National Chiao Tung University

Department of Computer Science

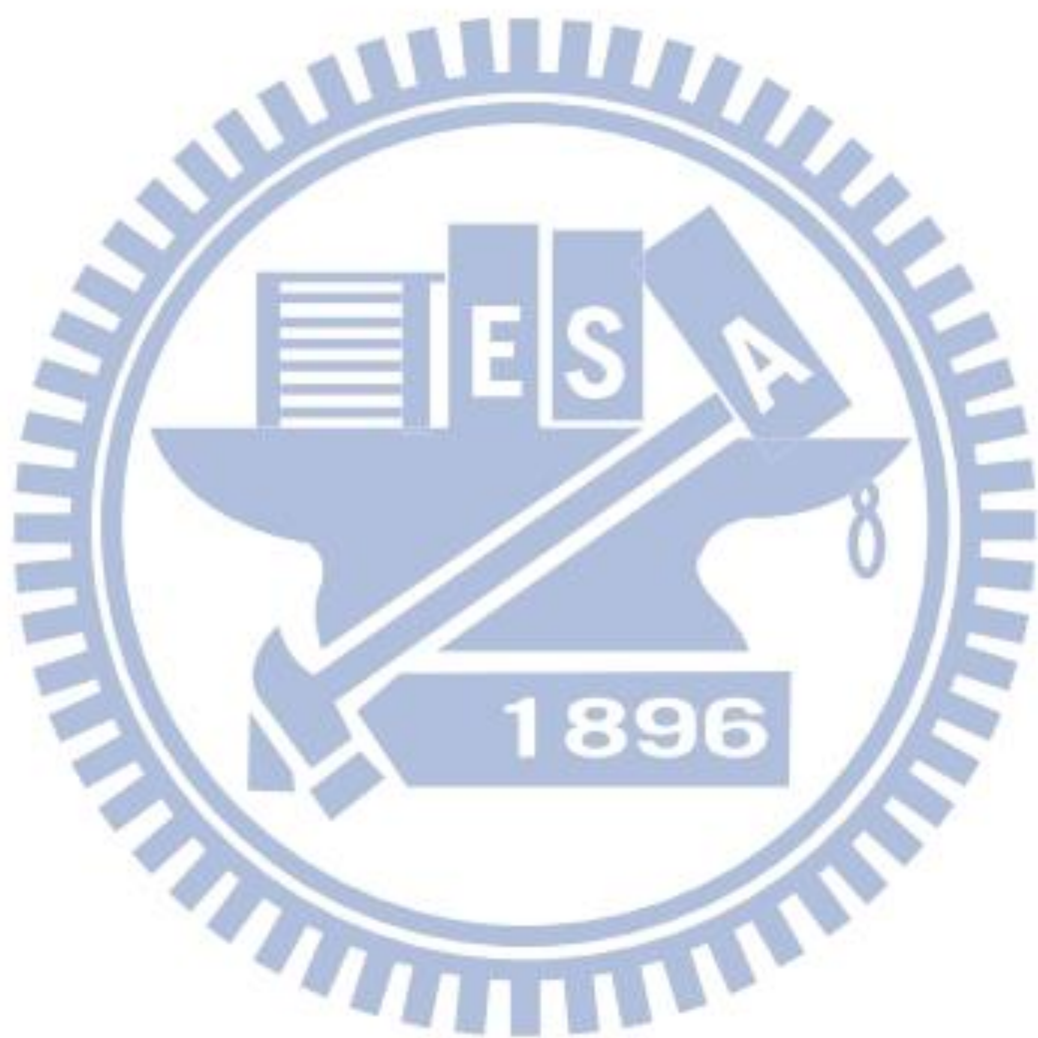
Dissertation

ProbeBuilder - Automating Probe Construction in Virtual Machine Introspection through Uncovering Opaque Kernel Data Structures

ProbeBuilder - 透過挖掘隱藏作業系統核心資料結構以自動化虛擬機器外部觀察的探針建構

Student: Chi-Wei Wang
Advisor: Shiuhpyng Shieh

July, 2014



ProbeBuilder - 透過挖掘隱藏作業系統核心資料結構以自動化虛擬機器外部觀察的探針建構

ProbeBuilder – Automating Probe Construction in Virtual Machine Introspection through Uncovering Opaque Kernel Data Structures

研 究 生：王繼偉　Student: Chi-Wei Wang
指導教授：謝續平　Advisor: Shiuhpyng Shieh

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

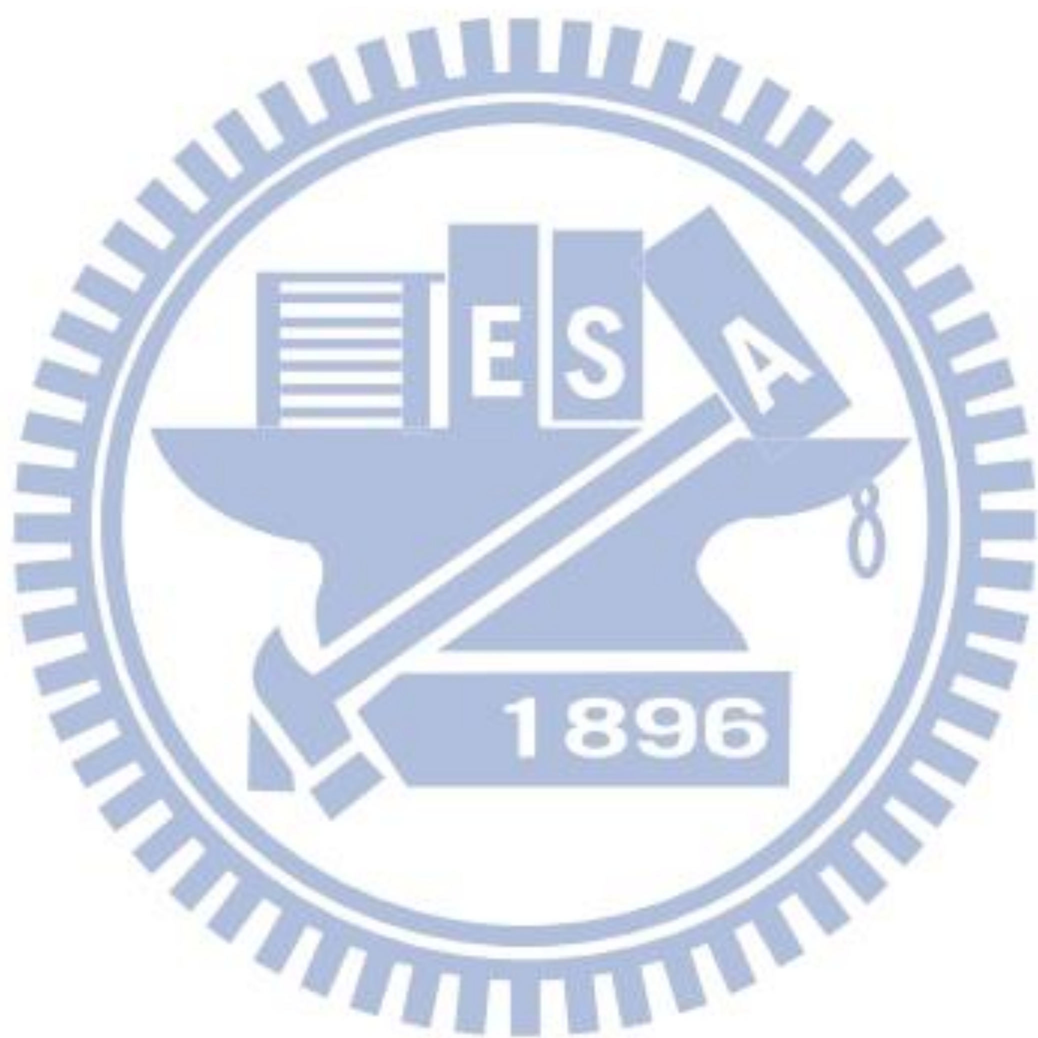
A Dissertation
Submitted to Department of Computer Science
College of Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy
in

Computer Science

July 2014

Hsinchu, Taiwan, Republic of China

中華民國一百零三年七月


E
S
A
1896

ProbeBuilder - Automating Probe Construction in Virtual Machine Introspection through Uncovering Opaque Kernel Data Structures

Student: Chi-Wei Wang Advisors: Shiuhpyng Shieh

Department of Computer Science
National Chiao Tung University

ABSTRACT

VM-based inspection tools generally implement probes in the hypervisor to monitor events and the state of kernel of the guest system. The most important function of a probe is to carve information of interest out of the memory of the guest when it is triggered. Implementing probes for a closed-source OS demands manually reverse-engineering the undocumented code/data structures in the kernel binary image. Furthermore, the reverse-engineering result is often non-reusable between OS versions or even kernel updates due to the rapid change of these structures. This dissertation proposes ProbeBuilder, a system automating the process to inference kernel data structures. Based on dynamic execution, ProbeBuilder searches for data structures matching the recursive "pointer-offset-pointer" pattern in guest memory. The sequences of these offsets, which are referred to as dereferences, are refined with a repetitive training process. ProbeBuilder further prepare stable probe locations for them with control flow analysis, and generate code snippets of probes for QEMU, KVM, and Xen. The experiment on Windows kernel shows that ProbeBuilder efficiently narrows hundreds of thousands of choices for kernel-level probes down to dozens, and the generated probes effectively capture both user-level and kernel activities. The finding allows analysts to quickly implement probes, facilitating rapid development/update of inspection tools for different OSes. With these features, ProbeBuilder is the first system capable of automatically generating practical probes that extracts information through dereferences to opaque kernel data structures.

ProbeBuilder - 透過挖掘隱藏作業系統核心資料結構以自動化虛擬機器外部觀察的探針建構

學生: 王繼偉 指導教授: 謝續平

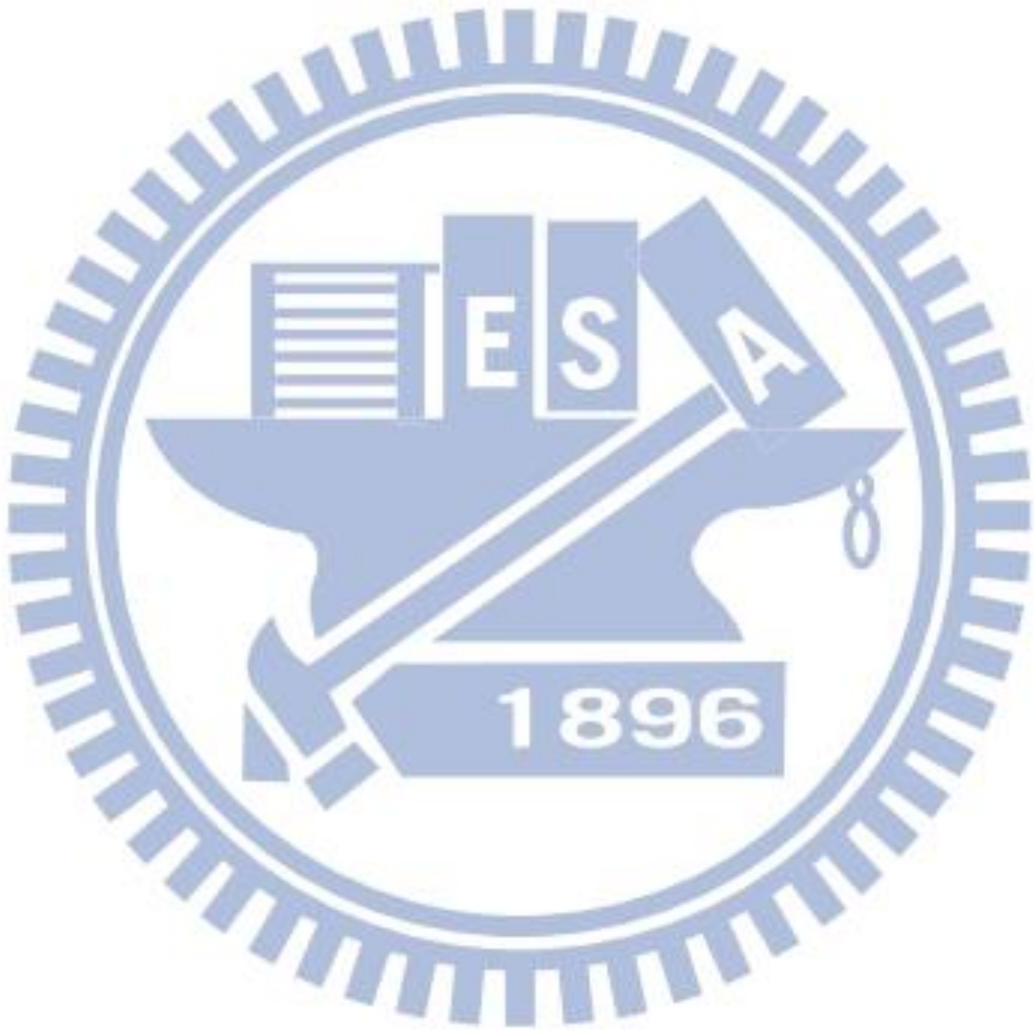
國立交通大學
資訊科學與工程研究所

摘　要

Virtual Machine Introspection, VMI 為一藉由虛擬機器運行目標程式，由虛擬機器外部進行行為觀察之分析方法。而此種分析工具為了能攔截客戶端系統事件並監視作業系統核心狀態，皆需要在虛擬機器管理器(Hypervisor)中插入程式探針(Probe) 。插入程式探針的目的在於，使客戶端作業系統內的程式執行流程觸及目標點時，虛擬機器管理器能暫停其執行並取得控制權。更重要的是，程式探針必須能從客戶端機器的記憶體內挖掘出與該事件有關的資訊。然而，若要為原始程式碼不公開的商用作業系統實作程式探針，往往需要對其核心進行手動軟體逆向工程，以得知其內部的程式流程與資料結構。更甚者，作業系統核心的頻繁更新，以及整體作業系統的更新，經常導致其程式與資料結構改變，因此逆向工程所得之結果，往往無法重覆利用。本篇論文提出 ProbeBuilder，為一自動化推斷作業系統核心程式與資料結構之系統化方法。經由動態執行，ProbeBuilder 在客戶端機器的記憶體中，不斷挖掘遞迴的「指標-偏移量-指標」的資料模式，以搜尋可能的資料結構。此外，透過程式流程分析， ProbeBuilder 可為所發現的資料結構，產生相對應的探針位置，並自動生成可插入 QEMU,KVM 以及 Xen 的程式片段，達成自動化的探針建構。經實驗驗證，ProbeBuilder 可自動為 Windows 作業系統快速地產生數十至數百的程式探針，並且可正確地捕捉使用者層級與核心層級的事件。本論文所提出之方法將可為分析人員利用，為不同的作業系統或核心版本，快速進行 VMI 工具之開發與更新。本論文所提出的系統核心資料結構挖掘方法，讓 ProbeBuilder 成為第一個具有自動化探針建購功能的系統。

Dedicated to my parents

獻給我的父母



Acknowledgement

Though the following dissertation focuses on the reconstruction of kernel-level data structures, it is actually based on a collection of resulted products of research topics that I explored during my Ph.D. career. The development and implementation of these systems could never be completed without the help, support, guidance and efforts of a lot of people.

Firstly, I would like to express my sincerest gratitude to my advisor, Professor Shiuhpyng Shieh (謝續平教授) who gave me the chance to explore the topic of software security and malware analysis. His altitude and strong experience on academic research helped me explore my ideas in a depth that I could never imagine. His guidance on academic writing also helped me transform my projects into academic publication. Working in the Distributed System and Network Security Laboratory, DSNS Lab, which is led by Prof. Shieh, I was able to meet and cooperate with the best researchers and students in the field. In my graduate life he unreservedly gave me both abundant research resources and full financial support, which are indispensable for me to complete these projects.

Chia-Wei Wang (王嘉偉), Chia-Wei Hsu (許家維), Bing-Han Li (李秉翰), and Chong-Kuan Chen (陳仲寬) are also Ph.D. candidates working in DSNS Lab. It is truly fortunate to work with these talented programmers and creative researchers. We co-worked on many research topics and projects in these years. Many thoughts and ideas are generated and gradually refined in the discussion among us. The life in DSNS Lab is always accompanied with responsibilities due to the close cooperation between our laboratory and industries or government institutions. Without their support, implementations of these systems would take much longer. They have my sincerest appreciation and best wishes on their graduation.

Ming-Hua Kuo (郭明華) works in the laboratory as our administrative assistant. She is one of the most intellectual and capable woman I have ever known. The laboratory would never operate as smoothly as it does right now without her effort. I already miss bickering with her after all these years.

My parents, 王敏玉 and 王良知, gave me unconditional love and support throughout my life. They made to ensure that I had all the resources to pursue excellence since I was a child. They receive my deepest appreciation and love for their dedication and the faith in me.

Over the past eight years I met lots of people in DSNS Lab. We co-worked, played, travelled, and laughed (got pissed-off, also) together. I will always treasure these moments. For 慧雯，政仲，經偉，均翰，穎昌，汎勳，芳瑜，慶峯，昀閔，子建，晏如，佳惠，宗賢，居安，怡嫻，善新，麟鈞, Sky, Michael, and all other friends that I met in this laboratory, I would like to say:

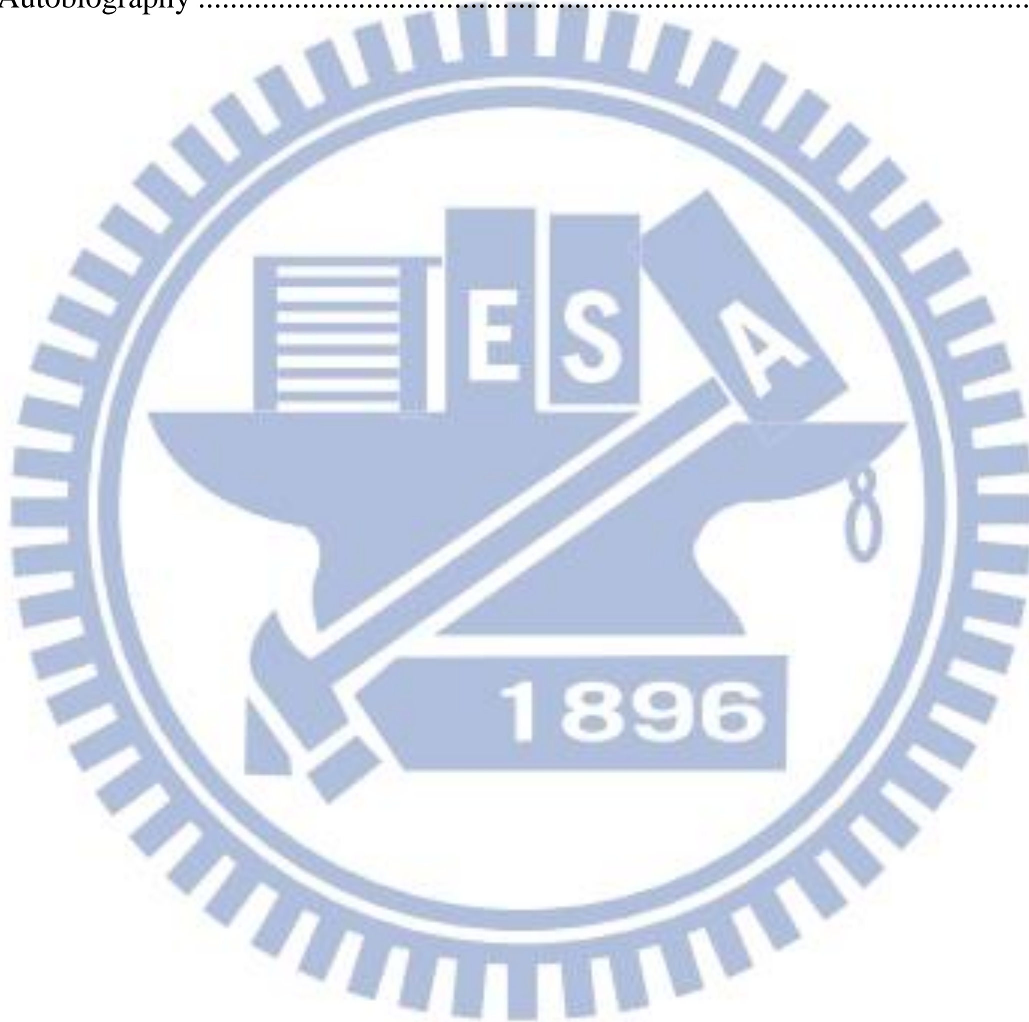
"Thank you for making the life of this nerd more colorful than he could've ever dreamed of."

