $0x1,%edx
28  80480f7:   84 c0                   test   %al,%al
29  80480f9:   88 04 0a                mov    %al,(%edx,%ecx,1)
30  80480fc:   75 f3                   jne    0x80480f1
31  80480fe:   89 d8                   mov    %ebx,%eax
32  8048100:   5b                      pop    %ebx
33  8048101:   5d                      pop    %ebp
34  8048102:   c3                      ret
35  ...
36  8048110:   b8 14 00 00 00          mov    $0x14,%eax
37  8048115:   cd 80                   int    $0x80
38  8048117:   c3                      ret
```

(b) Disassembly code of the `example` binary

Fig. 3.1.: An example showing how REWARDS works

`0x8048118` (line 9). Since `0x8048118` is in the address range of the `.rodata` section (it is actually the starting address of string "`hello world`"), `ebp-4` can be typed as a pointer, based on the heuristics that instruction executions using similar immediate values within a code or data section are considered type sinks. Note that the type of the pointer is not yet known. At line 10, `foo` calls `0x8048110`. Inside the body of the function invocation (lines 36-38), our algorithm detects a `getpid` system call (a type sink) with `eax` being `0x14` at line 36. The return value of the function call is resolved as `pid_t` type (i.e., register `eax` at line 11 is typed `pid_t`). When `eax` is copied to address `0x8049124` (a global variable in `.bss` section as shown in Figure 3.1(c)), the algorithm further resolves `0x8049124` as `pid_t`. Before the function call `0x80480e0` at line 15 (`strcpy`), the parameters are initialized in lines 12-14. As `ebp-4` has been typed as a pointer at line 9, the data flow in lines 12 and 13 dictates that location `esp+4` at line 13 is a pointer as well. At line 14, as `0x8049128` is in the global variable section and of a known type, location `esp` has an unknown pointer type. At line 15, upon the call to `strcpy` (a type sink),

both `esp` and `esp+4` are resolved to `char*`. Through a *backward* transitive resolution, `0x8049128` is resolved as `char`, `ebp-4` as `char*`, and `0x8048118` as `char`. Also at line 26, inside the function body of `strcpy`, the instruction "`movzbl (%edx),%eax`" can be used as another type sink as it moves between the `char` variables.

When the program finishes, we resolve all data types (including function arguments, and those implicit variables such as return address and stack frame pointer) as shown in Figure 3.1(d). The derived types for variables in `.rodata`, `.bss` and functions are presented in the figure. Each function is denoted by its entry address. `fun_0x080480b4`, `fun_0x08048110`, and `fun_0x080480e0` denote `foo`, `my_getpid`, and `strcpy`, respectively. The number before each derived type denotes the offset. The variables are listed in increasing order of their addresses. Type `stack_frame_t` indicates a frame pointer stored at that location. Type `ret_addr_t` means that the location holds a return address. Such semantic information is useful in applications such as vulnerability fuzz. Locations that are not accessed during execution are annotated with the `unused` type. In `fun_0x080480e0`, the two `char*` below the `ret_addr_t` represent the two actual arguments of `strcpy`.

3.2 Detailed Design

Now we describe the design of REWARDS. We first identify the type sinks used in REWARDS and then present the on-line type propagation and resolution algorithm, which will be enhanced by an off-line procedure that recovers more variable types not reported by the on-line algorithm. Finally, we present a method to construct a typed hierarchical view of memory layout.

3.2.1 Type Sinks

A type sink is an execution point of a program where the types (including semantics) of one or more variables can be directly resolved. In REWARDS, we identify three categories of type sinks: (1) system calls, (2) standard library calls, and (3) type-revealing instructions.

System calls. Most programs request OS services via system calls. Since system call conventions and semantics are well-defined, the types of arguments of a system call are known from the system call's specification. By monitoring system call invocations and returns, REWARDS can determine the types of parameters and return value of each system call at runtime. For example, in Linux, based on the system call number in register `eax`, REWARDS will be able to type the parameter-passing registers (i.e., `ebx`, `ecx`, `edx`, `esi`, `edi`, and `ebp`, if they are used for passing the parameters). From this type sink, REWARDS will further type those variables that are determined to have the same type as the parameter passing registers. Similarly, when a system call returns, REWARDS will type register `eax` and, from there, those having the same type as `eax`. In our type propagation and resolution algorithm (Section 3.2.2), a type sink will lead to the recursive type resolution of relevant variables accessed before and after the type sink.

Standard library calls. With well-defined API, standard library calls are another category of type sink. For example, the two arguments of `strcpy` must both be of the `char*` type. By intercepting library function calls and returns, REWARDS will type the registers and memory variables involved. Standard library calls tend to provide richer type information than system calls. For example, Linux-2.6.15 has 289 system calls, whereas `libc.so.6` contains 2,016 functions (note some library calls wrap system calls).

Type-revealing instructions. A number of machine instructions that require operands of specific types can serve as type sinks. Examples in x86 are as follows: (1) *String instructions* perform byte-string operations, such as moving and storing (`MOVS/B/D/W`, `STOS/B/D/W`), loading (`LOADS/B/D/W`), comparison (`CMPS/B/D/W`), and scanning (`SCAS/B/D/W`). Note that `MOVZBL` is also used in string movement. (2) *Floating-point instructions* operate on floating-point, integer, and binary coded decimal operands (e.g. `FADD`, `FABS`, and `FST`). (3) *Pointer-related instructions* reveal pointers. For a `MOV` instruction with an indirect memory access operand (e.g., `MOV (%edx), %ebx` or `MOV [mem], %eax`), the value held in the source operand must be a pointer. Meanwhile, if the target address is within the range of data sections, such as `.stack`, `.heap`, `.data`, `.bss` or `.rodata`, the pointer must be a data pointer. If it is in the range of `.text`

(including library code), the pointer must be a function pointer. Note that the concrete type of such a pointer will be resolved through other constraints.

3.2.2 Online Type Propagation and Resolution Algorithm

Given a binary program, our algorithm reveals variable types, including both syntactic types (e.g., `int` and `char`) and semantics (e.g., `return address`), by propagating and resolving the type information along the data flow during program execution. Each type sink encountered leads to both direct and transitive type resolution of variables. More specifically, at the binary level, variables exist in either memory locations or registers without their symbolic names. Hence, the goal of our algorithm is to type these memory addresses and registers. We attach three *shadow variables* – as the type attribute – to each memory address at the byte granularity (registers are treated similarly): (1) *constraint set* is a set of other memory addresses that should have the same type as this address; (2) *type set* stores the set of resolved types of the address[1], including both syntactic and semantic types; (3) *timestamp* records the birth time of the variable currently in this address. For example, the timestamp of a stack variable is the time when the stack frame is allocated. Timestamps are needed because the same memory address may be reused by multiple variables (e.g., the same stack memory being reused by stack frames of different method invocations). More precisely, a variable instance should be uniquely identified by a tuple <address, timestamp>. These shadow variables are updated during program execution, depending on the semantics of executed instructions.

The on-line type propagation and resolution algorithm, Algorithm 1 on the previous page, takes appropriate actions to resolve types on the fly according to the instruction being executed. For a memory address or a register v, its constraint set is denoted as S_v, which is a set of <address, timestamp> tuples, and each representing a variable instance that should have the same type as v; its type set T_v represents the resolved types for v; and the birth time of the current variable instance is denoted as ts_v.

[1]We need a set to store the resolved types because one variable may have multiple compatible types.

Algorithm 1 *On-line Type Propagation and Resolution*

```
1: /* S_v: constraint set for memory cell (or register) v; T_v: type set of v; ts_v: time stamp of v; MOV(v,w): moving v to w;
   BIN_OP(v,w,d): a binary operation that computes d from v and w; Get_Sink_Type(v,i): retrieving the type of argument/operand v
   from sink i; ALLOC(v,n): allocating a memory region starting from v with size n – the memory region may be a stack frame or a
   heap struct; FREE(v,n): freeing a memory region – this may be caused by eliminating a stack frame or de-allocating a heap struct*/
2: Instrument(i){
3:     case i is a Type_Sink:
4:         for each operand v
5:             T ← Get_Sink_Type(v, i)
6:             Backward_Resolve (v, T)
7:     case i has indirect memory access operand o
8:             T_o ← T_o ∪ {pointer_type_t}
9:     case i is MOV(v, w):
10:        if w is a register
11:            S_w ← S_v
12:            T_w ← T_v
13:        else
14:            Unify(v, w)
15:    case i is BIN_OP(v, w, d):
16:        if pointer_type_t ∈ T_v
17:            Unify(d, v)
18:            Backward_Resolve (w, {int, pointer_index_t})
19:        else
20:            Unify3(d, v, w)
21:    case i is ALLOC(v, n):
22:        for t=0 to n − 1
23:            ts_{v+t} ← current timestamp
24:            S_{v+t} ← φ
25:            T_{v+t} ← φ
26:    case i is FREE(v, n):
27:        for t=0 to n − 1
28:            a ← v+t
29:            if (T_a) log (a, ts_a, T_a)
30:            log (a, ts_a, S_a)
31: }
32: Backward_Resolve(v,T){
33:    for < w, t > ∈ S_v
34:        if (T ⊄ T_w and t ≡ ts_w) Backward_Resolve(w,T-T_w)
35:    T_v ← T_v ∪ T
36: }
37: Unify(v,w){
38:    Backward_Resolve(v, T_w-T_v)
39:    Backward_Resolve(w, T_v-T_w)
40:    S_v ← S_v ∪ {< w, ts_w >}; S_w ← S_w ∪ {< v, ts_v >}
41: }
```

1. If the current execution point i is a type sink (line 3). The arguments/operands/return values of the sink will be directly typed according to the sink's definition (**Get_Sink_Type()** on line 5)[2]. Type resolution is then triggered by calling the recursive method **Backward_Resolve()**. The method recursively types all variables that should have the same type (lines 32-36): It tests if each variable w in the constraint set of v has been resolved as type T of v. If not, it recursively calls itself to type all the variables that should have the same type as w. Note that at line 34, it checks if the current birth timestamp of w is equal to the one stored in the constraint set to ensure the memory has not been re-used by a different variable. If w is re-used ($t \neq ts_w$), the algorithm does not resolve the current w. Instead, the resolution is done by a different off-line procedure (Section 3.2.3). Since variable types are resolved according to constraints derived from data flows in the past, we call this step backward type resolution.

2. If i contains an indirect memory access operand o (line 7), either through registers (e.g., using `(%eax)` to access the address designated by `eax`) or memory (e.g., using `[mem]` to indirectly access the memory pointed to by `mem`), then the corresponding operand will have a pointer *type tag* (`pointer_type_t`) as a new element in T_o.

3. If i is a move instruction (line 9), there are two cases to consider. In particular, if the destination operand w is a register, then we just move the properties (i.e., the S_v and T_v) of the source operand to the destination (i.e., the register); otherwise, we need to unify the types of the source and destination operands because the destination is now a memory location that may have already contained some resolved types. The intuition is that the source operand v should have the same type as the destination operand w if the destination is a memory address. Hence, the algorithm calls method **Unify()** to unify the types of the two. In **Unify()** (lines 37-41), the algorithm first unions the two type sets by performing backward resolution at lines 38 and 39. Intuitively, the call at line 38 means that if there are any new types in T_w that are not in T_v (i.e. T_w-T_v), those new types need to be propagated to v and transitively

[2]The sink's definition also reveals the semantics of some arguments/operands, e.g., a PID.

to all variables that share the same type as v, mandated by v's constraint set. Such unification is not performed if the w is a register to avoid over-aggregation.

4. If i is a binary operation, the algorithm first tests if an operand has been identified as a pointer. If so, it must be a pointer arithmetic operation, the destination must have the same type as the pointer operand and the other operand must be a pointer index – denoted by a semantic type `pointer_index_t` (line 18). The semantic type is useful in vulnerability fuzz to overflow buffers. If i is not related to pointers, the three operands shall have the same type. The method **Unify3()** unifies three variables. It is very similar to **Unify()** and hence not shown. Note that in cases where the binary operation implicitly casts the type of some operand (e.g., an addition of a float and an integer), the unification induces over-approximation (e.g., associating the float point type with the integer variable). In practice, we consider such cases reasonable and allow multiple types for one variable as long as they are compatible.

5. If i allocates a memory region (line 21), either a stack frame or a heap struct, the algorithm updates the birth time stamps of all the bytes in the region and resets the memory constraint set (S_v) and type set (T_v) to empty. By doing so, we prevent the type information of the old variable instance from interfering with that of the new instance at the same address.

6. If i frees a memory region (line 26), the algorithm traverses each byte in the region and prints out the type information. In particular, if the type set is not empty, it is emitted. Otherwise, the constraint set is emitted. Later, the emitted constraints will be used in the off-line procedure (Section 3.2.3) to resolve more variables.

Example. Table 3.1 presents an example of executing our algorithm. The first column shows the instruction trace with the numbers denoting timestamps. The other columns show the type sets and the constraint sets after each instruction execution for three sample variables, namely, the global variable $g1$ and two local variables $l1$ and $l2$. For brevity, we abstract the calling sequence of `strcpy` to a `strcpy` instruction. After the execution

enters method M at timestamp 10, the local variables are allocated and hence both $l1$ and $l2$ have the birth time of 10. The global variable $g1$ has the birth time of 0. After the first `mov` instruction, the type sets of $g1$ and $l1$ are unified. Since neither was typed, the unified type set remains empty. Moreover, $l1$, together with its birth time 10, is added to the constraint set of $g1$ and vice versa, denoting they should have the same type. Similar actions are taken after the second `mov` instruction. Here, the constraint set of $l1$ has both $g1$ and $l2$. The `strcpy` invocation is a type sink and $g1$ must be of type `char*`, the algorithm performs the backward resolution by calling **Backward_Resolve()**. In particular, the variable in S_{g1}, i.e. $l1$, is typed to `char*`. Note that the timestamp 10 matches ts_{l1}, indicating the same variable is still alive. Transitively, the variables in S_{l1}, i.e. $g1$ and $l2$, are resolved to the same type. Note that if the backward resolution was not conducted, we would not be able to resolve the type of $l2$ because when the move from $l1$ to $l2$ (timestamp 12) occurred, $l1$ was not typed and hence $l2$ was not typed.

Table 3.1: An example of running the online algorithm. Variable $g1$ is a global, $l1$ and $l2$ are locals.

Instruction	T_{g1}	S_{g1}	ts_{g1}	T_{l1}	S_{l1}	ts_{l1}	T_{l2}	S_{l2}	ts_{l2}
10 enter M	ϕ	ϕ	0	ϕ	ϕ	10	ϕ	ϕ	10
11 mov $g1$, $l1$	ϕ	{<$l1$,10>}	0	ϕ	{<$g1$,0>}	10	ϕ	ϕ	10
12 mov $l1$, $l2$	ϕ	{<$l1$,10>}	0	ϕ	{<$g1$,0>,<$l2$,10>}	10	ϕ	{<$l1$,10>}	10
...	...	...	...	...	...	...	...	...	...
100 strcpy($g1$,)	{char*}	{<$l1$,10>}	0	{char*}	{<$g1$,0>,<$l2$,10>}	10	{char*}	{<$l1$,10>}	10

Table 3.2: An example of running the off-line type resolution procedure. The execution before timestamp 12 is the same as Table 3.1. Method N reuses $l1$ and $l2$

Instruction	T_{g1}	S_{g1}	ts_{g1}	T_{l1}	S_{l1}	ts_{l1}	T_{l2}	S_{l2}	ts_{l2}
...	...	...	...	...	...	...	...	...	...
12 mov $l1$, $l2$	ϕ	{<$l1$,10>}	0	ϕ	{<$g1$,0>,<$l2$,10>}	10	ϕ	{<$l1$,10>}	10
13 Exit M	ϕ	{<$l1$,10>}	0	ϕ	{<$g1$,0>,<$l2$,10>}	10	ϕ	{<$l1$,10>}	10
...	...	...	...	...	...	...	...	...	...
99 Enter N	ϕ	{<$l1$,10>}	0	ϕ	ϕ	99	ϕ	ϕ	99
100 strcpy($g1$,..)	{char*}	{<$l1$,10>}	0	ϕ	ϕ	99	ϕ	ϕ	99

3.2.3 Off-line Type Resolution

Most variables accessed during the binary's execution can be resolved by our online algorithm. However, there are still some cases in which, when a memory variable gets freed (and its information gets emitted to the log file), its type is still unresolved. We realize that there may be enough information from later phases of the execution to resolve those variables. We propose an off-line procedure to be performed *after* the program execution terminates. It is essentially an off-line version of the **Backward_Resolve()** method in Algorithm 1. The difference is that it has to traverse the log file to perform the recursive resolution.

Consider the example in Table 3.2. It shares the same execution as the example in Table 3.1 before timestamp 13. At time instance 13, the execution returns from M, de-allocating the local variables $l1$ and $l2$. According to the online algorithm, their constraint sets are emitted to a log file since neither is typed at that point. Later at timestamp 99, another method N is called. Assume it reuses $l1$ and $l2$, namely, N allocates its local variables at the locations of $l1$ and $l2$. The birth time of $l1$ and $l2$ becomes 99. Their type sets and constraint sets are reset. When the sink is encountered at 100, $l1$ and $l2$ are not typed as their current birth timestamp is 99, not 10 as in S_{g1}, indicating they are re-used by other variables. Fortunately, the variable represented by $< l1, 10 >$ can be found in the log and hence resolved. Transitively, $< l2, 10 >$ can be resolved as well.

3.2.4 Typed Variable Abstraction

Our algorithm is able to annotate memory locations with syntax and semantics. However, multiple variables may occupy the same memory location at different times and a static variable may have multiple instances at runtime[3]. Hence it is important to organize the inferred type information according to abstract, location-independent variables other than specific memory locations. In particular, primitive global variables are represented by

[3]A local variable has the same life time of a method invocation, and a method can be invoked multiple times, giving rise to multiple instances.

their offsets to the base of the global sections (e.g., `.data` and `.bss` sections). Stack variables are abstracted by the offsets from their residence activation record, which is represented by the function name (as shown in Figure 3.1).

For heap variables, we use the execution context, i.e., the PC (instruction address) of the allocation point of a heap structure plus the call stack at that point, as the abstraction of the structure. The intuition is that the heap structure instances allocated from the same PC in the same call stack should have the same type. The fields of the structure are represented by the allocation site and field offsets. As an allocated heap region may be an array of a data structure, we use the recursion detection heuristics in [19] to detect the array size. Specifically, the array size is approximated by the maximum number of accesses *by the same PC* to unique memory locations in the allocated region. The intuition is that array elements are often accessed through a loop in the source code and the same instruction inside the loop body often accesses the same field across all array elements. Finally, if heap structures allocated from different sites have the same field types, we will heuristically cluster these heap structures into one abstraction.

3.2.5 Constructing Hierarchical View of In-Memory Data

An important feature of REWARDS is to construct a hierarchical view of a memory snapshot, in which the primitive syntax of individual memory locations, as well as their semantics and the integrated hierarchical structure are visually represented. This is highly desirable in applications like memory forensics as interesting queries (e.g., "`find all IP addresses`"), can be easily answered by traversing the view. So far, REWARDS is able to reverse engineer the syntax and semantics of data structures, represented by their abstractions. Next, we present how we leverage such information to construct a hierarchical view.

Our method works as follows. It first types the top level global variables. In particular, a root node is created to represent a global section. Individual global variables are represented as children of the root. The edges are annotated with offset, size, primitive

type, and semantics of the corresponding children. If a variable is a pointer, the algorithm further recursively constructs the sub-view of the data structure being pointed to, leveraging the derived type of the pointer. For instance, assume a global pointer p is of type `T*`, our method creates a node representing the region pointed to by p. The region is typed based on the reverse engineered definition of `T`. The recursive process terminates when none of the fields of a data structure is a pointer. The stack is similarly handled: a root node is created to represent each activation record. Local variables of the record are denoted as children nodes. Recursive construction is performed until all memory locations through pointers are traversed. Note that all live heap structures can be reached (transitively) through a global pointer or a stack pointer. Hence, the above two steps essentially also construct the structural views of live heap data.

Our method can also type some of the unreachable memory regions, which represent "dead" data structures (e.g., activation records of previous method invocations whose space has been freed but not reused.) Such dead data is as important as live data as they disclose what had happened in the past. In particular, our method scans the stack beyond the current activation record to identify any pointers to the code section, which often denote return addresses of method invocations. With a return address, the function invocation can be identified and we can follow the aforementioned steps to type the activation record.

3.3 Implementation

We implemented REWARDS using PIN-2.6 [69], with 12.1K lines (LOC) of C code and 1.2K LOC of Python code. REWARDS is able to reveal variable semantics. In our implementation, variable semantics are represented as special *semantic tags* complementary to regular type tags such as `int` and `char`. Both semantic tags and regular tags are stored in the variable's type set. Tags are enumerated to save space. The vast diversity of program semantics makes it infeasible to consider them all. Since we are mainly interested in forensics and security applications, we focus on the following semantic tags: (1) file system related (e.g., FILE pointer, file descriptor, file name, file status); (2)

network communication related (e.g., socket descriptor, IP address, port, receiving and sending buffer, host info, msghdr); and (3) operating systems related (e.g., PID, TID, UID, system time, system name, and device info).

Meanwhile, we introduce some of our own semantic tags, such as `ret_addr_t` indicating that a memory location is holding a return address, `stack_frame_t` indicating that a memory location is holding a stack frame pointer, `format_string_t` indicating that a string is used in format string argument, and `malloc_arg_t` indicating an argument of `malloc` function (similarly, `calloc_arg_t` for `calloc` function, etc.). Note that these tags reflect the properties of variables at those specific locations and hence do not participate in the type information propagation. They can bring important benefits to our targeted applications.

REWARDS needs to know the program's address space mapping, which will be used to locate the addresses of global variables and detect pointer types. In particular, REWARDS checks the target address range when determining if a pointer is a function pointer or a data pointer. Thus, when a binary starts executing with REWARDS, we first extract the coarse-grained address mapping from the `/proc/pid/maps` file, which defines the ranges of code and data sections including those from libraries, and the ranges of stack and heap (at that time). Then for each detailed address mapping such as `.data`, `.bss` and `.rodata` for all loaded files (including libraries), we extract the mapping using the API provided by PIN when the corresponding image file is loaded.

3.4 Evaluation

We have performed two sets of experiments to evaluate REWARDS: one is to evaluate its correctness, and the other is to evaluate its time and space efficiency. All the experiments were conducted on a machine with two 2.13Ghz Pentium processors and 2GB RAM running Linux kernel 2.6.15.

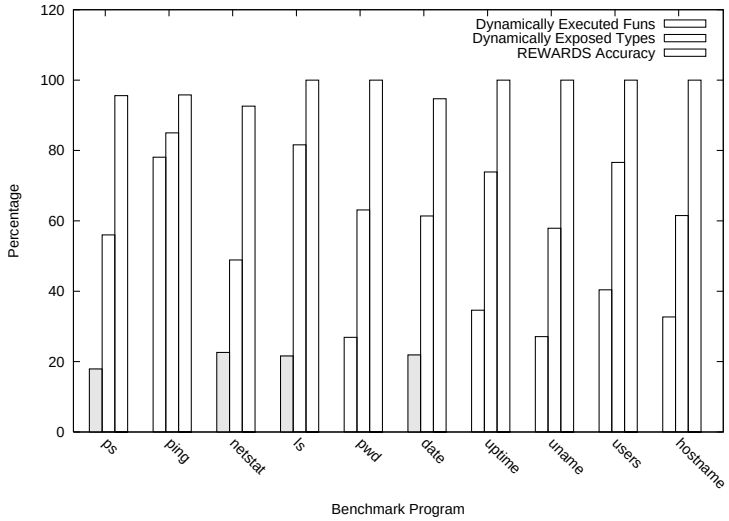
We select 10 widely used utility programs from the following packages: procps-3.2.6 (with 19.1K LOC and containing command `ps`), iputils-20020927 (with 10.8K LOC and

containing command `ping`), net-tools-1.60 (with 16.8K LOC and containing `netstat`), and coreutils-5.93 (with 117.5K LOC and containing the remaining test commands such as `ls`, `pwd`, and `date`). The reason for selecting these programs is that they contain many data structures related to the operating system and network communications. We run these utilities without command line option except `ping`, which is run with a localhost and a packet count 4 option.

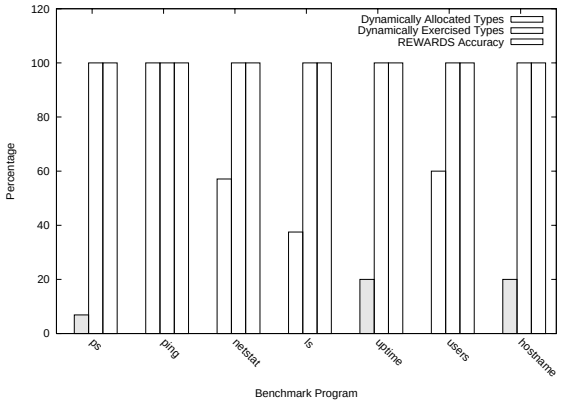
3.4.1 Effectiveness

To evaluate the reverse engineering accuracy of REWARDS, we compare the derived data structure types with those declared in the program source code. To acquire the oracle information, we recompile the programs with debugging information, and then use `libdwarf` [70] to extract type information from the binaries. The `libdwarf` library is capable of presenting the stack and global variable mappings after compilation. For instance, global variables scattering in various places in the source code will be organized into a few data sections. The library allows us see the organization. In particular, `libdwarf` extracts stack variables by presenting the mapping from their offsets in the stack frame and the corresponding types. For global variables, the output by `libdwarf` is program virtual addresses and their types. Such information allows us to conduct direct and automated comparison. Note that we only verify the types in `.data`, `.bss`, and `.rodata` sections, other global data in sections such as `.got`, `.ctors` are not verified. For heap variables, since we use the execution context at allocation sites as the abstract representation, given an allocation context, we can locate it in the disassembled binary, and then correlate it with program source code to identify the heap data structure definition, and finally compare it with REWARDS's output. Although REWARDS extracts variable types for the entire program address space (including libraries), we only compare the results for user-level code.

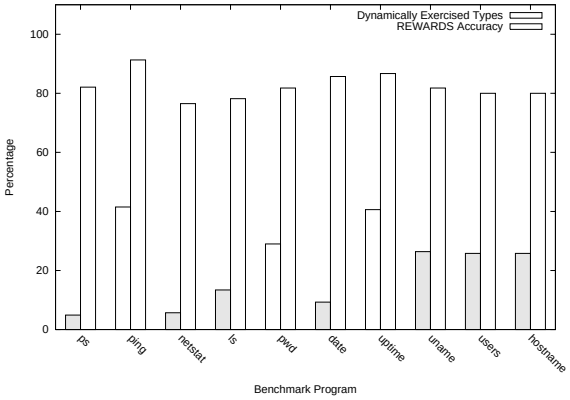
The result for stack variables is presented in Figure 3.2(a). The figure presents the percentage of (1) functions that are actually executed, (2) data structures that are used in the



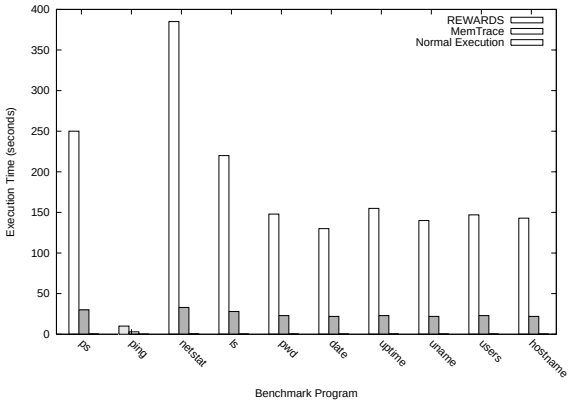
(a) Accuracy on Stack Variables



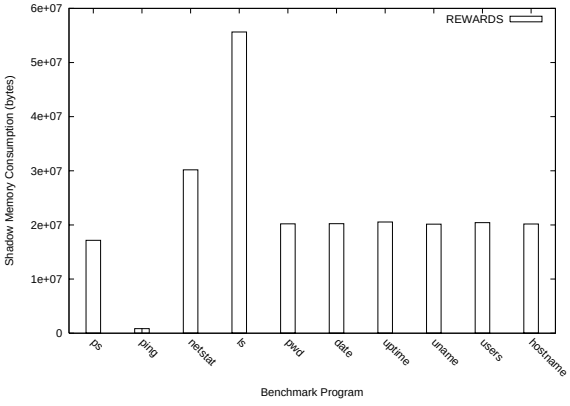
(b) Accuracy on Heap Variables



(c) Accuracy on Global Variables



(d) Performance Overhead



(e) Space Overhead

Fig. 3.2.: Evaluation results for REWARDS accuracy and efficiency

executed functions (over all structures declared in those functions), and (3) data structures whose types are accurately recovered by REWARDS (over those in (2)). At runtime, it is often the case that even though a buffer is defined in the source code with size n, only part of the n bytes are used. Consequently, only those used are typed (the others are considered `unused`). We consider the buffer is correctly typed if its bytes are either correctly typed or `unused`. From the figure, we can observe that, due to the nature of dynamic analysis, not all functions or data structures in a function are exercised and hence amenable to REWARDS. More importantly, REWARDS achieves an average of 97% accuracy (among these benchmarks) for the data structures that get exercised. For heap variables, the result is presented in Figure 3.2(b), the bars are similarly defined. REWARDS's output perfectly matches the types in the original definitions when they are exercised. Note some of the benchmarks are missing in Figure 3.2(b) (e.g., `date`) because their executions do not allocate any user-level heap structures. The result for global variables is presented in Figure 3.2(c), and REWARDS achieves over 85% accuracy.

To explain why REWARDS cannot achieve 100% accuracy, we carefully examined the benchmarks and identified the following three reasons:

- **Hierarchy loss**. If a hierarchical structure becomes flat after compilation, we are not able to identify its hierarchy. This happens to structures declared as global variables or stack variables. And the binary never accesses such a variable using the base address plus a local offset. Instead, it directly uses a global offset (starting from the base address of the global data section or a stack frame). In other words, multiple composite structures are flattened into one large structure. In contrast, such flattening does not happen to heap structures.

- **Path-sensitive memory reuse**. This often happens to stack variables. In particular, the compiler might assign different local variables declared in different program paths to the same memory address. As a result, the types of these variables are undesirably unified in our current design. A more thorough design would use a *path-sensitive* local offset to denote a stack variable.

- **Type cast**. It is possible that a variable type is casted to another one. For example, a float type variable could be casted to an integer type. As such, we will observe different semantic use of one variable, and if this variable is propagated to others, we will over propagate the types. Currently, we do not have a sound solution to this problem, and we just conservatively propagate the types.

Despite the imperfect accuracy, REWARDS still suits our targeted application scenarios, i.e., memory forensics and vulnerability fuzzing. For example, although REWARDS outputs a flat layout for all global and stack variables, we can still conduct vulnerability fuzzing because the absolute offsets of these variables are sufficient; and we can still construct hierarchical views of memory images as pointer types can be obtained.

3.4.2 Performance Overhead

We also measured the time and space overhead of REWARDS. We compared it with (1) a standard memory trace tool, MemTrace (shipped along with PIN-2.6) and (2) the normal execution of the program, to evaluate the performance overhead. The result is shown in Figure 4.8. Note the normal execution data is nearly not visible in this figure because they are very small (roughly at the 0.01 second level). We can observe that REWARDS causes slow-down in the order of ten times compared with MemTrace, and in the order of thousands (or tens of thousands) times compared with the normal execution.

For space overhead, we are interested in the space consumption by shadow type sets and constraint sets. Hence, we track the peak value of the shadow memory consumption. The result is shown in Figure 3.2(e). We can observe that the shadow memory consumption is around 10 Mbytes for these benchmarks. A special case is `ping`, which uses much less memory. The reason is that it has fewer function calls and memory allocations, which is also why it runs much faster than the other programs shown in Figure 4.8.

3.5 Summary

In this chapter, we have presented REWARDS, a reverse engineering system that automatically reveals data structures in a binary based on dynamic execution. REWARDS involves an algorithm that performs data flow-based type attribute forward propagation and backward resolution. Driven by the type information derived, REWARDS is also capable of reconstructing the structural and semantic view of in-memory data layout. Our evaluation using a number of real-world programs indicates that REWARDS achieves over 80% accuracy in revealing data structures accessed during an execution.

4. SIGGRAPH: DISCOVERING DATA STRUCTURE INSTANCES USING GRAPH-BASED SIGNATURES

In this chapter, we present the design of SigGraph, the second component in our framework, which aims to discovering data structure instances by scanning memory with the corresponding data structure signatures. To this end, it explores the points-to relation between data structures as signatures and enables the brute force scanning of memory. Brute force scanning requires effective, robust signatures of kernel data structures. Existing approaches often use the value invariants of certain fields as data structure signatures. However, they do not fully exploit the rich points-to relations between data structures. In our technique, a signature is a graph rooted at the subject data structure with edges reflecting the points-to relations with other data structures.

We first formally define our problem in Section 4.1, then present the detailed techniques on how we generate such graph based signatures from Section 4.2 to Section 4.5, followed we present the evaluation result in Section 4.7. Finally we conclude in Section 4.8.

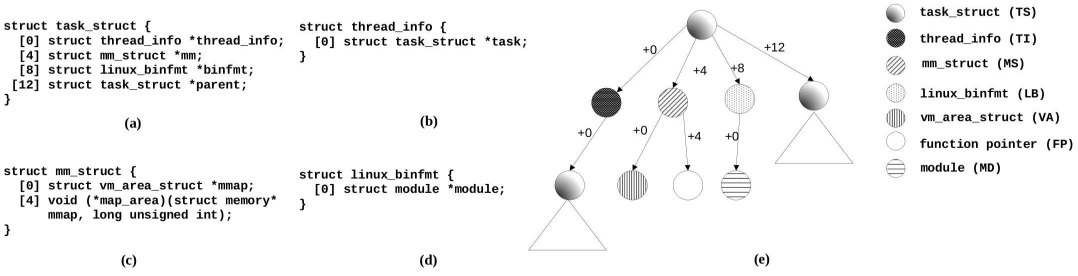


Fig. 4.1.: A working example of kernel data structures and a graph-based data structure signature. The triangles indicate recursive definitions

4.1 Problem Statement

As described in Chapter 2, the goal of SigGraph is to infer the relevant data structure instances given a memory dump. Basically, it exploits the inter-data structure points-to relations to generate non-isomorphic data structure signatures. To better understand the key idea behind SigGraph, let consider seven simplified Linux kernel data structures, four of which are shown in Figure 4.1(a)-(d). In particular, `task_struct(TS)` contains four pointers to `thread_info(TI)`, `mm_struct(MS)`, `linux_binfmt(LB)`, and `TS`, respectively. `TI` has a pointer to `TS` whereas `MS` has two pointers: One points to `vm_area_struct(VA)` (not shown in the figure) and the other is a function pointer. `LB` has one pointer to `module(MD)`.

At runtime, if a pointer is not `null`, its target object should have the type of the pointer. Let $S_T(x)$ denote a boolean function that decides if the memory region starting at x is an instance of type T and let $*x$ denote the value stored at x. Take `task_struct` data structure as an example, we have the following rule, assuming all pointers are not `null`.

$$\begin{aligned} S_{\mathtt{TS}}(x) \quad \rightarrow \quad & S_{\mathtt{TI}}(*(x+0)) \ \wedge S_{\mathtt{MS}}(*(x+4)) \wedge \\ & S_{\mathtt{LB}}(*(x+8)) \ \wedge S_{\mathtt{TS}}(*(x+12)) \end{aligned} \tag{4.1}$$

It means that if $S_{\mathtt{TS}}(x)$ is true, then the four pointer fields must point to regions with the corresponding types and hence the boolean functions regarding these fields must be true. Similarly, we have the following

$$S_{\mathtt{TI}}(x) \quad \rightarrow \quad S_{\mathtt{TS}}(*(x+0)) \tag{4.2}$$

$$S_{\mathtt{MS}}(x) \quad \rightarrow \quad S_{\mathtt{VA}}(*(x+0)) \wedge S_{\mathtt{FP}}(*(x+4)) \tag{4.3}$$

$$S_{\mathtt{LB}}(x) \quad \rightarrow \quad S_{\mathtt{MD}}(*(x+0)) \tag{4.4}$$

for `thread_info`, `mm_struct`, and `linux_binfmt`, respectively. Substituting symbols in rule (4.1) using rules (4.2), (4.3) and (4.4), we further have

$$\begin{aligned} S_{\mathtt{TS}}(x) \rightarrow \quad & S_{\mathtt{TS}}(*(*(x+0)+0)) \ \wedge \ S_{\mathtt{VA}}(*(*(x+4)+0)) \wedge \\ & S_{\mathtt{FP}}(*(*(x+4)+4)) \ \wedge \ S_{\mathtt{MD}}(*(*(x+8)+0))) \\ & \wedge S_{\mathtt{TS}}(*(x+12)) \end{aligned} \tag{4.5}$$

The rule corresponds to the graph shown in Figure 4.1(e), where the nodes represent pointer fields with their shapes denoting pointer types; the edges represent the points-to relations with their weights indicating the pointers' offsets; and the triangles represent recursive occurrences of the same pattern. It means that if the memory region starting at x is an instance of `task_struct`, the layout of the region must follow the graph's definition. Note that the inference of rule (4.5) is from left to right. However, we observe that the graph is so unique that the reverse inference ("bottom-up") tends to be true. In other words, we can use the graph as the signature of `task_struct` and perform the *reverse* inference as follows.

$$\begin{aligned} S_{\mathsf{TS}}(x) \leftarrow \quad & S_{\mathsf{TS}}(*(*(x+0)+0)) \ \wedge \ S_{\mathsf{VA}}(*(*(x+4)+0)) \wedge \\ & S_{\mathsf{FP}}(*(*(x+4)+4)) \ \wedge \ S_{\mathsf{MD}}(*(*(x+8)+0))) \\ & \wedge S_{\mathsf{TS}}(*(x+12)) \end{aligned} \tag{4.6}$$

Different from the global memory mapping techniques (e.g., [25, 30, 52, 53, 55, 56, 58]) SigGraph aims at deriving unique signatures for *individual* data structures for brute force kernel memory scanning. Hence we face the following new challenges:

- **Avoiding signature isomorphism** Given a static data structure definition, we aim to construct its points-to graph as shown in the `task_struct` example. However, it is possible that two distinct data structures may lead to *isomorphic* graphs that cannot be used to distinguish instances of the two data structures. Hence our new challenge is to identify the sufficient and necessary conditions to avoid signature isomorphism between data structures.

- **Generating signatures** Meanwhile it is possible that one data structure may have *multiple* unique signatures, depending on how (especially, how deep) the points-to edges are traversed when generating a signature. In particular, among the valid signatures of a data structure, finding the minimal signature that has the smallest size while retaining uniqueness (relative to other data structures) is a combinatorial optimization problem. Finally, it is desirable to *automatically* generate a scanner for each signature that will perform the corresponding data structure instance recognition on a memory image.

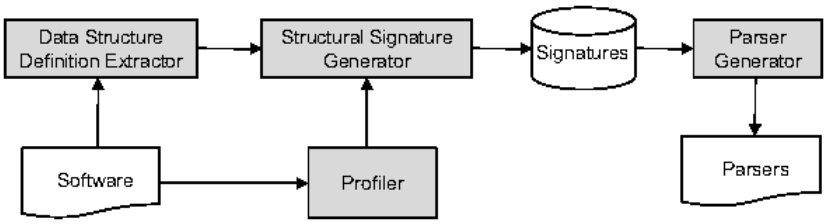


Fig. 4.2.: SigGraph system overview

- **Improving recognition accuracy** Although statically a data structure may have a unique signature graph, at runtime, pointers may be `null` whereas non-pointer fields may have pointer-like values. As a result the data structure instances in a memory image may not fully match the signature. We need to handle such issues to improve recognition accuracy.

An overview of the SigGraph is shown in Figure 4.2. It consists of four key components: (1) data structure definition extractor, (2) dynamic profiler, (3) signature generator, and (4) scanner generator. To generate signatures, SigGraph first extracts data structure definitions from the OS source code. This is done automatically through a compiler pass (Section 4.2). To handle practical issues such as `null` pointers and `void*` pointers, the *profiler* identifies problematic pointer fields via dynamic analysis (Section 4.5). The *signature generator* checks if non-isomorphic signatures exist for the data structures and if so, generates such signatures (Section 4.3). The generated signatures are then automatically converted to the corresponding kernel memory scanners (Section 4.4), which are the "product" shipped to users. A user will simply run these *scanners* to perform brute-force scanning over a kernel memory image (either memory dump or live memory), with the output being the instances of the data structures in the image.

4.2 Data Structure Definition Extraction

SigGraph's *data structure definition extractor* adopts a compiler-based approach, where the compiler pass is devised to walk through the source code and extract data structure

definitions. It is robust as it is based on a full-fledged language front-end. In particular, our development is in `gcc-4.2.4`. The compiler pass takes abstract syntax trees (ASTs) as input as they retain substantial symbolic information [71]. The compiler-based approach also allows us to handle data structure in-lining, which occurs when a data structure has a field that is of the type of another structure; After compilation, the fields in the inner structure become fields in the outer structure. Furthermore, we can easily see through type aliases introduced by `typedef` via ASTs.

The output of the compiler pass is the data structure definitions – with offset and type for each field – extracted in a canonical form. The pass is inserted into the compilation work-flow right after data structure layout is finished (in `stor-layout.c`). During the pass, the AST of each data structure is traversed. If the data structure type is `struct` or `union`, its field type, offset, and size information is dumped to a file. To precisely reflect the field layout after in-lining, we flatten the nested definitions and adjust offsets.

We note that source code availability is *not* a fundamental requirement of SigGraph. For a close-source OS (e.g., Windows), if debugging information is provided along with the binary, SigGraph can simply leverage the debugging information to extract the data structure definitions. Otherwise, data structure reverse engineering techniques (e.g., REWARDS [59], TIE [72], or HOWARD [73]) can be leveraged to extract data structure definitions from binaries.

4.3 Signature Generation

Suppose a data structure T has n pointer fields with offsets f_1, f_2, ..., f_n and types t_1, t_2, ..., t_n. A predicate $S_t(x)$ determines if the region starting at address x is an instance of t. The following production rule can be generated for T:

$$\begin{aligned} S_T(x) \quad \rightarrow \quad & S_{t_1}(*(x+f_1)) \wedge S_{t_2}(*(x+f_2)) \wedge ... \\ & \wedge S_{t_n}(*(x+f_n)) \end{aligned} \tag{4.7}$$

Brute force memory scanning is based on the *reverse* of the above rule: Given a kernel memory image, we hope to identify instances of a data structure by trying to match the

```
struct A {
  [0]  struct B * a1;
    ...
  [12] struct C * a2;
    ...
  [18] struct D * a3;
}
```

```
struct X {
    ...
  [8]  struct Y  * x1;
    ...
  [36] struct BB * x2;
    ...
  [48] struct CC * x3;
    ...
  [54] struct DD * x4;
}
```

```
c80b20e0: 00 00 00 00 01 20 00 32  0a 00 00 00 00 ae ff 00
c80b20f0: c8 40 30 b0 00 00 00 00  00 10 00 00 c8 40 42 30
c80b2100: 00 00 c8 41 00 22 00 00  00 10 00 00 00 00 00 00
```

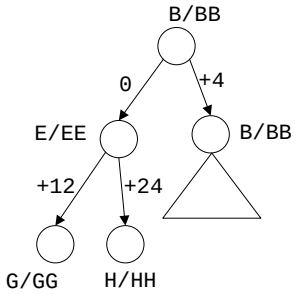
(a) Insufficiency of pointer layout uniqueness

```
struct B {
  [0]  E * b1;
  [4]  B * b2;
}

struct E {
    ...
  [12] G * e1;
    ...
  [24] H * e3;
}
```

```
struct BB {
  [0]  EE * bb1;
  [4]  BB * bb2;
}

struct EE {
    ...
  [12] GG * ee1;
    ...
  [24] HH * ee3;
}
```

```
struct D {
    ...
  [4]  I * d1;
}

struct DD {
    ...
  [8]  II * dd1;
}
```

(a) definitions



(b) structures of B and BB

(b) Data structure isomorphism

Fig. 4.3.: Examples illustrating the signature isomorphism problem

right-hand side of the rule (as a signature) with memory content starting at *any* location. Although it is generally difficult to infer the types of memory at individual locations based on the memory content, it is more feasible to infer if a memory location contains a pointer and hence to identify the layout of pointers with high confidence. This can be done recursively by following the pointers to the destination data structures. As such, the core challenge in signature generation is to find a finite graph induced by points-to relations (including pointers, pointer field offsets, and pointer types) that *uniquely* identifies a target data structure, which will be the root of the graph. For convenience of discussion, we assume for now that pointers are not `null` and they each have an explicit type (i.e., not a `void` pointer). We will address the cases where this assumption does not hold in Section 4.5.

As noted earlier, two distinct data structures may have isomorphic structural patterns. For example, if two data structures have the same pointer field layout, we need to further look into the "next-hop" data structures (we call them *lower layer* data structures) via the points-to edges. Moreover, we observe that *even though the pointer field layout of a data structure may be unique (different from any other data structure), an instance of such layout in memory is not necessary an instance of that data structure.* Consider Figure 4.3(a), data structures `A` and `X` have different layouts for their pointer fields. If the program has only these two data structures, it appears that we can use their one level pointer structures as their signatures. However, this is not true. Consider the memory segment at the bottom of Figure 4.3(a), in which we detect three pointers (the boxed bytes). It appears that $S_A(0xc80b20f0)$ is true because it fits the one-level structure of `struct A`. But it is possible that the three pointers are instead the instances of fields `x2`, `x3`, and `x4` in `struct X` and hence the region is part of an instance of `struct X`. In other words, a pattern scanner based on `struct A` will generate false positives on `struct X`. The reason is that the structure of `A` coincides with the sub-structure of `X`.

To better model the isomorphism issue, we introduce the concept of *immediate pointer pattern* (IPP) that describes the one-level pointer structure as a string such that the aforementioned problem can be detected by deciding if an IPP is the substring of another IPP.

Definition 4.3.1 *Given a data structure T, let its pointer field offsets be f_1, f_2, ..., and f_n, pointing to types t_1, t_2, ..., and t_n, respectively. Its* immediate pointer pattern, *denoted as $IPP(T)$, is defined as follows. $IPP(T) = f_1 \cdot t_1 \cdot (f_2 - f_1) \cdot t_2 \cdot (f_3 - f_2) \cdot t_3 \cdot ... \cdot (f_n - f_{n-1}) \cdot t_n$.*

We say an $IPP(T)$ is a sub-pattern of $IPP(R)$ if $g_1 \cdot r_1 \cdot (f_2 - f_1) \cdot r_2 \cdot (f_3 - f_2) \cdot ... \cdot (f_n - f_{n-1}) \cdot r_n$ is a substring of $IPP(R)$, with $g_1 >= f_1$ and r_1, ..., r_n being any pointer type.

Intuitively, an IPP describes the types of the pointer fields and their intervals. An $IPP(T)$ is a sub-pattern of $IPP(R)$ if the pattern of pointer field intervals of T is a sub-pattern of R's, disregarding the types of the pointers. It also means that we cannot distinguish an instance of T from an instance of R in memory if we do not look into the

lower layer structures. For instance in Figure 4.3(a), $IPP(A) = 0 \cdot B \cdot 12 \cdot C \cdot 6 \cdot D$ and $IPP(X) = 8 \cdot Y \cdot 28 \cdot BB \cdot 12 \cdot CC \cdot 6 \cdot DD$. $IPP(A)$ is a sub-pattern of $IPP(X)$.

Definition 4.3.2 *Replacing a type t in a pointer pattern with "($IPP(t)$)" is called one pointer expansion, denoted as $\xrightarrow{t}$. A pointer pattern of a data structure T is a string generated by a sequence of pointer expansions from $IPP(T)$.*

For example, assume the definitions of B and D can be found in Figure 4.3(b).

$$\begin{aligned} IPP(A) = \quad & 0 \cdot B \cdot 12 \cdot C \cdot 6 \cdot D \\ & \xrightarrow{B} \boxed{0 \cdot (0 \cdot E \cdot 4 \cdot B) \cdot 12 \cdot C \cdot 6 \cdot D}^{(1)} \\ & \xrightarrow{D} \boxed{0 \cdot (0 \cdot E \cdot 4 \cdot B) \cdot 12 \cdot C \cdot 6 \cdot (4 \cdot I)}^{(2)} \end{aligned} \tag{4.8}$$

Strings (1) and (2) above are both pointer patterns of A. The pointer patterns of a data structure are candidates for its signature. As one data structure may have many pointer patterns, the challenge becomes to algorithmically identify the unique pointer patterns of a given data structure so that instances of the data structure can be identified from memory by looking for satisfactions of the pattern without causing false positives. If efficiency is a concern, the minimal pattern should be identified.

Existence of Signature. The first question we need to answer is whether a unique pointer pattern exists for a given data structure. According to the previous discussion, given a data structure T, if its IPP is a sub-pattern of another data structure's IPP (including the case in which they are identical), we cannot use the one-layer structure as the signature of T. We have to further use the lower-layer data structures to distinguish it from the other data structure. However, it is possible that T is not distinguishable from another data structure R if their structures are isomorphic.

Definition 4.3.3 *Given two data structures T and R, let the pointer field offsets of T be f_1, f_2, ..., and f_n, pointing to types t_1, t_2, ..., and t_n, respectively.; the pointer field offsets of R be g_1, g_2, ..., and g_m, pointing to types r_1, r_2, ..., and r_m, respectively.*

T and R are isomorphic, denoted as $T \bowtie R$, if and only if

(1) $n \equiv m$;

(2) $\forall 1 \leq i \leq n \boxed{f_i \equiv g_i}^{(2.1)} \wedge (\boxed{t_i \bowtie r_i}^{(2.2)}$
$\vee \boxed{\textit{a cycle is formed when deciding } t_i \bowtie r_i}^{(2.3)})$.

Intuitively, two data structures are isomorphic, if they have the same number of pointer fields (Condition (1)) at the same offsets (2.1) and the types of the corresponding pointer fields are also isomorphic (2.2) or the recursive definition runs into cycles (2.3), e.g., when $t_i \equiv T \wedge r_i \equiv R$.

Figure 4.3(b) (i) shows the definitions of some data structures in Figure 4.3(a). The data structures whose definitions are missing from the two figures do not have pointer fields. According to Definition 4.3.3, $B \bowtie BB$ because they both have two pointers at the same offsets; and the types of the pointer fields are isomorphic either by the substructures ($E \bowtie EE$) or by the cycles ($B \bowtie BB$).

Given a data structure, we can now decide if it has a unique signature. As mentioned earlier, we assume that pointers are not `null` and are not of the `void*` type.

Theorem 4.3.1 *Given a data structure T, if there does not exist a data structure R such that*
<1>$IPP(T)$ is a sub-pattern of $IPP(R)$, and
<2> Assume the sub-pattern in $IPP(R)$ is $g_1 \cdot r_1 \cdot (f_2 - f_1) \cdot r_2 \cdot (f_3 - f_2) \cdot ... \cdot (f_n - f_{n-1}) \cdot r_n$, $t_1 \bowtie r_1$, $t_2 \bowtie r_2$, ... and $t_n \bowtie r_n$.

T must have a unique pointer pattern, that is, the pattern cannot be generated from any other individual data structure through expansions.

Proof For each data structure R different from T, either condition <1> or <2> is not satisfied according to the preconditions of the theorem.

If <1> is not satisfied, $IPP(T)$ can be used to distinguish T from R.

If <2> is not satisfied, there must be an i such that t_i is not isomorphic to r_i. There must be a minimal k, after k level of expansions, the pointer pattern of t_i is different from r_i's, disregard the type symbols. We say one level of expansion is to expand along all type

symbols for one step. $IPP(T)$ can be considered as the pointer pattern of T with $k = 0$ level of expansion.

Since there are finite number of data structures, we can always identify the maximal among all the k values. Lets denote it as k_{max}. Hence, the pointer pattern of T after k_{max} levels of expansions can distinguish T from any other individual data structure. ■

The proof of Theorem 4.3.1 is shown above. Intuitively, the theorem specifies that T must have a unique pointer pattern (i.e., a signature) as long as there is not an R such that $IPP(T)$ is a sub-pattern of $IPP(R)$ and the corresponding types are isomorphic.

If there is an R satisfying conditions <1> and <2> in the theorem, no matter how many layers we inspect, the structure of T remains identical to part of the structure of R, which makes them indistinguishable. In Linux kernels, we have found a few hundred such cases (about 12% of all data structures). Fortunately, most of those are data structures that are rarely used or not widely targeted according to OS security and forensics literature.

Note that two isomorphic data structures may have different concrete pointer field types. But given a memory image, it is unlikely for us to know the concrete types of memory cells. Hence, such information cannot be used to distinguish the two data structures. In fact, concrete type information is not part of a pointer pattern. Their presence is only for readability.

Consider the data structures in Figure 4.3(a) and Figure 4.3(b). Note all the data structures whose definitions are not shown do not have pointer fields. $IPP(A)$ is a sub-pattern of $IPP(X)$, $B \bowtie BB$ and $C \bowtie CC$. But D is not isomorphic to DD because of their different immediate pointer patterns. According to Theorem 4.3.1, there must be a unique signature for A. In this example, pointer pattern (2) in Equation (4.8) is a unique signature. If we find pointers that have such structure in memory, they must indicate an instance of A.

Finding the Minimal Signature. Even though we can decide if a data structure T has a unique signature using Theorem 4.3.1, there may be multiple pointer patterns of T that can distinguish T from other data structures. Ideally, we want to find the minimal pattern as it incurs the minimal parsing overhead during brute force scanning. For example, if the offset

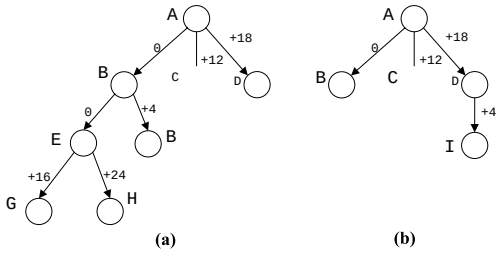


Fig. 4.4.: If the offset of field `e1` (of type `struct G`) in `E` is changed to 16, `struct A` will have two possible signatures (detailed data structure definitions in Figure 4.3)

of field `e1` (of type `struct G`) in `E` is 16, `struct A` will have two possible signatures as shown in Figure 4.4. They correspond to the following pointer patterns:

$$0 \cdot (0 \cdot (16 \cdot G \cdot 8 \cdot H) \cdot 4 \cdot B) \cdot 12 \cdot C \cdot 6 \cdot D$$

and

$$0 \cdot B \cdot 12 \cdot C \cdot 6 \cdot (4 \cdot I)$$

The first one is generated by expanding B and then E, and the second one is generated by expanding D. Either one can serve as a unique signature of A.

In general, finding the minimal unique signature is a combinatorial optimization problem: *Given a data structure T, find the minimal pointer pattern of T that cannot be a sub-pattern of any other data structure R, that is, cannot be generated by pointer expansions from a sub-pattern of $IPP(R)$.* The complexity of a general solution is likely in the NP category. In this paper, we propose an approximate algorithm (Algorithm 1) that guarantees to find a unique signature if one exists, though the generated signature may not be the minimal one. It is a breadth-first search algorithm that performs expansions for all pointer symbols at the same layer at one step until the pattern becomes unique.

The algorithm first identifies the set of data structures that may have $IPP(T)$ as their sub-patterns (lines 3-5). Such sub-patterns are stored in set $distinct$. Next, it performs breadth-first expansions on the pointer pattern of T, stored in s, and the patterns in $distinct$,

Algorithm 2 An approximate algorithm for signature generation

Input: Data structure T and set K of all kernel data structures considered
Output: The pointer pattern that serves as the signature of T.

1: s= $IPP(T)$
2: **let** $IPP(T)$ be $f_1 \cdot t_1 \cdot (f_2 - f_1) \cdot t_2 \cdot ... \cdot (f_n - f_{n-1}) \cdot t_n$
3: **for** each sub-pattern p=$g_1 \cdot r_1 \cdot (f_2 - f_1) \cdot r_2 \cdot (f_3 - f_2) \cdot ... \cdot (f_n - f_{n-1}) \cdot r_n$ in $IPP(R)$ of each structure $R \in (K - \{T\})$ with $f_1 <= g_1$ **do**
4: $distinct$=$distinct \cup \{p\}$
5: **end for**
6: **while** $distinct \neq \phi$ **do**
7: s=expand(s)
8: **for** each $p \in distinct$ **do**
9: p=expand(p)
10: **if** p is different from s disregarding type symbols **then**
11: $distinct$= $distinct - p$
12: **end if**
13: **end for**
14: **end while**
15: return s

expand(s)

1: **for** each type symbol $t \in s$ **do**
2: s= replace t with "($IPP(t)$)"
3: **end for**
4: return s

until all patterns can be distinguished. It is easy to infer that the algorithm will eventually find a unique pattern if one exists.

For the data structures in Figures 4.3(a) and 4.3(b), the pattern generated for A by the algorithm is

$$0 \cdot (0 \cdot E \cdot 4 \cdot B) \cdot 12 \cdot C \cdot 6 \cdot (4 \cdot I) \tag{4.9}$$

It is produced by expanding B and D in $IPP(A)$.

Generating Multiple Signatures. In some use scenarios, it is highly desirable to generate *multiple* signatures for the same data structure. A common scenario is that some pointer fields in a signature may not be dependable. For example, certain kernel malware may corrupt the values of some pointer fields and, as a result, the corresponding data structure instance will not be recognized by a signature that involves those pointers.

SigGraph mitigates such a problem by generating multiple unique signatures for the same data structure. In particular, if certain pointer fields in a data structure are potential targets of malicious manipulation, SigGraph will *avoid* using such fields during signature generation in Algorithm 1. For example, if field `e1`'s offset in `struct E` is 16 and field

`a3` (of type `struct D`) in `struct A` is not dependable, Algorithm 2 will adapt (not shown in the pseudo-code) by *pruning* the sub-graph rooted at field `a3` in Figure 4.4(a).

4.4 Scanner Generation

Given a data structure signature (i.e., a pointer pattern), SigGraph will automatically generate the corresponding memory scanner, which will be shipped to end users for brute force kernel memory scanning. To automatically generate scanners, we describe all signatures using a context-free grammar (CFG). Then we leverage `yacc` to generate the scanners. The CFG is described as follows.

$$\begin{aligned} Signature &:= \textbf{number} \cdot Pointer \cdot Signature \mid \epsilon \\ Pointer &:= \textbf{type} \mid (Signature) \end{aligned} \tag{4.10}$$

In the above grammar, **number** and **type** are terminals that represent numbers and type symbols, respectively. A $Signature$ is a sequence of **number** $\cdot$ $Pointer$, in which $Pointer$ describes either the **type** or the $Signature$ of the data structure being pointed to. It is easy to see that the grammar describes all the pointer patterns in Section 4.3, such as the signature of A generated by Algorithm 1 (Equation (4.9)).

Scanners can be generated based on the grammar rules. Intuitively, when a **number** symbol is encountered, the field offset should be incremented by **number**. If a **type** is encountered, the scanner asserts that the corresponding memory contain a pointer. If a '(' symbol is encountered, a pointer dereference is performed and the scanner starts to parse the next-level memory region until the matching ')' is encountered. A sample scanner generated for the signature in Equation (4.9) can be found in Figure 4.5. Function `isInstanceOf_A` decides if a given address is an instance of A; `assertPointer` asserts that the given address must contain a pointer value, otherwise an exception will be thrown and function `isInstranceOf_A` will return 0. The `yacc` rules to generate scanners are elided for brevity.