Table of Contents

Page

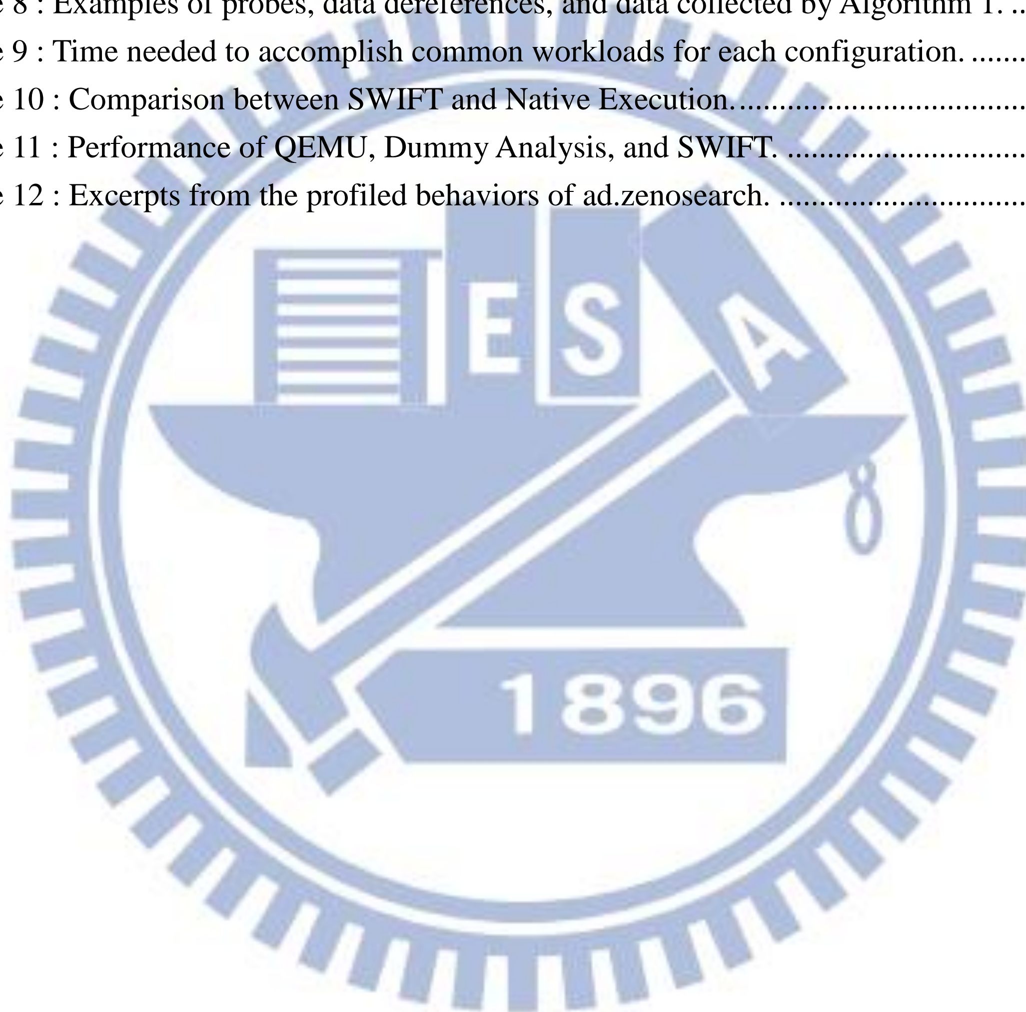
English Abstract i
Chinese Abstract ii
Acknowledgement iv
Table of Contents v
List of Tables vii
List of Figures viii
List of Algorithms ix
I. Introduction 1
1.1 Contribution 7
1.2 Synopsis 8
II. Related Work 10
III. System Design of ProbeBuilder 14
3.1 Data Dereference Analysis 15
3.1.1 Implementation of the Predicate P 20
3.2 Control Flow Graph Builder 22
3.3 Control Flow Graph Analysis 24
3.4 Code Generator 27
IV. System Design of SWIFT 28
4.1 Encoding Information Flows of Instructions 30
4.1.1 IA-32 Instruction Data-Flow Modeling 31
4.1.2 IA-32 Indirect Memory Access 37
4.2 Delivering IF-codes and Memory Addresses 38
4.3 Optimization 40
4.3.1 OPT1 : Delayed-Delivering on a Per-Block Basis 40
4.3.2 OPT2 : Stack-based Indirect Accessing 42
4.4 Peripherals 44
V. Evaluation 46
5.1 Probe Generation 47
5.2 Effectiveness of Generated Probes 51
5.2.1 Monitoring User-Space Activities 51
5.2.2 Monitoring Kernel-Space Activities 51
5.3 Performance 52
5.4 Malware Analysis with SWIFT 61
VI. Applications – Kernel Rootkit Recognition 70
6.1 MrKIP Internals 71
6.1.1 Behavior Profiler 72

6.1.2 Pattern Generator ... 74
6.1.3 Pattern Recognizer ... 79
6.2 Experiments ... 81
6.2.1 Case Study : Srizbi ... 81
6.2.2 Effectiveness of Recognition ... 83
VII. Limitation and Discussion ... 85
VIII. Conclusion ... 88
Bibliographies ... 90
Autobiography ... 98



List of Tables

Table 1 : An exemplified procedure to create probe w/ ProbeBuilder 4
Table 2 : The probes generated by ProbeBuilder.. 5
Table 3 : Typical Aggregated IA-32 Information Flow. 32
Table 4 : Information Flow of Common IA-32 Instructions 34
Table 5 : Remainder candidates after each run of Algorithm 1. 47
Table 6 : Remainder candidates after Algorithm 2 and Algorithm 3. 48
Table 7 : Eliminated non-dedicated functions. 49
Table 8 : Examples of probes, data dereferences, and data collected by Algorithm 1. 50
Table 9 : Time needed to accomplish common workloads for each configuration. 57
Table 10 : Comparison between SWIFT and Native Execution. 58
Table 11 : Performance of QEMU, Dummy Analysis, and SWIFT. 59
Table 12 : Excerpts from the profiled behaviors of ad.zenosearch. 73

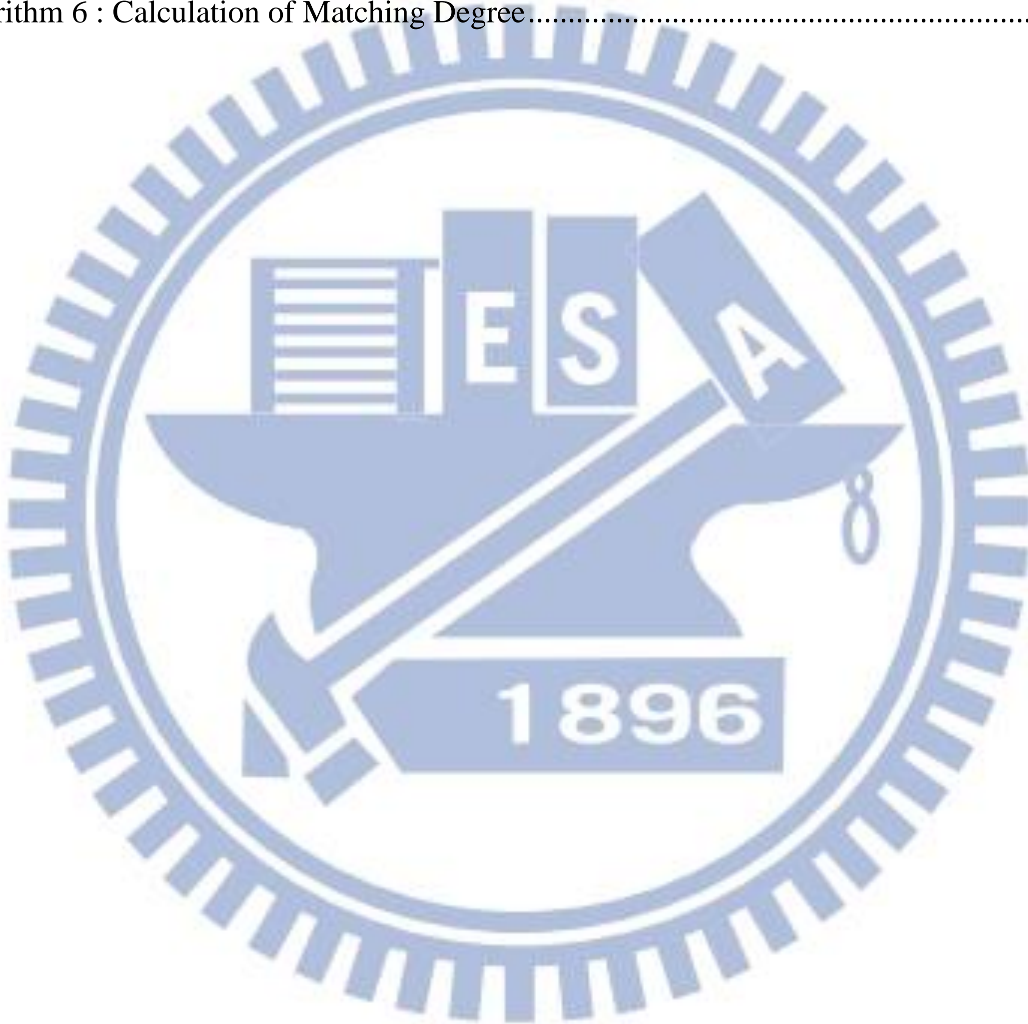


List of Figures

Figure 1 : A probe example. 2
Figure 2 : An overview of ProbeBuilder workflow. 15
Figure 3 : A simplified example of the search process. 19
Figure 4 : A typical approach to label arguments of a user-level API as taint source. 21
Figure 5 : Examples of undesired probe candidates. 23
Figure 6 : An example of the output of Algorithm 3 26
Figure 7 : Example of the generated probe for KVM. 27
Figure 8 : An overview on the basic system architecture of SWIFT. 29
Figure 9 : Representative instructions of aggregated information flow. 33
Figure 10 : Formats of IF-codes. 36
Figure 11 : Circular queue for IF-code delivering. 38
Figure 12 : An improved system design with OPT1. 41
Figure 13 : Scenarios of EBP-based memory address with offset within range -1024~+512. 43
Figure 14 : Overhead imposed by different configurations. 55
Figure 15 : The common workload overhead comparison between each configuration. 57
Figure 16 : Plots of emulation speed versus time and queue usage versus time. 60
Figure 17 : MrKIP Architecture 72
Figure 18 : Constructed model for Srizbi. 82
Figure 19 : Cumulative Ranking. 84

List of Algorithms

Algorithm 1 : System Emulation with Dereference Analysis...17
Algorithm 2 : Search for Leading Nodes...24
Algorithm 3 : Elimination of Non-Dedicated Code Blocks 25
Algorithm 4 : Calculation of the physical address generated by EBP-based accessing. 43
Algorithm 5 : Behavior to HMM state. 78
Algorithm 6 : Calculation of Matching Degree...80



Chapter 1

Introduction

Virtual machine introspection, or VMI, has received much attention in digital forensics and malware analysis. Executing unknown programs inside a dynamically created, isolated virtual machine, VMI systems allow analysts to observe the behaviors exposed by the subject program. Unlike conventional monitors and profilers, which co-exist with the subject program in the same execution environment, VMI is generally implemented at the level of the virtual machine manager, or the so-called hypervisor, that is, a VMI-based monitor (VMM) executes even beneath the operating system, allowing analysts to observe without causing interference.

There are various types of VMI applications. A specific type of VMI usage, that is, *probing*, is particularly useful for program behavior analysis. Probing refers to the process to suspend the guest virtual machine at specific moments, in which the state of the guest system can be collected and analyzed. Probing is used to monitor system events as well as program behaviors. In order to correctly capture every single event without manual operation, a trapping mechanism is generally needed to transfer execution to the hypervisor every time sensitive library functions are entered.

Although different virtualization methodologies (e.g. emulation, binary translation, hardware-assisted virtualization, etc.) have completely different trapping mechanisms, all of them need the same piece of information to be activated: the location of the trap. The trap location is generally specified with a memory address. Once the address is specified, the trapping mechanism is responsible of hijacking the execution every time the program counter

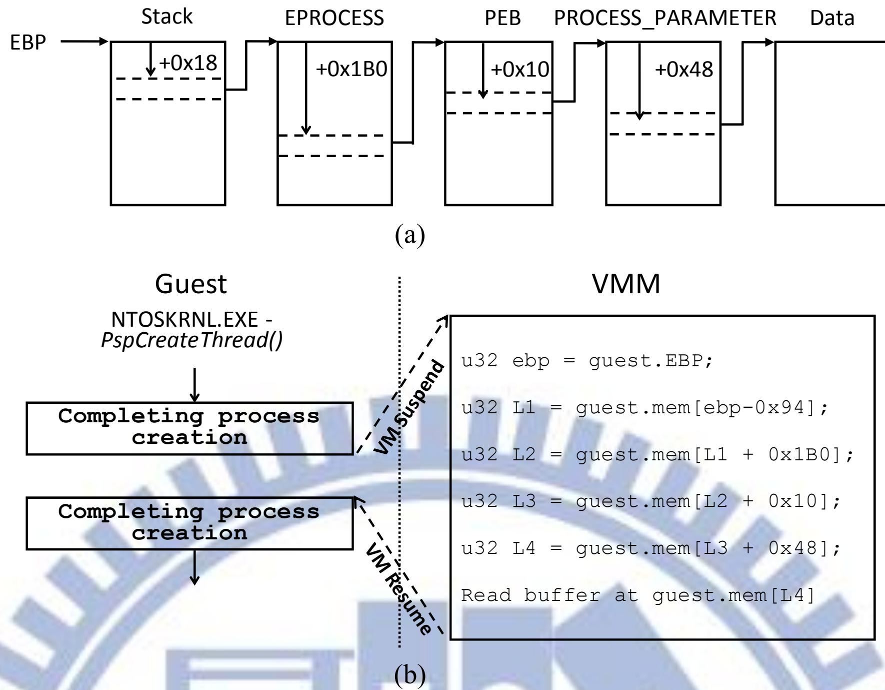


Figure 1 : A probe example.

(a) A revealed internal kernel data structure (dereference) from stack parameter to target data.
(b) A probe placed in *PspCreateThread()* to retrieve command line parameters through walking this dereference.

of the guest system matches the specified address. The implementation of traps on different virtualization platforms had been widely discussed in Sandbox-based analysis tools [1][2], since they often adopt probing to profile malware behaviors. This dissertation focuses on another issue: finding suitable location for placing the trap (probe).

Inserting probes in high-level functions like Win32 API or system calls can easily lead to incomplete profile results. First, the powerful functionality of modern operating systems allows a task to be done through different API or system calls. Exhaustively placing probes in all high-level functions is time consuming and error-prone due to the large number of these functions. Secondly, rootkits often invade the OS kernel to conceal their existence and disguise their malicious activities. Trojans Srizbi [55] and Stuxnet exemplify rootkits implemented in the kernel space. Merely probing high-level functions can be easily circumvented.

Instead, kernel functions are excellent candidates for placing probes due to their inevitability in all execution paths. Yet, kernel-level probing is a much more difficult task. The

following two challenges are often encountered.

Data Structure Inference – A useful probe extracts data of interests when it is triggered For instance, a probe logging process creation activities is often demanded to capture the program image path and the command line parameters. However, this usually demands traversing through undocumented kernel data structures. As shown in Figure 1(b), given a probe placed on the thread creation routine *PspCreateThread()* in the Windows kernel, it takes the traversing steps shown in Figure 1(a) to reach the buffer storing the command line parameter string. For conciseness, these traversing steps are referred as *dereferences*. Reverse-engineering the undocumented kernel function *PspCreateThread()* leads to the fact that the pointer to the data structure EPROCESS is the fifth function parameter, which locates at `[EBP-0x94]` by the standard calling convention. The rest of the steps are acquired through reverse-engineering undocumented data structures. In addition, the revealed dereferences vary between kernel versions since they are only for internal use. A simple update could easily invalidate all the effort.

Execution Flow Inference –Generally speaking, each probe is dedicated to profiling a single class of behaviors. Therefore, a probe must be placed on and only on the execution path of that specific behavior. For instance, *MmCreateProcessAddressSpace()* is a good candidate for profiling Windows process creation. It is used to create the page table of a process being created, that is, it is only invoked during process creation. This function is referred to as *dedicated* to process creation. Unfortunately, these kernel functions are often insufficiently documented. Facing a close-source operating system like Windows, one must reverse-engineer the OS kernel to reveal its execution flow to determine whether a code chunk is dedicated to a subject behavior.

Table 1 : An exemplified procedure to create probe w/ ProbeBuilder

Target	Sequence of steps
Guest	Run *notepad.exe*.
Guest	Acquire the PID of the *notepad.exe* process.
Guest	Enter a unique string *str1* in the notepad.
Guest	Open the “Save As” dialog, and name it with a unique string, *str2*.
ProbeBuilder	Specify search pattern 1: Fixed-Pattern, PID
ProbeBuilder	Specify search pattern 2: Regular-Expression, *str1*
ProbeBuilder	Specify search pattern 3: Regular-Expression, *str2*
Guest	Click the “save” button.

In this dissertation, a novel system, ProbeBuilder, is proposed. ProbeBuilder is the first system capable of automatically generating practical probes that extracts information through dereferences to opaque kernel data structures. It can minimize the effort of kernel-level probing through resolving the two aforementioned challenges. ProbeBuilder is able to automatically infer the kernel execution flow and data structures that are traversed in occurrence of an event or behavior. Initially, to identify the data of interest in memory, ProbeBuilder requests an operator to specify the identification method. Currently, ProbeBuilder supports the following three methods: fixed pattern, regular expression, and taint tracking. Based on the QEMU emulator, ProbeBuilder exercises the guest operating system and mines for valid dereferences to the data of interest.

We give an example herein to demonstrate the usability of ProbeBuilder. Supposed that an analyst is requested to implement a kernel-level probe to monitor file-writing operations in a file system. The probe is expected to capture three attributes, namely the path to the accessed file, the data written, and the ID of the process that issues the request. Table 1 illustrates a sequence of steps taken to create this probe with ProbeBuilder. The first column indicates where each step is taken. As shown, only eight steps are needed, where the human operator

Table 2 : The probes generated by ProbeBuilder

EIP	Dereference	Data
0x804e7461 IopUpdateWrite TransferCount	esp +16 +12 +0 +60	0xAC,0x0E (3756)
	esp +0 +84 +36 +60	__ProbeContent__\0a\00
	esp +12 +36 +52	\0P\0r\0o\0b\0e\0T\0e\0s\0t\0.\0t\0x\0t\0
0xf9926d22 <Unknown>	esp +4 +120 +0 +32	0xAC,0x0E (3756)
	esp +4 +192 +36 +12	__ProbeContent__\0a\00
	esp +20 +0 +40 +16	\0P\0r\0o\0b\0e\0T\0e\0s\0t\0.\0t\0x\0t\0

involvement is minimal. The entire procedure only takes a few hours to complete. The results in Table 2 illustrate the output of ProbeBuilder. Each generated probe consists of a code location, dereferences to the three attributes, and the string captured in the process. With these pieces of information, the code generator of ProbeBuilder can generate corresponding code snippets. The demonstrated procedure shows that ProbeBuilder effectively convert the difficult task of inspecting the OS kernel to a trivial user-space routine.

The most flexible way to specify data of interest is through the third method provided by ProbeBuilder: taint tracking. Taint tracking, also known as dynamic information flow tracking, DIFT, has been a widely-adopted analysis technique for software testing, malware analysis, and intrusion detection [3][4][5][6]. Using emulation [7] or binary instrumentation [8][9], executed instructions and accesses on memory or peripherals can therefore be monitored and analyzed. An application of this technique is taint analysis where CPU registers, memory cells, and sectors of hard-disks are augmented with a “dirty bit” to indicate whether a memory byte is tainted or not. The states of these bits are updated according to information flow caused by data movement or calculation.

Although the effectiveness of DIFT and taint analysis has been demonstrated in much past research [10][11][12][13], it comes at the cost of high performance overhead. Although various research works toward software-based DIFT speeding-up were proposed in the last few years,

they are limiting their scopes in one or several individual user-level process [14][15][16][17][18][19][20]. Since ProbeBuilder focuses on the data structure and execution flow inside the kernel of the operating system, none of the existing approaches can be applied.