```
1 int isInstanceOf_A(void *x){
2   x=x+0;
3   {
4      y=*x;
5      y=y+0'
6      assertPointer(*y);
7      y=y+4;
8      assertPointer(*y);
9   }
10  x=x+12;
11   assertPointer(*x);
12   x=x+6;
13   {
14      y=*x;
15      y=y+4;
16      assertPointer(*y);
17   }
18   return 1;
19 }
```

Fig. 4.5.: The generated scanner for `struct A`'s signature in Equation (4.9)

Considering Non-pointer Fields. So far, a scanner considers only the positive information from the signature, which indicates the fields that are supposed to be pointers. But it does not consider the implicit negative information, which indicates the fields that are supposed to be non-pointers. In many cases, such negative information is needed to construct robust scanners.

For example, assume that a data structure T has a unique signature $0 \cdot A \cdot 8 \cdot B \cdot 4 \cdot C$. If there is a pointer array that stores a consecutive sequence of pointers, even though T's signature is unique and has no structural conflict with any other data structures, the scanner of T will mistakenly identify part of the array as an instance of T.

To handle such cases, the scanner should also assert that the non-pointer fields must not contain pointers. Hence the scanner for T's signature becomes the following. Method `assertNonPointer` asserts that the given address does not contain a pointer. As such, the final scanner code for identify data structure T will be:

```
1 int isInstanceOf_T(void *x){
2   x=x+0;
3   assertPointer(*x);    // field of type "A *"
4   x=x+4;
5   assertNonPointer(*x); // field of non-pointer
6   x=x+4;
7   assertPointer(*x);    // field of type "B *"
8   x=x+4;
9   assertPointer(*x);    // field of type "C *"
10 }
```

4.5 Handling Practical Issues

We have so far assumed the ideal case for SigGraph. However, when applied to large system software such as the Linux kernel, SigGraph faces a number of practical challenges, in particular,

1. **Null pointers:** It is possible that a pointer field have a `null` value, which cannot be distinguished from other non-pointer fields, such as integer or floating point fields with value 0. If 0 is considered a pointer value, a memory region with all 0s would satisfy any immediate pointer patterns, which is clearly undesirable.

2. **Void pointers:** Some of the pointer fields may have a `void*` type and they will be resolved to different types at runtime. Obviously, our signature generation algorithm cannot handle such case.

3. **User-level pointers:** It is also possible that a kernel pointer point to the user space, e.g., the field `set_child_tid` and `clear_child_tid` in `task_struct`, and the `vdso` field in `mm_struct` all point to user space. The difficulty is that user space pointers have a very dynamic value range due to the larger user space, which makes it hard to distinguish them from non-pointer fields.

4. **Special pointers:** A pointer field may have non-traditional pointer value. For example, for the `list_head` data structure, Linux kernel uses `LIST_POISON1` with value `0x00100100` and `LIST_POISON2` with value `0x00200200` as two special pointers to verify that no one uses un-initialized list entries. Another special value `SPINLOCK_MAGIC` (`0xdead4ead`) also widely exists in some pointer fields such as in data structure `radix_tree`.

5. **Pointer-like values:** Some of the non-pointer fields may have values that resemble pointers. For example, it is not an uncommon coding style to cast a pointer to an integer field and later cast it back to a pointer.

6. **Undecided pointers:** Union types allow multiple fields with different types to share the same memory location. This creates problems when pointer fields are involved.

7. **Rarely accessed data structures:** Algorithm 1 in Section 4.3 treats all data structures equally and tries to find unique signatures for all kernel data structures. However, some of the data structures are rarely used and hence the conflicts caused by them may not be so important.

We find that most of the problems above boil down to the difficulty in deciding if a field is a pointer or non-pointer. Interestingly, the following observation leads to a simple solution: pruning a few noisy pointer fields does not degenerate the uniqueness of the graph-based signatures. Even though a signature after pruning may conflict with some other data structure signatures, we can often perform a few more refinement steps to redeem the uniqueness. As such, we devise a dynamic profiling phase to eliminate the undependable pointer/non-pointer fields.

Our profiler (Figure 5.2) relies on LiveDM [74], a dynamic kernel memory mapping system, to keep track of dynamic kernel data structures at runtime. Based on QEMU [75], LiveDM tracks kernel memory allocation and deallocation events. More specifically, we focus on slab objects by hooking the allocation and deallocation functions such as `kmem_cache_alloc` and `kmem_cache_free` at the VMM level. The function arguments and return values are retrieved to obtain memory ranges of these objects. Their types are acquired by mapping allocation call sites to kernel data types via static analysis. We then track the life time of these objects and monitor their values.

We monitor the values of a kernel data structure's fields to collect the following information: (1) How often a pointer field takes on a value different from a regular non-null pointer value; (2) How often a non-pointer field takes on a non-null pointer-like value; (3) How often a pointer has a value that points to the user space. In our experiments, we profile a number of kernel executions for long periods of time (hours to tens of hours).

Based on the above profiles, we revise our signature generation algorithm with the following refinements: (1) excluding all the data structures that have never been allocated in

our profiling runs so that structural conflicts caused by these data structures can be ignored; (2) excluding all the pointer fields that have the `void*` type or fields of union types that involve pointers – in other words, these fields are declared undependable (Section 4.3), which is done by annotating them with a special symbol. Note that they should *not* be considered as non-pointer fields either and method `assertNonPointer` discussed in Section 4.4 will not be applied to such fields; (3) excluding all the pointer fields that have ever had a `null` value[1] or a non-pointer value during profiling; as well as all non-pointer fields that ever have a pointer value during profiling. Neither `assertPointer` nor `assertNonPointer` will be applied to these fields; (4) allowing pointers to have special value such as `0x00100100` or `0x00200200`.

We point out that dynamic profiling and signature refinement is performed only during the *production* of SigGraph-based signatures/scanners. It is *not* performed by end-users, who will simply run the scanners on memory images. We do note that the SigGraph signatures/scanners are kernel-specific, as different OS kernels may have different data structure definitions and runtime access characteristics. In fact, Section 6.4 shows that different versions of the same OS kernel may have different signatures for the same data structure.

4.6 Implementation

We implemented SigGraph in C and Python. For the data structure definition extraction component, we instrumented `gcc-4.2.4` for our purpose. Specifically, we walked through the AST of each data structure at the moment when `gcc` finishes the layout allocation (`stor-layout.c`). If the data structure type is `struct` or `union`, we dumped its field type, offset, and size information in individual files, which are indexed by a tuple of <`name`, `file`>, that is we used the data structure name, and declaration file names stored in the AST to index the `struct` or `union`. Also, if the field is an embedded

[1]We note that such exclusion will *not* remove important pointer fields in critical kernel data structures such as lists and trees, where non-zero magic values are used to indicate list/tree termination or initialization.

data structure, we will eliminate the hierarchy, adjust the offset, and expand them to finally get a flattened data structure definition.

For the signature generation component, we implemented using the mix of python and C code, as python provides an easier way to parse data structure definitions extracted from `gcc`, and C is more efficient when doing graph comparison. Our experience showed that python code will take a few days to complete the graph comparison of the whole data structure but C code just takes less than one hour. Our scanner generator is lex/yacc based, and the generated scanners are in C. The entire implementation has around 10K lines of C code and 6K lines of Python code.

4.7 Evaluation

We have performed four sets of evaluation of SigGraph. The first one is the signature uniqueness evaluation that answers the question of whether there exist our graph based signatures. The second is signature effectiveness evaluation that answers how effective our signature is when used to scan memory images. The third one is to evaluate the diversity of our signatures as we could we have multiple signatures for one data structure. Finally we evaluate the performance overhead of SigGraph.

4.7.1 Signature Uniqueness

Table 4.1: Experimental results of signature uniqueness test

Kernel version	#Total structs	Signature Statistics			
		#Pointer structs	#Unique Sig.	Percent	$\overline{S}$
2.6.15-1	8850	3597	3229	89.76%	2.31
2.6.18-1	11800	4882	4305	88.18%	2.45
2.6.20-15	14992	6096	5395	88.50%	2.54
2.6.24-26	15901	6427	5645	87.83%	2.47
2.6.31-1	26799	9957	8683	87.20%	2.73

We first test if unique signatures exist for kernel data structures. We test 5 popular Linux distributions (from Fedora Core 5 and 6; and Ubuntu 7.04, 8.04 and 9.10), with the corresponding kernel version shown in the first column of Table 4.1. We compile these

Table 4.2: Detail statistics on our static signatures

Kernel version	Number of signatures in different steps												
	1	2	3	4	5	6	7	8	9	10	11	12	13
2.6.15-1	1355	823	461	229	76	194	85	4	1	0	1	-	-
2.6.18-1	1820	1057	382	410	159	337	121	9	3	5	1	1	-
2.6.20-15	2137	1311	680	236	407	501	106	9	1	5	1	1	-
2.6.24-26	2172	1316	761	475	624	248	37	7	1	0	3	1	-
2.6.31-1	3364	1951	696	319	1492	494	344	19	1	0	1	1	1

kernels using our instrumented `gcc`. Observe that there are quite a large number of data structures in different kernels, ranged from 8850 to 26799. Overall, we find nearly 40% of the data structures have pointer fields, and nearly 88% (shown in the 5^{th} column) of the data structures with pointer fields have unique signatures. Because of graph isomorphism, there are data structures that do not have any unique signature, and the percentage for these data structures is around 12%. For the average steps ($\overline{S}$) performed in pointer pattern expansion to generate the unique signatures, the numbers are shown in the 6^{th} column. Note that these are all static numbers before the dynamic refinement.

In Table 4.2, we show the number of unique signatures of various depths, obtained by taking various number of expansion steps along the points-to relations. For example, kernel 2.6.15-1 has 1355 data structures that have unique one-level signatures and 823 data structures that have unique two-level signatures.

4.7.2 Signature Effectiveness

To test the effectiveness of SigGraph, we take Linux kernel 2.6.18-1 as a working system, and show how the generated signatures can detect data structure instances. We choose 23 widely used kernel data structures shown in the 2^{nd} column of Table 4.3. We choose these data structures because: (1) They are the most commonly examined data structures in existing literature [25, 40–42, 52–55]; (2) They are important data structures that can represent the status of the system in the aspects of process, memory, network and file system; from these data structures, we can reach most other kernel objects; and (3)

They contain pointer fields. Note that when scanning for instances of these data structures, other data structures – as part of the pointer patterns – are also traversed.

To ease our presentation, we assign an ID to each data structure, which is shown in the 3^{rd} column of Table 4.3. We use F to represent the set of fine-grained fields, and P to represent the set of pointer fields. A fine-grained field is a field with a primitive type (not a composite data type such as a `struct` or an array). Then, we present the corresponding total number of fields $|F|$ and pointers $|P|$ in the 5^{th} and 6^{th} columns, respectively.

Experiment Setup

We perform two sets of experiments. We first use our profiler to automatically prune the undependable pointer/non-pointer fields, generate refined signatures, and then detect the instances. After that we perform a comparison run with value invariant-based signatures (Section 4.7.2) to further confirm the effectiveness of SigGraph.

Memory snapshot collection: The first input of the effectiveness test is the snapshots of physical memory, which are acquired by instrumenting QEMU [75] to dump them on demand. We set the size of the physical RAM to 256M.

Ground truth acquisition: The second input is the ground truth data of the kernel objects under study. We leverage and modify a kernel dump analysis tool, the RedHat `crash` utility [60], to analyze our physical memory image and collect the ground truth, through a data structure instance query interface driven by our Python script. Note that to enable `crash`'s dump analysis, the kernel needs to be rebuilt with debugging information.

Profiling run: In all our profiling runs, the OS kernel is executed under normal workload and monitored for hours, with the goal of achieving good coverage of kernel data access patterns. However, it is unlikely that the profiling runs be able to capture the complete spectrum of patterns. As our future work, we will leverage existing techniques for software test generation to achieve better coverage.

Dynamic Refinement

In this experiment, we carry out the dynamic refinement phase as described in Section 4.5. The depth and size of signatures before and after pruning are presented in the "SigGraph Signature" columns in Table 4.3, with D being the depth and $\sum |P|$ the number of pointer fields. Note that the signature generation algorithm has to be run again on the pruned data structure definitions to ensure uniqueness. Observe that since pointer fields are pruned and hence the graph topology gets changed, our algorithm has to perform a few more expansions to redeem uniqueness, and hence the depth of signatures increases after pruning for some data structures, such as `task_struct`.

Value Invariant-based Signatures

To compare SigGraph-based signatures with value invariant-based signatures [32, 40–42], we also implement a basic value-invariant signature generation system. More specifically, we generally derive four types of invariants for each field including (1) *zero-subset:* a field is included if it is always zero across all instances during training runs; (2) *constant:* a field is always constant; (3) *bitwise-AND:* the bitwise AND of all values of a field is not zero, that is, they have some non-zero common bits; and (4) *alignment:* if all instances of a field are well-aligned at a power-of-two (other than 1) number.

To derive such value invariants for the data structures, we perform two types of profiling: one is access frequency profiling (to prune out the fields that are never accessed by the kernel) and the other is to sample their values and produce the signatures. The access frequency profiling is done by instrumenting QEMU to track memory reads and writes. Sampling is similar to the sampling method in our dynamic refinement phase.

All the data structures under study turn out to have value invariants. The statistics of these signatures are shown in the last four column of Table 4.3. The total numbers of zero-subset, constant, bitwise-AND, and alignment are denoted as $|Z|$, $|C|$, $|B|$, and $|A|$, respectively.

Table 4.3: Summary of data structure signatures for Linux kernel 2.6.18-1

Category	Static Properties of the Data Structure					SigGraph Signature				Value Invariant Signature			
	Data Structure Name	ID	Size	$\|F\|$	$\|P\|$	Statically Derived		Dynamically Refined		$\|Z\|$	$\|C\|$	$\|B\|$	$\|A\|$
						D	$\sum\|P\|$	D	$\sum\|P\|$				
Processes	task_struct	1	1408	354	81	1	81	2	233	269	17	55	244
	thread_info	2	56	15	4	2	91	2	45	5	2	4	5
	key	3	100	27	9	4	117	4	69	5	2	7	11
Memory	mm_struct	4	488	121	23	1	23	2	26	39	41	62	68
	vm_area_struct	5	84	21	10	4	1444	4	60	15	0	3	17
	shmem_inode_info	6	544	135	51	1	51	2	147	32	24	51	41
	kmem_cache	7	204	51	39	3	295	3	36	8	0	4	9
File System	files_struct	8	384	50	41	3	3810	3	13	38	4	8	9
	fs_struct	9	48	12	7	2	121	2	68	2	7	8	7
	file	10	164	40	11	5	17034	5	3699	15	4	12	17
	dentry	11	144	63	16	5	27270	5	1444	44	4	14	16
	proc_inode	12	452	112	49	1	49	3	455	27	16	33	41
	ext3_inode_info	13	612	151	58	1	58	2	166	59	27	50	53
	vfsmount	14	108	27	23	4	6690	4	1884	4	0	20	24
	inode_security_struct	15	60	16	6	7	277992	7	8426	1	1	3	2
	sysfs_dirent	16	44	11	7	4	1134	4	61	3	0	4	8
Network	socket_alloc	17	488	121	54	1	54	2	142	28	8	21	37
	socket	18	52	13	7	5	45907	5	2402	1	4	10	6
	sock	19	436	114	48	1	48	2	149	21	42	59	34
Others	bdev_inode	20	568	141	65	1	65	2	166	22	13	31	39
	mb_cache_entry	21	36	12	8	6	27848	6	6429	2	1	4	6
	signal_struct	22	412	99	25	2	395	2	90	41	30	38	44
	user_struct	23	52	13	4	6	586	6	394	1	0	1	2

Table 4.4: Experimental results of SigGraph signatures and value invariant-based signatures

ID	Data Structure Name	$\|I\|$	SigGraph Signature				Value Invariant Signature			
			$\|R\|$	FP'	FP	FN	$\|R\|$	FP'	FP	FN
1	task_struct	88	88	0.00%	0.00%	0.00%	88	0.00%	0.00%	0.00%
2	thread_info	88	88	0.00%	0.00%	0.00%	93	6.45%	6.45%	1.08%
3	key	22	22	0.00%	0.00%	0.00%	19	0.00%	0.00%	15.79%
4	mm_struct	52	54	3.70%	0.00%	0.00%	55	5.45%	0.00%	0.00%
5	vm_area_struct	2174	2233	2.64%	0.40%	0.00%	2405	9.61%	7.52%	0.00%
6	shmem_inode_info	232	232	0.00%	0.00%	0.00%	226	0.00%	0.00%	2.65%
7	kmem_cache	127	127	0.00%	0.00%	0.00%	5124	97.52%	97.52%	0.00%
8	files_struct	53	53	0.00%	0.00%	0.00%	50	0.00%	0.00%	6.00%
9	fs_struct	52	60	13.33%	0.00%	0.00%	60	13.33%	0.00%	0.00%
10	file	791	791	0.00%	0.00%	0.00%	791	0.00%	0.00%	0.00%
11	dentry	31816	38611	17.60%	0.01%	0.00%	31816	0.00%	0.00%	0.00%
12	proc_inode	885	885	0.00%	0.00%	0.00%	470	0.00%	0.00%	88.30%
13	ext3_inode_info	38153	38153	0.00%	0.00%	0.00%	38153	0.00%	0.00%	0.00%
14	vfsmount	28	28	0.00%	0.00%	0.00%	28	0.00%	0.00%	0.00%
15	inode_security	40067	40365	0.74%	0.00%	0.00%	142290	71.84%	70.93%	0.00%
16	sysfs_dirent	2105	2116	0.52%	0.52%	0.00%	88823	97.63%	97.63%	0.00%
17	socket_alloc	75	75	0.00%	0.00%	0.00%	75	0.00%	0.00%	0.00%
18	socket	55	55	0.00%	0.00%	0.00%	49	0.00%	0.00%	12.24%
19	sock	55	55	0.00%	0.00%	0.00%	43	0.00%	0.00%	27.90%
20	bdev_inode	25	25	0.00%	0.00%	0.00%	24	0.00%	0.00%	4.17%
21	mb_cache_entry	520	633	17.85%	0.00%	0.00%	638	18.50%	0.00%	0.00%
22	signal_struct	73	73	0.00%	0.00%	0.00%	72	0.00%	0.00%	1.39%
23	user_struct	10	10	0.00%	0.00%	0.00%	10591	99.91%	99.91%	0.00%

Results

The final results for each signature when brute force scanning a test image is shown in Table 4.4. The 3^{rd} column shows the total number of true instances of the data structure, which is acquired by the modified `crash` utility [60]. The $|R|$ column shows the number of data structure instances detected by the scanning. Due to the limitation of `crash`, the ground-truth instances are live, namely reachable from global or stack variables. On the other hand, brute force scanning can further identify freed-but-not-yet-reallocated objects that are not reachable from global or stack variables. Such freed objects detected would be counted as false positives (FPs) when compared with the ground truth from `crash`. As such, we present two FP numbers: (1) $|FP'|$ for those false positives that include the freed objects and (2) $|FP|$ for those that do not include the freed objects (hence $|FP'| \geq |FP|$). The false negative FN indicates those missed by scanning but present among the ground truth objects from `crash`.

```
struct task_struct{                    struct vm_area_struct {
   [156] struct mm_struct *active_mm;     [0] struct mm_struct *vm_mm;
   [160] struct linux_binfmt *binfmt;     [4] long unsigned int vm_start;
   [164] long int exit_state;             [8] long unsigned int vm_end;
   [168] int exit_code;                  [12] struct vm_area_struct *vm_next;
   [172] int exit_signal;                [16] pgprot_t vm_page_prot;
   [176] int pdeath_signal;              [20] long unsigned int vm_flags;
   [180] long unsigned int personalit; ...
}                                      }

0xc035dc9c <init_task+156>: 0xce8e04e0  0x00000000  0x00000000  0x00000000
0xc035dcac <init_task+172>: 0x00000000  0x00000000  0x00000000  0x00000000
0xc035dcbc <init_task+188>: 0x00000000  0x00000000  0xc035dc00  0xc035dc00
0xc035dccc <init_task+204>: 0xc12f1704  0xc12f1704  0xc035dcd4  0xc035dcd4
0xc035dcdc <init_task+220>: 0xc035dc00  0x00000000  0x00000000  0x00000000
0xc035dcec <init_task+236>: 0x00000000  0x00000000  0x00000000  0x00000000
0xc035dcfc <init_task+252>: 0x00000000  0x00000000  0x00000000  0x00000000
0xc035dd0c <init_task+268>: 0x00000000  0x00000000  0x00000000  0x00000000
0xc035dd1c <init_task+284>: 0x00000000  0x02bf54e4  0x00000000  0x002eff84
0xc035dd2c <init_task+300>: 0x00000000  0x00000000  0x00000000  0x00000000
0xc035dd3c <init_task+316>: 0x00000000  0x00000000  0x00000000  0x00000000
0xc035dd4c <init_task+332>: 0xc035dd4c  0xc035dd4c  0xc035dd54  0xc035dd54
```

Fig. 4.6.: False positive analysis of `vm_area_struct`

Among the 23 data structures, SigGraph perfectly (namely with accuracy and completeness) identifies all instances of 16 of the data structures when freed objects are considered FPs (i.e., both FP' and FN are zero); whereas value invariant signatures perfectly identify only 5 of the data structures. When freed objects are not considered FPs, 20 data structures can be perfectly identified by SigGraph whereas value invariant signatures perfectly identify 9. We also note that, with the exception of `dentry`, SigGraph signatures achieve equal or (much) lower false positive rate than value invariant-based signatures. No FNs are observed for SigGraph, while some are observed for the value invariant-based approach.

False Positive Analysis. Table 4.4 shows that SigGraph results in false positives ($|FP|$) for three of the 23 data structures: `vm_area_struct`, `dentry`, and `sysfs_dirent`. We carefully examine the memory snapshot and identify the reasons as follows.

- **`vm_area_struct`** We have 9 false positives (FPs) among the 2233 detected instances. After dynamic refinement, some pointer fields are pruned, such as the pointer field at offset 12 (as shown in Figure 4.6). The resultant signature consists of a pointer field at offset 0 (`mm_struct`), followed by a sequence of non-pointer fields, and so on. However, field `task_struct` starting from offset 156 has the same pointer pattern as that of `vm_area_struct` *except* that offset 160 is a pointer. Unfortunately, in some rare cases that are *not* captured by our profiler, the pointer field at offset 160 becomes 0, leading to the 9 FPs.

```
struct dentry {
    [0] atomic_t d_count;
    [4] unsigned int d_flags;
    [8] raw_spinlock_t raw_lock;
   [12] unsigned int magic;
   [16] unsigned int owner_cpu;
   [20] void *owner;
   [24] struct inode *d_inode;
   [28] struct hlist_node d_hash;
   [36] struct dentry *d_parent;
   ...
   [84] long unsigned int d_time;
   [88] struct dentry_operations *d_op;
   ...
}

fp1
0xc72bdf48: 0x00000000  0x00000010  0x00000001
0xc72bdf58: 0xffffffff  0xffffffff  0x00000000  0x00000000
0xc72bdf68: 0x00200200  0xc710e1c8  0x57409b84  0x00000009
0xc72bdf78: 0xc72bdfb4  0xc72bdf7c  0xc72bdf7c  0xc72bdef4
0xc72bdf88: 0xc017b72e  0xc72bdf8c  0xc72bdf8c  0xc72bdf94
0xc72bdf98: 0xc72bdf94  0x00000000  0x00000000  0xcf91fe00

fp2
0xcb1d5088: 0x00000000  0x00000010  0x00000001  0xdead4ead
0xcb1d5098: 0xffffffff  0xffffffff  0x00000000  0x00000000
0xcb1d50a8: 0x00200200  0xcb80ebc8  0xe50e3f24  0x0000000a
0xcb1d50b8: 0xcb1d50f4  0xcb1d50bc  0xcb1d50bc  0xcb1dcf84
0xcb1d50c8: 0xc017b72e  0xcb1d50cc  0xcb1d50cc  0xcb1d50d4
0xcb1d50d8: 0xcb1d50d4  0x026a0005  0x00000000  0xcf91fe00

true
0xc001c0a8: 0x00000000  0x00000000  0x00000001  0xdead4ead
0xc001c0b8: 0xffffffff  0xffffffff  0x00000000  0xc67617f4
0xc001c0c8: 0xc12a0e7c  0xc727faa8  0xbfbb9195  0x00000009
0xc001c0d8: 0xc001c114  0xc001c16c  0xc05b9f5c  0xc001c174
0xc001c0e8: 0xc727faec  0xc001c0ec  0xc001c0ec  0xc001c0f4
0xc001c0f8: 0xc001c0f4  0x8bfffff9  0x00000000  0xcf91fe00
```

(a) False positives of `dentry`

```
struct sysfs_dirent {
   [0] atomic_t s_count;
   [4] struct list_head s_sibling;
  [12] struct list_head s_children;
  [20] void *s_element;
  [24] int s_type;
  [28] umode_t s_mode;
  [32] struct dentry *s_dentry;  [pruned]
  [36] struct iattr *s_iattr;    [pruned]
  [40] atomic_t s_event; }

fp1
0xcffaeffc: 0x00000000  0xcffa3800  0xcffaf800  0xcffa3808
0xcffaf00c: 0xcffaf808  0xcffc2800  0x00000000  0x00000000
0xcffaf01c: 0xcfd9bde0  0x00000008  0x70008086

fp2
0xcffaf7fc: 0x00000000  0xcffaf000  0xc03709a8  0xcffaf008
0xcffaf80c: 0xcffc2814  0xcffc2800  0x00000000  0x00000000
0xcffaf81c: 0xcfd9be60  0x00000000  0x12378086

fp3
0xcffa37fc: 0x00000000  0xcffa3000  0xcffaf000  0xcffa3008
0xcffa380c: 0xcffaf008  0xcffc2800  0x00000000  0x00000000
0xcffa381c: 0xcfd9bd60  0x00000009  0x70108086

fp4
0xcffa2ffc: 0x00000000  0xcffa2800  0xcffa3800  0xcffa2808
0xcffa300c: 0xcffa3808  0xcffc2800  0x00000000  0x00000000
0xcffa301c: 0xcfd9bce0  0x0000000b  0x71138086

fp5
0xcffa27fc: 0x00000000  0xcffa2000  0xcffa3000  0xcffa2008
0xcffa280c: 0xcffa3008  0xcffc2800  0x00000000  0x00000000
0xcffa281c: 0xcfd9bc60  0x00000010  0x00b81013