An ideal approach is to decouple the analysis task from the system emulation so that the two tasks can be performed in parallel. However, in reality the analysis task has heavy data and control-flow dependency on the outcome of the emulation process. Due to register-indirect addressing and virtual address translation, memory addresses are unpredictable and can only be acquired after being generated by emulator's MMU. Consequently, causal relation and data dependency are introduced. Furthermore, to track information flows correctly, analysis must follow the execution path of the emulator, and therefore control-flow dependency is introduced. Delivering physical address to the analysis thread after each instruction execution is an intuitive but apparently inefficient approach since the massive data exchange between the two threads could sabotage the benefit of decoupling. Things become even worse when control-flow dependency enters the picture. Since most IA-32 instructions could lead to exceptions such as page faults and privilege violation, control-flow transfer could happen for each instruction. If analysis thread wishes to follow execution path of the emulator, it must be informed of whether an instruction has been successfully executed, which introduces dependency on a per-instruction basis.

To accelerate the taint tracking process of ProbeBuilder, an efficient system-wide information flow tracking platform, SWIFT, is also proposed in this dissertation. Two novel approaches were proposed to aggressively eliminate both data and control-flow dependency between emulator and analysis thread. For data (accessed physical address) dependency, SWIFT alleviates it with the fact that many memory accesses are EBP-based addressing and the value of EBP itself is changed less frequently. This observation is leveraged to make the physical addresses of such memory accesses can be calculated by the analysis helper itself. On the other hand, to reduce control-flow dependency, a communication mechanism is proposed by informing the analysis helper of the execution path transfer on a per-block basis. Consequently, fewer message transfers are required. The proposed approach maintains

correctness even if exceptions are introduced. Our evaluations indicate that SWIFT operates 2.74~7.48 times faster than conventional interleaved design while being benchmarked by PassMark Performance Test 6.0. Although the performance penalty on CPU-bound tasks is still high (12.74X~35.55X) in comparison with native execution, the overhead is mainly attributed to the inherent emulation nature.

1.1 Contribution

The drastic evolution of malware and frequent change of operating systems can easily make VMI systems out of date. To keep up, analysts need to continuously update their VMI systems in an extremely fast pace, and lots of manual effort are invested in reverse-engineering. The methods proposed in my dissertation make the following major novel contributions, benefiting analysts in their development process.

- ProbeBuilder automatically discovers the recursive dereference structures in the kernel. This novel feature helps automatically generate practical probes. The only portion requiring human assistance is the specification of the method to identify the data of interest.
- ProbeBuilder also generates accurate locations to probe after exercising the system. Control-flow analysis is applied to guarantee that generated probes will locate on location is dedicated to the subject behavior.
- The generated probes can profile kernel-level activities that conventional approaches cannot deal with.
- All malware profiler (ThreatExpert, Anubis) development can benefit from the result of ProbeBuilder since it eliminates the demand for reverse-engineering and greatly reduces the effort of implementing probes for a new OS and transplanting them between OS versions.
- A decoupled design, SWIFT, is proposed to accelerate system-wide, taint tracking by executing analysis task and system emulation in parallel.
- Approaches are proposed to aggressively eliminate the needs for the taint tracking

thread to communicate with the emulator. The performance is hence accelerated due to less L2-cache confliction.

- Information flows incurred by IA-32 instructions are studied in detail. The result is used to propose a concise encoding format to preserve complete IA-32 instruction information flows. Meanwhile, the conciseness enables efficient processing.

1.2 Synopsis

In this subsection the organization of this dissertation is described. Since virtual machine introspection and dynamic information flow tracking are very active research fields, in Section 2 research work related to VMI applications, DIFT, and DIFT acceleration are introduced to help readers understand the state of the art in VMI development and taint tracking.

Section 3 surrounds the central topic of this dissertation: ProbeBuilder. The description of Section 3 is organized in a top-down paradigm. An overall architecture of ProbeBuilder is firstly given. The architecture illustrates the input and the output of ProbeBuilder. Like any other large systems, ProbeBuilder consists of sub-modules, each of which is further elaborated in the sub-sections.

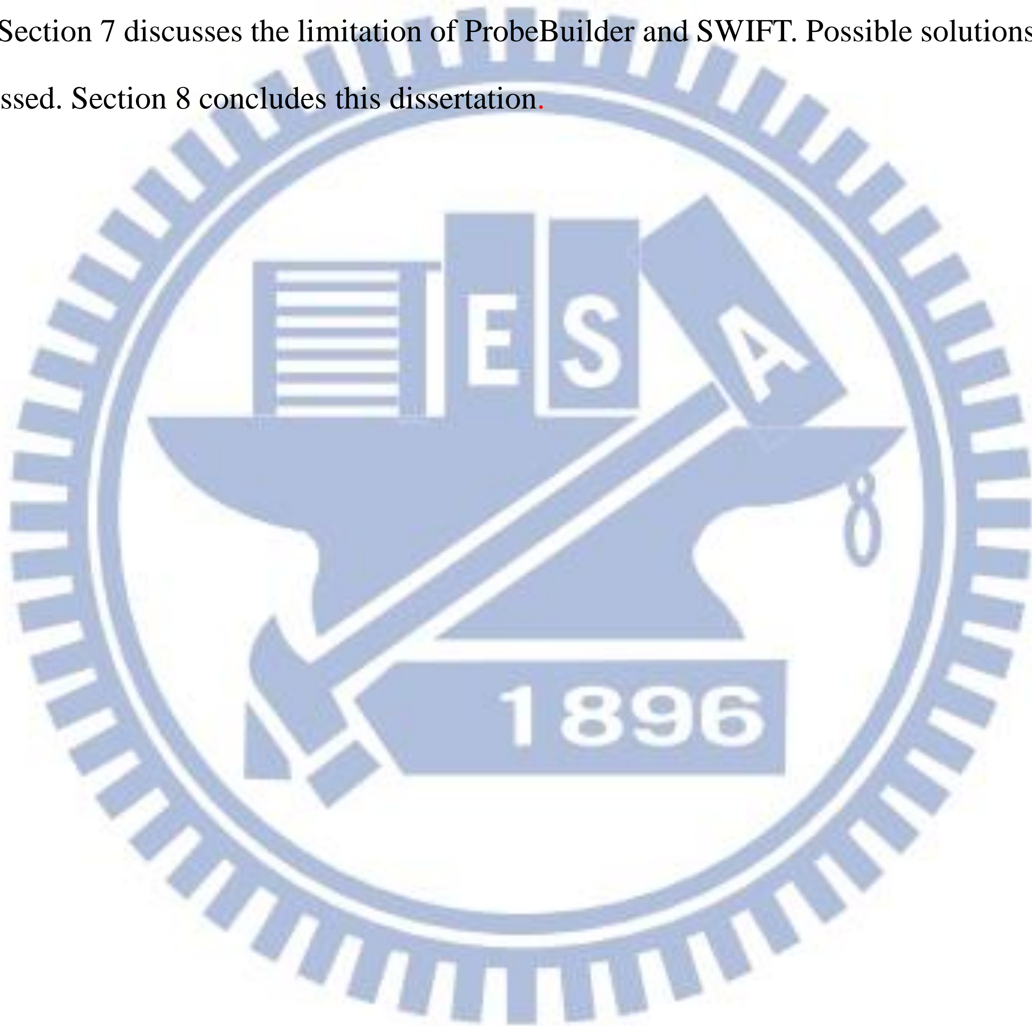
An important functionality of ProbeBuilder is allowing users to label data of interest through taint tracking. To provide practical taint tracking speed, a decoupled design of DIFT, namely SWIFT, is proposed. Note that the design of SWIFT does not rely on ProbeBuilder. It provides standalone use for any taint-based analysis. Therefore, the design of SWIFT is separately introduced in Section 4. And each proposed acceleration technique is described in the subsection.

To evaluate the proposed methods, experiments are conducted on both ProbeBuilder and SWIFT. The effectiveness of ProbeBuilder is tested through checking if user-level and kernel-level activities can be logged by the generated probes, and the results are compared with the log generated by commercial tools. The performance of SWIFT is benchmarked with both commercial test suite and common workload, and the result is compared with the famous taint-

tracking system TEMU. These experiments are included in Section 5.

Section 6 introduces a practical application of ProbeBuilder and SWIFT: recognizing rootkit with kernel function invocation patterns. This system firstly utilizes the probes generated by ProbeBuilder to automatically generate a kernel-level activity monitor. Then, the monitor is executed on top of SWIFT again to profile the tainted argument of kernel functions. The sequence of the occurrence of tainted kernel functions can be used to recognize rootkits.

Section 7 discusses the limitation of ProbeBuilder and SWIFT. Possible solutions are also discussed. Section 8 concludes this dissertation.



Chapter 2

Related Work

The conventional approach to implementing a probe is through hooking. A hook is a chunk of code injected into a program to intercept execution flow. Advanced rootkits and anti-virus software often compete with each other on the depth they implant hooks, seeking the priority of execution [21]. SIM [22], Lares [23], PsycoTrace [24], Process Implanting [25] realize probing with this technique. With the assistance of virtualization and emulation, probes can now be implemented in the hypervisor, providing better stealthiness. Sandbox-based analysis tools [1][2][26] generally adopt this approach to intercept malware behaviors. VMwatcher [27], VMDriver [28], VSyscall [29] also use similar techniques to intercept critical system events for inspection. However, the "out-of-box" implementation paradigm does not re-solve the issue of depth competition. For instance, a VM-based probe placed on *EnumProcess*(), which is a user-level Win32 API, will be always misguided by the root-kits manipulating the kernel function *NtQuerySystemInformation*(), even if the probe itself is implemented in the hypervisor. Consequently, the emergence of hypervisor-level implementation does not relax the need for deeper probes. All work above focus on application of probes, instead of their construction.

To ease the pain of semantic reconstruction in VMI, Virtuoso [30] executes the user-level profiling tools in the guest operating system and collects the code trace generated by QEMU for repetitive use. LiveWire [31], VMST [32], and VIX [33][34] automate VMI through executing the profiling process in a separated shadow VM and redirecting its memory access to the target VM. HookMap [35] and HookSafe [36] are dedicated to dis-covering indirect-branch-

based hooks to hijack execution flows. TZB [37] analyzes memory access patterns to acquire code locations that construct strings matching the specified, fixed pattern. However, neither does it reveal recursive pointer dereference structures nor the execution flow. ProbeBuilder simultaneously resolves these two issues, making it powerful for VMI probe construction.

Dynamic information flow tracking has been widely studied. It is demonstrated as a powerful tool for malware behavior analysis [10][11][12][13][38] and software testing [8][9]. By executing the target executable in an emulated environment, these systems are allowed to watch each CPU instruction execution at runtime and to track information flows dynamically. Paranoma [10] is a generic, extensible whole-system DIFT analysis platform on top of which multiple analysis plug-ins such as Panorama have been developed. Although Paranoma is equipped with complete functionalities for malware analysis, its performance downgrade is severe. Due to interleaved system emulation and analysis routines, it runs about 20 times slower than native execution [10] when tainting is enabled. The same condition applies to all other work as well.

PinOS [39] is another binary instrumentation framework proposed for whole-system dynamic analysis. With Intel hardware virtualization support (VT-x) and dynamic binary translation, PinOS is capable of instrumenting CPU execution at instruction granularity. However, the usage of virtualization cannot prevent its execution from being encumbered by interleaved analysis. 7 out of 9 performance benchmarks in [39] shows a 48X~121X slowdown even when PinOS runs with no instrumentation.

Aftersight [40] records non-deterministic events of a virtualized environment so that events can be replayed later. Since event recording produces much lighter overhead than the analysis does, the system can provide online services. However, Aftersight has a different intention from SWIFT because Aftersight tries to postpone the heavy-weight task rather than to accelerate the process of dynamic analysis.

Due to the performance issue, optimization on DIFT received extensive attention within past few years. Designs sped up with additional customized hardware are discussed widely. Hardware architectures with native taint propagation support are proposed in [41][42]. In these

architectures, a taint tag is augmented for every value in memory, cache, and the processor. While executing instructions, the modified processor automatically propagates these tags and hence dirty data flows are tracked. However, the hardware extension requires corresponding modification on operating systems. The modification can be huge or even impossible on certain closed-source operating systems, such as Windows. Besides, since only architectures of processors and memory are extended, taint analysis cannot track information flow in peripherals.

Another effective approach is to decouple the analysis process from the program execution itself. There are two ways to realize the idea. One is to extend processors so that instructions executed on one core are recorded and en-queued in a hardware message queue, and processes running on other cores could "peek" logged instructions with a special de-queuing instruction. Processors with this capability are called Log-Based Architecture, LBA [14][15][16][17]. The most significant advantage of LBA-based CPUs is that the overhead of instruction tracing is eliminated by hardware. However, the mechanism mentioned above makes them suitable for process-level testing and monitoring, but not for system-wide information flow tracking. The other decoupling methodology is to dynamically instrument instructions to execute so that instructions can be logged or monitored [18][19][20]. Since the binary instrumentation framework all limit their scope within one process, they cannot be applied in system-wide information flow tracking, either.

Using aggressive dynamic binary instrumentation and optimization, LIFT [19] performs DIFT-based security checks on applications. LIFT analyzes the loaded process before execution and tries to locate code regions which can only interact with safe data. Since unsafe input can usually enter the system through certain data paths, fewer blocks need to be instrumented. However, performing security checks and malware behavior analysis differ in two ways, which make their methodologies unsuitable in our applications. To perform malware analysis, massive amount of data will be considered as unsafe inputs, such as the whole body of the malware. Therefore, eliminating safe code regions becomes much less likely. Minemu [43] is an efficient process-level taint tracker. However, tracking information flows inside one process cannot benefit observing effects exerted by the malware in the operating system. Although most

infections of the kernel are caused by a user space program, there are pure kernel-level malicious programs. In fact, Srizbi, which is a Trojan program responsible for 40% of all the spam on the Internet in 2008, hides its file and sends out spams without any user-space components [55]. This kind of malware can only be analyzed at the system level.

Demand emulation [44] performs DIFT with emulation still. It accelerates by removing tainted pages from the page table and switching to virtualization once none of CPU registers contains tainted information and multiple untainted memory pages are consequently accessed. When accessing a tainted page, it will fall back to emulation to track information flows. Demand emulation is intended for system-level DIFT. However, it does not truly solve the problem that emulation with analysis enabled is catastrophically slow. As indicated in their original paper, frequent switches between virtualization and emulation even lead to worse performance overhead than pure emulation due to unpredictably frequent memory accesses. In addition, demand emulation requires modification on guest system so that tainted and untainted data would be placed on different pages, since it relies on page fault to leave virtualization.

In the technical report PTT, Ermolinskiy et al [45][45] proposed another novel system-level DIFT tool. It implements the technique of demand emulation and proposes a concept of separating the analysis and the emulation. Their work partially covered the decoupling idea, but there are clear differences distinguishing our contribution from PTT. First of all, formal taint propagation rules are provided in SWIFT, but PTT lacks of these information. In addition to the decoupled design, two more optimization techniques are proposed in SWIFT. One is the per-block-basis delivering and the other one is the elimination of EBP (or ESP) base memory address delivering. Meanwhile, our work consists of not only the decoupled design but also the methods to apply such a design to practical malware analysis. Unlike SWIFT, PTT only demonstrates the possibility of decoupling whole-system taint tracking.

Chapter 3

System Design of ProbeBuilder

In this section, the architecture of ProbeBuilder is introduced. An overview of the ProbeBuilder workflow is depicted in Figure 2, The workflow is traversed using the same example illustrated in Section 1. However, for conciseness, it is assumed that only the second search pattern exists, namely the regular expression specifying the keyed-in string. These search patterns are referred to as a *predicate P*: *Addr* → {TRUE, FALSE}. They are used to instruct ProbeBuilder how to determine whether the data stored on a given memory address are of interest or not.

The process starts with the data dereference analysis module, which is built upon the QEMU emulator installed with Windows XP. The system is exercised either manually or programmatically to force the kernel to execute the behavior that the operator wish to create probes for. Along the execution, the data dereference analysis is invoked at the entry of each code block. The intermediate output will be a set of pairs consisting of a deferencing point and the corresponding deferencing steps. The deferencing point is simply an instruction address, and the deferencing steps preserve the dereferences to the string matching the predicate. Let's assume that the data dereference analysis gives the following output.

```
< 0x804f54af, (ESP, +4, +192, +36, +12)>
< 0x804f13ba, (ESP, +4, +120, +4, +32)>
```

Note that the module of data dereference analysis guarantees that if the CPU is "freezed" every time it executes through the instruction at `0x804f54af` (and, of course, under the same

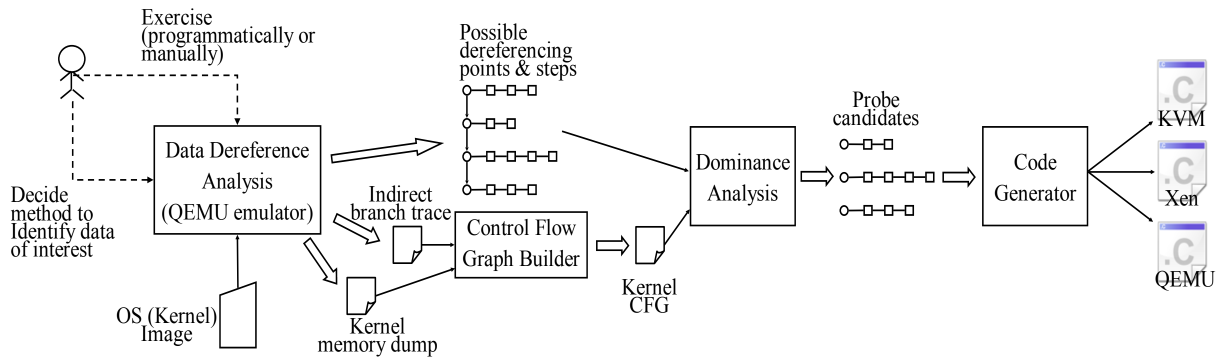


Figure 2 : An overview of ProbeBuilder workflow.

process context) and take the dereference `*((((ESP+4)+192)+36)+12)`, ProbeBuilder will (almost) always find the string matching the given regular expression.

However, not every one of these results can be used as probes. For instance, `0x804f13ba` could be the location of function *RtlCopyMemory*(), which is invoked widely as long as a buffer-copy is needed. Placing a probe here can lead to false positive output. The control flow graph, CFG, builder and the dominance analysis are designed to eliminate these non-dedicated code locations. The CFG builder constructs the graph from memory dump of the kernel, not the image file on the hard disk, so that certain control flows bound at runtime can be also captured. Indirect branches (e.g. `jmp EAX`) are filled with the indirect branch traces recorded during the execution of the emulator. The control flow analysis utilizes the generated control flow graph to eliminate dereferencing points not dedicated to the subject behavior. With the discovered dereference structures, the code generator is hence able to produce code snippets, which can be directly inserted into the hypervisor.

Sections 3.1 to 3.4 will introduce the details of the three core modules illustrated in Figure 2: data deference analysis, control flow graph builder, control flow graph analysis, and code generator respectively.

3.1 Data Dereference Analysis

This section illustrates the method that the data dereference analysis module uses to identify valid dereference sequence to data of interest. As aforementioned, a useful probe must extract numerous attributes accompanied with the probed behavior or event. Therefore, this

component plays the most essential role in ProbeBuilder.

The example listed in Figure 1(a) shows an intuitive, general principle of discovering a valid dereference sequence to data. The sequence in general starts with some register, say *r*, because the processor has to load the address into the register before it can access that memory location. The register often can be the stack (or frame) register if the pointer to the outermost data structure is a passed argument or a local variable on the stack. In this case, an non-zero offset σ_1 is often added to the value of stack (or frame) register to address that object, as the first +0x18 offset shown in Figure 1(a). However, the pointer to the outermost data structure is not necessarily stored on the stack. It can be directly loaded into other registers. In this case, the offset σ_1 added to the register is simply zero.

If the data of our interest reside in the outermost data structure, the dereference sequence ends up with *r* and σ_1. Otherwise, the sequence can continue with more offsets σ_2, σ_3, ..., σ_n, which are used to traverse through the subsequent data structures.

The data dereference analysis is described below. For formal presentation, the definitions and notations will be introduced first. Let's assume that the machine provides a set of general purpose registers *Reg*, and a virtual memory address space *Addr*. A machine $S = (pc, R, M, P)$ is a 4-tuple, where *pc* is the value of the program counter, R: $Reg \rightarrow Value$ gives the value of CPU registers. M: $Addr \rightarrow Value$ returns the memory content at the specified address in a special way. Instead of returning a single byte, *M* returns the whole word at the address. Note that the word size is architecture-dependent, and this abstraction makes the analysis applicable on different architectures. P: $Addr \rightarrow$ {TRUE, FALSE} is a predicate indicating whether the data (byte) at that address is of interest. More discussion on this predicate is given in section 3.1.1. Note that *pc*, *R*, *M*, and *P* change along with the system execution because the execution modifies the system state. Fortunately, the analysis only utilizes the state at the present instant. Therefore, *pc*, *R*, *M*, and *P* always reflect the present state of the machine. The state transition is based on the architecture of the emulated machine, which is out of the scope of this dissertation.

In addition to *pc*, *R*, *M*, and *P*, another special predicate $Valid$: $Addr \rightarrow$ {TRUE, FALSE},

Algorithm 1 : System Emulation with Dereference Analysis

```
Global:
    S = (pc, R, M, P)
    D is a table mapping addresses to a set of dereference sequences.
Constants:
    SRCH_DEPTH defines the maximal depth to search for a valid dereference sequence.
    SRCH_RANGE[] is an array giving the search range at each level
    K is the size of a pointer on the target architecture.
```

```
AnalysisLoop()
1  while TRUE
2  | if D[pc] = NIL then
3  | | D[pc] ← ∅
4  | | foreach r ∈ Reg
5  | | | for offset ← -SRCH_WIDTH[0] to SRCH_WIDTH[0] step K
6  | | | | if Valid( R(r) + offset )
7  └ └ └ └ Search( 1, r, <offset>, M( R(r) + offset ) )
8  | else
9  | | foreach p ∈ D[pc]
10 | | | if not Valid( R( p.r) + p.σ[0] )
11 | | | └ goto 19
12 | | | m ← M( R(p.r) + p.σ[0] )
13 | | | for i ← 1 to p.σ.length - 1
14 | | | | if not Valid( m + p.σ[i] )
15 | | | | └ goto 19
16 | | | └ m ← M( m + p.σ[i] )
17 | | | p.f ← p.f + 1
18 | | | continue
19 └ └ └ D[pc] ← D[pc] \ {p}
20 └ emulate from pc until branch or jump is encountered.

Search(d, r, σ, m)
21 if d = SRCH_DEPTH
22 └ return
23 for offset ← 0 to SRCH_WIDTH[d] step K
24 | if Valid( m + offset )
25 | | if not P( m + offset )
26 | | └ Search( d + 1, r, σ || <offset>, M( m + offset ) )
27 | | else
28 | | | p ← allocate an empty dereference sequence
29 | | | p.r ← r, p.σ ← σ, p.f ← 0
30 └ └ └ D[pc] ← D[pc] ∪ {p}
```

indicates whether the virtual memory address is an accessible address, meaning whether the address is actually mapped within the current page table. Since the page directory structure is also stored in the memory, *Valid* can be implemented solely with queries to *M*. Therefore, *Valid* is not considered as an extra input.

The analysis aims at collecting possible dereference sequences to data of interest. As aforementioned, a dereference sequence *p* begins with a register *r* and a series of offsets σ_1, σ_2, ..., σ_n, denoted as *p.r* and *p.σ*[.], respectively. Note that *p.σ*[.] is a list structure, and the notation || denotes concatenation with another list <.>.

The analysis procedure is shown in Algorithm 1. Lines 1 and 20 in AnalysisLoop() show that the analysis procedure between lines 2 and 19 is invoked before each dynamic code block is executed. The idea is to sweep through the memory region that is currently pointed by some register. In the region, the analysis searches for any data (words) that can be viewed as a pointer to a valid memory address. If any pointers are found, the procedure recursively traverses through them and repeats the similar sweeping operation until the maximal search depth SRCH_DEPTH is reached.

Line 2 checks whether the current code block has been analyzed before or not by looking up the table *D* with the value of the program counter. If not, the search process will be activated. Lines 4 through 5 enumerate all combinations of CPU register and possible offsets, and use them as starting points for the search. As indicated in the algorithm, the scanned offset ranges from 0 to SRCH_WIDTH[d]. The range contains negative offsets so that both local variables and passed arguments on the stack can be covered. The parameter K controls the increase to the variable *offset* in each iteration. Since the analysis searches for pointers, K is assigned with the pointer size of the target architecture in this dissertation. Line 6 checks if $R(r) + offset$ points to a valid memory address. If so, the recursive routine Search() is started on line 7.

The routine Search() begins on line 21 by checking if it has reached the maximal search depth. If not, on line 23 it scans offset ranges from *m* to *m*+SRCH_WIDTH[*d*], where *m* is the base address of the search range and *d* indicates the current depth of the recursion. Line 24 queries the predicate *Valid* to determine whether pointer *m+offset* refers to a valid memory address or not, just like line 6 does. This prevents the recursion from walking on an invalid data path. On line 25, the predicate *P* is queried to determine if any target data locate on $m + offset$. If not, it recursively invokes Search(), passing the current path $\sigma||offset$ as the base address for the search range at the next level. Note that in Search() the enumeration is confined in the scope from 0 to MAX_WIDTH because it is assumed that in consequent dereferencing steps, the passed-in argument m is the base address of some data structure. The offset should be always positive since it is supposed to point to its member variables.

Unfortunately, the collected dereferences may not be always valid because *S* only reflects a transient state. The pointer in each dereferencing step can be changed and becomes invalid. To eliminate unstable dereferences, lines 9-19 are executed. When *pc* has been included in *D*, the procedure enumerate through every paths collected in the previous search. Each dereferencing path is walked through again so that its validity on each step is verified. Each time a dereference passes through this test its counter *p.f* is increased. To find truly stable dereferences, the exercise should be repeated multiple times. These dereferences survived every elimination test with *p.f* greater than some threshold will be selected for later processing.

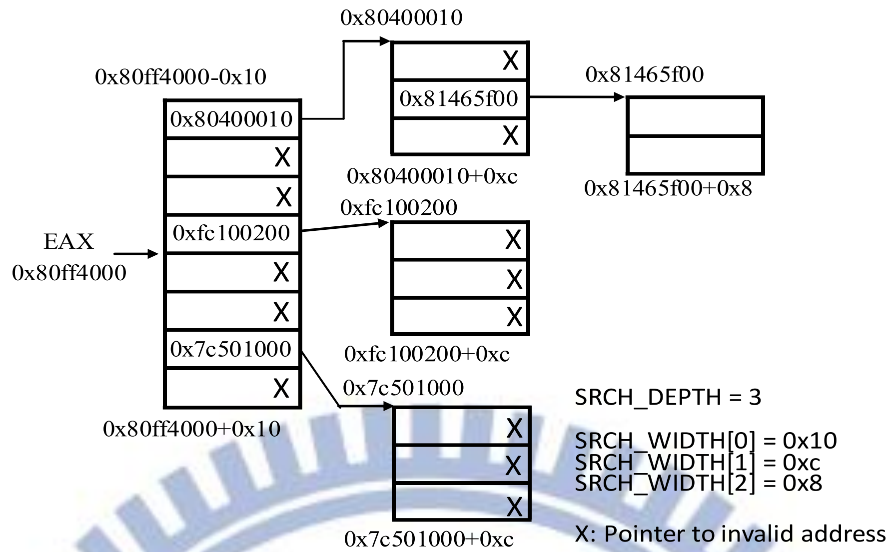


Figure 3 : A simplified example of the search process.

Figure 3 shows an example of the search process, which is given with the SRCH_DEPTH and the SRCH_WIDTH array depicted in the figure. The search starts with the register `EAX`, which contains value `0x80ff4000`. At the first level, the range [`0x80ff4000-0x10`, `0x80ff4000+0x10`] is searched, as described in algorithm 1. At deeper levels, only [*m*, *m*+SRCH_WIDTH[*d*] is searched. Note that the search width becomes smaller at deeper levels. This is an optimization based on the fact that the size of a child object is usually smaller than that of its parent. Otherwise, the search tree will quickly exhaust the memory. However, the search range at the last level should be enlarged again so that those data prefixed other strings. In our implementation, SRCH_DEPTH is set to 5 and SRCH_WIDTH is set to [512, 256, 128, 64, 512]

Also note that the search is only performed after a branch or jump instruction. The analysis collects only the dereference sequences that can be traversed at the beginning of a dynamic code block. The per-block basis is employed mainly due to the performance issue. Invoking the procedure at every instruction will lead to unacceptable overhead. In addition, checking every instruction is often overkill because between two adjacent instructions the machine state S does not vary significantly enough to expose different dereference sequences.

3.1.1 Implementation of the Predicate *P*

ProbeBuilder queries the predicate *P* to check whether it reaches any data of operators' interests. The predicate is simply a C function returning a Boolean value, and ProbeBuilder sets no limit on its implementation. Operators can always create new predicates to fit their needs. Nevertheless, ProbeBuilder provides three basic predicates: fixed pattern matching, regular expression, and taint checking, each of which has its pros and cons. Here their characteristics, usage, and application are discussed.

A. Fixed Pattern Matching

An operator can specify a fixed sequence of characters as the predicate. For efficiency, under this mode ProbeBuilder will execute the predicate, which is originally executed on line 25 in Algorithm 1, before the for-loop on line 23. Namely, the pattern matching is shifted out of the loop and is executed only once in every invocation of Search(). This mode works in high performance and gives no false positives, yet it also demands operators to know exactly the pattern of the data of their interests. A suitable application scenario is matching against the data buffer of a transmitted TCP/UDP packet. The buffer content will always stay unchanged since it is fed into the user-level API *send()*. If the immutability of the data can be predicted, fixed pattern matching should be used for higher performance.

B. Regular Expression

Likewise, under this mode ProbeBuilder also move up the execution of the predicate before the for-loop, and only the memory region *M*[0..SRCH_WIDTH[*d*]] is matched against the expression. A regular expression provides more flexibility than a fixed pattern does, however it may also incur false positives and extra overhead, depending on the expression specified. This mode should be chosen when the mutation rules of the data are predictable. For instance, Windows kernel often performs conversion between Unicode and multi-byte strings. These variants can be captured with regular expression.

C. Taint Checking

In certain cases the mutation rules of the target data may not be easily predictable. For

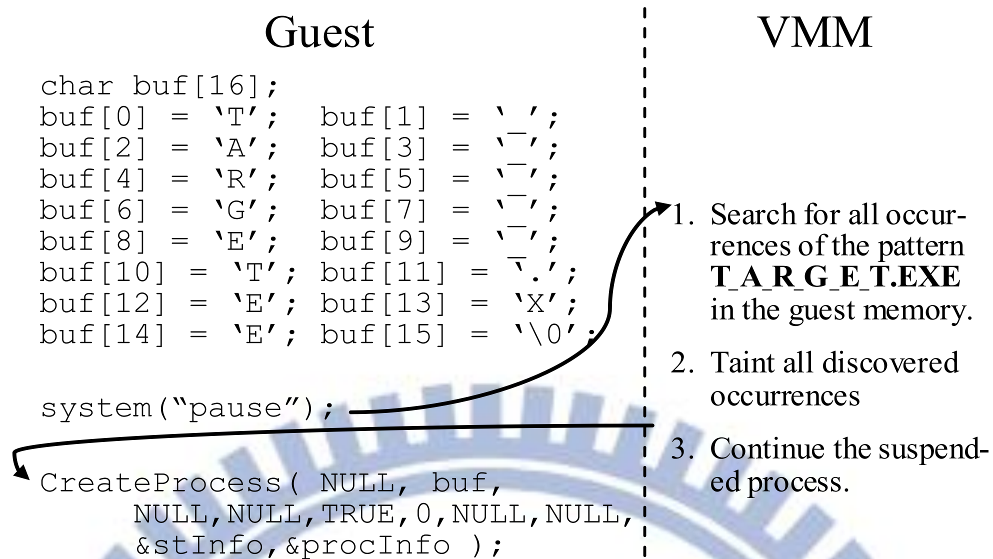


Figure 4 : A typical approach to label arguments of a user-level API as taint source.

instance, the DNS look-up module has to convert the domain name to match the format of question portion of DNS query message (e.g. a query "`google.com`" will be converted to "`\6google\3com\0`"). ProbeBuilder provides a fine-grained dynamic taint tracker, SWIFT, which is introduced later in this dissertation, for operators to perform a fussier matching. The utilization of taint analysis is straight-forward. P: $Addr \rightarrow$ {TRUE, FALSE} simply returns the status of the taint tag on that address. This mode searches comprehensively since all data flows are tracked, but it also inevitably incur high overhead and large amount of false positives/negatives.

Before taint tracking starts, operators have to manually specify contaminant sources. Here possible contaminant sources and their application scenarios are discussed.

User-Level API Parameters

Using API parameters as contaminant sources is extremely effective to reveal valid dereferences in kernel space. For example, in *CreateProcess*(), tainting the argument specifying the program path to execute can quickly identify the kernel functions dedicated to process creation. Since user-level APIs are exported and well documented, tainting these arguments is a trivial task.

Figure 4 shows a typical approach to tainting arguments of a user-level API. In this case,

the program path of *CreateProcess*() is targeted. Initially two programs are prepared. One is a dummy program with a unique file name. In this example, it is named as "**`T_A_R_G_E_T.EXE`**" and it will exits immediately after being executed. The other program is built to execute the operations listed in the left column of Figure 4. As illustrated, it constructs in the buffer the file name of the dummy program, suspends until operators press any key, and then finally invoke *CreateProcess*() to execute the command string constructed in the buffer.

Hard Disk Sectors and NIC RX buffer

ProbeBuilder also supports the functions of specifying any hard disk sectors and the RX buffer of the network adaptor as taint sources. This feature allows operators to quickly locate kernel functions used for disk-reading or packet-receiving.

We refer to a predicate as *byte-wide* if it returns TRUE only when the byte at the specified address is tainted. A byte-wide predicate provides the highest precision. However, under certain circumstances a byte-wide predicate may miss the data of interest. For example, the converted DNS query string "`\6google\3com\0`" can fail the predicate because the first byte '`\6`' is the result through control-flow-based calculation, which is difficult to deal with taint analysis. To cope with this issue, ProbeBuilder allows operators to specify the checking range of the predicate, that is, the predicate checks all the taint tags in the byte sequence locating in [*Addr*, *Addr*+*n*), where n is the length specified by operators. In our implementation, *n* is set to 8 to cover the whole 64-bit machine word of the IA-32 architecture.

3.2 Control Flow Graph Builder

This section discusses the reason why control flow analysis is needed and how the control flow graph is constructed for the later analysis components.

Consider the program structure shown in Figure 5. Both functions *NtWriteFile()* and *CmSetValueKey()* for modifying Windows registry entries utilize *RtlCopyMemory()* to perform data movement. At first sight, placing a probe on *RtlCopyMemory()* seems to be a good choice for capturing file writing operation. However, it will also capture the data from *CmSetValueKey()*, causing the mixture of produced profile with unwanted information. In

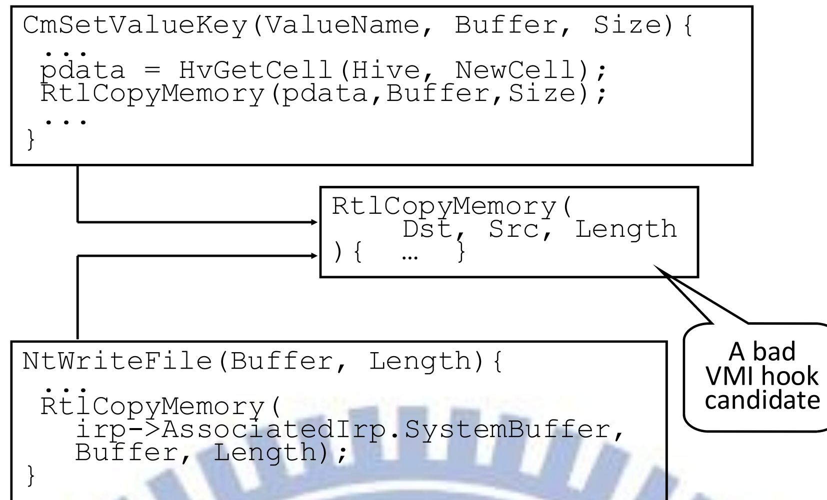


Figure 5 : Examples of undesired probe candidates.

addition, *RtlCopyMemory()* is extremely frequently used by other parts of the system. Placing a probe in this function will produce massive unwanted data.

The probe candidate refinement procedure begins by converting the kernel image to a graph with code blocks as nodes, and branches as edges. In the graph, there should be a sub-graph containing all nodes used to implement the subject behavior (e.g. registry modification). This sub-graph is referred to as *behavior sub-graph*. Through reachability analysis, probe candidates reachable without walking through this sub-graph should be identified and eliminated. Details are described below.

The control flow graph builder construct the control flow graph, CFG, for the kernel image from two sources. One is the static CFG, acquired through statically disassembling the runtime memory dump of the kernel. The other source is the trace of indirect branches occurred during the execution of data dereference analysis. The collected indirect branches are transformed to edges and added to the static CFG.

The collection of indirect branches is realized through monitoring the exit of a dynamic code block emulated by QEMU. If the last instruction emulated is one of the following classes: 1) indirect jump, 2) conditional branch, or 3) procedure call, and the target address is not immediately encoded in the instruction, it is collected. All other cases are not counted as indirect branches. Control flow transfers caused by interrupts (emulated) or exceptions will not be

Algorithm 2 : Search for Leading Nodes.

Input:
C : A set of memory addresses (EIP) of hook candidates.
G : A kernel CFG with code-block granularity.
Output:
T : A set of leading nodes, which are not descendants of any other nodes in C respective to G.

```
ComputeLeadingNodes()
1 T ← {}
2 foreach c ∈ C
3 │ r ← True
4 │ foreach d ∈ C \ {c}
5 │ │ if c is reachable from d respect to G
6 │ │ │ r ← False
7 └ └ └ break
8 │ if r
9 └ └ T ← T ∪ {c}
```

included in the final CFG. Procedure-return instructions are not included, either.

3.3 Control Flow Graph Analysis

Recall that the analysis requires behavior sub-graph to be distinguished from the rest of kernel CFG. Unfortunately, it is a difficult problem because the subject behavior itself is often ambiguously or informally defined. For instance, the term "process creation" is a quite loose term referring to the process of loading executable image into memory, parsing the executable header, creating memory address space for the process, and transferring execution to its entry point. To determine the scope of these tasks inevitably requires human knowledge of the code structure.

To resolve this issue, a heuristic method to approximate the sub-graph is required. Note that the kernel CFG built upon the granularity of code blocks. The CFG can be simplified by merging the nodes belonging to the same function, thereby constructing a function-level CFG. According to our study, recursion is seldom used by the kernel, and this function-level CFG is nearly acyclic. Based on this observation, the behavior sub-graph is approximated with all nodes reachable from the probe candidates which are not descendants of any other candidates in the graph. Note that this approximation is neither sound nor complete since these *leading candidate nodes* are not guaranteed to be in the precise behavior sub-graph.

Algorithm 3 : Elimination of Non-Dedicated Code Blocks

Input:
$G = (V, E)$: A kernel CFG with code-block granularity.
C : A set of hook candidates. ($C \subseteq G.V$)
T : A set of leading nodes. ($T \subseteq G.V$)
Output:
D : A subset of C, excluding non-dedicated nodes.

```
EliminateNonDedicated()
1  D ← {}
2  G^R ← Reverse every edge in G
3  G' ←({v| v ∈G.V ∧ v ∉T }, {(u, v)| (u, v)∈ G.E ∧ u ∉T ∧ v ∉T })
4  foreach f ∈ C
5  |  reach_b_rev ← {v | v is a reachable node from f in G^R.}
6  |  dedicated ← True
7  |  foreach n ∈ reach_b_rev
8  |  |  if f is reachable from n respect to G'
9  |  |  |  dedicated ← False
10 |  └  └  break
11 |  if dedicated
12 └  └ D ← D ∪ {f}
```

The algorithm of selecting *leading nodes* is listed in Algorithm 2. The loop on line 2 iterates through each candidate. Lines 3-7 check if it can be reached by any other candidates. If not, it will be collected as a leading node into T. If the reachability predicate used on line 5 is simply realized with a depth-first or a breadth-first search on G, the complexity of Algorithm 2 will be in $O(|C|^2|G|)$. However, the size of G, namely the block-level kernel CFG, contains millions of nodes (i.e., code blocks). It is time-consuming to conduct the reachability analysis directly on G. Fortunately, the process can be accelerated in light of the fact that programs are organized into functions. Code blocks can be grouped into larger units (functions), and each unit has a single entrance. The reachability can be computed in two steps: first on an inter-procedural CFG and then on an intra-procedural CFG.