fp6
0xc037099c: 0x00000000  0xcffc2800  0xcffc2800  0xcffaf800
0xc03709ac: 0xcffa2000  0xc0327d70  0x00000000  0x00000124
0xc03709bc: 0xc01de4bc  0x00000000  0x00000000
```

(b) False positives of `sysfs_dirent`

Fig. 4.7.: False positive analysis of `dentry` and `sysfs_dirent`

- **`dentry`** We have 2 FPs of `dentry`, which are shown in Figure 4.7(a). We consider these two instances as FPs because they cannot be found in either the pool of live objects or the pool of freed objects. However, if we carefully check each field's value, especially the boxed ones: `0xdead4ead` (`SPINLOCK_MAGIC` at offset 12) and `0xcf91fe00` (a pointer to `dentry_operations` at offset 88), we cannot help but thinking that these are indeed `dentry` instances instead of FPs. We believe that they belong to the case where the slab allocator has freed the memory page of the destroyed `dentry` instances.

- **`sysfs_dirent`** We have 6 FPs of `sysfs_dirent` among the 2116 detected instances. The detailed memory dumps of the 6 FP cases are shown in Figure 4.7(b). After our dynamic refinement, the fields at offsets 32 and 36 are pruned because they often contain `null` pointers. And the final signature entails checking two `list_head` data structures followed by a `void*` pointer (at offsets 4, 8, 12, 16 and 20, respectively) and checking four non-pointer fields. Note that each `list_head`

has only two fields: previous and next pointer. There are 6 memory chunks that match our signature in the test memory image. But the chunks are not part of the ground truth. We suspect that these chunks are *aggregations* of multiple data structures and the aggregations coincidentally manifest the same pattern.

Summary: In this experiment, SigGraph achieves zero FN and (much) lower FP rates. Intuitively, the reasons are the following: (1) SigGraph-base signatures are structure-oriented and thus tend to be more stable than value-oriented approaches. And their uniqueness can be algorithmically determined – that is, we can expand a signature along available points-to edges to achieve uniqueness. (2) SigGraph-based signatures are more "informative" as each signature includes information about *other* data structures; whereas a value-based signature only carries information about itself.

4.7.3 Multiple Signatures

One powerful feature of SigGraph is that multiple signatures can be generated for the same data structure (Section 4.3). We perform the following experiments with the `task_struct` data structure to verify that. In each experiment, we exclude one of the 38 pointer fields of `task_struct` (considering that pointer corrupted) before running Algorithm 1. In each of the 38 experiments, the algorithm is still able to compute a unique, alternative signature for `task_struct`. Next, we increase the number of corrupted pointer fields from 1 to 2, and conduct $C_{38}^{2} = \binom{2}{38}$ runs of Algorithm 1 (exhausting the combinations of the two pointers excluded). The algorithm is still able to generate a valid signature for each run.

The above experiments indicate that SigGraph is robust in the face of corrupted pointer fields. However, the robustness does have its limit. At the other extreme, we exclude 37 of the 38 pointer fields of `task_struct` and conduct $C_{38}^{37} = \binom{37}{38} = 38$ runs of Algorithm 1. Among the 38 runs, Algorithm 1 only generates valid signatures in 4 runs, where one of the following pointers is retained: `fs_struct`, `files_struct`, `namespace`, and `signal_struct`.

4.7.4 Performance Overhead

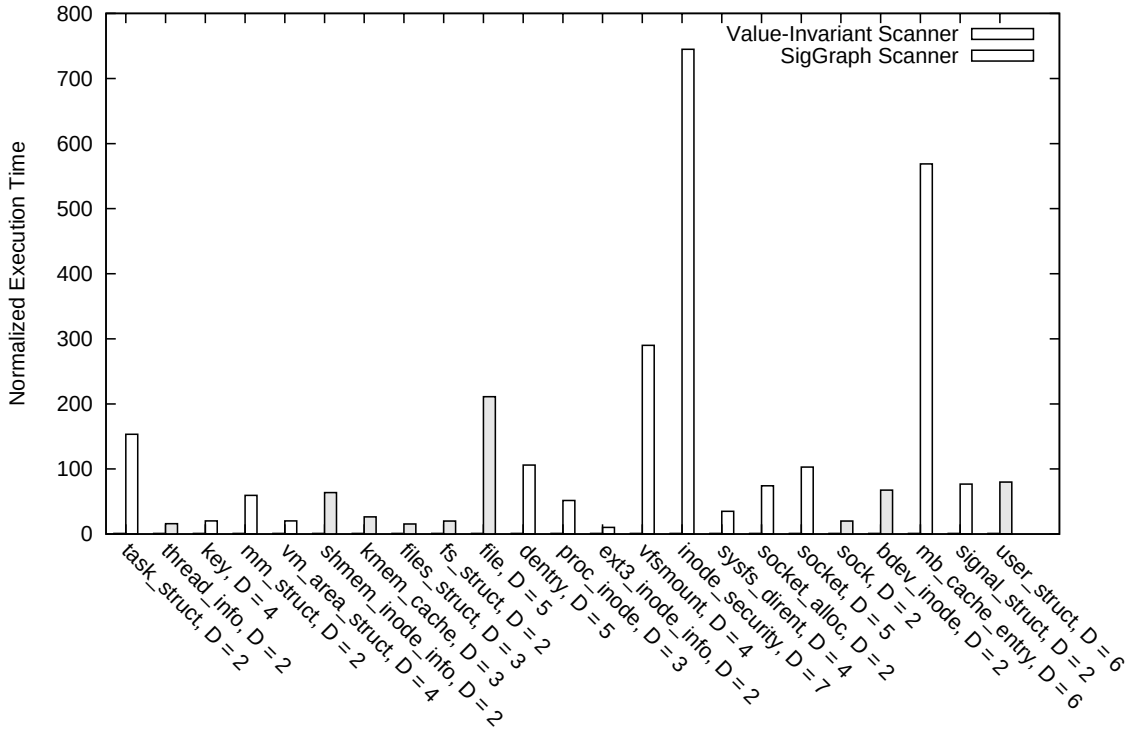


Fig. 4.8.: Memory scanning performance

Since SigGraph may be used for online live memory analysis, we measure the overhead of memory scanning using SigGraph signatures. We run both SigGraph-generated scanners and the value invariant-based scanners on the testing image (256MB) in a machine with 3GB RAM and an Intel Core 2 Quad CPU (2.4GHz) running Ubuntu-9.04 (Linux kernel 2.6.28-17). The final result of the normalized overhead (compared with value-invariant) is shown in Figure 4.8.

As expected, value-invariant scanners always outperform SigGraph scanners. The main reason is that: A SigGraph scanner needs to conduct address translation whenever there is a memory de-reference, which is not needed by the value invariant scanner. If the depth of a SigGraph signature is relatively low (e.g., $D = 2$), the SigGraph scanner will be roughly 10-20 times slower than the corresponding value invariant scanner. Greater depth

often leads to higher overhead because more nodes will need to be examined and more address translation needs to be performed. The cases of `inode_security` ($D = 7$) and `mb_cache_entry` ($D = 6$) are such examples. Thus, for data structures with low-depth signatures, their SigGraph scanners can be used online. For example, in our experiment, it takes only a few seconds to scan `fs_struct`, `thread_info`, and `files_struct`, and less than one minute to scan `task_struct`.

For data structures with a greater depth (due to isomorphism elimination) such as `inode_security` and `mb_cache_entry`, the scanning time is longer (e.g., about 15 minutes when we scan a 256MB memory image using the scanner for `inode_security`). However, we argue that such cost is acceptable in the context of computer forensics, where accuracy and completeness is more important than efficiency. Moreover, the scanning time can be reduced by various optimizations such as parallelization or having a pre-scanning phase to preclude unlikely cases.

4.8 Summary

In this chapter, we have presented SigGraph, a framework that systematically generates graph-based, non-isomorphic data structure signatures for brute force scanning of kernel memory images. Each signature is a graph rooted at the subject data structure with edges reflecting the points-to relations with other data structures. SigGraph-based signatures complement value invariant-based signatures for more accurate recognition of kernel data structures with pointer fields. Moreover, SigGraph differs from global memory mapping-based approaches that have to start from global variables and require reachability to all data structure instances from them.

5. DIMSUM: DISCOVERING DATA STRUCTURE INSTANCES USING PROBALISTIC INFERENCE

In this chapter, we present DIMSUM, the third component in our framework, to enable the recognition of data structure instances from un-mappable memory. Such un-mappable memory could be (1) the entire free pages of the system, (2) the memory swap file, or (3) a corrupted memory dump. Existing memory mapping-guided techniques do not work in that scenario as pointers in the un-mappable memory cannot be resolved. To address this problem, we thus present our *probabilistic inference*-based approach DIMSUM.

5.1 DIMSUM Overview

Given a set of memory pages and the specification of a target data structure, DIMSUM will identify instances of the data structure in those pages with quantifiable confidence. More specifically, it automatically builds graphical models based on boolean constraints generated from the data structure and the memory page contents. Probabilistic inference is performed on the graphical models to generate results ranked with probabilities. DIMSUM has the following observations.

5.1.1 Key Observation

DIMSUM was first motivated by the “dead memory pages left by terminated processes” scenario. More specifically, we notice that, when a process is terminated, neither Windows nor Linux operating system clears the content of its memory pages. We believe one of the reasons is to avoid memory cleansing overhead. Moreover, Chow et al. [76] found that many applications let sensitive data stay in memory after usage instead of “shredding” them. Even if an application performs data “shredding”, it is still possible that a crash

happens before the shredding operation, leaving some sensitive data in the dead memory pages.

Second, we also observe that dead pages may remain intact for a non-trivial duration, which we call their *death-span*. In fact, we observe that the death-span of the dead pages of a Firefox process can last up to 50 minutes after the process terminates, in a machine with 512 MB RAM, as shown in Figure 5.1. If the machine has a larger RAM or the workload after Firefox's termination is not as memory-intensive, the death-span of dead pages may be even longer. A similar study on the age of freed user process data on Windows XP (SP2) [77], has shown that large segments of pages can survive for nearly 5 minutes in a lightly loaded system; and smaller segments and single pages may be found intact for up to 2 hours.

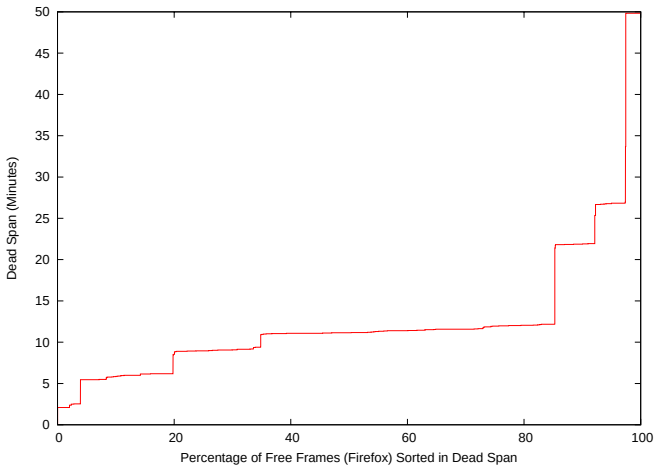


Fig. 5.1.: Death-span of free frames from a terminated Firefox process

Finally, we observe that, for a terminated process, the corresponding memory mapping information maintained by the OS kernel, such as the process control block and page table, are likely to disappear (i.e., be reused) very quickly. The much shorter dead-span of kernel objects (typically in a few seconds) – contrary to that of dead application pages – is due to the fact that kernel objects are maintained as slab objects by the kernel [78], which uses LIFO as the memory recycling policy; whereas memory pages of processes are managed by

the buddy system [78] that groups memory frames into lists of blocks having 2^k contiguous frames, and hence page frames tends to have longer dead-span.

We point out that the above scenario is *not the only one* that involves memory pages without mapping information. Another interesting scenario is to analyze the Coldboot images as demonstrated in [79]: After a machine with modern DRAM is powered off, the content of the DRAM will disappear *gradually* instead of immediately, making it possible to obtain partial memory image with no or partial memory mapping information.

5.1.2 Challenges

Compared with existing approaches, DIMSUM raises a number of new challenges. The first challenge is the absence of memory mapping information. Consequently, given a set of memory pages, there is little hint on which pages belong to which process, let alone the sequencing of physical pages in the virtual address space of a process. Even if we can identify some pointers in a page, we still cannot follow those pointers without the address mapping information.

The second challenge is that DIMSUM may accept an incomplete subset of memory pages of a process as input. In this case the application data that reside in the absent pages cannot be recovered. However, such data could be useful for the recognition of application data that reside in the input pages, especially when a pointer-based memory forensics technique is employed.

The third challenge is the absence of type/symbolic information for dead memory. To map the raw bits and bytes of a memory page to meaningful data structure instances, type information is necessary. For example, if the content at a memory location is 0, its type could be integer, floating point, or even pointer. If these bits and bytes belong to the live memory, symbolic information is available and they can be typed through reference path (as in [30]). To DIMSUM, however, such information is not available.

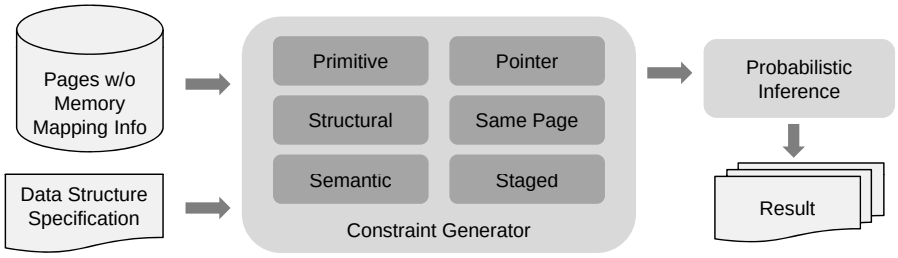


Fig. 5.2.: Overview of DIMSUM

5.1.3 Overview

To address the above challenges, we take a *probabilistic inference* and *constraint solving* approach. Fig. 5.2 shows the key components and operations of DIMSUM. The input of the system includes: (1) a subset of memory pages from a computer and (2) the specifications of data structure(s) of interest. Note that a data structure specification includes field offset and type information, which can be obtained from either application documentation, debugging information, or reverse engineering [59, 72, 73].

A key component of DIMSUM, the *constraint generator*, transforms the data structure specification into constraint templates that are instantiated by the input memory pages. These templates describe correlations dictated by data structure field layout, and include *primitive*, *pointer*, *structural*, *same-page*, *semantic*, and *staged* constraints (Section 5.3).

Next, the *probabilistic inference* component automatically transforms all the constraints into a factor graph [67], and efficiently computes the marginal probabilities of all the candidate memory locations for the data structure of interest. Finally, it outputs the result based on the probability rankings.

5.2 DIMSUM Design

The essence of DIMSUM is to formulate the data structure recognition problem as a probabilistic constraint solving problem. We first use a working example to demonstrate the basic idea, which relies on solving boolean constraints.

```
struct utmplist {
    00: short int ut_type;
    04: pid_t ut_pid;
    08: char ut_line[32];
    40: char ut_id[4];
    44: char ut_user[32];
    76: char ut_host[256];
    332: long int ut_etermination;
    336: long int ut_session;
    340: struct timeval ut_tv;
    348: int32_t ut_addr_v6[4];
    364: char __unused[20];
    384: struct utmplist *next;
    388: struct utmplist *prev;
}
```

Fig. 5.3.: Data structure definition of our working example

Ideally, our technique takes (1) the data structure specification such as the one defined in Fig. 5.3, which is the `utmplist` data structure showing a list of last logged users in a Linux utility program `last` and (2) a set of memory pages, and then tries to identify instances of the data structure in the pages. The idea is to first generate a set of constraints from the given data structure. For example, given the predicate definitions presented in Table 5.1 and assuming a 32 bit machine, the generated constraint for the `utmplist` structure would be:

$$\begin{aligned}
\texttt{utmplist}(a) \quad \rightarrow\ & I_{\texttt{ut_type}}(a) \wedge I_{\texttt{ut_pid}}(a+4) \wedge \\
& C_{\texttt{ut_line}}(a+8)[32] \wedge C_{\texttt{ut_id}}(a+40)[4] \wedge \\
& C_{\texttt{ut_user}}(a+44)[32] \wedge C_{\texttt{ut_host}}(a+76)[256] \wedge \\
& I_{\texttt{ut_session}}(a+336) \wedge I_{\texttt{ut_etermination}}(a+332) \wedge \\
& I_{\texttt{ut_tv.tv_sec}}((a+340)) \wedge I_{\texttt{ut_tv.tv_usec}}((a+344)) \wedge\ . \\
& I_{\texttt{ut_addr_v6}}((a+348)[4]) \wedge C_{\texttt{__unused}}((a+364)[20]) \wedge \\
& P_{\texttt{next}}(a+384) \wedge \texttt{utmplist}(*(a+384)) \wedge \\
& P_{\texttt{prev}}(a+388) \wedge \texttt{utmplist}(*(a+388)) \wedge \\
& *(a+4)_{ut_pid} \geq 0
\end{aligned} \tag{5.1}$$

Note that the subscripts are used to denote field names. Intuitively, the above formula means that if the location starting at a denotes an instance of `utmplist`, the location at a contains an integer, the location at $a+4$ contains an integer as well, $a+8$ contains a char array with size 32, and so on. The constraint also dictates that the locations pointed-to by pointers at $a+384$ and $a+388$ contain instances of `utmplist` as well. These

Table 5.1: Predicate definitions used throughout the paper

Predicate	Definitions
$\tau(x)$	The region starting at x is an instance of a user-defined type τ
$I(x)$	The location at x is an integer.
$F(x)$	The location at x is a floating point value.
$D(x)$	The location at x is a double floating point value.
$S(x)$	The location at x is a string.
$C(x)$	The location at x is a char.
$P(x)$	The location at x is a pointer.
$T(x)[y]$	The location at x is an array of size y, with each element of type T.

are called *structural constraints* as they are derived from the type structure. We may also have *semantic constraints* that predicate on the range of the value at an address. The term at the end of the constraint specifies that field `ut_pid` should have a non-negative value. Semantic constraints can be provided by the user based on domain knowledge.

Besides the above constraints, we also extract a set of *primitive constraints* by scanning the pages. These constraints specify what primitive type each location has. We consider seven primitive types: *int*, *float*, *double*, *char*, *string*, *pointer* and *time*. Here, we leverage the observation that deciding if a location is an instance of a primitive type, such as a pointer, can often be achieved by looking at the value. Suppose that addresses 0, 4, 8, 12, 16 have been determined to contain integer, integer, non-negative integer, char array with size 16, the primitive constraints $I(0)$, $I(4)$, $I(8)$, $C(12)[16]$ (defined in Table 5.1) are generated. By conjoining the structural, semantic, and primitive constraints, we can use a solver to produce satisfying valuations for $utmplist(a)$, which essentially identifies instances of the given type. With the above constraints, $a = 0$ is not an instance because $C(a + 8)[32]$ is not satisfied. In contrast, $a = 4$ may be one.

5.2.1 Practical Problems

However, the basic design faces a number of practical problems in the context of DIMSUM. In particular:

Uncertainty in Primitive Constraints: While values of primitive types have certain attributes, it is in general hard to make a binary decision for a type predicate by looking at the value. In such cases, we expect that our technique is able to reason with probabilities.

Absence of Page Mappings: As discussed in Section 5.1, a pointer value is essentially a *virtual* address. Without memory mapping information, for constraints like $S(*a)$, we cannot identify the page being pointed to by a and thus cannot decide if a points to a string.

Incompleteness: We may see only part of a data structure, e.g., some elements in a linked list may be missing. Our system should be able to resolve constraints for such cases.

5.2.2 Probabilistic Inference

To address the above issues, we formulate the problem as a probabilistic inference problem [66, 67]. Initial probabilities are associated with individual constraints, representing the user's view of uncertainty. The probabilities are efficiently propagated, aggregated, and updated over a graphical representation called *factor graph* (FG) [67]. After convergence, the final updated probabilities of interesting boolean variables can hence be queried from the FG. Next we use an example to explain.

We simplify the case in the Fig. 5.3 by considering only the pointer fields, i.e., fields at offsets 384 and 388. For a given address a, let boolean variable x_1, x_2, and x_3 denote $T_{utmplist}(a)$, $P_{next}(a + 384)$, and $P_{prev}(a + 388)$, respectively. The structural constraint is simplified as follows.

$$x_1 \ \rightarrow \ x_2 \wedge x_3 \tag{5.2}$$

Assume the structural pattern is unique across the entire system, meaning that there are no data structures across the system with the same structural pattern. In particular for the above pattern, if we observe two consecutive pointers in memory, we can be assured that they must be part of an instance of `struct utmplist`, we have the following constraint.

$$x_1 \ \leftarrow \ x_2 \wedge x_3 \tag{5.3}$$

With this constraint, when we observe $x_2 = 1$ and $x_3 = 1$, we can infer $x_1 = 1$, meaning that there is an instance of `struct utmplist` at the given address a. If $x_2 = 1$ and $x_3 = 0$, we infer that $x_1 = 0$.

In general, assume there are m constraints C_1, C_2..., and C_m on n boolean variables x_1, x_2, ..., and x_n. Functions f_{C_1} f_{C_2}, ..., and f_{C_n} describe the valuation of the constraints. For instance, let C_1 be Equation (5.2), $f_{C_1}(x_1 = 1, x_2 = 1, x_3 = 0) = 0$. Since all the constraints need to be satisfied, the function representing the conjunction of the constraints is hence the product of the individual constraint functions, as shown in Equation (5.4).

$$f(x_1, x_2, ..., x_n) = f_{C_1} \times f_{C_2} \times ... \times f_{C_m} \tag{5.4}$$

In DIMSUM, we often cannot assign a boolean value to a variable or a constraint. Instead, we can make an observation about the likelihood of a variable being true. For instance, from the value stored at offset $a + 384$, we can only say that it is likely a pointer. Moreover, if the structural pattern of $T_{utmplist}$ is not unique, i.e., other data structures may also have such a pattern, we can similarly assign a probability to constraint (5.3) according to the number of data structures sharing the same pattern.

Assume we use a set of boolean variables x_1, x_2, ..., x_n to represent type predicates. Probabilities are associated with variables and constraints. In our previous example, assume that we are 100% sure that $x_1 \rightarrow x_2 \wedge x_3$ (C_1); 80% sure that $x_1 \leftarrow x_2 \wedge x_3$ (C_3) because other data structures manifest a similar structural pattern; 90% sure that x_2 is a pointer (C_2); 90% sure that x_3 is a pointer (C_4). We have probabilistic functions:

$$f_{C_1}(x_1, x_2, x_3) = \begin{cases} 1 & \text{if } (x_1 \rightarrow x_2 \wedge x_3) = 1 \\ 0 & \text{otherwise} \end{cases} \tag{5.5}$$

$$f_{C_2}(x_2) = \begin{cases} 0.9 & \text{if } x_2 = 1 \\ 0.1 & \text{otherwise} \end{cases} \tag{5.6}$$

$$f_{C_3}(x_1, x_2, x_3) = \begin{cases} 0.8 & \text{if } (x_1 \leftarrow x_2 \wedge x_3) = 1 \\ 0.2 & \text{otherwise} \end{cases} \tag{5.7}$$

$$f_{C_4}(x_3) = \begin{cases} 0.9 & \text{if } x_3 = 1 \\ 0.1 & \text{otherwise} \end{cases} \tag{5.8}$$

With these probabilistic constraints, the joint probability function is defined as follows [66, 67].

$$p(x_1, x_2, ..., x_n) = \frac{f_{C_1} \times f_{C_2} \times ... \times f_{C_m}}{Z} \tag{5.9}$$

$$Z = \sum_{x_1,...,x_n} (f_{C_1} \times f_{C_2} \times ... \times f_{C_m}) \tag{5.10}$$

In particular, Z is the normalization factor [66, 67].

It is often more desirable to further compute the marginal probability $p_i(x_i)$ as follows.

$$p_i(x_i) = \sum_{x_1} \sum_{x_2} ... \sum_{x_{i-1}} \sum_{x_{i+1}} ... \sum_{x_n} p(x_1, x_2, ..., x_n) \tag{5.11}$$

In other words, the marginal probability is the sum over all variables other than x_i. Variable x_i often predicates on a given address having the type we are interested in. Hence, in order to discover the instances of the specific type, DIMSUM orders memory addresses by their marginal probabilities.

Table 5.2: Boolean constraints with probabilities

x_1	x_2	x_3	$f_{C_1}(x_1, x_2, x_3)$	$f_{C_2}(x_2)$	$f_{C_3}(x_1, x_2, x_3)$	$f_{C_4}(x_3)$
0	0	0	1	0.1	0.8	0.1
0	0	1	1	0.1	0.8	0.9
0	1	0	1	0.9	0.8	0.1
0	1	1	1	0.9	0.2	0.9
1	0	0	0	0.1	0.8	0.1
1	0	1	0	0.1	0.8	0.9
1	1	0	0	0.9	0.8	0.1
1	1	1	1	0.9	0.8	0.9

Consider the previous example. Table 5.2 presents the values of the four probability constraint functions for all possible variable valuations.

$$\begin{aligned} p(x_1 = 1) &= \frac{\sum_{x_2,x_3} f_{C_1}(1, x_2, x_3) \times f_{C_2}(x_2)}{\sum_{x_1,x_2,x_3} f_{C_1}(x_1, x_2, x_3) \times f_{C_2}(x_2)} \\ &= \frac{0 \times 0.1 + 0 \times 0.1 + 0 \times 0.9 + 1 \times 0.9}{1 \times 0.1 + 1 \times 0.1 + ... + 1 \times 0.9} \\ &= \frac{0.9}{2.9} = 0.31 \end{aligned} \tag{5.12}$$

$$\begin{aligned} p(x_2 = 1) &= \frac{1 \times 0.9 + 1 \times 0.9 + 0 \times 0.9 + 1 \times 0.9}{2.9} \\ &= 0.93 \end{aligned} \tag{5.13}$$

Assume only constraints C_1 and C_2 are considered, Equation (5.12) describes the computation of the marginal probability of $p(x_1 = 1)$, i.e., the probability of the given address being an instance of `struct utmplist`. Equation (5.13) describes the marginal probability of $p(x_2 = 1)$. Note that it is different from the initial probability 0.9 in f_{C_2}. Intuitively, the value assigned in f_{C_2} is essentially an observation, which does not necessarily reflect the intrinsic probability. In other words, the initial probability in f_{C_2} is what we believe and it reflects only a local view of the constraint, whereas the computed probability represents a global view with all initial probabilities over the entire system being considered.

Similarly, when all four constraints are considered, we can compute $p(x_1 = 1) = 0.71$. Intuitively, compared to considering only C_1 and C_2, now we also have high confidence on x_3 (C_4) and we have confidence that as long as we observe x_2 and x_3 being true, x_1 is very likely true (C_3). Such information raises the intrinsic probability of x_1 being true.

Note that depending on the number of variables and the number of constraints, the computation entitled by Equation (5.11) could be very expensive because it has to enumerate the combinations of variable valuations. A *Factor graph* [63, 66, 67] is a representation for a probability function that allow for highly efficient computation. In particular, a factor graph is a bipartite graph with two kinds of nodes. A *factor node* represents a factor in the function, e.g., f_{C_i} in Equation (5.9). A *variable node* represents a variable in the function,

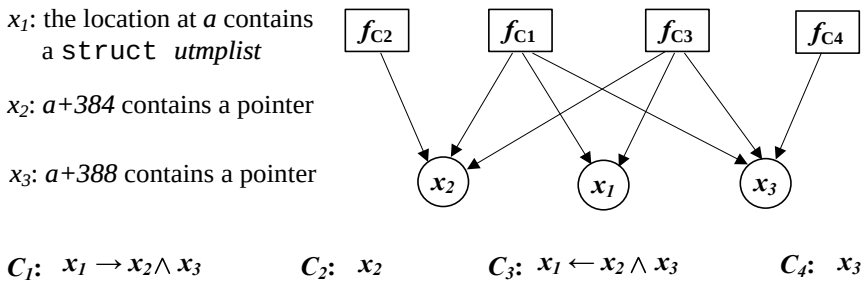


Fig. 5.4.: Factor graph example

e.g., x_i in Equation (5.9). Edges are introduced from a factor to the variables of the factor function. Fig. 5.4 presents the factor graph for the probability function for the previous example. The *sum-product* algorithm [66,67] can leverage factor graphs to compute marginal probabilities in a highly efficient way. The algorithm is iterative. In particular, probabilities are propagated between adjacent nodes through message passing. The probability of a node is updated by integrating the messages it receives. The algorithm terminates when the probabilities become stable. At a high level, one can consider initial probabilities as energy applied to a mesh such that the mesh transforms to strike a balance and minimize free energy. Probabilistic inference has a wide range of successful applications in artificial intelligence, information theory and debugging [62, 63]. In this paper, DIMSUM is built on a probabilistic inference framework called *Infer.NET* [65].

In order to conduct probabilistic reasoning using FG, we first assign a boolean variable to each type predicate, indicating if a specific address holds an instance of a given type. We create a variable for each type of interest for each memory location. In other words, if there are n data structures of interest and m memory locations, we would generate $n * m$ boolean variables. We will introduce a pre-processing phase that can reduce variables needed by reducing m. Then constraints are introduced. Constraints are essentially boolean formula on the boolean variables. Initial probabilities are assigned to these constraints to express uncertainty. Constraints and initial probability assignments are programmed as scripts in *Infer.NET*. FGs are constructed by these scripts. The *Infer.NET* engine conducts inference on the FGs. After that, data structure instances can be identified by querying the

probabilities of the corresponding boolean variables. We report those within the highest probability cluster to the user.

5.3 Generating Constraints

We now explain how we model the memory forensics problem with constraints. The constraints fall into the following categories: *primitive constraints* that associate initial probabilities to individual boolean variables; *structural constraints* that describe field structures; *pointer constraints* that describe dependencies between a data structure and those being pointed to by its pointer fields; *same-page constraints* dictating multiple data structures reside in the same physical page; *semantic constraints* that are derived from the semantics of the given data structures. All these constraints are associated with initial probabilities. They are conjoined and updated by the inference engine.

5.3.1 Primitive Constraints

Primitive constraints allow us to assign initial probabilities to boolean variables. Sample primitive constraints are f_{C_2} and f_{C_4} in Eq. (5.6) and (5.8) in Section 5.2. A primitive constraint is translated to a factor node in FG. It has only one outgoing edge to the boolean variable (Fig. 5.4). We consider the following primitive types: `int`, `float`, `double`, `char`, `string`, `pointer` and `time`.

Pointer: To decide the initial probability of a boolean variable denoting that a memory location is a pointer, we check whether the value of 4 contiguous bytes starting at a given location is within the virtual address space of a process (e.g., in the `.data`, `.bss`, `.heap`, and `.stack` sections). If true, we assign a HIGH initial probability (0.9) to the primitive pointer constraint, representing we believe the given location is likely a pointer. The other primitive constraints for the same location would be assigned LOW (0.1) initial probability. Note that setting HIGH/LOW initial probabilities is a standard practice in probabilistic inference. They do not reflect the intrinsic probabilities of boolean variables but rather what we believe. The absolute values of initial probabilities are hence *not* meaningful.

NULL pointers have the special value 0 that could be confused with a char or an integer, we will discuss how to handle them later.

String: To decide the initial probability of a string (a `char` array), we inspect the bytes starting with the given location. Firstly, a string ends with a NULL byte. Secondly, a string often contains printable ASCII ($[32, 126]$) or some special characters such as carriage return (CR), new-line (LF), and tab (Tab). If the two conditions are satisfied, the string constraint is set to HIGH, and other primitive constraints are set to LOW. It is possible that the bytes starting at x look like both a string and an integer. A unique advantage of probabilistic inference is that we can assign HIGH probabilities to multiple primitive constraints on x. Intuitively, it means we believe it could be multiple types. Assigning multiple HIGH probabilities regarding the same memory location allows the location playing different roles during inferencing and we do not need to make the decision upfront on if the location is a string or an integer. The inference process will eventually make the decision, by considering the probabilities from other parts of the FG through their dependencies.

Char: If a field with a `char` type is packed with other fields, that is, it is not padded to the word boundary, it becomes hard to disambiguate a char value from a byte that is just part of an integer or a floating point value. We have to set the probability to HIGH for all these primitive constraints. Fortunately, a `char` field is usually padded. Hence, we can limit our test to offsets aligned with the word boundary. More particularly, we only assign a HIGH probability to locations whose four bytes values fall into $\{0, 255\}$.

Int: Compared to the above primitive types, integers have fewer attributes to allow disambiguation. Theoretically, any four bytes could be a legitimate 32-bits integer value. In some cases, we are able to leverage semantic constraints to avoid assigning HIGH probabilities. For instance, it is often possible to find out from the data structure specification that an integer timeout field must have the value within 0-2^{10}. We could use such semantic information to assign LOW probabilities to values outside the range.

Float/double: According to the standard of floating-point format representation defined in IEEE 754 [5], we know the numerical value n for a `float` variable is: $n = (1 - 2s) \times$

$(1 + m \times 2^{-23}) \times 2^{e-127}$, where s is a sign bit (zero or one), m is the significand (i.e., the fraction part), and e is the exponent. Fig. 5.5 shows this representation.

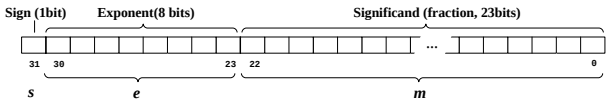


Fig. 5.5.: Float point representation

Now if we examine the value of a floating point variable, suppose $s = 0$ and $e = 0$, then the numerical value is very small, and it is within $[0, 2^{-126}]$. Thus, we could infer that most floating point values have their leftmost 9 bits set with at least one bit. If all the leftmost 9 bits have been set with 1 (i.e., $s = 1$, $e = 255$), then the numerical value for such floating point variable is within $[-2^{128}, -2^{105}]$, which is a very large negative value. If the sign bit is 0 (i.e., $s = 0$, $e = 255$), then the numerical value is within $[2^{105}, 2^{128}]$, which is a very large positive value. In practice, we *believe* floating point values rarely fall into such ranges.

Therefore, we check the hexadecimal value at page offset x, that is the $*x$. If $*x <$ `0x007fffff`, `0x7f800000` $< *x <$ `0x7f8fffff`, or `0xff800000` $< *x <$ `0xffffffff`, we set the initial probability of $F(x)$ to LOW, otherwise HIGH. The `double` type is handled similarly. The details are elided.

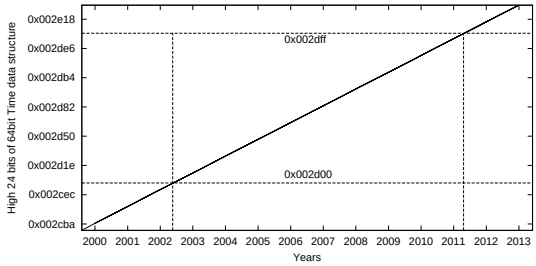


Fig. 5.6.: Common high bits in a time data structure

Time: Time data structures are often part of many interesting data structures. A time data structure maintains the cumulative time units (e.g., seconds or microseconds) since

a specific time in the past. Its bit representation has a general property that high bits are less frequently updated than lower bits. It allows us to create constraints to infer time data structures by using common bit fields for all time values during a given period.

For example, Fig. 5.6 shows the values of the highest 24 bits of a time data structure of 64 bits over a period of time. During the period between mid-2002 and mid-2011, the highest 24 bits have the common value, *0x002d*. These constraints can be used to infer time object instances. Similarly, in 32-bit Unix systems, the time data structure has 32 bits. The four highest bits are updated around every 8.5 years.

Lastly, zeros present an interesting case for us because it could have multiple meanings: an integer with the value 0; an empty string; a null pointer; and so on. We assign HIGH probabilities to all these types except for cases in which the fields in vicinity are also having zero values. The reason is that consecutive zeros often imply unused memory regions. In particular, if the number of consecutive zeros exceed the size of the data structure we are interested in, the probability is set to LOW. In general, the probability is inversely proportional to the length of consecutive zeros.

5.3.2 Structural Constraints

DIMSUM takes the specification of a set of data structures as input. The specification includes the field offsets and field types of the data structures. For instance, if data structure T of interest has a pointer field of T_x type, T_x's definition is transitively included as well. Then we translate each type into a boolean structural constraint describing the dependencies between the data structure and its fields. Eventually, the boolean constraints are modeled into the factor graph automatically.

A structural constraint is intended to denote the dependence that as long as a given location x is an instance of T, then x's offsets must be of the fields types described by T's specification. An example of such constraint was introduced in Eq. (5.1) in the beginning of Section 5.2. In particular, for each memory location, we introduce a boolean variable to predicate if it is an instance of T. We also introduce a factor node to represent the

constraint. Edges are introduced between the factor node and the newly introduced variable and the variables describing the corresponding primitive field types. These variables were introduced in the previous step when primitive constraints were generated. A sample factor graph after such process is the subgraphs rooted at f_{C_1} in Fig. 5.4. Since the constraint is always certain, meaning as long as x is of T type, its offsets must follow the structure dictated by T's definition. The probability of structural constraints are always 1.0, meaning that they must hold (see Eq. (5.5) in Section 5.2).

5.3.3 Pointer Constraints

If a field $a + f$ is a pointer $T*$, in the structural constraint, besides forcing $a + f$ to be a pointer, we should also dictate $*(a + f)$ be of T type. In particular, we will add boolean variables $T(*(a + f))$ to the structural constraint. Note that T could belong to primitive types, user defined types, or function pointers. Variables $\texttt{utmplist}(*(a + 384))$ and $\texttt{utmplist}(*(a + 388))$ in Equation (5.1) are examples. Ideally, these variables have been introduced at the time when we typed the page of the pointer target (e.g., the page that $*(a + 384)$ points to), we only need to introduce edges from the factor node to such variables. However, since we do not have page mapping information, it is impossible to identify the physical location of the pointer target and the corresponding boolean variable.

We observe that the lower 12 bits of a virtual address indicates the offset within a physical page. Hence, while we cannot locate the concrete physical page corresponding to the given address, we can look through all physical pages and determine if there are some pages that have the intended type at the same specified offset.

From now on, we denote a memory location with symbol a^p, with a being the page offset and p the physical page ID. Hence, a boolean variable predicating a location a^p has type T is denoted as $T(a^p)$. For pointer constraints, we introduce boolean variables predicating merely on offsets. In particular, $T(*((a + f)^p)\&0x0fff)$ represents that there is at least one physical page that has a type T instance at the page offset (the least 12 bits) of

the pointer target at location $(a + f)^p$. We call such boolean variables the *offset variables* and the previous variables considering both offsets and page IDs the *location variables*.

We further introduce pointer constraints that are an implication from an offset variable to the disjunction of all the location variables with the same offset, to express the "there is at least one" correlation. The probability of the constraint is not 1.0 as it is likely there is not such a physical page if the page has been re-allocated and overwritten. Ideally, the probability is inversely proportional to the duration between the process termination and the analysis. In this paper, we use a fixed value δ to represent that we believe in δ probability such a remote page is present. With pointer constraints, we are able to construct an FG that connects variables in different physical pages and perform global inference such that probabilities derived from various places can be fused together.

Example. Let's revisit the example in Section 5.2.2. Regular variables x_1, x_2, and x_3 now denote $utmplist(a^p)$ for a given page offset a, $P_{next}((a + 384)^p)$, and $P_{prev}((a + 388)^p)$, respectively. Superscript p can be considered as the id of the physical page. Offset variables y_1 and y_2 represent $utmplist(*((a+384)^p)\&0x0fff)$ and $utmplist(*((a+388)^p)\&0x0fff)$. Constraint C_1 (i.e. Equation (5.2)) is extended to the following.

$$x_1 \rightarrow x_2 \wedge x_3 \wedge y_1 \wedge y_2 \tag{5.14}$$

The probability of f_{C_1} remains 1.0. Assume we have three physical pages p, q, and r in DIMSUM's memory page input. Let $b = *((a + 384)^p)\&0x0fff$ and $c = *((a + 388)^p)\&0x0fff$, the page offsets of the pointers stored at $(a + 384)^p$ and $(a + 388)^p$.

Let x_4, x_5 and x_6 denote $utmplist(b^p)$, $utmplist(b^q)$, and $utmplist(b^r)$, respectively; and x_7, x_8 and x_9 denote $utmplist(c^p)$, $utmplist(c^q)$, and $utmplist(c^r)$. These variables are created when typing pages p, q and r. The pointer constraints are thus represented as follows.

$$(C_5) \quad y_1 \rightarrow x_4 \vee x_5 \vee x_6 \tag{5.15}$$

$$(C_6) \quad y_2 \rightarrow x_7 \vee x_8 \vee x_9 \tag{5.16}$$

The factor for C_5 is defined as follows.

$$f_{C_5}(y_1, x_4, x_5, x_6) = \begin{cases} \delta & \text{if } (y_1 \rightarrow x_4 \vee x_5 \vee x_6) = 1 \\ 1 - \delta & \text{otherwise} \end{cases} \tag{5.17}$$

Recall δ reflects our overall belief of the completeness of the input memory pages. Factor f_{C_6} can be similarly defined and hence omitted. Fig. 5.7 presents the FG enhanced with the pointer constraint C_5. Observe that while many constraints (e.g., primitive constraints) are local to a page, the pointer constraint C_5 and the enhanced structural constraint C_1 correlate information from multiple pages. For instance, the probability of x_5 in page q can be propagated through the path $x_5 \Rightarrow f_{C_5} \Rightarrow y_1 \Rightarrow f_{C_1} \Rightarrow x_1$ to the goal variable x_1. The probability of x_1, which is the fusion of all the related probabilities, indicates if we have an instance `utmplist` at the given address a.

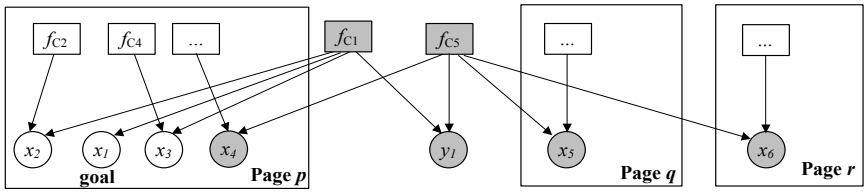


Fig. 5.7.: The factor graph enhanced with a pointer constraint. Constraints C_3 and C_6 are elided for readability. The modified part is highlighted. Constraints and variables local to a page are boxed.

5.3.4 Same-Page Constraints

We observe that the values of multiple pointer fields may imply that the points-to targets are within the same page. For instance in `struct utmplist` in Fig. 5.3, if the higher 20 bits of the addresses stored in fields $a + 384$ and $a + 388$ are identical, we know their points-to targets must be within the same page. Hence, if we observe field $a + 384$ in page q and $a + 388$ in page r hold instances of `utmplist`, they should not be considered as support for a in p holds an instance of `utmplist`. We leverage same-page constraints to reduce false positives.

If the values of multiple pointer fields are within the same page, these pointers should not have individual pointer constraints. Instead, we introduce a joint pointer constraint that dictates the objects being pointed to by the pointers must reside in the given offsets of the same page. In our running example, the structural constraint in Equation (5.14) is changed to the following.

$$x_1 \ \rightarrow \ x_2 \wedge x_3 \wedge y_{1\cdot 2} \tag{5.18}$$

Variable $y_{1\cdot 2}$ represents a joint offset variable. It represents that there is at least one physical page that has `utmplist` instances at offsets specified by $b = *((a+384)^p)\&\mathit{0x0fff}$ and $c = *((a+388)^p)\&\mathit{0x0fff}$. The joint constraint is hence the following.

$$y_{1\cdot 2} \ \rightarrow \ (x_4 \wedge x_7) \ \vee \ (x_5 \wedge x_8) \ \vee \ (x_6 \wedge x_9) \tag{5.19}$$

5.3.5 Semantic Constraints

Besides the aforementioned constraints, there could exist semantic constraints imposed by the data structure definitions. For example, a field `pid` tends to have value ranges from 0 to 40000; an unused fields tends to have zero values. Meanwhile, it is also possible that a particular data structure field has value invariant. As such, semantic constraints can be used to prune unmatched fields.

5.3.6 Staged Constraints

The previous discussion implies that we need to create many boolean variables for each memory location. In particular, for each offset in every page, we introduce variables to predicate on its various primitive types and types of interest. Constraints are introduced among these variables, describing any possible dependencies. The order of introducing the constraints is *irrelevant*. The entailed FG is often very large and takes a lot of time to resolve. We develop a simple preprocessing phase to reduce the number of variables and constraints. In particular, we first scan each input page and construct primitive constraints,

describing if each offset is an integer, a char, a pointer, etc. In the second step, we construct structural and other constraints. We avoid introducing a variable predicating on if a base address a is of type T if any of the corresponding field primitive constraints has a LOW probability. We leverage the observation that such inference is simple and does not need FG to proceed.

5.4 Implementation