The elimination of non-dedicated nodes is performed as described in Algorithm 3. The algorithm takes as input the same kernel CFG G, the set C of probe candidates, and the set T of leading nodes given by algorithm 2. In the beginning, another two graphs are created from G. On line 2, G^R is created by revering every edge in G. On line 3, G' is built through removing all nodes of T and the edges attached to them from G. The algorithm guarantees that each node f in the output set D can never be reached (from any other node in G) without passing at least

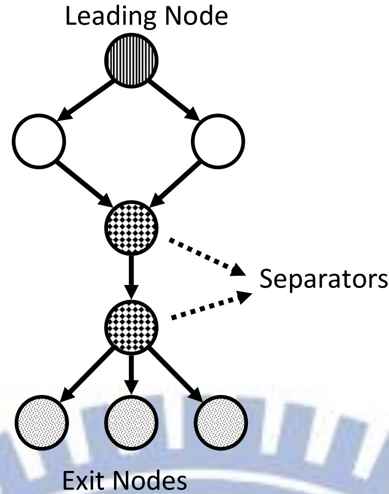


Figure 6 : An example of the output of Algorithm 3

one of the nodes of *T*. The reason is as follows. If *f* is included in D, the constraint on line 8 must have not been fulfilled in all iterations of line 7. Since line 7 loops through all ascendants of *f* (computed on line 5), and the predicate on line 8 checks if *f* can be reached from them in *G'*, which contains no node in *T*, the property holds.

The output of Algorithm 3, *D*, is a subset of *G.V*. Computing the reachability between all pairs of vertices in *D* with respect to the original *G* gives a simplified control flow graph. Namely, a new graph *F* can be generated by defining:

$$F.V := D \text{ and}$$

$$(u, v) \in F.E \text{ iff } (u, v) \in G.E, \text{ where } u \in F.V \text{ and } v \in F.V$$

Figure 6 gives an example of the output of Algorithm 3. The exit nodes are simply those vertices without any descendants. To minimize the effort of probe insertion, it is essential to find the minimal set of vertices in *F* that separates the leading nodes and exits nodes. This is a classical vertex separator problem, which is proved to be NP-Hard [46]. Fortunately, in practice *F* only consists of hundreds of vertices, and the size of the separating vertices is usually less than 3. In this dissertation brute-force is used to test if the elimination of a node will cause the separation of leading nodes and exits nodes.

With non-dedicated nodes removed, ProbeBuilder passes the output of Algorithm 3 and their dereference paths generated in Algorithm 1 to code generator to generate code snippets

```
1  int gen_probe_804e7461(unsigned char* buf, size_t len)
2  {
3    int ret = 0;
4    unsigned long ptr = kvm_register_read(vcpu, VCPU_REGS_RSP);
5
6    unsigned int offset[]= {12, 36, 52};
7    unsigned int offsets = 3;
8
9    int i;
10   gpa_t pb_gpa;
11
12   for(i = 0; i < offsets-1; ++i) {
13    pb_gpa = kvm_mmu_gva_to_gpa_system(vcpu, ptr+offset[i], NULL);
14    if((ret=kvm_read_guest(vcpu->kvm, pb_gpa, &ptr, sizeof(ptr))) != 0)
15       return ret;
16   }
17   pb_gpa = kvm_mmu_gva_to_gpa_system(vcpu, ptr+offset[i], NULL);
18   if((ret = kvm_read_guest(vcpu->kvm, pb_gpa, buf, len)) != 0)
19    return ret;
20
21   return 0;
22 }
```

Figure 7 : Example of the generated probe for KVM.

that can be inserted into the hypervisor.

3.4 Code Generator

Figure 7 gives an instance of the code snippets generated by ProbeBuilder for KVM. The corresponding probe information `<0x804e7461,ESP +12 +36 +52>`, which is shown in Table 2. The generated snippet follows the dereference path of the final output of Algorithm 3 and copies data of interests to the specified buffer. Users can directly invoke the function *gen_probe_804e7461()* in the hypervisor. As shown, line 4 initializes the pointer with the value of the ESP register of guest. The deference path is stored in the array created in lines 6 and 7. Lines 12-16 iteratively dereferences the memory with the discovered offsets. In addition, a validity check is performed in each round to avoid invalid memory access. A non-zero return value of the function *kvm_read_guest()* (provided by KVM kernel module) indicates a failure. Finally lines 17-18 copy the data of interest from the memory of guest to the specified buffer.

The generated code of probes shows that ProbeBuilder can be practically applied to the hypervisor for the VMI usage. Yet, the mechanism to trap the VM back to the hypervisor is hypervisor-dependent, and this issue is considered out of the scope of this dissertation.

Chapter 4

System Design of SWIFT

As aforementioned, ProbeBuilder provides a flexible method to mark data of interest through the taint tracking functionality. Unfortunately, existing system-wide taint trackers commonly suffer from the performance issue, and research on its improvement either rely on support from customized hardware or only deal with information flow tracking inside one process. To realize practical taint tracking, a system-wide, fast dynamic information flow tracking technique is indispensable for ProbeBuilder. SWIFT, a decoupled design for system emulation and DIFT, is therefore proposed to resolve this issue.

Before proceeding to the rest of this section, certain preliminary knowledge and terminology about dynamic binary translation and QEMU are introduced to lay down the basis of our system. QEMU is system-wide emulator, which is capable of emulating the whole operation of certain architecture on other machines. To be specific and concise, the emulated architecture will be referred to as the *guest* and the machine running the emulation as the *host* in the rest of the dissertation. Any hardware devices and mechanisms existing in the architecture of guest machine such as registers, MMU, or peripherals are realized by software. The two architectures need not to be different. In our application, both the emulated guest and the hosting machine are of IA-32 architecture. Note that such coincidence does not make binary programs inside the guest machine directly runnable. Having the instruction "`add EAX, 1`" executed inside the emulated guest machine, it is expected that the emulated EAX register, not the real one of the host machine, will be increased. Therefore, all instructions must be translated before

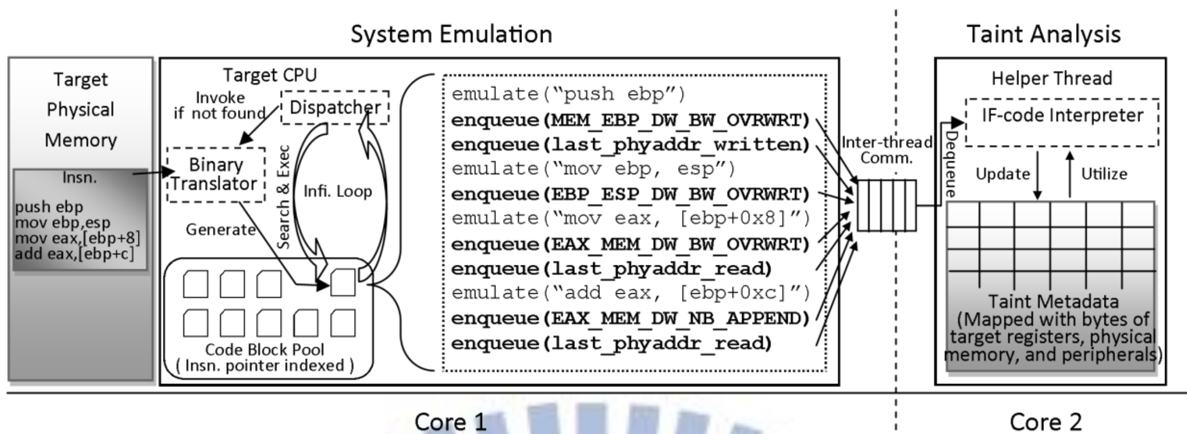


Figure 8 : An overview on the basic system architecture of SWIFT.

execution so they can reflect expected behaviors. QEMU adopts dynamic binary translation to perform the translation. In addition, the translation is done “on the fly.” Namely, the binary translator would be invoked when the emulator encounters a code region which had not been translated. The translation is done on a per-block basis. Namely, the process of translating instructions continues until a branch or jump instruction is discovered. All guest instruction sequences (the branch or jump included) translated in the process above forms a *basic block*, and the binary code generated for actual execution on the host will be the corresponding *code block*. Code blocks are stored in a hash table for next time use, since the translation is computationally expensive.

The original architecture of QEMU is included in the left part of Figure 8. On core 1, the system emulation is executed. As any hardware processor operating in a fetch-decode-execute loop, the software emulator also behaves similarly. The dispatcher always tries to search the code block pool with its instruction pointer to locate next block to execute. If the corresponding code block is found, it will be invoked. The emulation starts and then returns to the dispatcher after the code block finishes its task. Recall that all code blocks end with emulation of any jump or branch instructions. The dispatcher selects next code block to execute according to current target CPU status, and the process above repeats itself. However, if nothing is found in the search, the binary translator will be invoked to translate the basic block.

Previous DIFT acceleration research inject desired binary routines directly into code

blocks to perform specific task after examining each instruction during the translation phase. For system-wide taint analysis, these routines propagate taint status of registers and physical memory addresses which the instruction accesses. This approach, which adopted in prior works, has been demonstrated to be effective yet inefficient because the injected code usually involve with complicated computation. This encumbers system emulation in both explicit and implicit ways. The injected analysis routines could perform tasks as complicated as the emulation itself. In addition, the alternation between system emulation and analysis makes software optimization much more difficult or even disables coherent hardware acceleration such as cache mechanism.

Instead of being injected directly into basic blocks, analysis routines in SWIFT are executed by another helper thread. Additional code is injected only for delivering information flows and physical memory addresses for the helper thread to accomplish its analysis task. Decoupling the analysis from the system-wide emulation enables SWIFT to shift the analysis workload such as updating the taint map or security check onto a different core.

Figure 8 also gives the basic system architecture of SWIFT. While generating code blocks for emulation, the binary translator of SWIFT in the meantime extracts information flow semantics of instructions. With the proposed DIFT model, extracted semantics are converted to the so-called *IF-codes*, such as `MEM_EBP_DW_BW_OVRWRT` or `EBP_ESP_DW_BW_OVRWRT`, for delivery. The binary translator also injects code for delivering IF-codes to the helper thread. Note that the delivery is always done immediately after the instruction which generate the information flow is emulated. Through inter-thread communication the helper thread running on core 2 can therefore perform corresponding taint analysis or security checks.

In the following subsections, the encoding technique used to encode information flows of instructions of IA-32 is introduced. Then, two important optimization techniques are proposed to aggressively eliminate message exchange between system emulation and taint analysis.

4.1 Encoding Information Flows of Instructions

Since taint analysis is decoupled and executed on another thread, the information that should be passed to the thread to accomplish the task must be identified. Passing raw

instructions executed by the emulator, like LBA-based architecture does, could be an option. This preserves most complete information for analysis, but the helper thread will be required to decode these raw instructions by itself to perform corresponding analysis tasks. Another option is passing only data necessary for information flow tracking. This approach loses part of the information due to discarded instructions. However, it preserves the performance because the tedious decoding process can be removed from the helper thread. In SWIFT, the latter approach is chosen to meet the performance requirement.

4.1.1 IA-32 Instruction Data-Flow Modeling

As aforementioned, an IA-32 instruction frequently uses an operand as both input and output. In these cases, a single information flow will be sufficient to describe it. Since the goal is to track system-wide dynamic information flow at byte-granularity, source and destination will always refer to a single byte of registers, memory, hard disks and network interface buffers.

In addition to the source and destination, another essential factor should be included in modeling information flows. Consider a data transfer instruction such as “`mov`”. The two operands obviously serve as the source and the destination in the information flow, and the original information in the destination operand is overwritten. For a binary arithmetic operation such as addition or bitwise exclusive-or, however, one of the operands will be used as both the input and the outcome variables. For example, the `EBX` register in the instruction “`xor EBX, EAX`“ is used as input in the exclusive-or operation and also storage for the output value. Instead of being overwritten, data of `EBX` is combined with the information flowing out of `EAX`. Therefore, the two flowing *effects*, which are referred to as *overwriting* and *appending*, should be encoded also.

An information flow always originates from a certain source, denoted by *A*, and flows into a destination, denoted by *B*. For the CPU instructions accessing registers and the memory, *A* and *B* can only be register names or memory addresses. For concise expression, the overwriting information flows and appending ones from *B* to *A* are denoted as $A \leftarrow B$ and $A \overset{+}{\leftarrow} B$, respectively.

Table 3 : Typical Aggregated IA-32 Information Flow.

Category	Information Flow	Notation
Byte-wise overwriting	$A[0] \leftarrow B[0]$, $A[1] \leftarrow B[1]$, $\dots$, $A[n] \leftarrow B[n]$	$A \Leftarrow_n B$
Byte-wise appending	$A[0] \stackrel{+}{\leftarrow} B[0]$, $A[1] \stackrel{+}{\leftarrow} B[1]$, $\dots$, $A[n] \stackrel{+}{\leftarrow} B[n]$	$A \stackrel{+}{\Leftarrow}_n B$
Incrementally mixed	$A[0] \stackrel{+}{\leftarrow} B[0]$, $A[1] \stackrel{+}{\leftarrow} A[0], A[1] \stackrel{+}{\leftarrow} B[1]$, $A[2] \stackrel{+}{\leftarrow} A[1], A[2] \stackrel{+}{\leftarrow} B[2]$, $\dots$, $A[n] \stackrel{+}{\leftarrow} A[n-1], A[n] \stackrel{+}{\leftarrow} B[n]$	$A \stackrel{\Delta}{\Leftarrow}_n B$
All mixed-up	$T \leftarrow A[0], T \stackrel{+}{\leftarrow} A[1], \dots, T \stackrel{+}{\leftarrow} A[n]$, $T \stackrel{+}{\leftarrow} B[0], T \stackrel{+}{\leftarrow} B[1], \dots, T \stackrel{+}{\leftarrow} B[n]$, $A[0] \leftarrow T, A[1] \leftarrow T, \dots, A[n] \leftarrow T$ $B[0] \leftarrow T, B[1] \leftarrow T, \dots, B[n] \leftarrow T$	$A \stackrel{*}{\Leftarrow}_n B$

However, complicated information flows could be generated by a single instruction. If the encoding format encodes only the information flow of a single byte, the complication could lead to lengthy expressions. A few more observations can be leveraged to avoid this. Although our system tracks information flows at byte-granularity, IA-32 instructions always perform operations on specific widths such as byte, word (2 bytes), double-word (4 bytes), or quad-word (8 bytes). Specifying the width of operands directly in the encoding of information flows can therefore bring us a much more succinct processing.

In Table 3 are listed typical information flows from *B* to *A* when the operand width and information flow effect of an instruction are considered simultaneously. In the formula, *A* and *B* are both variables of *n*-byte, and the lowest significant byte of *A* is denoted as *A*[0]. For each category, a notation is also defined for succinct expression in later context. The information flows falling into one of these categories are referred to as *multi-byte* information flows.

In Figure 9, each category is demonstrated with a representative instruction. An edge in

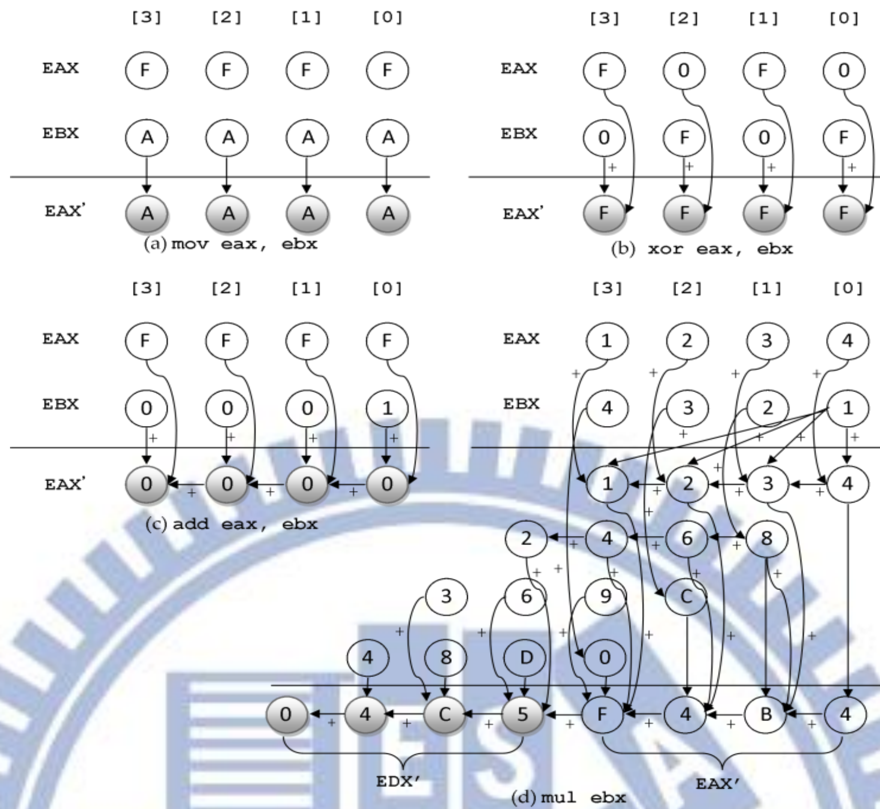


Figure 9 : Representative instructions of aggregated information flow.
(a) byte-wise overwriting, (b) byte-wise appending, (c) incrementally mixed, and (d) all mixed-up. Note that some edges are neglected in (d) for clearer depiction because they do not affect the net-effect inferred.

the figure represents the information flow from the sourcing byte to the destined byte in the calculation. One way to tell whether a variable *A* influences the other variable *B* is checking existence of a directed path from *A* to *B*. In Figure 9(a), the *byte-wise overwriting* flow caused by a `mov` instruction is shown. This flow copies data byte-wisely from the source to the destination and overwrite destination operands. In Fig. 2(b) is given an example of a *byte-wise appending* information flow, which can be caused by the `xor` instruction. It performs exclusive-or on the `EAX` and `EBX` yet uses `EAX` as the output operand. The *incrementally-mixed* flow is used to depict the situation that all bytes are influenced by those with lower or equal significance. A good example would be arithmetic addition or subtraction, in which higher bytes are influenced by all lower bytes due to the carry. The detailed flows are shown in Fig. 2(c).

Table 4 : Information Flow of Common IA-32 Instructions

Instructions	Information Flow
mov,cmovXX,push,pop,pushad,popad,movsXX, lodsXX,stosXX,inXX,outXX,lds,les,lfs,lgs, lss,fld,fst,fstp,fcmovXX	$DST \Leftarrow_n SRC$
and,or,xor	$DST \overset{+}{\Leftarrow}_n SRC$
add,adc,sub,sbb	$DST \overset{\Delta}{\Leftarrow}_n SRC$
inc,dec,neg,movzx,movsx	$DST \overset{\Delta}{\Leftarrow}_n DST$
mul,imul	$EAX \overset{\Delta}{\Leftarrow}_n MULTIPLIER$ $EDX \overset{*}{\Leftarrow}_n EAX$
div,idiv	$EDX \overset{*}{\Leftarrow}_n DIVISOR$ $EDX \overset{*}{\Leftarrow}_n EAX$ $EAX \Leftarrow_n EDX$
xchg,cmpxchg,cmpxchg8b	$\mathrm{T} \Leftarrow_n DST$ $DST \Leftarrow_n SRC$ $SRC \Leftarrow_n T$
xadd	$\mathrm{T} \Leftarrow_n DST, T \overset{\Delta}{\Leftarrow}_n SRC,$ $DST \Leftarrow_n SRC, SRC \Leftarrow_n T$
jXX	$EIP \Leftarrow_n SRC$
call	$MEM \Leftarrow_n EIP$ $EIP \Leftarrow_n SRC$
ret	$EIP \Leftarrow_n MEM$
enter	$MEM \Leftarrow_n EBP$ $EBP \Leftarrow_n ESP$
leave	$EBP \Leftarrow_n MEM$
lea	$DST \Leftarrow_n SRC1$ $DST \overset{\Delta}{\Leftarrow}_n SRC2$
(all other FPU insn.)	$DST \overset{*}{\Leftarrow}_n SRC$

The complicated information flows caused by multiplication are shown in Fig. 2(d). Note that the entire output would be 8 bytes due to the multiplication on the two double-words. As it

turns out, category (c) would suffice to describe the net information flowing effect on the four lower bytes (stored back in EAX). Note that all four higher bytes (stored in EDX) are influenced by all bytes of the two input operands. These flows are referred to as *all-mixed-up* ones. It turns out that information flow of most IA-32 instructions can be totally described with these four categories and operand widths.

Table 4 lists IA-32 instructions and their information flow encoded in the notation invented. Instruction mnemonics are listed in the first column. The second column shows their operand formats. Note that herein the Intel syntax is adopted for the explanation. Namely, the destination operand is always encoded in op1. Rows 1-6 show the encoded information flows of all the data movement and arithmetic operations. In rows 7-8 the case for data exchanging instructions are given. To correctly model their flows, an extra variable T is introduced, which does not exist in the architecture, to store the transient status. Rows 9-12 give the data flows incurred by procedure-related instructions. The lea instruction deserves a little more attention. In modern operating systems which adopt the flat memory model, this instruction simply assigns op1 with the value of op2 + op3. Since most operating systems such as Windows and Linux adopt this model, our interpretation for this instruction holds.

With all discussion above, we are now ready to encode any information flow caused by IA-32 instructions with its destination, source, width, and effect. Figure 10 shows the two encoding formats of information flows. In Figure 10(a), the format used to describe multi-byte ones is depicted. Fields *D* and *S* specify types of source and destination. *WTH* and *EFF* are used to specify the operand width n and the category in effect mentioned in Table 3. For the information flows that cannot be properly described by multi-byte rules, the format depicted in Figure 10(b) should be used, which lacks *WTH* but provides two fields *D_OFF* and *S_OFF* to specify offsets of referred bytes.

0					8	16	24	32
0	0	D	S	WTH	EFF	DST REG	SRC REG	UNUSED

D:{0:REG,1:MEM}
WTH:{00:BYTE,01:WORD,
10:DWORD,11:QWORD}

S:{0:REG,1:MEM}
EFF:{00: $\Leftarrow_n$,01: $\overset{+}{\Leftarrow}_n$,
10: $\overset{\Delta}{\Leftarrow}_n$,11= $\overset{*}{\Leftarrow}_n$}

(a)

0						8	16	24		32
0	1	D	S		E	DST REG	SRC REG	D OFF	S OFF	

D:{0:REG,1:MEM}
S:{0:REG,1:MEM}
D_OFF:{0~15}
S_OFF:{0~15}
E:{0: $\leftarrow$,1: $\overset{+}{\leftarrow}$ }

(b)

Figure 10 : Formats of IF-codes.

(a) Format for describing multi-byte information flows. (b) Format for describing information flow from a single byte to another.

For registers, DST_REG and SRC_REG store their identities. The two fields cover general-purpose registers (EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI), segment registers (CS, DS, ES, SS, FS, and GS), program counter (EIP), floating-point registers (ST0~ST7), control registers (CR0~CR4), debug register (DR0~DR7), GDTR, IDTR, TR and LDTR. MMX registers, XMM registers, MTRRs, MSRs, and any other registers are viewed as a null source and sink for information flow, that is, information flowing out of them is always labeled as clean, and information flowing into them is not tracked. In addition, instructions of MMX, SSE, SSE2, and SSE3 extension are ignored. The simplification is simply an implementation issue. Tracking information flow for the enormous instructions and registers introduced by these extensions requires huge implementation efforts but brings less effectiveness since they are seldom used. Note that 8 bits are leaved for the DST_REG and SRC_REG fields so that DIFT for these instructions and registers can always be implemented incrementally without modifying the current IF-code design. Note that these encoded information flows are simply 32-bit integers, and they could be generated when translating an instruction. These codes are referred to as *IF-codes*.

Although field *DST_REG* and *SRC_REG* seemed useless for memory operands, they keep

important information for the next special case.

4.1.2 IA-32 Indirect Memory Access

Information can be propagated in multiple ways. Information flows that have been discussed so far can be all attributed to the direct relation between input and output variables in a calculation. Namely, these inputs are directly used in the calculation to generate the output. However, for those registers used as a base or an index in an instruction using indirect addressing mode, their values indeed influence the result yet they do not take part in the calculation directly.

Indirect memory access occurs due to dereferencing pointers or accessing arrays. Tracking information flows caused by indirect memory access can generate massive unwanted false alarms. To avoid this problem, previous studies simply ignore them or limit the tracking depth on indirection. However, this could totally invalidate a DIFT system because Windows operating systems use table look-up extremely frequently in string conversion routines such as *RtlMultiByteToUnicodeN(),* and *RtlUnicodeToMultiByteN*().

After investigating these conversion functions, two facts can be observed. First, these conversions are done mostly by table look-up, which is actually an indirect memory-read operation with a register used as the index in address calculation. Secondly, these functions are used mainly for Unicode/Multibyte character set conversion. Since UTF-8 and UTF-16 are the most common implementations for Unicode characters, tables that these instructions look up are composed of 8-bit or 16-bit characters. Therefore, the indexing register is encoded in *DST_REG* or *SRC_REG* to propagate their taint status for IF-codes with BYTE or WORD as the width and $\Leftarrow_n$ as the effect.

The approach above is based on the assumption of the non-existence of UTF-32 encoding, which holds only for Windows. However, the condition is no longer valid for Linux. In Linux, UTF-32 is utilized in certain parts of the system, and the character conversion is done with a DWORD indirect memory access. Therefore, these indirect information flows cannot be easily distinguished as those in Windows. Further investigation is needed to determine how to hook

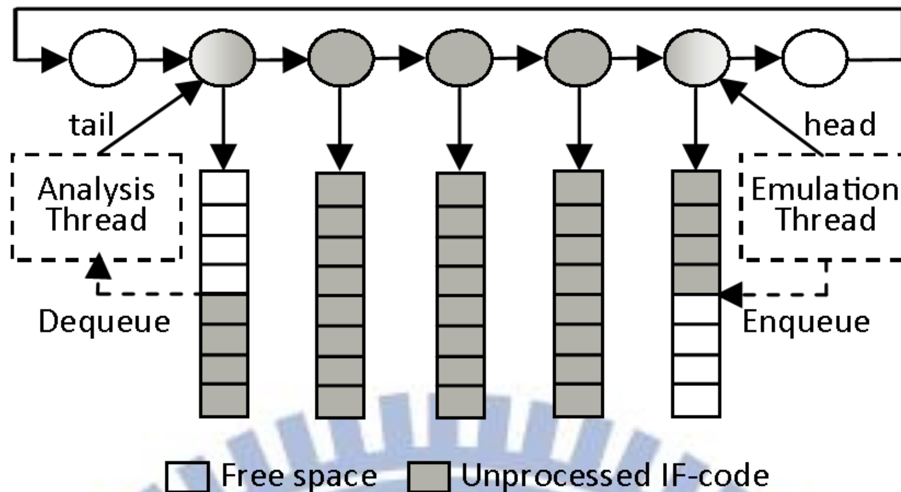


Figure 11 : Circular queue for IF-code delivering.

these conversion functions of glibc. These flows are left for the future enhancement and implementation.

4.2 Delivering IF-codes and Memory Addresses

In SWIFT, binary code to assist the helper thread, instead of analysis itself, is injected. The major task of injected binary is delivering IF-codes of each emulated instruction to the helper thread, so the helper could analyze information flows of the original program and hence the overall performance could be improved since the analysis could be processed in parallel by another core.

However, since the delivering will be done for each information flow incurred by all emulated instructions, the mechanism must be efficient enough or it could impose a new overhead on the emulation. To be light-weight, the one-way communication is achieved through a circular queue residing on a shared memory region as depicted in Figure 11. To utilize the queue more efficiently, entries in the circular queue are actually pointers to chunks of continuous space. In code blocks, en-queuing routines are injected so that IF-codes are delivered after each instruction is emulated. Next chunk will be asked for once the current chunk is full. By selecting 4 KB as the chunk size and aligning all chunks on 4 KB boundary, the en-queuing could be accomplished with following code snippet injected.

```
  mov EAX, dword ptr [LOC_enqptr]
  mov dword ptr [EAX], CONST_IFCODE
  add EAX, 0x4
  and EAX, 0xfff
  jne L1
  call nextblock
L1:
  mov dword ptr [LOC_enqptr], EAX
```

Note that `CONST_IFCODE` will be decided in the trans-lation phase.

So far we have introduced how an information flow is extracted and encoded. Nevertheless, the system-wide information flow tracking cannot be done unless addresses of memory variables are also tracked. Recall that a memory access can be indirect. In an indirect memory access, the memory address depends on the value in a certain register, and it is hence impossible to predict these addresses in advance. Therefore, watching addresses of memory operands is postponed until runtime.

Prior DIFT works based on binary instrumentation framework such as PIN or StarDBT watch virtual addresses of memory operands. This approach seems intuitive and may even be the only feasible method since the binary instrumentation tool can only monitor a user-level process. Instead, SWIFT watches physical addresses of memory operands. Since QEMU provides software MMU for system-wide emulation, watching physical addresses of accessed memory is nothing harder than watching virtual addresses. QEMU is modified so that any physical memory addresses generated by the software MMU emulation will be recorded in *last_phyaddr_written* and *last_phyaddr_read*, depending on whether operation is writing or not. To deliver a memory address just read, the following codes are injected.

```
  mov EAX, dword ptr [LOC_enqptr]
  mov EDX, dword ptr [last_phyaddr_read]
  mov dword ptr [EAX], EDX
  add EAX, 0x4
  and EAX, 0xfff
  jne L2
  call nextblock
L2:
```

```
mov dword ptr [LOC_enqptr], EAX
```

4.3 Optimization

Although approaches proposed so far successfully decouple the execution of system emulation and analysis task, the en-queuing operations still incur performance downgrade in three ways. First, extra instructions injected for en-queuing inherently introduce latency in the emulation process and the helper thread. Secondly, the massive memory accesses to the queue can consume hardware cache or causes cache misses generated by the producer-consumer communication. Thirdly, the more data en-queued, the faster the circular queue will be saturated. Therefore, reducing en-queuing operations can accelerate the emulation operation in multiple ways. Two optimizations were proposed to aggressively remove them.

4.3.1 OPT1 : Delayed-Delivering on a Per-Block Basis

Avoiding en-queuing IF-codes frequently can bring us substantial performance improvement because they form the major part of messages delivered to the helper thread. One possible optimization toward this is to deliver IF-codes on a per-block basic. While translating a basic block, the translator could group all IF-codes generated into a special entity, called the *IF-code block*. In the IF-code block, IF-codes are stored in the order as corresponding instructions are arranged. Instead of delivering IF-codes between every emulated instruction, code are only injected in the beginning of the code block to inform the helper thread which code block is being emulated so the correct IF-code block can be traced. In doing so only one en-queuing operation is needed to deliver all IF-codes for a whole code block.

However, above optimization does not reflect correct information flows always. Consider the third emulated instruction of the exampling basic block listed in Figure 8. Since the "`mov EAX, [EBP+8]`" instruction accesses a memory address indirectly, it can potentially lead to a page fault exception as long as EBP register contains an inappropriate value. To emulate such behaviors correctly, a code block must exit itself when things go wrong. Therefore, only first two IF-codes would be effective in such a case because rest instructions had not been emulated. Since an exception is unpredictable, the amount of effective IF-codes may vary between each

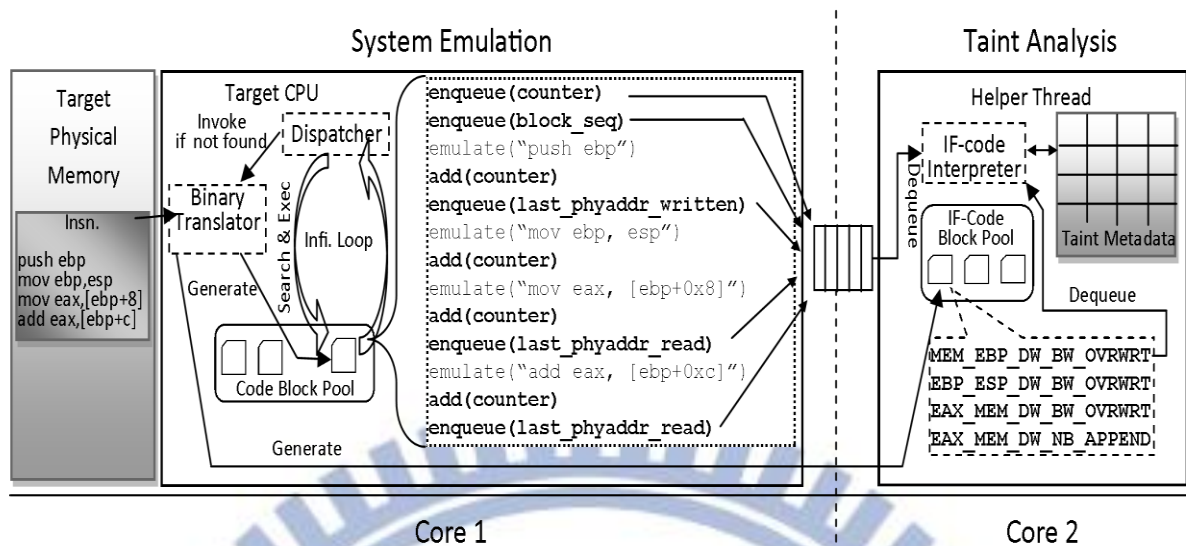


Figure 12 : An improved system design with OPT1.

Communications between the two threads are reduced since IF-codes are no longer delivered in execution of code blocks.

execution of the code block.

To amend the problem, a counter should be added to accumulate instructions emulated for each code block. Between each emulated instruction, code are injected to increase the counter so it reflects the amount of emulated instruction. The accumulation will continue until the code block exits. The value of the counter will be delivered to the helper thread in the very beginning of next code block. An example of this amendment is shown in Figure 12. In the beginning of every code block, the counter is delivered as the IF-code accumulation of the previous code block. Meanwhile, the sequence number of the current code block is also delivered. IF-codes are no longer delivered in execution of code blocks. Instead, they are only delivered once by the binary translator on a per-block basis. After that, only the counter is delivered so that the IF-code interpreter could decide how many IF-codes should be tracked for each execution of a code block. It is easy to see that communications between the two threads are reduced. In addition, the counter addition could be realized with the following instruction, which is far more concise than previous IF-code en-queuing routine.

```
inc dword ptr [LOC_counter]
```

OPT1 eliminates en-queuing operations effectively with the observation that once a basic

block is generated, its IF-codes would be fixed also. However, OPT1 cannot entirely eliminate the slow, per-code delivering mechanism. Even without exceptions, the total number of executed instructions of a code block can be still unpredictable if it contains conditional execution. The **CMOV** of the IA-32 ISA is such an instruction. When the emulator translates these instructions, it will fall back to the slow delivering mode. Fortunately, they are seldom used in ordinary programs, and hence their presences do not impede the acceleration of OPT1.

On the other hand, large amount of en-queuing operation are still needed to pass physical memory addresses. As stated earlier, these physical memory addresses can only be watched when the code block is being executed. To reduce the overhead incurred by memory address delivery, another optimization is proposed below.

4.3.2 OPT2 : Stack-based Indirect Accessing

The foundation of the second optimization relies on several phenomena observed on the frame pointer register and stack pointer register, namely EBP and ESP of the IA-32 architecture. Due to the conventional design of common compilers, this register stores the beginning address of an activation record and top of the stack in EBP and ESP respectively. In addition, their values usually change only when a new activation record is created. Referencing memory indirectly with these two registers is a common way to access local variables and function arguments. The third emulated instruction "`mov EAX, [EBP+8]`" listed in Figure 12 is a representative instance. More, in more than 90 percent of EBP or ESP-based memory accesses, their offsets distribute over the range from -1024 to +512 bytes. The clustering phenomenon is understandable since local variables and arguments usually locate near the beginning of the frame and occupy little space.

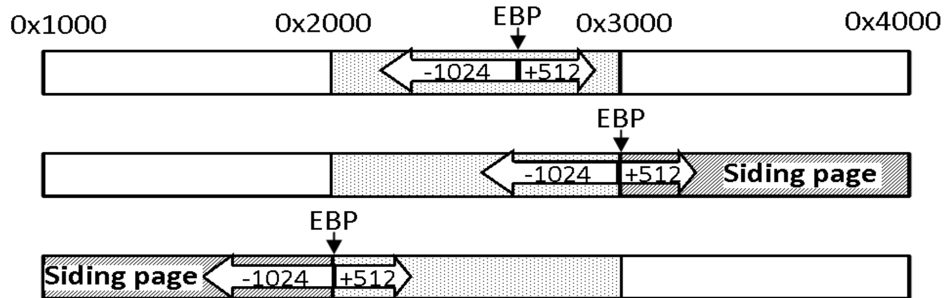


Figure 13 : Scenarios of EBP-based memory address with offset within range -1024~+512.

Algorithm 4 : Calculation of the physical address generated by EBP-based accessing.

Input:

offset : offset encoded in EBP-based accessing instruction.
ebp : current value of the register.
paddr_base : physical address of base page.

Output:

Physical address accessed by this instruction.

```
EliminateNonDedicated()
1  PAGEMASK = 0xFFFFF000
2  if (ebp & PAGEMASK) != (ebp+offset) & PAGEMASK
3  └ return paddr_siding + ((ebp+offset) & ~PAGEMASK)
4  else
5  └ return paddr_base + ((ebp+offset) & ~PAGEMASK)
```

In Figure 13 are depicted the three scenarios could occur when adding offsets within the range above to EBP. It is easy to see that two pages at most could be cross by the ranging offset. Note that although the two pages are continuous in virtual address space, their physical locations may be not due to the virtual memory mapping. For the two physical pages, the page pointed by EBP is referred to as the *base page* and the other one as the *siding page*. The physical address generated by EBP-based addressing with such offsets could be acquired using Algorithm 4. Note that the discussion above also applies on ESP-based indirect addressing.

Using the algorithm the helper thread could calculate the physical address generated by EBP (or ESP) -based addressing instructions as long as the instruction has a proper offset. Let's consider the four inputs needed for the algorithm. Since the offset is encoded in the instruction, it could be determined in translation phase and stored as part of the IF-code. This

observation saves us from en-queuing memory address for each EBP (or ESP) -based memory access. However, the benefit comes with the trade-off that the helper must possess correct *paddr_base* and *paddr_siding* for calculation. This is done by having the emulator perform virtual address translation and deliver translated addresses to the helper every time that EBP or ESP is modified. However, sometimes their values can change so frequently that the cost to perform address translation may attenuate the benefit.

To resolve the limitation, the following technique is used. Note that in most cases, both EBP and ESP point at locations inside the stack segment (if the program has one). Therefore, an out-of-box hook is implemented on the part in charge of segment allocation to acquire the physical pages mapped to pages of the stack segment. These physical pages are specially labeled so that we can identify whether EBP or ESP points at a labeled page when their values are modified. All the addresses of these physical pages are delivered to the helper thread only at the infrequent context switch or user/kernel mode switch. As a result, the helper can be informed of *paddr_base* and *paddr_siding* without demanding the emulator to perform address translation every time when EBP or ESP is modified. If EBP or ESP point at an unlabeled page, the delivering operation automatically falls back to the slower translate-then-deliver mode upon each EBP or ESP modification.