```
using System;
using MicrosoftResearch.Infer.Models;
using MicrosoftResearch.Infer.Distributions;
using MicrosoftResearch.Infer;
public class Simple_DIMSUM_Example
{
    static void Main()
    {
        //1. Declare Boolean Variables
        Variable<bool> x1 = Variable.Bernoulli(0.5).Named("x1");
        Variable<bool> x2 = Variable.Bernoulli(0.5).Named("x2");
        Variable<bool> x3 = Variable.Bernoulli(0.5).Named("x3");

        //2. Define Probabilitics Model
        Variable.ConstrainEqualRandom<bool, Bernoulli>
               (!x1 | (x2 & x3), new Bernoulli(1));
        Variable.ConstrainEqualRandom<bool, Bernoulli>
               (x2, new Bernoulli(0.9));
        Variable.ConstrainEqualRandom<bool, Bernoulli>
               (!(x2 & x3) | x1, new Bernoulli(0.8));
        Variable.ConstrainEqualRandom<bool, Bernoulli>
               (x3, new Bernoulli(0.9));

        //3. Create an Inference Engine
        InferenceEngine ie = new InferenceEngine();
        ie.ShowFactorGraph = true;

        //4. Compute Marginal Probabilities
        Console.WriteLine("x1: " + ie.Infer(x1));
    }
}
```

Fig. 5.8.: Sample code on using Infer.NET to model C_1 - C_4 in our working example and compute $p(x_1)$.

The key part of DIMSUM is the probabilistic inference component. We use Infer.NET, a framework for belief propagation with factor graphs as its internal model [65]. To use such framework, we did the following: (1) declare the boolean variables associated with each candidate memory cell, (2) define the corresponding constraints using their modeling API in C# code, (3) create an inference engine, and (4) execute the inference query over the boolean variables of interest.

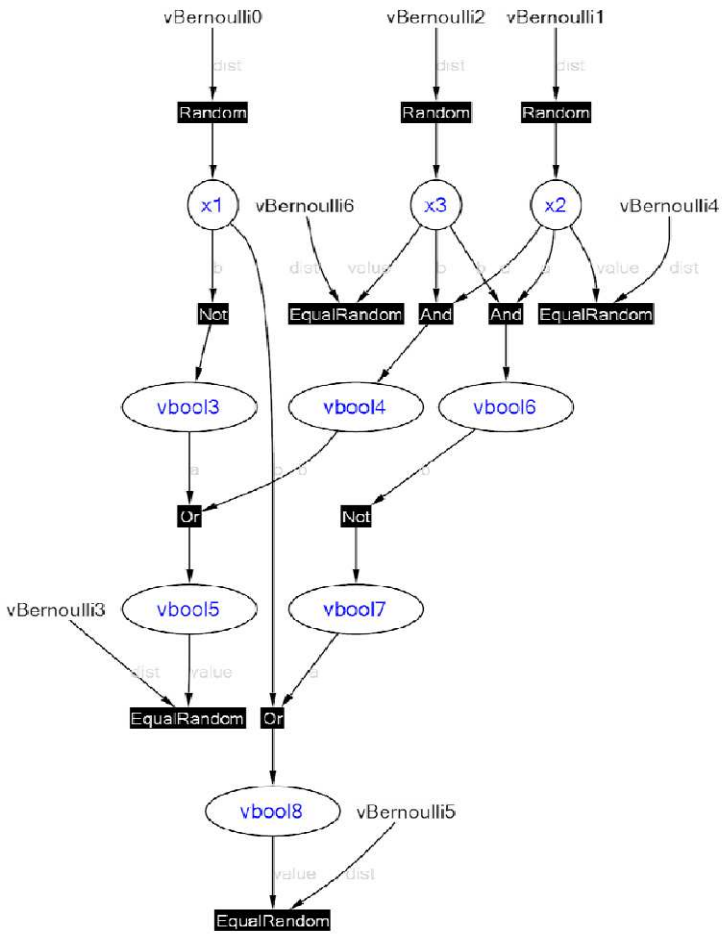


Fig. 5.9.: Factor graph of the example code from Infer.NET. Note each variable is denoted as a circle and each factor or constraint as a square. If a variable participates in the factor or the constraint, then an edge is shown between the corresponding circle and square.

We use our working example (Equation [2-8])) to demonstrate the process. As illustrated in Fig. 5.8, we declared three boolean variables $x1$, $x2$, and $x3$, initially with Bernoulli value of 0.5 (meaning they have a 50% possibility of being the instances as we have no observations yet). Lines 15-16 model Equation (5.5). Similarly, Equation (5.6), Equation (5.7) and Equation (5.8) are modeled from line 17 to line 22. After that, we created an inference engine (line 25), and visualized the factor graph (line 26) (for debug and understanding purpose). Finally, we computed the marginal probability of $x1$ (line 29), and eventually we got $p(x1) = 0.71$, which corresponds to the probability of the given

address being an instance of the type of interest. Also, in this factor graph, there are a total of 9 boolean variables (represented as circle), and 13 factors or constraints (represented as square). Some variables and factors are generated internally.

The implementation of our inference component is mainly in C#. When using our system, users need to provide the subject data structure specification and memory pages. DIMSUM then processes the pages, generates constraints, and compiles the constraints to C# code, which will further get compiled and linked with other supporting libraries from Infer.NET. Running the compiled binary delivers the final data structure instance(s) uncovered.

5.5 Evaluation

In our DIMSUM evaluation, we first present our experiment setup in Section 5.5.1, then the experimental results of discovering data structure instances *without memory mapping information* in Linux platform in Section 5.5.2. We also evaluate the sensitivity of the setting of the HIGH/LOW probabilities in Section 5.5.3. Finally we evaluate the cost of DIMSUM in Section 5.5.4.

5.5.1 Experiment Setup

Our evaluation scenario involve dead memory pages. Such dead pages come from terminated processes, and the virtual memory mapping information is no long available. Essentially, DIMSUM takes a set of (dead) physical pages, and identifies data structure instances in them.

To enable the evaluation, we have to first collect the ground truth so that we can compare it with the results reported by DIMSUM to measure false positives (FP) and false negatives (FN). We extract the ground truth in two steps: The first step is to extract data structure instances from the application process' virtual space via program instrumentation. In particular, given a data structure of interest, we instrument the program to log allocations and de-allocations of the data structure. Then, upon process termination, we visit the

log file to identify the data structure instances that have been deallocated but not yet destroyed. These are essentially the ground truth. The second step is to find the physical residence pages of these instances using page mapping information. The second step is needed as DIMSUM operates directly on physical pages. We implement the ground truth extraction component in QEMU [80] and an Android emulator (based on QEMU as well). Specifically, we trap the system call `sys_exit_group` to perform the extraction. Note that, on the Android platform, executables are in the form of byte code and their execution is object oriented. We have to tap into the emulator to translate object references to memory addresses.

The input to DIMSUM is all the dead memory in the system [1]. To acquire all dead pages across the system, we enhance QEMU to traverse kernel data structures such as memory zones and page descriptors.

To emulate the scenario where some dead pages – especially those containing data structures of interest or their supporting data (e.g. those data structures that are pointed to by pointers in the data structure of interest) – are reused for new processes, we vary the number of dead pages provided to DIMSUM. In our experiments, we study three settings: 33%, 67%, and 100%. For example, 33% means that we randomly select 33% of the dead pages as input to DIMSUM.

Comparison with value-invariant and SigGraph We also compare DIMSUM with other techniques that can be adopted for un-mappable memory forensics. The first technique to compare with is a value invariant approach similar to the approaches in [32, 41, 42], leveraging field value patterns to identify data structure instances. The patterns we use are mainly the value patterns for pointers and those derived from domain knowledge.

The second technique to compare with is a variant of SigGraph [31]. SigGraph is a brute force memory scanning technique. It leverages the points-to relations between data structures and uses a points-to graph rooted at a data structure as its signature for scanning. *Note that the original SigGraph relies on page mappings to traverse pointers and thus*

[1]Live memory forensics is outside the scope of this chapter. It can be achieved by techniques guided by page mapping such as SigGraph [31] (presented in Chapter 4 and KOP [30]).

cannot be applied to our "un-mappable memory" scenario. We implement a variant of SigGraph, called SigGraph$^+$, which tries to aggressively traverse pointers even without page mappings. In particular, during scanning, SigGraph$^+$ tries to traverse a pointer without mappings, it identifies the page local offset (the lower 12 bits) of the pointer value, say x, and then tries to look for a match at offset x among all dead pages. For instance, assume the graph signature of a type T is that its field f points to a type T_1. Assume the page offset of the pointer value at the f field is x. As along as it can find at least one page whose offset x is an instance of T_1, SigGraph$^+$ considers that an instance of T is identified.

5.5.2 Effectiveness

In the following, we present the experimental results of applying DIMSUM to discover (1) user login records, (2) browser cookies, (3) email addresses, and (4) messenger contacts from applications on Linux. A summary of these experiments is presented in Table 5.3. The specific data structures of interest, the applications, and the size of the target data structures are reported in the 1^{st}, 2^{nd}, and 3^{rd} column, respectively. The 4^{th} column reports the total number of input pages provided to DIMSUM, and the 5^{th} column shows the total number of true instances. We compare DIMSUM with value-invariant and SigGraph$^+$. Columns "#R", "FP%" and "FN%" report the total number of instances identified by the corresponding approaches, the False Positive (FP), and False Negative (FN) rate, respectively.

From this table, we make the following observations: (1) Value-invariant has high FPs and very low FNs, (2) SigGraph$^+$ has high FPs as well, and low FNs, (3) DIMSUM has significant less FPs and low FNs. On average, the FPs for value-invariant, SigGraph$^+$, and DIMSUM are 65.5%, 38.5%, and 19.0%, respectively; the FNs are 0.4%, 8.3%, and 5.4%. Note the real FP rate of DIMSUM may be lower than the reported number because the two 100% false positive cases (those with superscripts in Table 5.3) can be easily pruned because the absolute value of the probability is very low (below 0.5). More details will be discussed in the case study. Precluding these two cases, DIMSUM has only 8.0% FP.

Table 5.3: Summary on discovering data instances of interest for user applications in Linux. Note the two * false positives can easily be pruned by looking at the absolute value of the probability.

Data of Interest	Benchmark Program	Size	#Input Pages	#True Inst.	Value-Invariant			SigGraph+			DIMSUM				
					#R	FP%	FN%	#R	FP%	FN%	Factor Graph #Var	Factor Graph #FC	#R	FP%	FN%
Login record utmp	last 2.85	392	27266	8	48	83.3	0.0	6	0.0	25.0	507	709	8	0.0	0.0
			18186	6	46	87.0	0.0	2	0.0	66.7	435	609	6	0.0	0.0
			8898	0	40	100.0	0.0	0	0.0	0.0	405	567	1	100.0*	0.0
Browser Cookies	w3m 0.5.1	80	31303	23	93	76.3	0.0	35	34.3	0.0	1874	2613	22	0.0	4.3
			20848	23	93	76.3	0.0	35	34.3	0.0	1874	2613	22	0.0	4.3
			10423	0	70	100.0	0.0	9	100.0	0.0	1260	1782	9	100.0*	0.0
	chromium 8.0.552.0	44	45308	25	89	71.9	0.0	82	69.5	0.0	1068	1157	45	44.4	0.0
			30205	19	61	68.9	0.0	56	66.1	0.0	976	1037	38	50.0	0.0
			15103	9	49	81.6	0.0	43	79.1	0.0	784	833	16	43.8	0.0
Address Book	pine 4.64	144	33186	124	1216	90.3	4.8	229	48.5	4.8	13056	17607	101	0.0	18.5
			22123	96	1174	92.2	2.1	174	50.1	10.4	11468	15594	79	0.0	17.7
			11063	63	1142	94.5	0.0	88	56.8	39.7	8992	11683	42	0.0	33.3
	Sylpheed 3.0.3	48	46504	309	412	25.0	0.0	412	25.0	0.0	12040	16588	323	5.0	0.6
			31002	204	244	16.4	0.0	244	16.4	0.0	7223	9644	194	0.0	4.9
			15502	92	128	28.1	0.0	128	28.1	0.0	3537	4710	82	0.0	10.9
Contact List	pidgin 2.4.1	60	58743	300	491	38.9	0.0	485	38.8	1.0	8874	12543	297	0.0	1.0
			39163	198	259	23.6	0.0	254	22.8	1.0	5241	7521	196	0.0	1.0
			19580	98	130	24.6	0.0	126	23.0	1.0	2595	3724	97	0.0	1.0

A Case Study

We further zoom in on one case to concretize our discussion. In the study of utility program `last`, we acquired 8 true instances and 27266 input pages, including the 2 pages that contain the 8 true instances.

The detailed result with the three different settings are presented in Fig.s 5.10(a), 5.10(b), and 5.10(c), respectively. Note in these figures, the X-axis represents the page offset within a physical page. For DIMSUM, Y-axis represents the probability of a match. For value-invariant and SigGraph$^+$, since there is no probability associated, we just add "V" and "S" to the Y-axis to show their results. Also, in these figures, a ground truth is marked with $\times$.

A data point marked with *both* $\times$ and the symbol of the technique means the technique identifies a true positive (TP). For example, the data point marked with *both* $\times$ and Δ as indicated in the top cluster in Fig. 5.10(a) is a TP for DIMSUM. A point with only a technique symbol indicates a false positive (FP). For example, the nodes in the right bottom of Fig. 5.10(a) are FPs for the value-invariant approach. *Note that DIMSUM only reports nodes in the top cluster*. Hence, those DIMSUM data points that are not in the top cluster are not FPs, even though they are not marked with $\times$. A point with only $\times$ indicates a FN. For DIMSUM, any single $\times$ symbols that are not in the top cluster are FNs.

When 100% input pages are provided to DIMSUM (as shown in Fig. 5.10(a)), DIMSUM successfully identifies all ground truth without any FPs or FNs – in the top cluster of points whose probability is greater than 0.95. SigGraph$^+$ identifies 6 instances with 25% FN, and the value-invariant approach identifies 48 instances with 83.3% FP. Next, we randomly select 67% of the input pages. One page containing 2 true instances is precluded as the result of the random selection. The result is shown in Fig. 5.10(b). DIMSUM identifies all remaining 6 true instances in the top cluster. In contrast, SigGraph$^+$ in this case identifies only 2 true instances, because for the other 4 instances, their graph signatures are not complete due to the missing pages. In contrast, *DIMSUM is able to survive as it aggregates sufficiently high confidences from the fields in the remaining 67% pages.* Finally, when 33% pages are selected, all the true instances are precluded. DIMSUM

identifies one instance in its top cluster as shown in Fig. 5.10(c), which is a false positive. But we want to point out DIMSUM in the mean time determines that the instance has only a probability lower than 0.50. The user can easily discard such results.

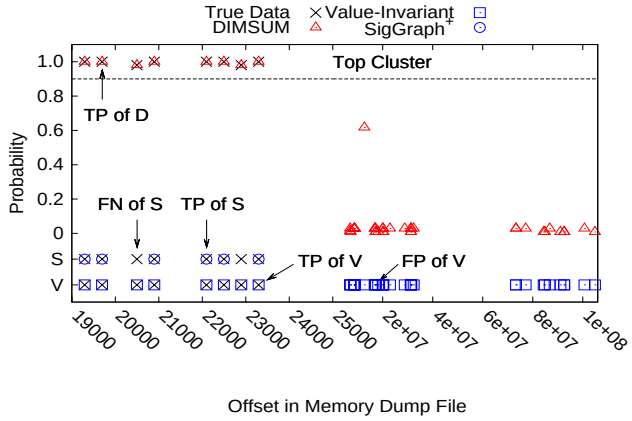
False Positive Comparison

Below we discuss the FP and FN comparison in more details.

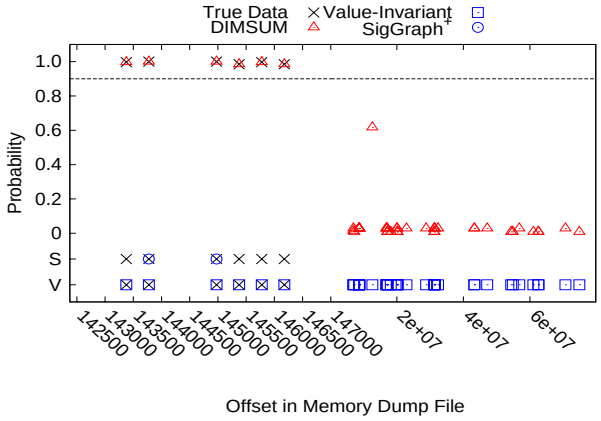
Value-invariant has high FPs because it only looks at the value patterns of the fields in the target data structure. It does not try to collect additional confidence from the child data structures (those pointed-to by the pointer fields in the target data structure). The end result is that it admits lots of bogus data structure instances.

SigGraph$^+$ also has high FPs. Recall that as an extension of SigGraph, SigGraph$^+$ also uses the points-to graph signature to search for instances of a data structure (Section 5.5.1). Given a pointer field "`T* f;`" of the data structure, it tries to confirm if `*(f)` holds an instance of `T`. However, since memory mapping is not available, `f` cannot be resolved, it aggressively looks for any instance of `T` among all pages at the page offset `f&0x0fff`. The consequence is that it may find such an instance that was indeed not pointed-to by `f`. The situation is particularly problematic when `T` is a popular type (e.g., string) so that there are instances of this type at almost any page offset. Another main reason is that it cannot propagate probabilities among different data structures like DIMSUM to reduce the mis-perception.

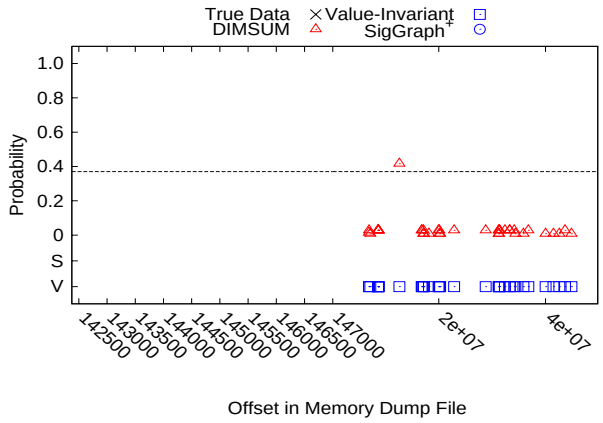
DIMSUM has low FPs. As explained in Section 5.5.2, the only case (`utmp` and the 33% setting) with a 100% FP rate indeed has a very low probability, and is hence an easy-to-prune FP. The result strongly supports the effectiveness of DIMSUM. Probabilistic inference indeed allows global reasoning over all the connected data structures, collecting and aggregating confidence from all over the places, eventually distinguishing the true positives. The DIMSUM FPs for `chromium` are mainly caused by the simplicity of the cookie data structure. In other words, DIMSUM does not have a lot of sources to collect enough confidence to distinguish true positives from others. Interestingly, for the 5% FPs



(a) `last` (100%)



(b) `last` (67%)



(c) `last` (33%)

Fig. 5.10.: Effectiveness evaluation of DIMSUM for discovering user login record data structure `utmp`.

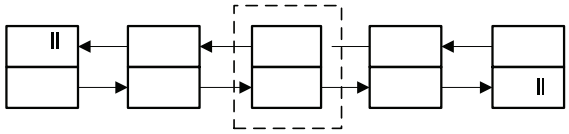


Fig. 5.11.: An abstraction of the `utmp` case. The node in the middle is missing.

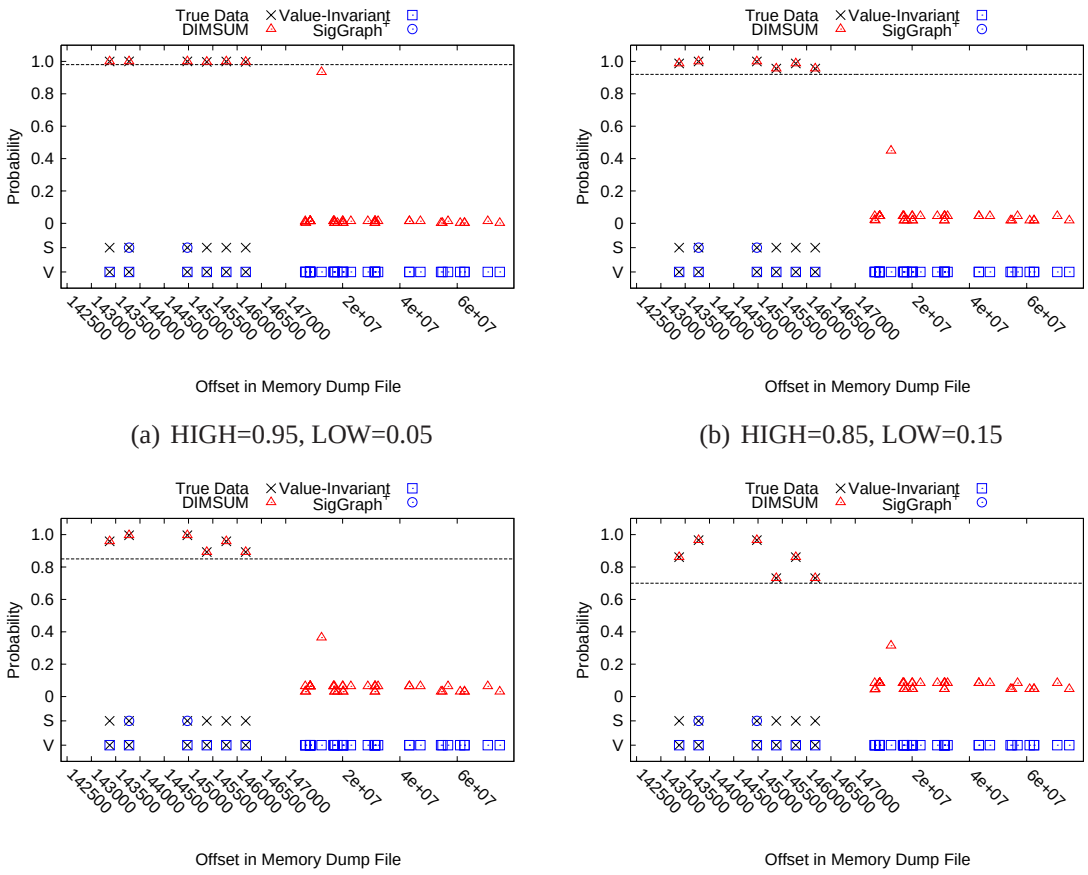
in `Sylpheed`, they are mainly caused by the fact the some garbage pages happen to have some instances that satisfy our constraints, and when the garbage pages are not selected (33% and 67% cases), these FPs are gone.

False Negative Comparison

Value-invariant has the lowest FNs. This is understandable as it is the least restricting method. It admits everything that appears to be an instance of the target data structure based on their value patterns.

Both SigGraph$^+$ and DIMSUM have high FNs for the `pine` case. The main reason is the lack of support due to the missing pages, especially for the settings of 33% and 67%. In other words, the child data structures are not present in the pages provided to these techniques. Another reason for FNs is cross page data structures. There are some data structure instances spanning two pages. None of these techniques including DIMSUM currently handle cross-page data structures because consecutive virtual pages do not correspond to consecutive physical pages. We will leave it to our future work. It contributes to the FNs for the 100% setting.

In some cases, DIMSUM is even superior to the less restricting SigGraph$^+$ in terms of FNs, for example, the `utmp` structure in `last-2.85`. The main reason is that SigGraph$^+$ is doing binary reasoning, and hence a piece of memory is either an instance of interest or not. In contrast, DIMSUM does not draw binary conclusions but rather collects little pieces to gradually form the right picture. Fig. 5.11 abstracts the case. The whole linked list represents the `utmp` linked list. And it is freed. The node in the middle is missing



(a) HIGH=0.95, LOW=0.05

(b) HIGH=0.85, LOW=0.15

(c) HIGH=0.80, LOW=0.20

(d) HIGH=0.75, LOW=0.25

Fig. 5.12.: The threshold impact on the experimental result

(the page was reused). The graph signature used in SigGraph$^+$ is a node with its preceding node and succeeding node, meaning an instance of `utmp` is recognized if the `prev` and `next` pointers also point to instances of `utmp`. In this case, SigGraph$^+$ cannot recognize the head or tail due to the null pointers. It cannot recognize the 3^{rd} node as it is missing. As a result, it can not recognize the 2^{nd} or 4^{th} nodes either. In contrast, DIMSUM never makes binary judgements on individual nodes. Instead, it models them into a network of constraints. In this case, two factor graphs, one containing the 1^{st} and 2^{nd} nodes and the other the 3^{rd} and 4^{th} are formed and resolved. Aggregating probabilities in the two graphs indeed sufficiently identifies the true positives.

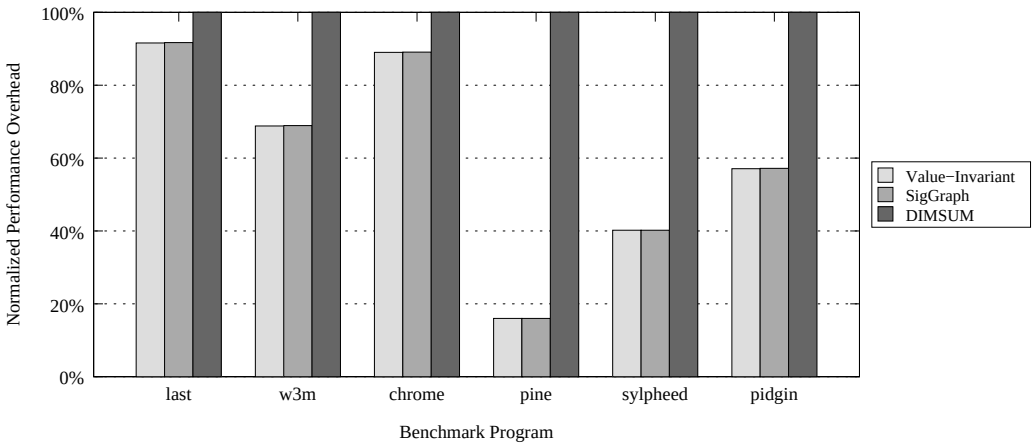


Fig. 5.13.: Comparison of the normalized execution time

5.5.3 Sensitivity on the Threshold

By default, we set the threshold probabilities HIGH=0.90, and LOW =0.10. To study the impact of these threshold variables on the final result, we take the second case of discovering `utmp` instances (Fig. 5.10(b)) as an example, and change the values of HIGH and LOW and observe the result. We make four different settings, namely setting HIGH=0.95 and LOW=0.05, HIGH=0.85 and LOW=0.15, HIGH=0.80 and LOW=0.20, and HIGH=0.75 and LOW=0.25. The result is illustrated in Fig. 5.12(a)-Fig. 5.12(d). We could see for all these settings, the top cluster still contains the true instances. The results for other cases are similar. The conclusion is hence that the result is not sensitive to the thresholds.

5.5.4 Performance Overhead

In this experiment, we study the cost of DIMSUM. The performance data is collected in a Windows Vista system with 2GB memory and a 2.16Ghz CPU. The result is presented in Fig. 5.13. All execution times are normalized based on DIMSUM's time. We find that the execution time of DIMSUM is reasonable, in comparison with that of the value-invariant approach (44.8% of DIMSUM) and SigGraph$^+$ (45.1% of DIMSUM). Also observe that it is much slower in Android platform because there are large amount of strings (in UNI-

CODE form) in the snapshot; the string searching and constraint resolving takes more time. The space overhead is decided by the size of factor graphs. We could not get the precise memory consumption as Infer.NET has its own memory management system. We instead present the number of variables (#Var) and the number of constraints or factors (#FC) in the 12^{th} and 13^{th} columns respectively in Table 5.3.

5.6 Summary

Uncovering semantic data of interest in memory pages without memory mapping information is a desirable capability in computer forensics. Such a capability is realized by DIMSUM, a system that extracts the data structure instances – with confidence – from a set of memory pages without memory mapping information. In this chapter, we have presented the design, implementation, and evaluation of DIMSUM. Our experimental results show that DIMSUM achieves higher accuracy than previous non-probabilistic approaches when discovering data from unmappable memory.

6. APPLICATIONS

There are many security applications of our framework. In this chapter, we demonstrate how our framework can be used for memory forensics, vulnerability discovery, kernel rootkit detection, and kernel version inference.

6.1 Memory Image Forensics

Memory image forensics is a process to extract meaningful information from a memory dump. Examples of such information are the IP addresses to which the application under investigation is talking and files being accessed. Data structure definitions play a critical role in the extraction process. For instance, without data structure information, it is hard to decide if four consecutive bytes represent an IP address or are just a regular value. All the components in our framework can support memory forensics, especially SigGraph and DIMSUM as forensics is the nature of their design that was demonstrated in Chapter 5. For REWARDS, it enables analyzing memory dumps for a binary without symbolic information. In the following, we demonstrate how REWARDS can be used to type reachable memory as well as some of the unreachable (i.e., dead) memory.

6.1.1 Typing Reachable Memory

In this case study, we demonstrate how we use REWARDS to discover IP addresses from a memory dump using the hierarchical view (Section 3.2.5). We run a web server `nullhttpd-0.5.1`. A client communicates with this server through `wget-1.10.2`. The client has IP `10.0.0.11` and the server has IP `10.0.0.4`. The memory dump is obtained from the server at the moment when a system call is invoked to close the client connection. Part of the memory dump is shown in Figure 6.1. The IPs are underlined in the

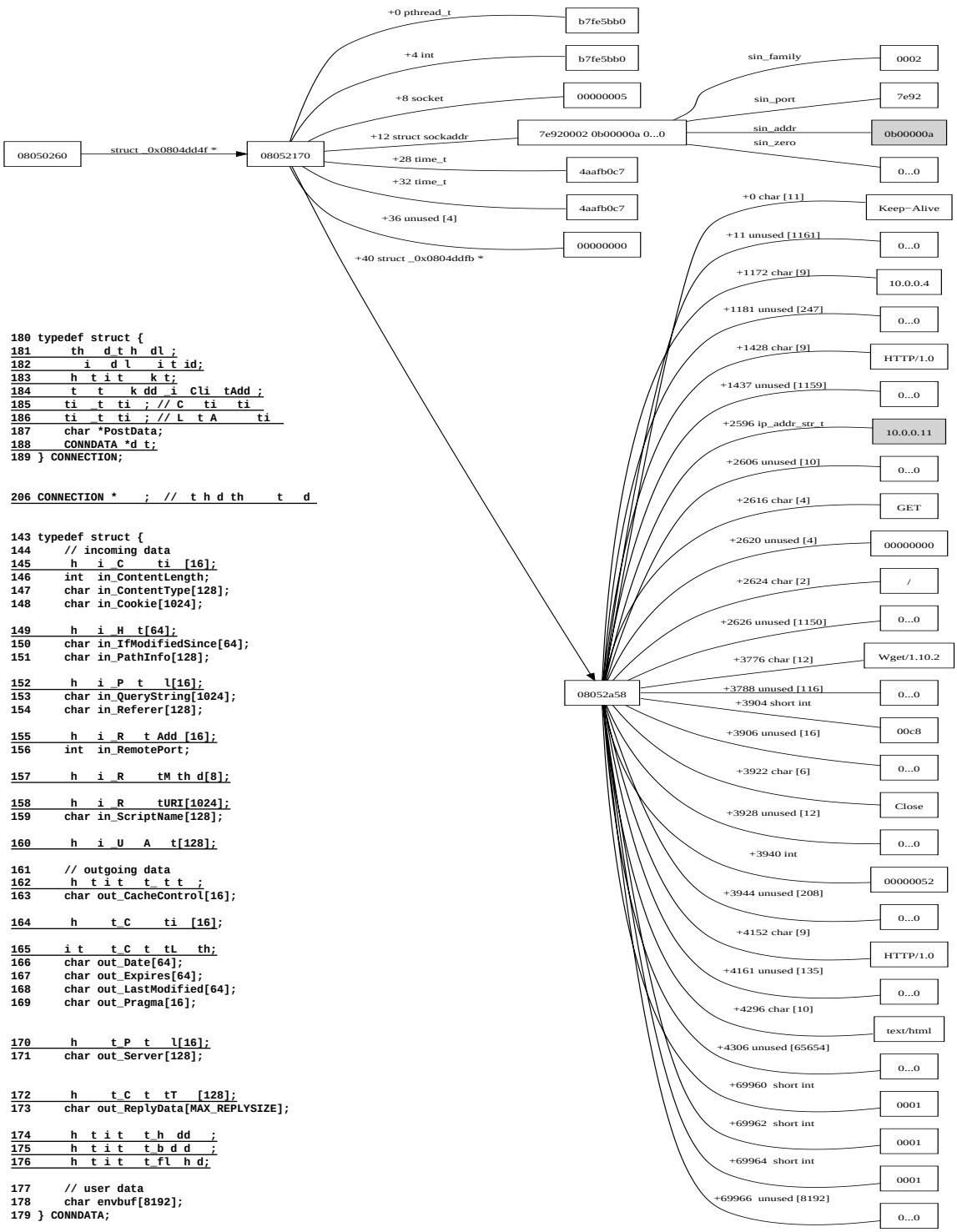
```
...
08052170  b0 5b fe b7 b0 5b fe b7  05 00 00 00 02 00 92 7e
08052180  0a 00 00 0b 00 00 00 00  00 00 00 00 c7 b0 af 4a
08052190  c7 b0 af 4a 00 00 00 00  58 2a 05 08 00 00 00 00
080521a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
...
08052a50  00 00 00 00 59 31 01 00  4b 65 65 70 2d 41 6c 69
08052a60  76 65 00 00 00 00 00 00  00 00 00 00 00 00 00 00
08052a70  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
08052ee0  00 00 00 00 00 00 00 00  00 00 00 00 31 30 2e 30
08052ef0  2  30 2  34 00 00 00 00  00 00 00 00 00 00 00 00
08052f00  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
08052fe0  00 00 00 00 00 00 00 00  00 00 00 00 48 54 54 50
08052ff0  2f 31 2e 30 00 00 00 00  00 00 00 00 00 00 00 00
08053000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
08053470  00 00 00 00 00 00 00 00  00 00 00 00 31 30 2e 30
08053480  2e 30 2e 31 31 00 00 00  00 00 00 00 00 00 00 00
08053490  47 45 54 00 00 00 00 00  2f 00 00 00 00 00 00 00