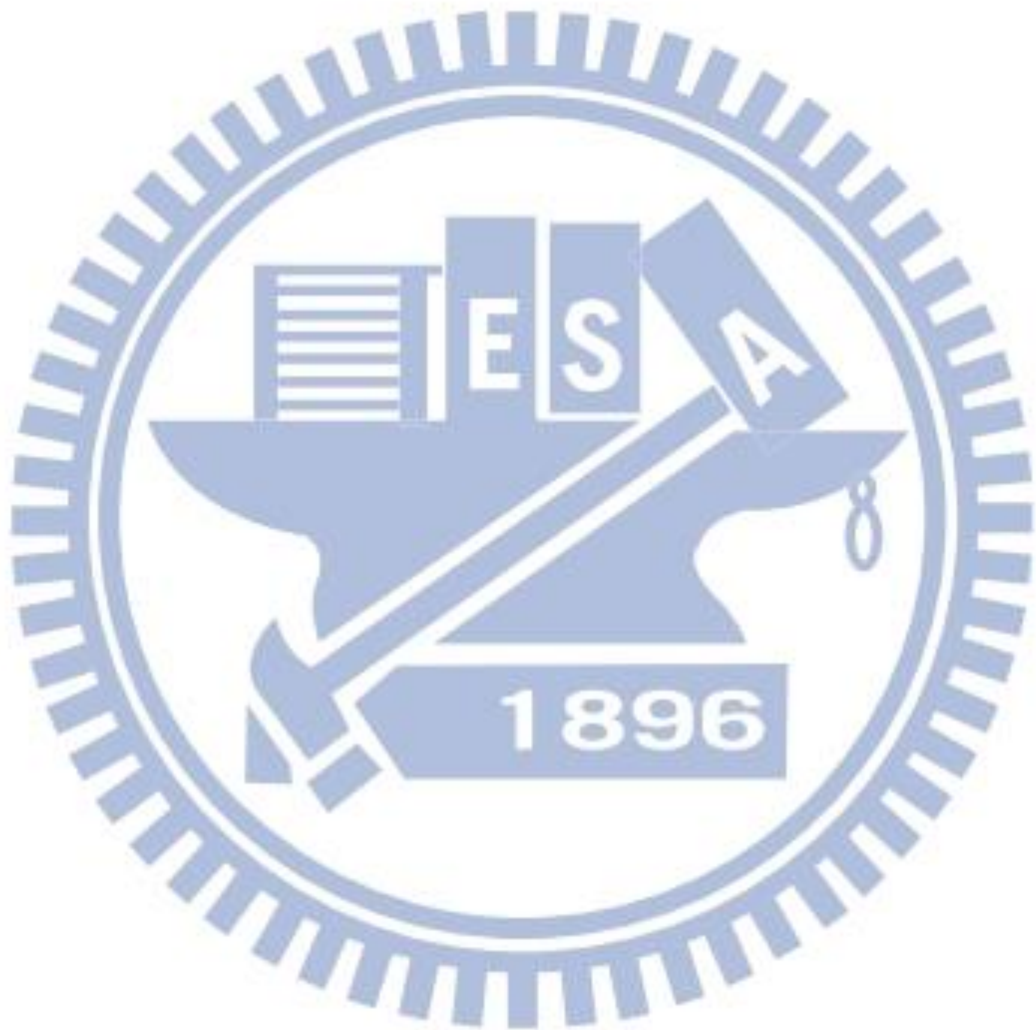
4.4 Peripherals

Tracking information flows across peripherals is a primary goal of SWIFT. For the time being SWIFT tracks information flows in hard disks and network interfaces

For hard disks, any DMA operation and port I/O between the hard disk buffer and memory are watched, and taint tags are propagated along the data movement. Their taint maps are stored hierarchically just as page directory mechanism in conventional MMU to avoid excessive memory consumption. In this way, no taint tags would be allocated to those sectors which had never been tainted.

The watching on network interfaces follows a similar pattern. Since packet exchange between the NIC and memory is usually done with DMA, only DMA operations are watched

in SWIFT implementation.



Chapter 5

Evaluation

Algorithm 1 is implemented on the QEMU emulator. The algorithm generally requires 8~12 Gigabytes of memory space to store the acquired dereference paths. Ubuntu 12.04 64-bit is chosen as the host environment and have 16 Gigabytes RAM installed so that the QEMU process can utilize as much memory as it demands. To implement Algorithm 2 and Algorithm 3, IDA Pro and IDAPython are used to parse the memory dump and construct the basic graph. Then, the graph is converted to a NetworkX graph object with edges of indirect branches added. The code generator is implemented within 127 lines of ruby scripts.

The effectiveness of ProbeBuilder is evaluated through executing the following six subject behaviors: process creation, file creation, registry creation, process termination, file deletion, and registry deletion. All these behaviors are performed with upper layer Win32 API with tainted arguments. Windows XP SP3 32-bit is installed as the guest operating system. ProbeBuilder generates probe locations for these behaviors and the corresponding data dereferences. To verify the correctness and quality of the automatically generated probes, for each behavior three probes are selected at random and implemented in another QEMU instance dedicated to behavior monitoring. In the QEMU instance, Process Monitor produced by Sysinternals and Wireshark are installed. The log trace generated by our probes is then compared with the ones generated by the above two tools. To demonstrate the strength of ProbeBuilder, a kernel-level VMI profiler is implemented using the generated probe locations and dereferences.

Table 5 : Remainder candidates after each run of Algorithm 1.

Behavior	1	2	3	4	5	6	7	8	9	10	11
Process Creation	474	311	308	303	300	295	294	281	277	276	-
File Creation	220	183	181	178	177	177	173	168	168	167	-
Registry Creation	83	70	64	62	58	-	-	-	-	-	-
Process Termination	42	38	35	-	-	-	-	-	-	-	-
File Deletion	104	100	99	99	98	-	-	-	-	-	-
Registry Deletion	86	67	67	66	62	60	55	51	-	-	-

5.1 Probe Generation

As aforementioned, Algorithm 1 will be executed repetitively to eliminate unstable probe locations. After that, graph analysis (Algorithm 2 and Algorithm 3) is used to eliminate non-dedicated code locations. To better illustrate effectiveness of the process, an entry in Table 5 shows the total number of remaining probe candidates for a specific behavior after runs of Algorithm 1. Since only identical probe locations and data dereferences are kept after each round, the total number decreases as the process continues. Note that each test is repeated 50 times to ensure the stability of sieved probe candidates. However, in all these tests the total number of probe candidates soon stabilized in less than 10 rounds. As an example, the test item for process termination takes only 3 rounds to converge. The experiment indicates two important facts. First, a large portion of candidates found in the first round are eliminated in later rounds. This shows the benefit of multi-run elimination. Secondly, the fact that all these numbers converge to stable points guarantees the existence of stable probe locations and data dereferences.

In Table 6, the analysis results of Algorithm 2 and Algorithm 3 are listed. The set of the

Table 6 : Remainder candidates after Algorithm 2 and Algorithm 3.

	Process Creation	File Creation	Registry Creation	Process Term.	File Deletion	Registry Deletion
Input	276	167	58	35	98	51
Top Nodes	32	20	6	4	27	12
Dedicated (Output)	176	88	14	22	58	25
Non-dedicated	100	79	44	13	40	26

probe candidates discovered by Algorithm 1 is used as an input variable *C* to Algorithm 2 and Algorithm 3, that is, the numbers listed in the first row of Table 6 are identical to the final stable numbers in Table 5. The second row in Table 6 gives the total number of leading nodes generated by Algorithm 2. The third row gives the total number of dedicated nodes as the final output of ProbeBuilder. As shown, a large portion of candidates are again eliminated by Algorithm 3. The test on registry creation filtered 44 non-dedicated probe candidates, leaving 14 candidates as the final answer.

To understand the effectiveness of the refinement phase, the eliminated probe candidates are inspected. However, the massive amount of code and its assembly form make manual examination extremely difficult. To simplify the task, the non-dedicated probe candidates (code blocks) are mapped back to the owner functions with IDA Pro. Functions with human-readable names recognized by IDA Pro and WinDbg are then collected. The functionality of these "named" functions are manually checked on MSDN, looking for those dedicated to the subject behavior. Subroutines called by these non-dedicated functions are also considered as non-dedicated.

The results of examination are shown in Table 7. For each subject behavior, a certain amount of functions are manually discovered to be non-dedicated. Their names are listed in the first column. Among the non-dedicated functions recognized by Algorithm 2 and Algorithm 3, the proportion of these manually verified functions *p* is listed in the second column. For instance, the test on file creation shows that 80% of discovered non-dedicated probe candidates are manually verified. Tests on behaviors like registry deletion and registry creation give low proportion of the successfully verified functions. However, the test merely investigates the

Table 7 : Eliminated non-dedicated functions.

	Names	*p*%
Process Creation	ExAcquireSharedWaitForExclusive, RtlRandom, RtlCopyUnicodeString, SePrivilegeCheck, FsRtlDoesNameContainWildCards, RtlEqualUnicodeString, ObfDereferenceObject, SeReleaseSecurityDescriptor, ObGetObjectSecurity, SeDeleteAccessState, ObOpenObjectByName, RtlLengthRequiredSid, RtlUpcaseUnicodeChar	32%
File Creation	CcUnpinData, RtlSplay, CcPinRead, CcRemapBcb, SeDeassignSecurity, CcPinMappedData RtlLookupElementGenericTableFullAvl, CcMapData, RtlCopyUnicodeString, ObOpenObjectByName, RtlAreBitsClear, RtlInsertElementGenericTableFullAvl	80%
Registry Creation	ExDisableResourceBoostLite, SeDeassignSecurity, CcPinMappedData, IoSetThreadHardErrorMode, SeAssignSecurity, RtlUpcaseUnicodeChar	27%
Process Termination	MmMapViewOfSection	77%
File Deletion	IoSetShareAccess, ObOpenObjectByName, IoIsOperationSynchronous, RtlCopyUnicodeString	63%
Registry Deletion	RtlCreateSecurityDescriptor, ExDisableResourceBoostLite, CcPinMappedData, SeQuerySecurityDescriptorInfo	13%

documented functions, which can be recognized by IDA Pro and WinDbg, and hence the number may be underestimated. Considering this fact, it is reasonable to deduce that this test has successfully verified the effectiveness of Algorithm 2 and Algorithm 3.

In Table 8, a few concrete examples of generated probes and their data dereferences are shown. During the execution of Algorithm 1, the first 512 bytes of the data captured by the taint-based predicate P are captured. However, for conciseness only the part before the terminating null character is listed. The first and the second columns give the probe locations and the corresponding dereference data paths, respectively. As expected, the large offsets in

Table 8 : Examples of probes, data dereferences, and data collected by Algorithm 1.

EIP	Dereference Path	Captured Data
Process Creation		
0x804d9050	ESP, +20, +0	T_A_R_G_E_T.exe\0
0x804e447f	EAX, +184, +16, +0	T\0_\0A\0_\0R\0_\0G\0_\0E\0_\0T\0.\0e\0x\0e\0\0\0
0x804efe53	ESP, +12, +160, +124, +6	C\0:\0\\0T\0_\0A\0_\0R\0_\0G\0_\0E\0_\0T\0.\0e\0x\0e\0\0\0
File Creation		
0x804e875a	ESI, +100, +52, +0	T\0_\0A\0_\0R\0_\0G\0_\0E\0_\0T\0.\0t\0x\0t\0\0\0
0x80577af2	ESP, +24, +20, +108, +60, +0	T_A_R_G_E_T.txt\0
Registry Creation		
0x804e8a61	ESP, +0, +0, +96, +60, +0	T\0_\0A\0_\0R\0_\0G\0_\0E\0_\0T\0.\0e\0x\0e\0\0\0
0x804edb22	ESP, +24, +32, +28, +60, +0	H\0K\0E\0Y\0_\0L\0O\0C\0A\0L\0_\0M\0A\0C\0H\0I\0N\0E\0\\0S\0O\0F\0T\0W\0A\0R\0E\0\\0M\0i\0c\0r\0o\0s\0o\0f\0t\0\\0W\0i\0n\0d\0o\0w\0s\0\\0C\0u\0r\0r\0e\0n\0t\0V\0e\0r\0s\0i\0o\0n\0\\0R\0u\0n\0\0\0
Process Termination		
0x804e917d	ESP, +20, +12, +0	T_A_R_G_E_T.exe\0
File Deletion		
0x804e875a	ESP, +4, +100, +52, +0	T\0_\0A\0_\0R\0_\0G\0_\0E\0_\0T\0.\0t\0x\0t\0\0\0
0x804e883c	EDI, +28, +148, +60, +6	C\0:\0\\0T\0_\0A\0_\0R\0_\0G\0_\0E\0_\0T\0.\0t\0x\0t\0\0\0
Registry Deletion		
0x80563db2	ESP, +8, +200, +0	T\0_\0A\0_\0R\0_\0G\0_\0E\0_\0T\0.\0e\0x\0e\0\0\0

dereference paths show the existence of huge data structures in Windows kernel. The given SRCH_WIDTH: [512, 256, 128, 64, 512] may not be enough to cover all possible dereferences. Nevertheless, it is not necessary to identify all of them in this application.

Note that our dummy program invokes the ANSI-string-based API. Yet, the corresponding

Unicode strings encoded by the operating system are also identified. In addition, prefixed Unicode strings are captured as well in process creation and file deletion. The results demonstrate the use of the taint-based predicate.

5.2 Effectiveness of Generated Probes

To verify effectiveness of the generated probes and dereferences, two experiments are performed.

5.2.1 Monitoring User-Space Activities

Since all the probes generated by ProbeBuilder locate in the OS kernel, they should produce profiles at least as complete as any user-level monitors. To verify this, for each behavior 3, probes are randomly selected out of the output of ProbeBuilder, and are manually implemented in another QEMU instance (without the functionality of ProbeBuilder). In that guest machine, the same OS image is installed. Meanwhile, Sysinternal Process Monitor v3.04 and API Monitor v2.0 from Rohitab are also installed. The system is then manually exercised for 30 minutes, producing more than one million activities recorded by the two commercial applications. The API trace logged by the probes of ProbeBuilder was compared with theirs.

The comparison shows that all the occurrences of these six behaviors reported by Process Monitor and API Monitor are also logged by the probes of ProbeBuilder. The API arguments are also correctly captured by the generated data dereferences. We discover that the probes of ProbeBuilder recorded more activities than the two user-space profiling tools, especially for the file creation behavior. A large portion of these extra records are confirmed as expected to be kernel activities since the probes reside in the kernel space. However, there still exist 0.21% of the total recorded activities with meaningless binary data which are considered as false positives.

5.2.2 Monitoring Kernel-Space Activities

The kernel-level probe shows its effectiveness against kernel activities. For evaluation, a kernel module is implemented to simulate a pure kernel-level Trojan. The following tasks are

performed in sequence: 1) Establish a TCP connection with a HTTP server controlled by us. 2) Create a dummy file `ProbeBuilderTest.txt` at `C:\` 3) Create a registry key `ProbeBuilderRegistryInjection` in the start-up program entries. Note that these tasks are executed purely in the kernel-space. This module is packed within a leading program in which the attached module is registered as a system service. The program is then profiled with the same QEMU instance used in the previous subsection. It is also uploaded to ThreatExpert and Anubis for comparison.

The result shows that only our profiling tool successfully captures all the three activities. ThreatExpert only identified the created registry key, and Anubis only logged the TCP connection. (The user-space activities of the leading program are captured by all three platforms) Since both ThreadExpert and Anubis captured at least one of the three kernel-level behaviors, the success execution of the kernel module is confirmed. This experiment not only demonstrates the effectiveness of ProbeBuilder but also the necessity of kernel-level probes.

Please note that this result does not imply that the probes generated by ProbeBuilder are more effective than those in existing VMI-based systems. Given sufficient time, any experienced analysts can discover probe locations and data dereferences through reverse-engineering. The contribution of ProbeBuilder is automating these procedures in an effective way.

5.3 Performance

ProbeBuilder utilizes emulation to monitor the system state before each code block. Under the taint checking mode, additional taint analysis must be performed for each executed instruction. The data dereference analysis (Algorithm 1) runs about 30 times slower than the native machine. It takes 1~3 minutes to complete an upper-layer API invocation. The time required by the control flow analysis (Algorithm 2 and Algorithm 3) heavily depends on the size of the probe candidates discovered by Algorithm 1, varying from 6 minutes (for process termination) to 167 minutes (for process creation). However, note that ProbeBuilder is designed to reduce the effort of manually building VMI tools. Compared with the enormous effort

generally required by manually reverse-engineer a closed-source kernel image, the execution time needed by ProbeBuilder is trivial. In addition, the probes and the data dereferences generated by ProbeBuilder should be transferred to systems implemented with faster emulation, virtualization, or even native machines, not directly on ProbeBuilder itself. Consequently, the schemes proposed in this dissertation do not impose overhead on the final application system.

On the other hand, the performance of SWIFT should be carefully evaluated since its contributions focus on the performance boost led by the optimizations proposed in this dissertation. To evaluate performance improvement attributed to techniques proposed in this study, both commercial test suites are used and common workloads to acquire benchmark scores for following configurations.

(a) Native QEMU

(b) SWIFT (decoupled design)

(c) SWIFT w/ only OPT1 enabled

(d) SWIFT w/ both OPT1 and OPT2 enabled

(e) QEMU with inline taint propagation

(f) TEMU (Based on QEMU Ver. 0.9.1)

To set up a baseline for our evaluations, a native version of QEMU, which our system base on, is tested in configuration (a). Note that neither KQEMU nor KVM was activated because we want to benchmark the performance of the pure emulation. In (b), solely the decoupling mechanism is enabled so that its performance advantage could be measured. Configuration (c) and (d) operate with the decoupled design as well as (b), yet OPT1 and OPT2 are enabled respectively. To understand how much performance gain could be achieved with the proposed schemes, we also set up a configuration (e), which inlines taint propagation routines of SWIFT directly in code blocks generated by original QEMU. The comparison also included TEMU, a well-known system-wide taint analysis system, as configuration (f) in this benchmark evaluation.

A little more explanation is needed to elaborate the goal of this evaluation. For configuration (b), (c), (d), and (e) all data in guest memory or received from network are labeled

as tainted. Although this leads to considerable memory usage since the companied shadow memory becomes as large as the allocated RAM size of the guest machine, it is necessary for measuring the performance gain under the worst case. Moreover, it is difficult to fairly compare TEMU with our scheme directly. TEMU is designed with extremely high flexibility, and it thus contains large taint record for each byte and many callbacks for additional plug-ins. These features inherently incur severe overhead on performance of TEMU. However, it is also difficult to port its code into SWIFT for direct comparison because TEMU is based on an older version of QEMU. Due to reasons above, only the taint propagation of TEMU is activated and remove any other plug-ins in configuration (f). In addition, no taint data is introduced in configuration (f) among all experiments.

All the evaluations are performed on an IBM System x3650, with one unit of Intel Xeon E5430 2.66 GHz Quad-Core Processor, 8GB DDR2 RAM, and a 150GB SATA-II hard disk installed. In each configuration one identical virtual machine snapshot is loaded into the emulator to ascertain fairness. The virtual machine is allocated with 512 MB RAM and a 10GB hard disk. Windows XP with service pack 3 is installed and booted in the snapshot. In addition, 512 MB are allocated for the IF-code delivering circular queue.

To perform information flow tracking SWIFT consumes more memory than the original emulator does. First of all, an extra 512 MB space was allocated to construct the circular queue for IF-code delivering. In addition, each byte in the guest memory is augmented with an extra shadow byte to preserve its taint status. Since in our evaluation every byte in memory is labeled as tainted, the shadow memory occupies the same size as the physical memory size of the guest. To enable OPT1, a shared 128MB memory region is pre-allocated to store IF-code blocks generated in the translation phase. We also force the emulator to flush all the code blocks when this region is full. Therefore, all these extra memory usage can be statically calculated to be 512+512+128=1152MB.

The first result of performance evaluation is acquired with PassMark Performance Test 6.0, which is an off-the-shelf commercial test suites adopted extensively in CPU and system benchmarking. Benchmark items could be categorized into CPU-intense jobs and memory-

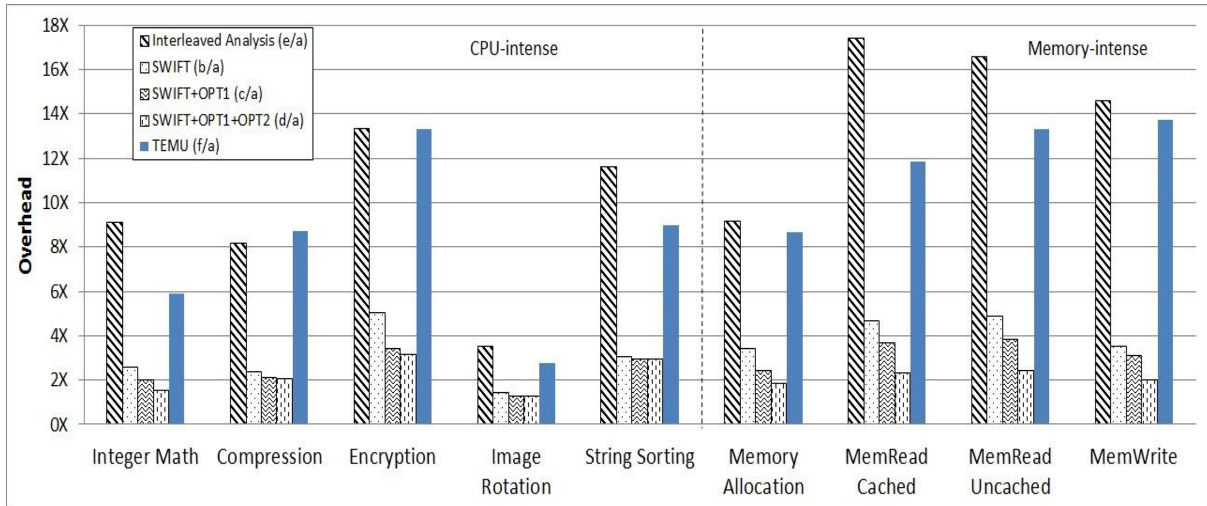


Figure 14 : Overhead imposed by different configurations.