080534a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
08053910  00 00 00 00 00 00 00 00  57 67 65 74 2f 31 2e 31
08053920  30 2e 32 00 00 00 00 00  00 00 00 00 00 00 00 00
08053930  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
08053990  00 00 00 00 00 00 00 00  c8 00 00 00 00 00 00 00
080539a0  00 00 00 00 00 00 00 00  00 00 43 6c 6f 73 65 00
080539b0  00 00 00 00 00 00 00 00  00 00 00 00 52 00 00 00
080539c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
08053a90  48 54 54 50 2f 31 2e 30  00 00 00 00 00 00 00 00
08053aa0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
08053b20  74 65 78 74 2f 68 74 6d  6c 00 00 00 00 00 00 00
08053b30  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
08063ba0  01 00 01 00 01 00 00 00  00 00 00 00 00 00 00 00
08063bb0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
...
```

Fig. 6.1.: Part of a memory dump from null-httpd. String and IP address are underscored.

figure. From the memory dump, it is very hard for human inspectors to identify those IPs without a meaningful view of the memory. We use REWARDS to derive the data structure definitions for `nullhttpd` and then construct a hierarchical view of the memory dump following the method described in Section 3.2.5.

The relevant part of the reconstructed view is presented in Figure 6.2(a). The root represents a pointer variable in the global section. The outgoing edge of the root leads to the data structure being pointed to. The edge label "`struct_0x0804dd4f *`" denotes that this is a heap data structure whose allocation PC (also its abstraction) is `0x0804dd4f`. According to the view construction method, the memory region being pointed to is typed according to the derived definition of the data structure denoted by `0x0804dd4f`, resulting in the second layer in Figure 6.2(a). The memory region starts at `0x08052170` and is denoted by the node with the address label. The individual child nodes represent the different fields of the structure (e.g. the first field is a thread id according to the semantic tag `pthread_t`, the fourth field (with offset +12) denotes a `sockaddr` structure). The last field (with offset +40) denotes another heap structure whose allocation site is `0x0804ddfb`. Transitively, our method reconstructs the entire hierarchy.

The extraction of IP addresses is translated into a traversal over the view to identify those with the IP address semantic tags. Along the path `08050260` → `08052170` → `7e9200...0` → `0x0b0000a`, a variable with the `sin_addr` type can be identified,




```
180 typedef struct {
181      th   d_t h  dl ;
182        i   d l    i t id;
183      h  t i t    k t;
184      t   t    k dd  i  Cli  tAdd ;
185     ti  _t  ti  ; // C   ti   ti
186     ti  _t  ti  ; // L  t A      ti
187     char *PostData;
188     CONNDATA *d t;
189 } CONNECTION;


206 CONNECTION *    ;  //  t h d th     t   d


143 typedef struct {
144     // incoming data
145      h   i  C     ti  [16];
146     int  in_ContentLength;
147     char in_ContentType[128];
148     char in_Cookie[1024];

149      h   i  H  t[64];
150     char in_IfModifiedSince[64];
151     char in_PathInfo[128];

152      h   i  P  t   l[16];
153     char in_QueryString[1024];
154     char in_Referer[128];

155      h   i  R   t Add [16];
156     int  in_RemotePort;

157      h   i  R     tM th d[8];

158      h   i  R     tURI[1024];
159     char in_ScriptName[128];

160      h   i  U   A   t[128];

161     // outgoing data
162      h  t i t   t  t t  ;
163     char out_CacheControl[16];

164      h     t C     ti  [16];

165     i t    t C  t  tL   th;
166     char out_Date[64];
167     char out_Expires[64];
168     char out_LastModified[64];
169     char out_Pragma[16];


170      h     t P  t   l[16];
171     char out_Server[128];


172      h     t C  t  tT   [128];
173     char out_ReplyData[MAX_REPLYSIZE];

174      h  t i t   t h  dd   ;
175      h  t i t   t b d d   ;
176      h  t i t   t fl  h d;

177     // user data
178     char envbuf[8192];
179 } CONNDATA;
```

(b) Data structure definition (a) Hierarchical view from REWARDS

Fig. 6.2.: Comparison between the REWARDS-derived hierarchical view and source code definition

which stores the client IP. The same IP can be identified along the path `08050260` → `08052170` → `08052a58` → `10.0.0.11` as well, with the field offset `+2596`. The field has the `ip_addr_str_t` tag, which is resolved at the return of a call to `inet_ntoa`. REWARDS can isolate the server IP `10.0.0.4` as a string along the path `08050260` → `08051170` → `10.0.0.4` with the field offset `+1172`. Interestingly, this field does not have a semantic tag related to an IP address. The reason is that the field is simply a part of the request string (the host field in HTTP Request Message), but it is not used in any type sinks that can resolve it as an IP. However, isolating the string also allows a human inspector to extract it as an IP.

To validate our result, we present in Figure 6.2(b) the corresponding symbolic definitions extracted from the source for comparison. Fields that are underlined are used during execution. In particular, struct `CONNECTION` corresponds to the abstraction `struct_0x0804dd4f` (node `08052170`) and struct `CONNDATA` corresponds to `struct_0x0804ddfb` (node `08052a58`). Observe that all fields of `CONNECTION` are precisely derived, except the pointer `PostData`, which is represented as an unused array in the inferred definition because the field is not used during execution. For the `CONNDATA` structure, all the exercised fields are extracted and correctly typed. Recall that we consider a field is correctly typed if its offset is correctly identified and its composition bytes are either correctly typed or unused.

```
bfffd140  05 00 00 00 6b 00 00 00  69 00 00 00 00 00 00 00
bfffd150  00 00 00 00 38 ea ff bf  00 00 00 00 00 00 00 01
bfffd160  2c 00 00 00 67 45 8b 6b  0e 00 00 00 00 00 00 00
bfffd170  0a 00 00 63 0f 27 00 00  9f 86 01 00 9f 86 01 00
bfffd180  1c ea ff bf 10 ea ff bf  6a f2 b2 4a 7a 4a 0e 00
bfffd190  22 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
bfffd1a0  6a f2 b2 4a 7a 4a 0e 00  f2 f3 8d 8c 00 00 00 00
bfffd1b0  00 00 00 00 00 00 00 00  01 00 00 00 02 00 00 00
bfffd1c0  64 6e 73 66 6c 6f 6f 64  00 00 00 00 00 00 00 00
bfffd1d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
bfffd5c0  c0 d1 ff bf 00 00 00 00  02 ca 04 08 00 00 00 00
bfffd5d0  00 00 00 00 00 00 00 00  02 ca 04 08 02 ca 04 08
bfffd5e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
bfffd5f0  00 00 00 00 00 00 00 00  00 00 00 00 04 d6 ff bf
bfffd600  64 6e 73 66 6c 6f 6f 64  00 00 00 00 00 00 00 00
bfffd610  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
bfffe5b0  00 00 00 00 00 00 00 00  0e 00 00 00 00 00 00 00
bfffe5c0  00 00 00 00 02 00 4e 34  0a 00 00 0b 00 00 00 00

bfffe5d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
bfffe5e0  00 00 00 00 00 00 00 00  00 00 00 00 e0 f5 ff bf
bfffe5f0  a0 2d 05 08 e0 f5 ff bf  a0 13 05 08 00 00 00 00
bfffe600  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
bfffea00  00 00 00 00 00 00 00 00  00 00 00 00 10 ea ff bf
bfffea10  01 00 00 00 00 00 00 00  e5 de f2 49 46 00 00 00
bfffea20  67 45 8b 6b 10 00 00 00  e8 be e6 71 0a 00 00 34
bfffea30  0a 00 01 33 0a 00 00 0b  0a 00 00 04 00 00 00 00
bfffea40  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
...
bffff5c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
bffff5d0  01 00 00 00 80 00 00 00  80 00 00 00 ff f7 ff bf
bffff5e0  00 00 00 00 00 00 00 00  f3 f7 ff bf 67 45 8b 6b
bffff5f0  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
bffff600  01 00 00 00 c0 f6 ff bf  28 f6 ff bf fb c7 04 08
bffff610  02 00 00 00 dc 3a 1f b6  d4 df 04 08 dc 3a 1f b6
bffff620  00 00 00 00 dc 3a 1f b6  88 f6 ff bf 2 d 0d b6
bffff630  02 00 00 00 b4 f6 ff bf  c0 f6 ff bf f6 5b ff b7
```

Fig. 6.3.: Memory dump for Slapper worm control program when exiting the control interface

6.1.2 Typing Dead Memory

In this case, we demonstrate how to type dead memory (i.e., memory regions containing dead variables) using the slapper worm bot-master program. Slapper worm relies on P2P communications. The bot-master uses a program called `pudclient` to control the P2P botnet, such as launching TCP-flood, UDP-flood, and DNS-flood attacks. Our goal is to extract evidence from a memory dump of `pudclient` from the attacker's machine.

Our experiment has two scenes: the investigator's scene and the attacker's scene. More specifically,

- Scene I: In the lab, the investigator runs the bot-master program `pudclient` to communicate with slapper bots to derive the data structures of `pudclient`.
- Scene II: In the wild, the attacker runs `pudclient` to control real slapper bots.

In Scene I, we run a number of slapper worm instances in a contained environment (at IP addresses ranging from `10.0.0.1 - 10.0.1.255`) using vGround [81], a Virtual Playgrounds for Worm Behavior Investigation. Then we launch `pudclient` with REWARDS and issue a series of commands such as listing the compromised hosts, and launching the UDPFlood, TCPFlood, and DNSFlood attacks. REWARDS extracts the data structure definitions for `pudclient`. Then, in Scene II, we run `pudclient` again without REWARDS. Indeed, the attacker's machine does not have any forensics tool running. Emulating the attacker, we issue some commands and then hibernate the machine. We then get the memory image of `pudclient` and use the data structure information derived in Scene I to investigate the image.

We construct the hierarchical view and try to identify IP addresses from the view. However, the hierarchical view can only map the memory locations that are alive, namely they are reachable from global and stack (pointer) variables. Here, we take an extra step to type the dead (unreachable) data. As described in Section 3.2.5, our technique scans the stack space lower than the current (the lowest and live) activation record and looks for values that are in the range of the code section, as they are very likely return addresses.

Four such values are identified. One example and its memory context is shown in Figure 6.3. In this memory dump snippet, the return address, as underlined, is located at address `0xbffff62c`. Our technique further identifies that the corresponding function invocation is to `0x0804a708`. Hence, we use the data structure definition of `fun_0x0804a708` to type the activation record. The definition and the typed values are shown in Table 6.1. Observe that a number of IPs (fields with `ip_addr_t`) are identified. We also spot the bot command "`dnsflood`" at `-9324` and `-8236`. Note that these two fields have the `input_t` tag as part of their derived definition, indicating they hold values from input.

6.2 Vulnerability Fuzzing

It is a challenging task to detect and confirm vulnerabilities in a given binary without symbolic information. Previously in [82], we proposed a dynamic analysis approach that can decide if a vulnerability suspect is true positive by generating a concrete exploit. The basic idea is to first use existing static tools to identify vulnerability candidates, which are often of large quantity; then benign executions are mutated to generate exploits. Mutations are directed by dynamic information called *input lineage*, which denotes the set of input elements that is used to compute a value at a given execution point, usually a vulnerability candidate. Vulnerability-specific patterns are followed during mutation. One example pattern is to exponentially expand an input string in the lineage of a candidate buffer with the goal of generating an overflow exploit. In that project, we had difficulty finding publicly available, *binary-level* vulnerability detectors to use as the front end. REWARDS helps address this issue by deriving both variable syntax and semantics from a subject binary. Next, we present our experience of using REWARDS to identify vulnerability suspects and then using our prior system (a fuzzer) to confirm them.

For this study, we design a static vulnerability suspect detector that relies on the variable type information derived by REWARDS. The result of the detector is passed to our lineage-based fuzzer to generate exploits. In the following, we present how REWARDS helps identify various types of vulnerability suspects.

Table 6.1: Result on the unreachable memory type using type fun_0x804a708

Offset	Type	Size	Mem Addr	Content	Offset	Type	Size	Mem Addr	Content
-9432	void*	4	bfffd154	38 ea ff bf	-9324	char[9],input_t	9	bfffd1c0	64 6e..64
-9428	char*	4	bfffd158	00 00 00 00	-8300	char*	4	bfffd5c0	c0 d1 ff bf
-9420	int	4	bfffd160	2c 00 00 00	-8236	char[9],input_t	9	bfffd600	64 6e..64
-9416	int	4	bfffd164	67 45 8b 6b	-8227	char[28]	28	bfffd609	00 .. 00
-9412	int	4	bfffd168	0e 00 00 00	-4236	void*	4	bfffe5a0	00 00 00 00
-9408	int	4	bfffd16c	00 00 00 00	-4156	struct_0x804834e*	4	bfffe5f0	a0 2d 05 08
-9404	ip_addr_t	4	bfffd170	0a 00 00 63	-4152	void*	4	bfffe5f4	e0 f5 ff bf
-9300	port_t	4	bfffd174	0f 27 00 00	-3104	char*	4	bfffea0c	10 ea ff bf
-9396	int	4	bfffd178	9f 86 01 00	-3088	char[16]	16	bfffea1c	46 00 00 00
-9392	int	4	bfffd17c	9f 86 01 00	-3068	ip_addr_t	4	bfffea30	0a 00 01 33
-9388	void*	4	bfffd180	1c ea ff bf	-3064	ip_addr_t	4	bfffea34	0a 00 00 0b
-9384	void*	4	bfffd184	10 ea ff bf	-3058	ip_addr_t	4	bfffea38	0a 00 00 04
-9376	timeval.tv_sec	4	bfffd18c	7a 4a 0e 00	-3054	ip_addr_t	4	bfffea3c	0a 00 00 04
	timeval.tv_usec	4	bfffd190	22 00 00 00	-0088	int	4	bfffff5d4	80 00 00 00
-9368	int	4	bfffd194	00 00 00 00	-0084	int	4	bfffff5d8	80 00 00 00
-9352	int	4	bfffd1a4	7a 4a 0e 00	-0080	int	4	bfffff5dc	ff f7 ff bf
-9348	int	4	bfffd1a8	f2 f3 8d 8c	-0004	stack_frame_t	4	bfffff628	88 f6 ff bf
-9344	int	4	bfffd1ac	00 00 00 00	+0000	ret_addr_t	4	bfffff62c	a2 de 0d b6
-9332	int	4	bfffd1b8	01 00 00 00	+0004	int	4	bfffff630	02 00 00 00
-9328	int	4	bfffd1bc	02 00 00 00	+0008	char*	4	bfffff634	b4 f6 ff bf

- **Buffer overflow vulnerability.** Buffer overflows could happen in three different places: stack, heap, and global areas. As such, we define three types of buffer overflow vulnerability patterns. Specifically, for stack overflow, if a stack layout contains a buffer and its content comes from user input, we consider it a suspect. Note that this can be easily facilitated by REWARDS's typing algorithm: A semantics tag `input_t` is defined to indicate that a variable receives its value from external input. *The tag is only susceptible to the forward flow but not the backward flow*. In the stack layout derived by REWARDS, if a buffer's type set contains an `input_t` tag, it is considered vulnerable. For heap overflow, we consider two cases: one is to exploit heap management data structure outside the user-allocated heap chunk; and the other is to exploit user-defined function pointers inside the heap chunk. Detecting the former case is simply to check if a heap structure contains a buffer field that is input-relevant, in a way similar to stack vulnerability detection. For the later case, the detector scans the derived layout of a heap structure to check the presence of both an input-relevant buffer field and a function pointer field. Vulnerabilities in the global memory region are handled similarly.

- **Integer overflow vulnerability.** Integer overflow occurs when an integer exceeds the maximum value that a machine can represent. Integer overflow itself may not be harmful (e.g., `gcc` actually leverages integer overflow to manipulate control flow path condition [83]), but if an integer variable is dependent on user input without any sanity check and it is used as an argument to `malloc`-family functions, then an integer overflow vulnerability is likely. In particular, overflowed values passed to malloc functions usually result in heap buffers being smaller than they are supposed to be. Consequently, heap overflows occur. For this type of vulnerabilities, our detector checks the actual arguments to the malloc family function invocations: if an integer parameter has both `malloc_arg_t` and `input_t` tags, an integer overflow vulnerability suspect will be reported.

- **Format string vulnerability.** The format string vulnerability pattern involves a user input flowing into a format string argument. Thus, we introduce a semantics tag `format_string_t`, which is only resolved at invocations to `printf`-family functions. If a variable's type set contains both `input_t` and `format_string_t` tags, a format string vulnerability suspect is reported.

Besides facilitating vulnerability suspect identification, the information generated by REWARDS can also help *composing exploits*. For instance, it is critical to know the distance between a vulnerable stack buffer and a return address (i.e., a variable with the `ret_addr_t` tag), in order to construct a stack overflow exploit. Similarly, it is important to know the distance between a heap buffer and a heap function pointer for composing a heap overflow-based code injection attack. Such information is provided by REWARDS.

Table 6.2: Number of vulnerability suspects reported with help of REWARDS

Program	#Buffer Overflow	#Integer Overflow	#Format String
ncompress-4.2.4	1	0	0
bftpd-1.0.11	3	0	0
gzip-1.2.4	3	0	0
nullhttpd-0.5.0	5	2	0
xzgv-5.8	3	8	0
gnuPG-1.4.3	0	3	0
ipgrab-0.9.9	0	5	0
cfingerd-1.4.3	4	0	1
ngircd-0.8.2	12	0	1

We applied our REWARDS-based detector to examine several programs shown in the 1^{st} column of Table 6.2. The detector reported a number of vulnerable suspects based on the aforementioned vulnerability patterns. The total number of vulnerabilities of each type is presented in the remaining columns. Observe that our detector does not produce many suspects for these programs and therefore can serve as a tractable front end for our fuzzer. The fuzzer then tries to generate exploits to convict the suspects. The details of each confirmed vulnerable data structure are shown in the 2^{nd} column of Table 6.3. The field symbols do not represent their symbolic names, which we do not know, but rather the type tags derived for these fields. For instance, `format_string_t` denotes that the field is essentially a format string; `sockaddr_in` indicates that the field holds a socket

Table 6.3: Result from our vulnerability fuzzer with help of REWARDS

Benchmark	Suspicious Data Structure	Input	Offset	Vulnerability Type
ncompress-4.2.4	fun_0x08048e76 { -1052: **char[13]**, -1039: unused[1023],... -0008: char*, -0004: stack_frame_t, +0000: ret_addr_t, +0004: char**}	argv[1]	{0..11}	Stack overflow
bftpd-1.0.11	fun_0x080494b8 { -0064: char*, -0060: **char[12]**, -0048: unused [44], -0004: stack_frame_t, +0000: ret_addr_t, +0004: char*}	recv	{0..3}	Stack overflow
gzip-1.2.4	bss_0x08053f80 { ... +244128: **char[8]**, +244136: unused[1016], +245152: char*,...}	argv[1]	{0..6}	Global overflow
nullhttpd-0.5.0	heap_0x0804f205 { +0000: char[11], +0011: unused[5], +0016: **int**,... }	recv	{607,608}	Integer overflow
	heap_0x0804c41f { +0000: **void[29]**, +0029: unused[1024]}	recv	{661..690}	Heap Overflow
xzgv-5.8	bss_0x0809ac80 { ... +91952: **int**, +91956: **int**,...}	fread	{4..11}	Integer overflow
gnuPG-1.0.5	fun_0x080673fc { -0176: char[6],unused[2], -0168: **int**,int,...}	fread	{2..5}	Integer overflow
	heap_0x080afec1 { +0000:int,..., +0036: **void[5]** }	fread	{6..10}	Heap overflow
ipgrab-0.9.9	fun_0x0804d06b { -0056: int, -0052: **int**, int,...}	fread	{20..23}	Integer overflow
	heap_0x0805a976 {+0000: **void[60]** }	fread	{40..100}	Heap overflow
cfingerd-1.4.3	fun_0x080496b8 { ..., -0440: struct sockaddr_in, -0424: **format_string_t[34]**, -0390: unused[174], -0216: char[4],,...}	read	{0..3}	Format String
ngircd-0.8.2	fun_0x0805f9a5 { ..., -0284: **format_string_t[76]** -0208: unused[204], -0004: stack_frame_t, +0000: ret_addr_t,...}	recv	{12..15}	Format String

Table 6.4: Experimental result on kernel rootkit detection

Rootkit Name	Target Object	Inside view	crash		SigGraph	
		# of obj.s	# of obj.s	Detected	# of obj.s	Detected
adore-ng-2.6	module	23	23	✗	24	✓
adore-ng-2.6'	task_struct	62	63	✓	63	✓
cleaner-2.6	module	22	22	✗	23	✓
enyelkm 1.0	module	23	23	✗	24	✓
hp-2.6	task_struct	56	57	✓	57	✓
linuxfu-2.6	task_struct	59	60	✓	60	✓
modhide-2.6	module	22	22	✗	23	✓
override	task_struct	58	59	✓	59	✓
rmroots	task_struct	56	N/A	✗	55	✓
rmroots'	module	23	N/A	✗	24	✓

address. The 3^{rd} column presents the input category that is relevant to the vulnerable data structure. For example, the `char[12]` buffer in `bftpd` denotes a packet received from outside (the `recv` category). Note that the input categories are conveniently implemented as semantics tags in REWARDS. The 4^{th} column `offset` represents the input offsets reported by our fuzzer. They represent the places that are mutated to generate the real exploits. The REWARDS-based vulnerability detector also emits vulnerability types (shown in the 5^{th} column) based on the vulnerability patterns matched. Consider the first benchmark `ncompress`: Its entry in the table indicates that the `char[13]` buffer inside a function starting with PC `0x08048e76` is vulnerable to the stack buffer overflow. The buffer receives values from the second command line option (`argv[1]`). Our data lineage fuzzer mutates the lineage of the buffer, which are the first 12 input items (offset 0 to 11) to generate the exploit. From the data structure in the 2^{nd} column, the exploit has to contain a byte string longer than 1,052 bytes to overwrite the return address at the bottom. Other vulnerabilities can be similarly apprehended.

6.3 Kernel Rootkit Detection

By uncovering the kernel objects in a kernel memory image, our second component, SigGraph, provides the semantic view of kernel memory for kernel rootkit detection. We note the convenience of using SigGraph: The user simply runs the data structure-specific scanners on a subject memory image to uncover kernel objects of interest.

Based on the kernel objects revealed by SigGraph, we then follow the existing "view comparison" methodology [30,34,74] for kernel rootkit detection: for a certain type of kernel object (data structure), we compare (1) the number and values of its instances revealed by SigGraph with (2) the relevant information returned by a corresponding system utility (e.g., `lsmod` and `ps` for kernel modules and processes, respectively). If a discrepancy between the two views is observed, we know that a certain kernel object(s) is being hidden, indicating a kernel rootkit attack.

Many kernel rootkits engage in kernel data hiding attacks [30,34,74], among which we experimented with eight representative real-world kernel rootkits that cover the spectrum of data hiding techniques, and the results are presented in the first eight rows in Table 6.4. SigGraph enabled the detection of all of them. Specifically, we use the original samples of `adore-ng-2.6`, `adore-ng-2.6'`, `override`, `enyelkm 1.0` and port those of `hp`, `linuxfu`, `modhide`, `cleaner` from Linux 2.4 to Linux 2.6 on which our experiment is based. All of the above rootkits, except `adore-ng-2.6'` and `override`, hide tasks or kernel modules by manipulating pointers. For example, `adore-ng` changes the connecting pointers of neighboring modules to hide its own module; and `enyelkm` calls a list function (`list_del`) that separates its own module from the module list. As a result, the number of kernel modules counted by `lsmod` is *one less* than the number of corresponding kernel objects revealed by SigGraph, with the missing one being the rootkit module itself.

We point out that the success of kernel rootkit detection in these experiments is attributed to SigGraph's provision of *multiple alternative* signatures (Section 4.7.3) for the same data structure. With the kernel rootkit's presence, some pointers from/to a kernel object may be corrupted and can no longer be used for signature matching. For example, kernel modules are connected by `list.next` and `list.prev` pointers, which are manipulated by rootkits. Fortunately, SigGraph is able to generate alternative signatures that do not involve those pointers. With such signatures, SigGraph scanners accurately recognize the kernel objects that are being hidden.

Finally, rootkits `adore-ng-2.6'` and `override` have different attack mechanisms. They hide processes by filtering out information about the hidden processes using injected

code – without manipulating kernel objects. SigGraph recognizes these objects using the default signature of `task_struct` without resorting to the alternative ones, which leads to the detection of such attacks via view comparison.

Comparison with techniques based on global memory mapping. A number of existing kernel rootkit detection techniques rely on building a graph that maps the *entire* live memory through pointers. The state of the art is KOP [30]. Based on Windows, it builds a global memory graph and resolves function pointers through an advanced points-to analysis. Due to the lack of its Linux implementation, we implement a basic system based on global memory mapping by extending the `crash` utility. As a core dump analysis infrastructure that resolves memory regions based on type information, `crash` is extendable for customized memory analysis. In particular, our extension involves a Python script to build a global memory graph by exploring the points-to relations. We consider a rootkit detected if the hidden kernel object (module or task) is reachable in the graph. Table 6.4 presents the results. The extended `crash` detects four out of the eight real-world rootkits. It is not a surprise that `crash` detects `adore-ng-2.6`' and `override` as they do not manipulate kernel object pointers. For `hp-2.6` and `linuxfu-2.6`, even though the rootkit tasks are hidden from the task list, they are still reachable via other data structures in the memory graph (more specifically via data structures for process scheduling). However, such alternative reachability is not available when running `adore-ng-2.6`, `cleaner-2.6`, `enyelkm 1.0`, and `modhide-2.6` and hence `crash` misses them.

We note that global memory graph-based techniques rely on each object's reachability from the root(s) of the graph. In other words, an object cannot be properly typed if it is not reachable from the root(s). As a result, it is conceivable that future rootkits may try to destroy such reachability. For example, a rootkit may identify a *cut* of the global memory graph and destroy (or obfuscate) the pointers along the cut. Consequently, objects not reachable from the original roots become unrecognizable. As an extreme example, we construct two such rootkits: `rmroots` and `rmroots`' (the last two rows in Table 6.4). They hide `task_struct` and `module` instances, respectively, and to destroy evidence at

the end of the attack, they "wipe out" the static kernel data structures listed in the kernel symbol table (`system.map`) so that the rest of the memory becomes unmappable.

In comparison, SigGraph shows better robustness against such an attack. In our experiment with the `rmroots` rootkit, there are 56 running processes right before the static kernel object wipe-out. Soon after the wipe-out, the system crashes due to pointer corruption and a kernel memory snapshot is taken. We run the extended `crash` on the kernel memory image, but it fails to construct the global memory graph due to the absence of static kernel objects. On the other hand, SigGraph is able to identify 55 instances of `task_struct`, *including* the one that was hidden before the wipe-out. The missing one is actually `init_task`, an instance of `task_struct` that has been cleared. For our experiment with `rmroots`', the SigGraph scanner successfully identifies all 24 kernel modules including the hidden one.

6.4 Kernel Version Inference

Another application of our framework is the determination of an OS kernel version based on a kernel memory snapshot. Consider the following scenario: a public cloud computing platform hosts virtual machines (VMs) with various OS kernels. In order to perform virtual machine introspection [26, 28, 34] on these guest VMs (e.g., for intrusion/malware detection and usage auditing), a prerequisite is to know the specific version of a guest's OS kernel [35–37]. The kernel type/version is critical to accurately interpreting the VM's system state and events by the VMM. However, such information is not always available to the cloud provider (e.g., the cloud provider only knows that a VM runs Linux but does not know which version).

Currently a guest kernel version can be determined via value-invariants (e.g., as adopted in [34]). We instead propose using SigGraph-based data structure signatures as a more accurate kernel version indicator. To validate our proposal, we take nine more Linux kernels ending with an even version number from 2.6.12 to 2.6.34. We select this range because they all work with our `gcc-4.2.4`-based implementation. If a selected version

has multiple sub-versions, we take the latest one. Together with the five Linux kernels already tested (marked with *), we have a total of 14 kernel versions, which are listed in the 1^{st} column of Table 6.5.

Table 6.5: Detailed field offsets of `task_struct` for kernel version inference

Linux kernel version	thread_info	process name	mm_struct		task_struct		list_head					R
			mm	active_mm	real_parent	parent	tasks	ptrace_children	ptrace_list	children	sibling	
2.6.12-6	4	436	108	112	152	156	84	92	100	160	168	✓
2.6.14-7	4	428	120	124	164	168	96	104	112	172	180	✗
2.6.15-1*	4	428	120	124	164	168	96	104	112	172	180	✗
2.6.16-62	4	432	120	124	164	168	96	104	112	172	180	✓
2.6.18-1*	4	428	152	156	196	200	128	136	144	204	212	✓
2.6.20-15*	4	404	128	132	172	176	104	112	120	180	188	✓
2.6.22-19	4	408	132	136	176	180	108	116	124	184	192	✓
2.6.24-26*	4	461	164	168	208	212	140	148	156	216	224	✓
2.6.26-8	4	505	188	192	232	236	164	172	180	240	248	✓
2.6.28-10	4	508	176	180	220	224	168	248	256	228	236	✓
2.6.30-1	4	496	220	224	268	272	192	296	304	276	284	✓
2.6.31-1*	4	500	220	224	268	272	192	296	304	276	284	✓
2.6.32-17	4	504	228	232	268	272	200	296	304	276	284	✓
2.6.34-2	4	512	220	224	276	280	192	304	312	284	292	✓

Version indicator selection. We first compile these kernels using the default configuration to obtain all of their data structure definitions. We then derive SigGraph-based signatures for all of the data structures. After that we try to select one data structure whose signatures in different kernel versions can be used to differentiate the kernel versions. The main requirements for such a data structure D are: (1) it should be commonly present in the execution of all kernels; and (2) its signatures should be distinctive across different kernels. In other words, for each kernel version i, we shall find a signature S_i of D that will recognize instances of D *in and only in* memory images of kernel version i. In the end, we are not able to find a single data structure that can differentiate all the 14 kernels due to the similarity among them. (In fact, we find that two of the kernels share the same data structure definitions.) However, we do find that data structure `task_struct` satisfies the above requirements for most of the kernels. The offsets and types of fields in `task_struct` involved in the signatures are presented from the 2^{nd} to 12^{th} columns in Table 6.5. We can see that there are only two kernels (2.6.14-7 and 2.6.15-1) that cannot be distinguished using `task_struct`'s signatures as shown in Column-R. To validate, we take snapshots

of these kernels and then scan the snapshots using the 13 distinct signatures. We succeed in uniquely identifying 12 of the 14 kernels. The two kernels that we cannot tell apart are two *consecutive* Linux kernels with no significant differences in data structure definitions.

7. LIMITATION AND FUTURE WORK

In this chapter, we discuss the limitations and outline the future work of our framework. We first examine the limitation of our data structure definitions reverse engineering, REWARDS, in Section 7.1, and then data structure instances reverse engineering, SigGraph, in Section 7.2 and finally DIMSUM in Section 7.3.

7.1 REWARDS

As a data structure definition reverse engineering component, REWARDS has a number of limitations:

- REWARDS is a dynamic analysis-based approach. Thus it cannot achieve full coverage of the data structures defined in a program. Instead, the coverage of REWARDS relies on those data structures that are actually created and accessed during a particular run of the binary. How to increase the coverage will be one of our future efforts. Inspired by the recent efforts from static analysis [72], we plan to investigate the combination of dynamic and static analysis in reverse engineering.

- REWARDS is not fully on-line as our timestamp-based on-line algorithm may leave some variables unresolved by the time they are de-allocated, and an off-line companion procedure is therefore needed to make the system sound. A fully on-line type resolution algorithm is our another future work.

- The current implementation of REWARDS is based on PIN, and it hence does not support the reverse engineering of kernel-level data structures. Using other binary instrumentation platforms, such as QEMU, could allow us to reverse engineer the kernel data structures. To port our REWARDS to a virtual machine monitor will be our another future effort.

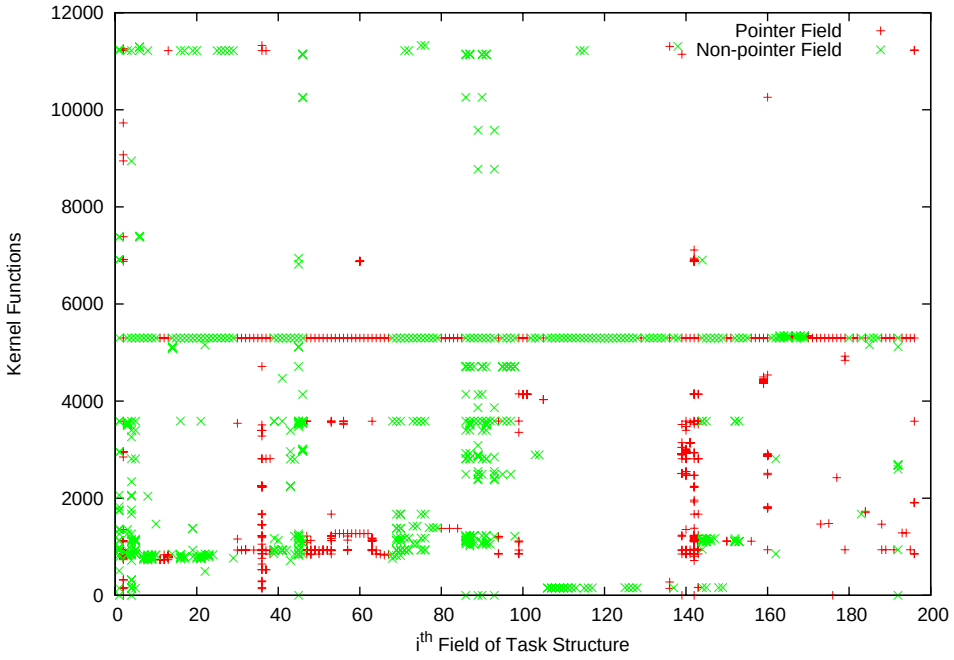
- Besides the general data structures, REWARDS has yet to support the extraction of other data types, such as the format of a specific type of files (e.g., ELF files, multimedia files), and browser-related data types (e.g., URL, cookies). Moreover, REWARDS does not distinguish between signed and unsigned integers in our current design.

7.2 SigGraph

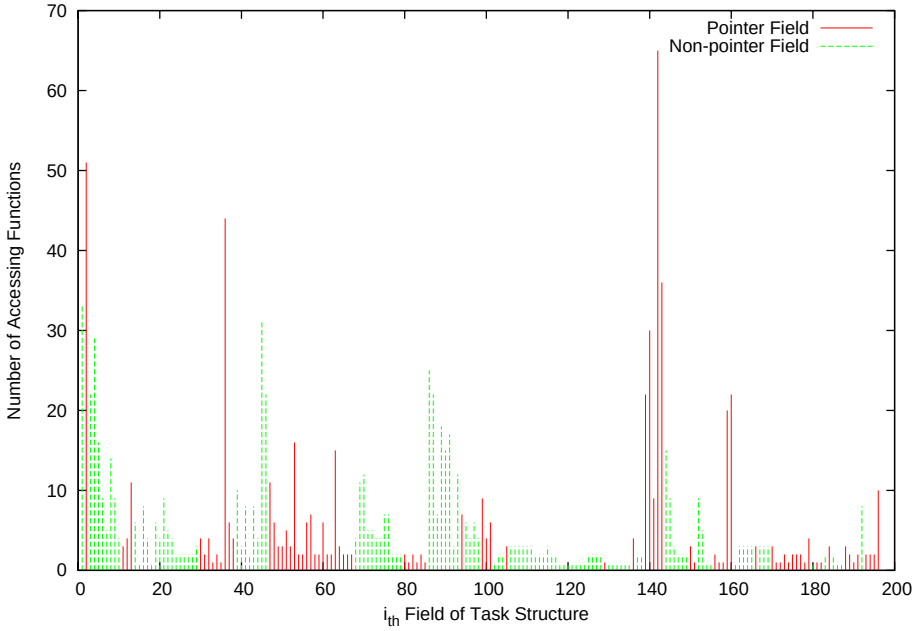
While SigGraph-based signatures are capable of identifying kernel data structure instances, it has limitations as well. We believe that there may be more sophisticated attempts to evade SigGraph in the future. We discuss below the possible attacks against SigGraph, assuming that the attacker has knowledge of SigGraph and has gained control of the kernel.

Malicious Pointer Value Manipulation. The first type of attacks are to manipulate pointers as SigGraph relies on the inter-data structure topology induced by pointers. However, compared to non-pointer values, pointers are more sensitive to mutation as changes to a pointer value may likely lead to kernel crashes. Note that re-pointing a pointer to another data structure instance of the same type may not affect SigGraph in discovering the mutated instance. While the attacker may try to manipulate pointer fields that are not used, recall that SigGraph has a dynamic refinement phase that gets rid of such unused or undependable fields before signature generation.

The attacker may try harder by destroying a pointer field after a reference and then by restoring it before its next reference. As such, it is likely that a memory snapshot may not have the true pointer value. However, carrying out such attacks is challenging as there may be many code sites in the kernel that access the pointer field. All such sites need to be patched to respect the original semantics of the kernel, which would require a complex and expensive static analysis on the kernel. To get an (under) estimate of the required efforts, we conducted a profiling experiment on `task_struct`. We collected the functions that access each field, including both pointers and non-pointers. The results are shown in Figure 7.1(a), We observe that most fields are accessed by at least 6 functions. Some fields



(a) Detailed field access functions



(b) Statistics of field access functions

Fig. 7.1.: Profiling accesses to the fields of `task_struct`

are accessed by 70 functions (the statistics are shown in Figure 7.1(b)). Note that these are only dynamic profiling numbers, the static counterparts may be even higher. Even if the attacker achieves some success, SigGraph can still leverage its multiple signature capability to avoid using those pointers that are easily manipulatable.

Malicious Non-Pointer Value Manipulation. Another possible way to confuse SigGraph is to mutate a non-pointer value to resemble that of a pointer. SigGraph has built-in protection against such attacks. First of all, the dynamic refinement phase will get rid of most fields that are vulnerable to such mutation. Moreover, compared to mutation within a domain, such as changing an integer field (with the range from 1 to 100) from 55 to 56, cross-domain mutation, such as changing the integer field to a pointer, has a much greater chance to crash the system. In the future, we plan to use fuzzing, similar to [32], to study how many fields allow such cross-domain value mutation. Meanwhile, we can effectively *integrate* SigGraph signatures with the value-invariant signatures (e.g., those derived by [32]) for the same data structure, which is likely to achieve even stronger robustness against malicious non-pointer manipulation.

Other Possible Attacks. The attacker may change the data structure layout to evade SigGraph. Without knowledge about the new layout, SigGraph will fail. However, such attacks are challenging. The attacker needs to intercept the corresponding kernel object allocations and de-allocations to change the layout at runtime. Furthermore, all accesses to the affected fields need to be patched.

SigGraph can help detect kernel rootkit attacks by identifying hidden kernel data structure instances in a given memory image. There are other types of kernel attacks that do not involve data hiding (e.g., BluePill [84]). SigGraph, as a kernel object scanner generator, is not applicable to the detection of such kernel attacks.

Also, SigGraph has a number of limitations. We examine each of them below and discuss possible future directions to address them.

First, not all data structures have pointer fields. However, value-invariant signatures will be able to handle them if their fields contain value invariants. As such, we believe a real-world memory analysis system should combine the value-invariant and graph-based

signatures to achieve the maximum coverage of data structures. Also, some data structures have neither value-invariant nor graph-based signature. In such cases, we have to explore other techniques. For example, it is challenging to identify the small-size data structure (e.g., the four bytes IP address) in the memory, but it is still possible to identify some instances of them if they are part of other composite data structures that do have graph-based signatures.

Second, SigGraph has a dynamic refinement component, and we cannot directly use the static signature (largely because of the `null` pointer issue). We cannot achieve completeness because of the nature of dynamic analysis. One solution would be to combine the semantics of the pointer fields and assign different weights statically, and then use them. For example, we could assign a semantic-known pointer field a heavier weight and ignore or assign a smaller weight to other pointer fields.

Third, if an un-initialized pointer exists and we fail to prune, SigGraph will have false negatives. Also, for the possible attacks discussed above, SigGraph may have false positives or false negatives in some cases.

Finally, we did not resolve the `void` pointers. To remedy this, we could take the approach proposed in KOP [30] to resolve the `void` pointers. Also, we could leverage the LiveDM [74] to dynamically track all of the kernel objects, and profile and resolve them.

7.3 DIMSUM

As a data structure instance reverse engineering component, DIMSUM has several limitations too. First, if a program chooses to always zero out its data after they are used (e.g., reset all de-allocated memory), it is unlikely that DIMSUM will recover meaningful information. This is a common limitation for all forensics techniques. However, we consider it to be relatively more tedious to clean up memory than clear up other types of evidence, such as screens and files. The user has to instrument memory management functions and intercept program exit signals. In the presence of memory swapping, the

user has to make sure the pages that get swapped out are destroyed as well. Moreover, if a program crashes or gets killed, cleaning up its memory may not be easy.

Second, DIMSUM currently does not fully make use of value invariant properties, such as a fine-grained range of an integer field of a type T is [x,y]. Instead, it can only leverage the weaker information that it is an integer field. Value invariant properties are usually acquired from profile or domain knowledge. While it is arguable to assume profiling and domain experts in memory forensics, the success of DIMSUM without value invariant properties illustrates its unique strength. Also, DIMSUM is able to deliver better results when value invariants are integrated.

Third, our current implementation in data structure transformation demands end-users to manually write down the specification based on our grammar, in order to automatically generate constraints and then the factor graphs. Part of our future work lies in making this process more automated.

Finally, DIMSUM currently does not have a systematic approach to identifying the data instances that cross pages. In our experiment, we encountered only six such cases for data structure address book in `pine` (partly because of the small size of the data structures of these benchmarks). We leave it as another future effort.

8. RELATED WORK

Our work is related to a large number of techniques, such as type inference, variable recovery, program understanding, vulnerability discovery, malware signature derivation, protocol reverse engineering, rootkit detection, kernel version inference, and memory forensics. In this chapter, we review and compare our technique with each of them.

8.1 Type Inference

Some programming languages, for instance ML, do not explicitly declare types. Instead, types are inferred from programs. Typing constraints are derived from program statements statically, and programs are typed by solving these constraints. There is a large body of type inference techniques, including the Hindley-Milner algorithm [46], the Cartesian Product algorithm [47], iterative type analysis [48], object-oriented type inference [49], aggregate structure identification [85], and dynamic heap type inference [58].

These techniques, like REWARDS [59], rely on type unification, namely, variables connected by operators shall have the same type. However, these techniques assume the program source code and they are static (i.e., typing constraints are generated from source code at compile time). For REWARDS, we only assumed binaries without symbolic information, in which high level language artifacts are all broken down to machine level entities, such as registers, memory addresses, and instructions. REWARDS relies on type sinks to obtain the initial type and semantics information. Variables are then typed through unification with type sinks during execution.

Abstract type inference [50] is to group typed variables according to their semantics. For example, variables that are meant to store money, zip codes, ages, etc., are clustered based on their intentions, even though they may have the same integer type. Such an intention is called an abstract type. The technique relies on the Hindley-Milner type

inference algorithm. Recently, dynamic abstract type inference was proposed [51] to infer abstract types from execution. Regarding the goal of performing semantics-aware typing, these techniques are similar to our technique. However, they work at the source code level, whereas ours works at the binary level. REWARDS [59] further derives syntactic type structures.

Dynamic heap type inference by Polishchuk et al. [58] focuses on typing heap objects in memory. REWARDS, SigGraph, and [58] do share some common insights, such as leveraging pointers. However, the latter focuses on type-inference of heap objects (for debugging) by assuming known start addresses and sizes for all of the allocated heap blocks; whereas REWARDS does not have such constraint, and also SigGraph has a different purpose that aims at uncovering all kernel objects (including heap, stack, and global) from a raw memory image. To uncover those objects, the user can simply execute the data structure-specific scanners on the raw memory image – without any runtime support; the techniques in [58], on the other hand, require collecting runtime information. Moreover, the different purpose of SigGraph raises the new challenge of avoiding structural isomorphism among the data structure signatures.

8.2 Variable Recovery

Variable Discovery Recently, Balakrishnan et al. [39, 86, 87] showed that analyzing executables alone can largely discover the syntactic structures of variables, such as sizes, field offsets, and simple structures. Their algorithm is based on the intuition that the memory accessing patterns in a program provide information about the location of data. They show that variable-like entities can be recovered by iterating Value-Set Analysis [39], a combined numeric-analysis and pointer-analysis algorithm, and Aggregate Structure Identification [85], an algorithm to identify the structure of aggregates. They have a tool called CodeSurfer/x86 [39, 86, 88, 89], a binary analysis platform, which makes use of both IDA Pro [90] and the CodeSurfer system [91] for building program-analysis and inspection tools.

Their technique entails points-to analysis and abstract interpretation at the binary level. This tool cannot handle obfuscated binaries and dynamically loaded libraries, and furthermore, the inaccuracy of the binary points-to analysis makes it hard to type heap variables. In comparison, our technique is relatively simple, and addresses the major hindrances to static analysis (e.g., points-to relations and dynamically loaded libraries) via dynamic analysis.

Decompilation Decompilation is a process of reconstructing program source code from lower-level languages (e.g., assembly or machine code) [92–94]. Tools like HexRay [95], offer a variety of techniques to help elevate low-level assembly instructions to higher level code. Decompililation usually involves reconstructing the variable types [96, 97]. By using unification, Mycroft [96] extends the Hindley-Milner algorithm [46] and delays unification until all constraints are available. Recently, Dolgova and Chernov [97] present an iterative algorithm that uses a lattice over the properties of data types for reconstruction.

All of these techniques are static and hence share the same limitations of static type inference and only derive simple syntactic structures. Moreover, they aim to find an execution-equivalent code and do not pay attention to whether the recovered types reflect the original declarations and have the same structures.

Principled Reverse Engineering Inspired by our REWARDS, most recently there have been two follow up works: HOWARD [73] and TIE [72]. HOWARD differs from REWARDS by looking at the internal data structures inside the binary. Also, it offers a loop detector, which is used to detect array accesses. TIE differs from REWARDS by statically analyzing the binary instead of dynamic analysis. As discussed, there are a number of challenges in statically analyzing the binary code. TIE addressed these challenges and proposed a novel type reconstruction algorithm for binary code based on BAP [98], another binary analysis platform. Unlike REWARDS and HOWARD, which are limited to the dynamic analysis of a single execution trace, TIE handles control flow and thus is amenable to providing complete coverage of the data structures.

8.3 Program Understanding

Our reverse engineering work is also related to program understanding, which aims to help programmers maintain and understand the legacy code [99–101]. There are also a variety of methods for profiling [102–104], testing [105, 106], slicing [107–109], and debugging [110, 111] of program behavior for a given binary or source code, from which they can build the cognitive models for program understanding. The basic cognitive models include (1) top-down understanding [112], (2) bottom-up understanding [113], (3) iterative hypotheses refinement [99], and (4) some combinations of the three [114]. Our technique can facilitate program understanding particularly for the bottom-up approaches, as demonstrated in Lackwit [50], a type inference based program understanding (note Lackwit requires program source code access).

8.4 Malware Signature Derivation

Data structures are one of the important and intrinsic properties of a program. Recent advances have demonstrated that data structure patterns can be used as a program's signature. In particular, Laika [33] shows a way of inferring the layout of data structure from a snapshot, and uses the layout as the signature. Their inference is based on unsupervised Bayesian learning and they assume no prior knowledge about program data structures. There are significant differences between Laika and REWARDS as Laika does not provide any semantic types of the data structure. Laika also does not aim to recover the correct definitions.

Meanwhile, Laika and SigGraph are substantially different in two ways: (1) Laika focuses on deriving a program's signature from data structure patterns; whereas SigGraph focuses on deriving the signatures of the data structures from the points-to relations among them. (2) Laika, by its nature, does not assume the availability of data structure definitions. On the contrary, data structure definitions are the input of SigGraph to generate data structure signatures.

For DIMSUM, the difference compared with Laika [33] is that Laika aims to derive the data structure definition inside a binary and uses it as the program signature. It starts with no knowledge of the data structure definition, and uses data instances to eventually cluster the data structure definitions. DIMSUM solves a completely different problem, namely, starts with data structure definitions and tries to find data instances in the memory. The modeling techniques are completely different. Furthermore, Laika relies on memory mapping when traversing the memory, and DIMSUM does not.

8.5 Protocol Reverse Engineering

Recent efforts in protocol reverse engineering involve using dynamic binary analysis, input data taint analysis in particular, to reveal the format of protocol messages, facilitated by instruction semantics (e.g., Polyglot [19]) or execution context (e.g., AutoFormat [20] and [115]). Recently, it has been shown that the BNF structure of a given protocol with multiple messages can be derived [21, 22, 116]; and the format of outgoing messages, as well as encrypted messages, can be revealed [117, 118]. In particular, REWARDS shares the same insight as Dispatcher [117] for type inference and semantics extraction.

These techniques share the same methodology with our system (i.e. making use of run-time information). However, most existing protocol format reverse engineering techniques focus on using program structure to reflect input syntactic structure. Comparing to these techniques, we share the same observation that a binary implementation contains a wealth of information on discovering the syntax and semantics of program data.

The difference among them is that Dispatcher and other protocol reverse engineering techniques mainly focus on live input and output messages, whereas our technique strives to reveal the general data structures in a program. We also care more about the detailed in-memory layout of the program data, which is motivated by our different targeted application scenarios.

8.6 Vulnerability Discovery

There is a large body of research in vulnerability discovery thorough fuzzing [7, 8], automated test case generation, model checking, or taint analysis such as BuzzFuzz [10], SNOOZE [11], SmartFuzz [9], Flayer [119], Archer [120], EXE [121], Vigilante [122], Bouncer [123], TaintCheck [124], BitScope [125, 126], IntScope [83], TaintScope [127], RICH [128], DART [129], CUTE [130], KLEE [131], and SAGE [132, 133]. REWARDS complements these techniques by enabling the identification of data structure pattern of vulnerability suspects directly from binaries.

Chevarista [134] is another project for automated vulnerability analysis on SPARC binary code. Chevarista demonstrates how to translate binary code to SSA form and model variable bounds by interval analysis to detect buffer overflow or integer overflow.

8.7 Kernel Rootkit Detection

Kernel-level rootkits pose a significant threat to the integrity of operating systems. Earlier research uses a specification-based approach deployed in hardware (e.g., [56]), virtual machine introspection (e.g., Livewire [26]), or binary analysis [135] to detect kernel integrity violations. Recent advances include the mapping and analysis of kernel memory images for control flow integrity checking [136] and kernel data integrity checking [30, 57]. To facilitate kernel data integrity checking, techniques have been proposed for deriving kernel data structure invariants [32, 43].

SigGraph is inspired by, and hence closely related to, the above efforts [30, 32, 43, 57]. In particular, Petroni et al. [57] proposed examining semantic invariants (such as “a process must be on either the wait queue or the run queue”) of kernel data structures to detect kernel rootkits. The key observation is that any violations of semantic invariants indicate kernel rootkit presence. But the semantic invariants are manually specified. Baliga et al. [43] proposed using the dynamic invariant detector Daikon [137] to extract kernel data structure constraints. The invariants detected include membership, non-zero, bounds, length, and subset relations. Dolan-Gavitt et al. [32] proposed a scheme for generating robust value

invariant-based kernel data structure signatures. Complementing these efforts, SigGraph leverages the points-to relations between kernel data structures for signature generation. As suggested in Section 7.2, SigGraph-based and value invariant-based signatures can be *integrated* to further improve brute force scanning accuracy.

Carbone et al. proposed KOP [30] which involves building a global points-to graph for kernel memory mapping and kernel integrity checking. The global graph is constructed via an advanced inter-procedural points-to analysis on OS source code. A few heuristics were proposed to better resolve function pointers. KOP is a highly effective system when the kernel source code and a powerful static analysis infrastructure are available. The main differences between SigGraph and KOP are the following: (1) Unlike KOP, SigGraph does not require complex points-to analysis (which often involves source code analysis) and instead only requires kernel data structure definitions. (2) KOP requires that data structure instances be reachable starting from the root(s) of the global points-to graph; whereas SigGraph does not require such global reachability and hence supports brute force memory scanning that can start at any kernel memory address. In particular, SigGraph may recognize kernel objects that are unreachable from global/stack variables. (3) To achieve robustness against pointer corruption, the global points-to graph heavily depends on a complete revelation of points-to relations between data structures; whereas SigGraph can generate multiple signatures for each data structure by *excluding* problematic pointers (e.g., `null` and `void*` pointers).

8.8 Kernel Version Inference

The goal of OS kernel version inference is to determine the specific OS of the machine on which it is running, which is quite similar to OS fingerprinting mainly launched by attackers to discover possible security vulnerabilities. The basic technique in OS fingerprinting is searching for OS-specific differences in the implementation of the TCP stack. The widely used tools include Nmap [35, 138] and Xprobe2 [37].

The difference compared with our work is that we directly take a specific kernel data structure signature to pinpoint an OS kernel version. Previously, it has been impossible for the attacker to probe the data structures of remote OS, but in the cloud computing scenario, it is quite possible for cloud providers to examine the guest OS memory. Though value-invariants have been used to fingerprint OS kernels (e.g., in [34]), our SigGraph-based fingerprinting technique has all the benefits of SigGraph over the value-invariant based technique, as demonstrated in Chapter 6.

8.9 Memory Forensics

Memory forensics is a particular type of digital forensics [139], which focuses on analyzing a memory image to interpret the state of the system. It has been evolving from basic techniques, such as string matching, to more complex methods, such as object traversal (e.g., [25, 30, 52, 53, 55]) and signature-based scanning (e.g., [32, 40–42, 140]).

Memory traversal approaches (e.g., KOP [30]) attempt to build a road-map of all data structures, starting from the global key objects and traversing along the points-to edges. However, such an approach has to resolve generic pointers such as `void*` and also cannot traverse further if a pointer is corrupted. SigGraph [31] complements those approaches by deriving context-free pointer-based signatures. Yet these techniques mostly work for live data because "dead" data cannot be reached by traversal due to missing page tables and unresolvable pointers. Signature scanning directly searches memory using signatures. A classic approach is to search specific strings in memory. Other notable techniques include PTfinder [41] for linear search of Windows memory to discover process and thread structures, Volatility [25] and Memparser [42] with more capabilities of searching other types of objects.

Signature-based scanning involves directly parsing the memory image using signatures. In particular, Schuster [41] presented PTfinder for linearly searching Windows memory images to discover process and thread structures, using manually created signatures. Similar to PTfinder, GREPEXEC [140], Volatility [40], and Memparser [42] are related systems

capable of searching for more types of objects. Dolan-Gavitt et al. [32] further proposed an automated technique to derive robust data structure signatures. Sharing the same goal of providing robust signatures for brute force memory scanning, SigGraph provides graph-based, provably non-isomorphic signatures (as well as the corresponding memory scanners) for individual kernel data structures.

Table 8.1: Capability comparison with existing techniques

Scenario	Value Invariant	KOP	SigGraph	DIMSUM
Live Object	✓	✓	✓	✓
Dead Object	✓			✓
w/o Mem Mapping	✓			✓
Brute-Force Scanning	✓		✓	✓

The difference between these techniques including SigGraph and DIMSUM, is summarized in Table 8.1: for live objects such as objects in OS kernel, which usually have memory mapping information (or the mapping information is easily recoverable), we could use all these techniques including DIMSUM. However, without memory mapping information, which is mostly the case for the dead object in free pages as well as the swapped page files, we could use value-invariant and DIMSUM. Except KOP which does not support brute force scanning, all these techniques can scan memory at arbitrary memory locations.

9. CONCLUSION

Data structure is one of the key aspects of a program. In this dissertation, we show that by dynamically analyzing the binary code, we can reverse engineer the syntax and semantics of data structures, and we hence develop a tool called REWARDS in Chapter 3 for this purpose. By exploiting the points to relations between data structure, we next propose SigGraph in Chapter 4, which can derive unique signatures for data structures and use them to scan memory and identify data structure instances. Finally, for those data instances that do not have memory mapping information, we develop DIMSUM in Chapter 5, which leverages Bayesian inference techniques to identify them.

REWARDS makes a first step in recovering both the syntax and semantic information of data structures. Given a binary executable, REWARDS executes the binary, monitors the execution, aggregates and analyze runtime information, and ultimately recovers the data structures observed in the execution. Besides leveraging the forward type propagation technique, for reverse engineering of program data structures, REWARDS involves both an on-line and off-line backward type resolution. REWARDS correctly handles the issues caused by memory re-use (e.g., a same stack address may be shared by multiple variables) by using timestamps. We have developed a prototype of REWARDS and used it to analyze a number of binaries. Our evaluation results show that REWARDS is able to reveal the types of the variables observed in a program's execution with over 80% accuracy. Furthermore, we demonstrated the versatility of REWARDS in a variety of application scenarios, such as memory image forensic analysis and binary fuzzing for vulnerability discovery.

After we have derived the data structure definitions from the application binary, the next step is how to use these data structures. We observed that the points-to relations between the data structures could be exploited by deriving the data structure signatures. We thus developed SigGraph, a system to extract the points-to graph and automatically generate the signatures for data structures. We have extensively evaluated SigGraph-based signatures

with several Linux kernels and verified the uniqueness of the signatures. Our signatures achieve close to zero false positives and zero false negatives when applied to data structure instance recognition in kernel memory images. Furthermore, our experiments showed that SigGraph works without global memory maps and in the face of a range of kernel attacks that manipulate pointer fields, demonstrating its applicability to kernel rootkit detection. Finally, we showed that SigGraph can be used as well to determine the version of a guest OS kernel, a key prerequisite of virtual machine introspection.

SigGraph can discover the data structure instances that are reachable, namely, the pointer addresses can be resolved and mapped. However, for data instances in unmappable memory, SigGraph does not work. Such un-mappable memory could be the entire free pages of the system, the memory swap file, or a corrupted memory dump. To address this problem, we presented a *probabilistic inference*-based approach called DIMSUM to enable the recognition of data structure instances from un-mappable memory. Given a set of memory pages and the specification of a target data structure, DIMSUM will identify instances of the data structure in those pages with quantifiable confidence. Our experiments with real-world applications on the Linux platform show that DIMSUM achieves better effectiveness (with over 20% accuracy improvement) than non-probabilistic approaches without memory mapping information.

Together, REWARDS, SigGraph, and DIMSUM present an integrated framework for reverse engineering of data structures, including data structure definitions and data structure instances. Meanwhile, they also suggest many interesting research problems. Below, we briefly conclude this dissertation with a list of a number of open problems in this research direction.

- **Binary code obfuscation** Reverse engineering has to deal with obfuscation [44]. There are many code obfuscation techniques to thwart static and dynamic code analysis, such as dead code insertion, code transposition, register assignment, instruction substitution [141], and code encryption and packing. These obfuscation techniques will have an impact on our REWARDS, especially when we want to apply

REWARDS to analyze malware. However, REWARDS is resilient to static code obfuscation by nature as it is a dynamic analysis system.

- **Data structure obfuscation** It is also possible to obfuscate the data structures, for example, shuffle the data structure field [45], or insert a garbage field. Such problems are a large threat to data structure reverse engineering and many data structure based applications. It is worthwhile to study the problem of data structure obfuscations and deobfuscations.

- **False data injection issue** In forensics scenarios, an attacker could attempt to generate *fake* data structure instances to thwart the use of SigGraph and DIMSUM. Although exploiting the points-to relation makes such attacks more difficult as the attacker would have to fake the *multiple* data structures involved in a graph signature and make sure that all of the points-to relations among these data structures are properly set up, and such an attack is totally possible. At this time there is no general solution to this problem, and it is also worthwhile to investigate how to remove the false injected data in forensics.

LIST OF REFERENCES

LIST OF REFERENCES

[1] D. E. Knuth, *The Art of Computer Programming, Volume 1 (3rd ed.): Fundamental Algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997.

[2] N. Wirth, *Algorithms + Data Structures = Programs*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1978.

[3] E. Chikofsky and I. Cross, J.H., "Reverse engineering and design recovery: A taxonomy," *Software, IEEE*, vol. 7, pp. 13 –17, Jan 1990.

[4] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[5] "Single precision floating-point format." http://en.wikipedia.org/wiki/Single_precision_floating-point_format.

[6] H. Etoh, "GCC extension for protecting applications from stack-smashing attacks (ProPolice)," *http://www.trl.ibm.com/projects/security/ssp/*, 2003.

[7] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," in *Proceedings of the Workshop of Parallel and Distributed Debugging*, pp. 9–19,, Academic Medicine, 1990.

[8] J. E. Forrester and B. P. Miller, "An empirical study of the robustness of Windows NT applications using random testing," in *Proceedings of the 4th Conference on USENIX Windows Systems Symposium*, (Seattle, Washington), pp. 6–6, USENIX Association, 2000.

[9] D. Molnar, X. C. Li, and D. A. Wagner, "Dynamic test generation to find integer bugs in x86 binary Linux programs," in *Proceedings of the 18th Conference on USENIX security Symposium*, SSYM'09, (Montreal, Canada), pp. 67–82, USENIX Association, 2009.

[10] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pp. 474–484, IEEE Computer Society, 2009.

[11] G. Banks, M. Cova, V. Felmetsger, K. C. Almeroth, R. A. Kemmerer, and G. Vigna, "Snooze: Toward a stateful network protocol fuzzer," in *Information Security Conference/Information Security Workshop*, pp. 343–358, 2006.

[12] X. Wang, Z. Li, J. Xu, M. K. Reiter, C. Kil, and J. Y. Choi, "Packet vaccine: Black-box exploit detection and signature generation," in *Proceedings of the 13th ACM Conference on Computer and Communication Security (CCS)*, (Alexandria, Virginia, USA), pp. 37–46, ACM Press, 2006.

[13] W. Cui, M. Peinado, H. J. Wang, and M. Locasto, "Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing," in *Proceedings of 2007 IEEE Symposium on Security and Privacy*, (Oakland, CA), May 2007.

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

[15] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: Automatic exploit generation," in *Proceedings of Network and Distributed System Security Symposium*, Feb. 2011.

[16] S. Heelan, "Msc computer science dissertation: Automatic generation of control flow hijacking exploits for software vulnerabilities," 2009.

[17] W. Cui, J. Kannan, and H. J. Wang, "Discoverer: Automatic protocol reverse engineering from network traces," in *Proceedings of the 16th USENIX Security Symposium (Security'07)*, (Boston, MA), August 2007.

[18] "The Protocol Informatics Project," *http://www.baselineresearch.net/PI/*.

[19] J. Caballero and D. Song, "Polyglot: Automatic extraction of protocol format using dynamic binary analysis," in *Proceedings of the 14th ACM Conference on Computer and and Communications Security (CCS'07)*, (Alexandria, Virginia, USA), pp. 317–329, 2007.

[20] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic protocol format reverse engineering through context-aware monitored execution," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, (San Diego, CA), February 2008.

[21] G. Wondracek, P. Milani, C. Kruegel, and E. Kirda, "Automatic network protocol analysis," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, (San Diego, CA), February 2008.

[22] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz, "Tupni: Automatic reverse engineering of input formats," in *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*, (Alexandria, Virginia, USA), pp. 391–402, October 2008.

[23] G. Palmer, "A road map for digital forensic research.," *Report from DFRWS 2001, First Digital Forensic Research Workshop*, pp. 27–30, August 2001.

[24] S. Peisert, M. Bishop, S. Karin, and K. Marzullo, "Toward models for forensic analysis," in *Proceedings of the Second International Workshop on Systematic Approaches to Digital Forensic Engineering*, (Washington, DC, USA), pp. 3–15, IEEE Computer Society, 2007.

[25] N. L. Petroni, Jr., A. Walters, T. Fraser, and W. A. Arbaugh, "Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory," *Digital Investigation*, vol. 3, no. 4, pp. 197 – 210, 2006.

[26] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proceedings Network and Distributed Systems Security Symposium (NDSS'03)*, February 2003.

[27] P. M. Chen and B. D. Noble, "When virtual is better than real," in *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, pp. 133–, 2001.

[28] B. D. Payne, M. Carbone, and W. Lee, "Secure and flexible monitoring of virtual machines," in *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)*, December 2007.

[29] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*, (Alexandria, Virginia, USA), pp. 128–138, ACM, 2007.

[30] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, "Mapping kernel objects to enable systematic integrity checking," in *The 16th ACM Conference on Computer and Communications Security (CCS'09)*, (Chicago, IL, USA), pp. 555–565, 2009.

[31] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang, "Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, (San Diego, CA), February 2011.

[32] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin, "Robust signatures for kernel data structures," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*, (Chicago, Illinois, USA), pp. 566–577, ACM, 2009.

[33] A. Cozzie, F. Stratton, H. Xue, and S. T. King, "Digging for data structures," in *Proceeding of 8th Symposium on Operating System Design and Implementation (OSDI'08)*, (San Diego, CA), pp. 231–244, December, 2008.

[34] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*, (Alexandria, Virginia, USA), pp. 128–138, ACM, 2007.

[35] L. G. Greenwald and T. J. Thomas, "Toward undetected operating system fingerprinting," in *Proceedings of the first USENIX Workshop on Offensive Technologies*, pp. 1–10, USENIX Association, 2007.

[36] M. Smart, G. R. Malan, and F. Jahanian, "Defeating TCP/IP stack fingerprinting," in *Proceedings of the 9th Conference on USENIX Security Symposium*, pp. 17–17, USENIX Association, 2000.

[37] O. Arkin, F. Yarochkin, and M. Kydyraliev, "The present and future of xprobe2: The next generation of active operating system fingerprinting. sys-security group," July 2003.

[38] Zero, CuTedEvil, and Crick *The art of disassembly, Free online book*.

[39] G. Balakrishnan and T. Reps, "Divine: Discovering variables in executables," in *Proceedings of International Conference on Verification Model Checking and Abstract Interpretation (VMCAI'07)*, (Nice, France), ACM Press, 2007.

[40] A. Walters, "The volatility framework: Volatile memory artifact extraction utility framework." https://www.volatilesystems.com/default/volatility.

[41] A. Schuster, "Searching for processes and threads in Microsoft Windows memory dumps," *Digital Investigation*, vol. 3, no. Supplement-1, pp. 10–16, 2006.

[42] C. Betz, "Memparser." http://sourceforge.net/projects/memparser.

[43] A. Baliga, V. Ganapathy, and L. Iftode, "Automatic inference and enforcement of kernel data structure invariants," in *Proceedings of the 2008 Annual Computer Security Applications Conference (ACSAC'08)*, (Anaheim, California), pp. 77–86, December 2008.

[44] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," *Technical Report 148, Department of Computer Science, University of Auckland*, 1997.

[45] Z. Lin, R. D. Riley, and D. Xu, "Polymorphing software by randomizing data structure layout," in *Proceedings of the 6th SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA'09)*, (Milan, Italy), July 2009.

[46] R. Milner, "A theory of type polymorphism in programming," *Journal of Computer and System Sciences*, vol. 17, pp. 348–375, 1978.

[47] O. Agesen, "The cartesian product algorithm: Simple and precise type inference of parametric polymorphism," in *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, (London, UK), pp. 2–26, Springer-Verlag, 1995.

[48] C. Chambers and D. Ungar, "Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs," in *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pp. 150–164, 1990.

[49] J. Palsberg and M. I. Schwartzbach, "Object-oriented type inference," in *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*, (Phoenix, Arizona, United States), pp. 146–161, ACM, 1991.

[50] R. O'Callahan and D. Jackson, "Lackwit: A program understanding tool based on type inference," in *Proceedings of the 19th International Conference on Software engineering*, (Boston, Massachusetts, United States), pp. 338–348, ACM, 1997.

[51] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst, "Dynamic inference of abstract types," in *Proceedings of the 2006 International Symposium on Software testing and analysis (ISSTA'06)*, (Portland, Maine, USA), pp. 255–265, ACM, 2006.

[52] A. Case, A. Cristina, L. Marziale, G. G. Richard, and V. Roussev, "Face: Automated digital evidence discovery and correlation," *Digital Investigation*, vol. 5, no. Supplement 1, pp. S65 – S75, 2008. The Proceedings of the Eighth Annual DFRWS Conference.

[53] P. Movall, W. Nelson, and S. Wetzstein, "Linux physical memory analysis," in *Proceedings of the FREENIX Track of the USENIX Annual Technical Conference*, (Anaheim, CA), pp. 23–32, USENIX Association, 2005.

[54] I. Sutherland, J. Evans, T. Tryfonas, and A. Blyth, "Acquiring volatile operating system data tools and techniques," *SIGOPS Operating System Review*, vol. 42, no. 3, pp. 65–73, 2008.

[55] J. Rutkowska, "Klister v0.3." https://www.rootkit.com/newsread.php?newsid=51.

[56] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot - A coprocessor-based kernel runtime integrity monitor," in *Proceedings of the 13th USENIX Security Symposium*, (San Diego, CA), pp. 179–194, August 2004.

[57] N. L. Petroni, Jr., T. Fraser, A. Walters, and W. A. Arbaugh, "An architecture for specification-based detection of semantic integrity violations in kernel dynamic data," in *Proceedings of the 15th USENIX Security Symposium*, (Vancouver, B.C., Canada), USENIX Association, August 2006.

[58] M. Polishchuk, B. Liblit, and C. W. Schulze, "Dynamic heap type inference for program understanding and debugging," *SIGPLAN Not.*, vol. 42, no. 1, pp. 39–46, 2007.

[59] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*, (San Diego, CA), February 2010.

[60] "Mission critical linux." http://oss.missioncriticallinux.com/projects/mcore/.

[61] B. Moghaddam, T. Jebara, and A. Pentland, "Bayesian face recognition," *Pattern Recognition*, vol. 33, pp. 1771–1782, November 2000.

[62] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler, "From uncertainty to belief: Inferring the specification within," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*, (Seattle, Washington), pp. 161–176, USENIX Association, 2006.

[63] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee, "Merlin: specification inference for explicit information flow problems," in *Proceedings of ACM SIGPLAN 2009 International Conference on Programming Language Design and Implementation (PLDI'09)*, pp. 75–86, 2009.

[64] N. E. Beckman and A. V. Nori, "Probabilistic, modular and scalable inference of typestate specifications," in *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*, 2011.

[65] T. Minka, J. Winn, J. Guiver, and A. Kannan, "Infer.NET 2.3," 2009. Microsoft Research Cambridge. http://research.microsoft.com/infernet.

[66] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference (2nd ed.)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1988.

[67] J. S. Yedidia, W. T. Freeman, and Y. Weiss, "Understanding belief propagation and its generalizations," *Exploring Artificial Intelligence In The New Millennium*, pp. 239–269, 2003.

[68] L. Dietz, V. Dallmeier, A. Zeller, and T. Scheffer, "Localizing bugs in program executions with graphical models," in *Proceedings of the 2009 Advances in Neural Information Processing Systems*, December 2009.

[69] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, (Chicago, IL, USA), pp. 190–200, 2005.

[70] "Libdwarf," *http://reality.sgiweb.org/davea/dwarf.html*.

[71] "Gnu compiler collection (gcc) internals," *http://gcc.gnu.org/onlinedocs/gccint/*.

[72] J. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, (San Diego, CA), February 2011.

[73] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A dynamic excavator for reverse engineering data structures," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, (San Diego, CA), February 2011.

[74] J. Rhee, R. Riley, D. Xu, and X. Jiang, "Kernel malware analysis with untampered and temporal views of dynamic kernel memory," in *Proceedings of the 13th International Symposium of Recent Advances in Intrusion Detection*, (Ottawa, Canada), September 2010.

[75] "QEMU: an open source processor emulator," *http://www.qemu.org/*.

[76] J. Chow, B. Pfaff, K. Christopher, and M. Rosenblum, "Understanding data lifetime via whole-system simulation," in *Proceedings of the 13th USENIX Security Symposium*, 2004.

[77] J. Solomon, E. Huebner, D. Bem, and M. Szezynska, "User data persistence in physical memory," *Digital Investigation*, vol. 4, no. 2, pp. 68 – 72, 2007.

[78] D. Bovet and M. Cesati, *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[79] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: Cold-boot attacks on encryption keys," in *Proceedings of the 17th USENIX Security Symposium*, (San Jose, CA), August 2008.

[80] B. Fabrice, "Qemu, a fast and portable dynamic translator," in *Proceedings of the 2005 USENIX Annual Technical Conference*, (Berkeley, CA, USA), USENIX Association, 2005.

[81] X. Jiang, D. Xu, H. J. Wang, and E. H. Spafford, "Virtual Playgrounds for Worm Behavior Investigation," *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection*, Sept. 2005.

[82] Z. Lin, X. Zhang, and D. Xu, "Convicting exploitable software vulnerabilities: An efficient input provenance based approach," in *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'08)*, (Anchorage, Alaska, USA), June 2008.

[83] T. Wang, T. Wei, Z. Lin, and W. Zou, "Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution," in *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS'09)*, (San Diego, CA), February 2009.

[84] B. Laurie and A. Singer, "Choose the red pill and the blue pill: A position paper," in *Proceedings of the 2008 Workshop on New Security Paradigms*, pp. 127–133, 2008.

[85] G. Ramalingam, J. Field, and F. Tip, "Aggregate structure identification and its application to program analysis," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL'99)*, (San Antonio, Texas), pp. 119–132, ACM, 1999.

[86] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *Proceedings of International Conference on Compiler Construction (CC'04)*, pp. 5–23, Springer-Verlag, 2004.

[87] T. W. Reps and G. Balakrishnan, "Improved memory-access analysis for x86 executables," in *Proceedings of International Conference on Compiler Construction (CC'08)*, pp. 16–35, 2008.

[88] T. Reps, G. Balakrishnan, J. Lim, and T. Teitelbaum, "A next-generation platform for analyzing executables," in *The Third Asian Symposium on Programming Languages and Systems*, (Tsukuba, Japan), 2005.

[89] T. Reps, G. Balakrishnan, and J. Lim, "Intermediate-representation recovery from low-level code," in *Proceedings of the 2006 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, (Charleston, South Carolina, USA), pp. 100–111, 2006.

[90] "The IDA Pro disassembler and debugger." http://www.hex-rays.com/idapro/.

[91] "CodeSurfer: A code browser that understands pointers, indirect function calls, and whole-program effects," *http://www.grammatech.com/products/codesurfer/*.

[92] C. Cifuentes, "Reverse Compilation Techniques," *PhD thesis, Queensland University of Technology*, 1994.

[93] M. V. Emmerik and T. Waddington, "Using a decompiler for real-world source recovery," in *Proceedings of the 11th Working Conference on Reverse Engineering*, pp. 27–36, 2004.

[94] P. T. Breuer and J. P. Bowen, "Decompilation: The enumeration of types and grammars," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1613–1647, 1994.

[95] "Hex-rays decompiler SDK." http://www.hex-rays.com/.

[96] A. Mycroft, "Type-based decompilation (or program reconstruction via type reconstruction)," in *Proceedings of the 8th European Symposium on Programming Languages and Systems (ESOP'99)*, (London, UK), pp. 208–223, Springer-Verlag, 1999.

[97] E. N. Dolgova and A. V. Chernov, "Automatic reconstruction of data types in the decompilation problem," *Program. Comput. Softw.*, vol. 35, no. 2, pp. 105–119, 2009.

[98] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *Proceedings of Computer Aided Verification (CAV)*, July 2011.

[99] R. Brooks, "Towards a theory of the comprehension of computer programs," *International Journal of Man-Machine Studies*, vol. 18, no. 6, pp. 543 – 554, 1983.

[100] T. A. Corbi, "Program understanding: Challenge for the 1990s," *IBM Systems Journal*, vol. 28, no. 2, pp. 294 –306, 1989.

[101] S. Tilley, S. Paul, and D. Smith, "Towards a framework for program understanding," in *Proceedings of the Fourth Workshop on Program Comprehension*, pp. 19–28, Mar 1996.

[102] T. Reps, T. Ball, M. Das, and J. Larus, "The use of program profiling for software maintenance with applications to the year 2000 problem," *SIGSOFT Software Engineering Notes*, vol. 22, no. 6, pp. 432–449, 1997.

[103] T. Ball and J. R. Larus, "Efficient path profiling," in *Proceedings of the 29th annual ACM/IEEE International Symposium on Microarchitecture (MICRO-29)*, (Paris, France), pp. 46–57, 1996.

[104] B. Calder, P. Feller, and A. Eustace, "Value profiling," in *Proceedings of the 30th annual ACM/IEEE International Symposium on Microarchitecture (MICRO-30)*, (Research Triangle Park, North Carolina, United States), pp. 259–269, 1997.

[105] J. Misurda, J. Clause, J. Reed, B. Childers, and M. Soffa, "Jazz: A tool for demand-driven structural testing," in *Proceedings of the 14th International Conference on Compiler Construction(CC'05)*, (Edinburgh, Scotland), pp. 242–245, 2005.

[106] G. Misherghi and Z. Su, "HDD: Hierarchical delta debugging," in *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, (Shanghai, China), pp. 142–151, 2006.

[107] F. Tip, "A survey of program slicing techniques," *Journal of Programming Languages*, vol. 3, pp. 121–189, 1995.

[108] M. Weiser, *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, 1979. University of Michigan.

[109] H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Dynamic slicing in the presence of pointers, arrays, and records," in *Proceedings of the4th ACM Symposium on Testing, Analysis, and Verification*, pp. 60–73, 1991.

[110] H. Agrawal, R. A. Demillo, and E. H. Spafford, "Debugging with dynamic slicing and backtracking," *Softw. Pract. Exper.*, vol. 23, pp. 589–616, June 1993.

[111] J. R. Bogdan Korel, "Application of dynamic slicing in program debugging," in *Proceedings of the International Symposium on Automated Analysis-driven Debugging (AADEBUG'97)*, no. 43-58, 1997.

[112] E. Soloway and K. Ehrlich, "Empirical studies of programming knowledge," *Software Engineering, IEEE Transactions on*, vol. SE-10, pp. 595 –609, sept. 1984.

[113] B. Shneiderman and R. Mayer, "Syntactic/semantic interactions in programmer behavior: A model and experimental results," *International Journal of Parallel Programming*, vol. 8, pp. 219–238, 1979. 10.1007/BF00977789.

[114] A. Von Mayrhauser and A. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, pp. 44 –55, aug 1995.

[115] Z. Lin and X. Zhang, "Deriving input syntactic structure from execution," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'08)*, (Atlanta, GA, USA), November 2008.

[116] P. Milani Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol Specification Extraction," in *IEEE Symposium on Security & Privacy*, (Oakland, CA), pp. 110–125, 2009.

[117] J. Caballero, P. Poosankam, C. Kreibich, and D. Song, "Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering," in *Proceedings of the 16th ACM Conference on Computer and and Communications Security (CCS'09)*, (Chicago, Illinois, USA), pp. 621–634, 2009.

[118] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace, "Reformat: Automatic reverse engineering of encrypted messages," in *Proceedings of 14th European Symposium on Research in Computer Security (ESORICS'09)*, (Saint Malo, France), LNCS, September 2009.

[119] W. Drewry and T. Ormandy, "Flayer: Exposing application internals," in *Proceedings of the first USENIX Workshop on Offensive Technologies*, (Boston, MA), pp. 1:1–1:9, USENIX Association, 2007.

[120] Y. Xie, A. Chou, and D. Engler, "Archer: Using symbolic, path-sensitive analysis to detect memory access errors," in *Proceedings of the 9th European Software Engineering Conference held jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-10)*, (Helsinki, Finland), pp. 327–336, 2003.

[121] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," in *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS'06)*, (Alexandria, Virginia, USA), pp. 322–335, ACM, 2006.

[122] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-end containment of internet worms," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, (Brighton, United Kingdom), pp. 133–147, 2005.

[123] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado, "Bouncer: Securing software by blocking bad input," in *Proceedings of the 21st ACM SIGOPS Symposium on Operating systems principles (SOSP'07)*, (Stevenson, Washington, USA), pp. 117–130, ACM, 2007.

[124] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS'05)*, (San Diego, CA), February 2005.

[125] D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin, "Bitscope: Automatically dissecting malicious binaries," 2007. Technical Report CMU-CS-07-133, Carnegie Mellon University.

[126] J. Caballero, P. Poosankam, S. McCamant, D. Babi ć, and D. Song, "Input generation via decomposition and re-stitching: Finding bugs in malware," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CC '10, (Chicago, Illinois, USA), pp. 413–425, ACM, 2010.

[127] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Proceedings of the 31st IEEE Symposium on Security and Privacy (Oakland'10)*, May 2010.

[128] D. Brumley, T. cker Chiueh, R. Johnson, H. Lin, and D. Song, "Efficient and accurate detection of integer-based attacks," in *Proceedings of the Network and Distributed System Security Symposium*, 2007.

[129] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, (Chicago, IL, USA), pp. 213–223, ACM, 2005.

[130] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*, (Lisbon, Portugal), pp. 263–272, ACM, 2005.

[131] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, (San Diego, CA), 2008.

[132] P. Godefroid, M. Levin, and D. Molnar, "Automated whitebox fuzz testing," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, (San Diego, CA), February 2008.

[133] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, (Tucson, AZ, USA), pp. 206–215, ACM, 2008.

[134] "Automated vulnerability auditing in machine code," *Phrack Magzine, Vol(64). http://www.phrack.com/issues.html?issue=64&id=8*.

[135] C. Kruegel, W. Robertson, and G. Vigna, "Detecting kernel-level rootkits through binary analysis," in *Proceedings of the 20th Annual Computer Security Applications Conference(ACSAC'04)*, pp. 91–100, 2004.

[136] N. L. Petroni, Jr. and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*, (Alexandria, Virginia, USA), pp. 103–115, ACM, October 2007.

[137] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Trans. on Software Engineering*, vol. 27, no. 2, pp. 1–25, 2001.

[138] Fyodor, "Remote os detection via TCP/IP fingerprinting (2nd generation). insecure.org," January 2007. http://insecure.org/nmap/osdetect/.

[139] B. D. Carrier and E. H. Spafford, "Automated digital evidence target definition using outlier analysis and existing evidence," in *Proceedings of the 5th Annual Digital Forensic Research Workshop*, 2005.

[140] "Grepexec: Grepping executive objects from pool memory," *Uninformed Journal, Vol(4)*, 2006.

[141] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," in *Proceedings of the 12th USENIX Security Symposium (Security'03)*, pp. 169–186, USENIX Association, USENIX Association, Aug. 2003.

VITA

VITA

Zhiqiang Lin received his BE degree in computer science from Nanjing University of Posts and Telecommunications in 2002, MS degree in computer science from Nanjing University in 2006, and PhD degree in computer science from Purdue University in 2011. His research interests mainly focus on systems and software security, with an emphasis on the development of program analysis and reverse engineering techniques, and their applications to OS kernel integrity enforcement, software vulnerability discovery, malicious code analysis, and computer forensics. In the fall of 2011, he will join the faculty of the University of Texas at Dallas as an Assistant Professor of Computer Science.