To present overhead imposed by each configuration more clearly, benchmark scores of configuration (b), (c), (d), and (e) are divided by the score of baseline configuration (a).

intense ones. To present overhead imposed by each configuration more clearly, benchmark scores of configuration (b), (c), (d), (e), and (f) are divided by the score of baseline configuration (a). As indicated in Figure 14, although the design with decoupled DIFT still imposes high overhead (1.43X ~ 5.00X) among all test items, it already outperforms configuration (e) significantly. When OPT1 and OPT2 are enabled, the overall performance downgrade can be reduced to 1.28X~3.16X, which are 2.74X~7.48X times faster than the interleaved design (f). The result demonstrates effectiveness of optimizations proposed in this dissertation. In addition, close scores between (e) and (f) give us faith on the representativeness of configuration (f).

There is an interesting fact presented in Figure 14. First of all, memory-intense benchmark items benefit a lot from OPT2, but no significant improvement is shown on CPU-intense ones. After analyzing instruction traces of those experiments, it is discovered that EBP-based memory accesses in the benchmark program occur less frequently than expected. In such cases, the overhead of delivering memory addresses cannot be effectively removed by the optimization.

Next, same configurations with common workload such as file transferring or source code compiling are benchmarked to further investigate the analysis overhead in real applications. Details of these workloads are explained below. All measurements are repeated certain times and average values are calculated. The number of repetition of each item is listed after the name

of the workload.

System Booting (50)

The time needed for booting Windows XP is measured. More precisely, the time elapsed from powering on the emulator until Windows loads Graphical Identification and Authentication, GINA, which brings up the Windows Security dialog for users to log on, is measured. It is chosen as the termination of the measurement because its loading represents that the booting-up sequence has come to an end.

Web Browsing (50)

Since web-browsing is an extremely frequent user behavior and a common way to get attacked by malware, this item is included in benchmark to investigate how our implementation can affect the browsing speed. The experiment is carried out by measuring the time needed for sequentially browsing top 50 websites, which are ranked by Alexa Internet, an authoritative Internet information provider. The sequential browsing mechanism is implemented with a Firefox plugin, which automatically visits next website once it receives an event of page loading complete.

Communication over SCP (20)

In this benchmark a large file is downloaded into the emulator through Secure Copy, a file transfer mechanism based on SSH protocol to provide confidentiality and authentication. The file consists of 120 MB random binary sequence and resides on a host locating in the same 100BASE-TX local area network. The benchmark is performed with Putty SCP, which is a Win32 implementation of the protocol. The time needed to accomplish the following command is measured.

```
pscp -l dummy_user -pw dummy_password \
192.168.0.254:dummy.dat ./
```

Kernel Compiling (5)

Compiling the kernel of an operating system is a common workload used to benchmark the overall performance of a computer. To be consistent with previous benchmark items, which all target on Windows environment, this experiment is carried out by building the Windows

Table 9 : Time needed to accomplish common workloads for each configuration.

	a	b (b/a)	c (c/a)	d (d/a)	e (e/a)	f (f/a)
System Booting (50)	27.7	62.1 (2.24X)	58.4 (2.11X)	53.4 (1.93X)	97.2 (3.51X)	195.0 (7.38X)
Web Browsing (50)	536.1	945.1 (1.76X)	730.6 (1.36X)	632.7 (1.18X)	1554.2 (2.90X)	1754.3 (3.34X)
SCP Communication(20)	164.6	386.5 (2.35X)	227.5 (1.38X)	214.6 (1.30X)	688.0 (4.18X)	722.8 (4.08X)
Kernel Compiling (5)	1364.7	3605.9 (2.64X)	2837.7 (2.08X)	2757.1 (2.02X)	5751.2 (4.21X)	14653.2 (12.83X)

Unit: sec

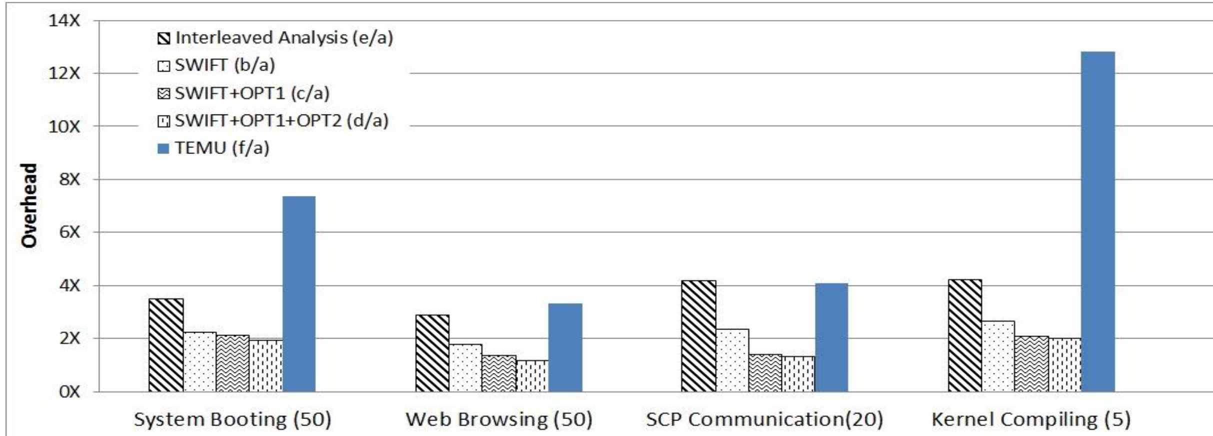


Figure 15 : The common workload overhead comparison between each configuration.

Research Kernel in the emulated machine.

Table 9 shows the time needed by each configuration to accomplish common workload. In addition, except for configuration (a) the overhead is calculated by dividing the time needed by the configuration by the time needed by configuration (a). To illustrate the overhead more clearly a comparison chart is given in Figure 15. As shown, configurations with analysis decoupled showed enormous performance advantage over those with inline analysis routines. In addition, effectiveness of the two optimizations are also demonstrated in configuration (c) and (d). The result shows that our system remains 50%~85% performance of a native emulator when both optimizations are enabled. Moreover, compared with configuration (e), a greater than 2X performance advantage is given by configuration (e) in nearly all commercial benchmark items and workload tests. The observation justifies that the investment of utilizing an addition CPU core is paid off.

To show the overhead imposed on the native machine by the system-wide emulation and the decoupled design, we also benchmarked the native performance. In Table 10, the comparison of native machine and SWIFT+OPT1+OPT2 is listed. The values shown in the

Table 10 : Comparison between SWIFT and Native Execution.

Benchmark	Unit	Native	(d)
CPU-intensive			
Integer Math	MOps./s	523.4	24.1(21.72X)
Compression	KBytes/s	3128.2	103.0(30.37X)
Encryption	Mbytes/s	13.5	0.4(33.75X)
Img Rotation	Images/s	123.5	9.7(12.73X)
String Sorting	1000 strings/s	1701.4	64.3(26.46X)
Memory-intensive			
Mem Alloc	Mbytes/s	2118.4	75.8(27.94X)
MemRead Cached	Mbytes/s	1371.2	66.5(20.62X)
MemRead Uncached	Mbytes/s	1340.2	62.2(21.55X)
MemWrite	Mbytes/s	1477.1	87.2(16.94X)
Common Workload			
System Booting	Sec.	10.3	53.4(5.18X)
Web Browsing	Sec.	231.7	632.7(2.73X)
SCP Comm.	Sec.	76.2	214.6(2.82X)
Kernel Compiling	Sec.	343.1	2757.1(8.03X)

group of CPU-intensive and memory-intensive tasks are the scores given by the PassMark benchmark suite. A higher value indicates a better performance. In the common workload group, the time needed to accomplish that task is listed. As indicated in the table, even if both optimizations were used, the system still suffers from huge performance penalty. The PassMark benchmark shows that the overhead can range from 12.74X up to 35.55X. However, in common workload the system generates lower overhead than it does in Passmark benchmark. The overhead ranges from 2.73X to 8.03X. The phenomenon is understandable because these common workloads do not demand CPU and memory as much as Passmark does. Instead, more time is spent on waiting for the I/O operations involved in network behaviors or file system accesses.

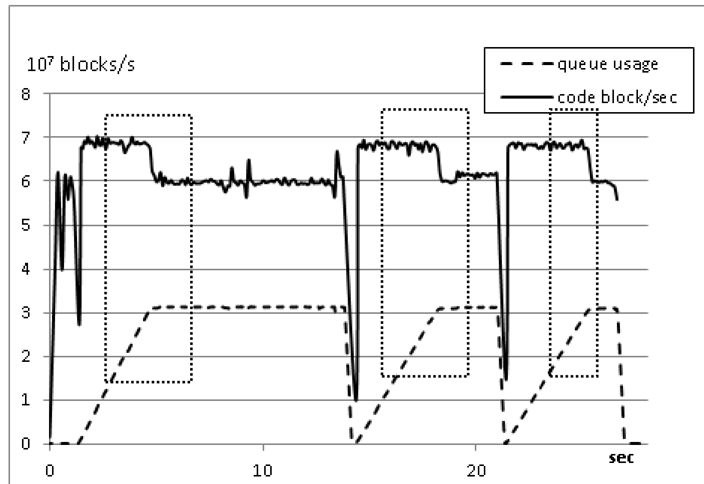
Obviously, the performance penalty is non-negligible even if the proposed decoupling and optimization techniques are used. The performance degradation can be attributed to the inherent emulation characteristics. However, by comparing configurations (a) and (d) we understand that there is still overhead imposed by DIFT on pure emulation. One direct cause of the overhead is the routines injected in code blocks. Although OPT1 and OPT2 aggressively eliminate code injections, there are still plenty of them. We also suspect that the overhead is

Table 11 : Performance of QEMU, Dummy Analysis, and SWIFT.

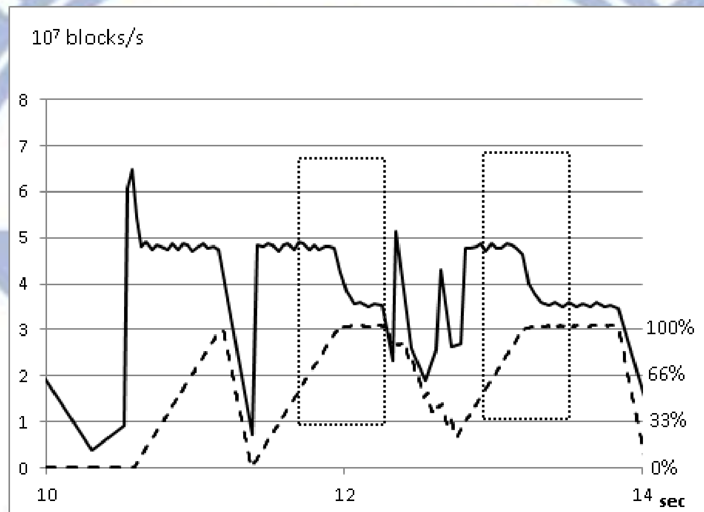
Benchmark	Unit	(a)	Dummy	(d)
CPU-intensive				
Integer Math	MOps./s	37.3	31.2	24.1
Compression	KBytes/s	213.2	168.2	103.0
Encryption	Mbytes/s	1.2	0.7	0.4
Img Rotation	Images/s	12.4	11.6	9.7
String Sorting	1000 strings/s	189.0	127.8	64.3
Memory-intensive				
Mem Alloc	Mbytes/s	140.3	113.4	75.8
MemRead Cached	Mbytes/s	155.0	124.2	66.5
MemRead Uncached	Mbytes/s	150.5	108.6	62.2
MemWrite	Mbytes/s	171.7	130.9	87.2

also contributed by a saturated IF-code delivering queue. In current implementation, the analysis thread has not been optimized. Therefore, the speed of the analysis thread, which consumes IF-codes, cannot keep up with the emulator. Once the queue is saturated, the emulator must wait for the analysis thread. To investigate whether it is the code injection or the saturation degrades the performance, we implemented a dummy analysis thread, which performs nothing but solely consumes received IF-codes. With the dummy analysis thread, the queue is never saturated, and we can measure the overhead incurred solely by the code injected in the emulator. The result is shown in Table 11. As shown in the table, the fully-operational taint tracking thread causes substantially larger overhead than the dummy helper thread does. The observation indicates that, in current implementation, the overhead imposed by DIFT on pure emulation is mainly contributed by a slow taint tracker.

To verify our conjecture, we set up an experiment with two additional profilers in SWIFT. One is installed in the emulation thread to measure how many code blocks are emulated per second. The other is installed in the DIFT thread to measure the distance between the en-queuing pointer and the de-queuing pointer periodically. The distance indicates the usage of the message-delivering queue. By plotting the two kinds of measurements versus the system clock time on the same graph, we can study the correlation between the emulation speed and the queue usage. It is worth noting that in our experiment we allocated a 512 MB queue. This



(a)



(b)

Figure 16 : Plots of emulation speed versus time and queue usage versus time. Plot (a) and (b) are acquired in the process of running Passmask CPU integer operation and encryption benchmark respectively. The patterns circled by dashed boxes indicate that the emulation performance drop significantly when the queue becomes saturated. Namely, the emulation thread will still be cumbered by the taint tracking thread eventually.

assures that the saturation (if any) can be observed even if an impractically large queue is given.

The result is shown in Figure 16. Figure 16 contains two plots, which are acquired in the process of running Passmask CPU integer operation and encryption benchmark, respectively. In each plot, the emulation speed versus time graph and queue usage versus time graph are plotted. The patterns circled by dashed boxes indicate that the emulation performance drop when the queue becomes saturated. Namely, the emulation thread will still be cumbered by the

DIFT thread eventually. Namely, the emulation thread will still be cumbered by the DIFT thread eventually. The data indicates an interesting fact worth noting. In both plots, the drop is not negligible. Namely, the current taint tracking implementation in SWIFT is not able to keep up with the emulation part, and it indeed eventually incur significant overhead on long CPU-bound tasks.

The finding above gives us a direction to the possible improvement. Due to the implementation effort, in the current implementation of SWIFT the taint tracking thread adopts a threaded dispatching way to interpret received IF-code blocks. Although the IF-codes are already stored in a very concise format, certain decoding steps are still required to inter-pret. To eliminate overhead incurred by the decoding step, it is possible to choose dynamic binary translation over interpretation. Namely, we can enhance the IF-code block generation module so that it produces not only IF-code blocks but also native binary codes to track taints. With the help of OPT1, it would be trivial for the DIFT thread to verify whether a code block is emulated without unexpected exceptions and to execute the corresponding taint tracking binary codes. The idea above is totally feasible, but it needs further implementation and evaluation. We leave it as future work.

5.4 Malware Analysis with SWIFT

Although SWIFT is originally designed for accelerate the taint tracking predicate *P* of ProbeBuilder, it also provides practical use for malware behavior analysis. In this section, we introduce behavior profiling functionalities and their designs.

In the emulated IA-32 environment, Professional edition of Windows XP with SP3 is installed as the guest operating system. The emulator is powered on with a snapshot of the guest system already running so the booting sequence of the guest OS is skipped. The analysis process starts with having the file to examine imported into the file system in the emulated machine. While being imported, each byte written into the emulated hard disks will be labeled as tainted. Having all sectors occupied by the target tainted, we execute the target so that its effects on the whole operating system can therefore be revealed. Along the emulation, IF-codes will be

generated by the emulator and delivered continuously to the analysis thread as previously stated. In our implementation, we give the system only one minute for execution and then freeze it to finish analysis.

Code Injection/Unpacking/Kernel-Level Execution

Identifying instructions residing in tainted memory regions can reveal various suspicious behaviors such as code injection or unpacking. Whenever such an instruction is discovered, we could check the value of CR3 to determine in which process the instruction is executed. If it resides in processes other than the executable being analyzed, code injection is detected. Otherwise, it may be an unpacking behavior. In addition, we can also check the CPU privilege to detect execution in kernel space.

The most intuitive idea toward this is to check whether an instruction fetched by the emulated processor locates at a tainted memory location. However, the idea above requires that the emulator can always access up-to-date taint tags before fetching instructions. It is obviously infeasible because taint tags may have not been updated due to the un-synchronized cooperative pattern between the emulator and the analysis thread. To solve this problem without introducing synchronization issues, the security check should be placed in the helper so that the status of taint tags is assured to be up-to-date.

Here we state our solution toward this problem. Recall that when OPT1 is enabled the emulator generates IF-code blocks while translating instructions. In the IF-code block, we store not only IF-codes but also the physical address of the first instruction and the value of CR3 register in that basic block. In this way, the helper could always detect dirty code execution by checking whether that physical address has been tainted every time it accesses an IF-code block. And the recorded CR3 value can help us determine in which process the instruction resides. Note that we taint data written by dirty code with a different taint color so that unpacked code can be distinguished from the original program loaded from disks.

Outgoing Traffic

Many malware perform network activities. However, those traffics which are not induced by the analyzed item can be annoying and distracting for analysts. These “noise traffics”, such

as NetBEUI messages broadcasted periodically by Windows, are usually generated spontaneously by the operating system itself. With taint analysis, our system can filter out traffics which are irrelevant to the target being examined and generate more concise and accurate report. To capture tainted packets we check each DMA operation which copies data to the TX-buffer of the emulated NIC as stated previously. To display content of the traffic, the first 256 bytes of each packet is recorded so that header information and part of the payload can be traced later.

Dirty Code Execution

One interesting problem in taint analysis is how to propagate taint tags when the executed instruction locates in a tainted memory region. Such cases indicate that potentially problematic codes are about to gain full control of the emulated environment, and therefore extra care should be taken when we propagate taint tags because crafted programming techniques could be used to evade information flow tracking.

In our implementation we took an aggressive propagation policy for dirty code execution by tainting every memory cell written by instruction residing in tainted memory regions. This approach assures that all memory modifications done directly by those dirty codes are tracked by dirty taint tags. We set this rule with the highest priority in taint propagation to resist taint laundering. We do not claim that this approach disables all circumventions because dirty codes may still be able to modify memory without being tracked by taking advantage of other clean code blocks already existing in the system such as external libraries.

The second kind of analysis is done at the end of the analysis process instead. They are postponed due to following reasons. First, these analysis items cost so much time that activating them on-the-fly could slow down the analysis thread dramatically. In addition, these items are used to identify persistent behaviors such as file modification. Therefore, postponing these analysis items until the end of the analysis cannot jeopardize effectiveness of our system. We show such items below.

File Creation/Modification

The most representative behavior of malicious persistence is modification on the file

system. To sustain their survival after rebooting process, malware always need to implant themselves into non-removable storage such as hard disks. Therefore, showing these modifications is essential for malware analysis. However, to present the result of taint analysis in a more readable way, tainted sectors in hard disks are reversely mapped to file objects which possess these sectors. The mapping could be done through parsing the metadata and the file allocation table of the file system. We realize the translation by integrating The Sleuth Kit (TSK), which is a popular open source toolset for disk forensics, with our system. With the functionality provided by TSK, files possessing tainted sectors can be easily inferred in the end of analysis.

Registry Creation/Modification

Another important behavior should be carefully profiled is modification on the registry. The registry is a database which stores configurations for Windows itself or various applications. Resembling files and directories in a file system, these data are well-organized in a hierarchical way. An entry stored in registry has its key, value, and path. The key and the value of an entry in registry can be viewed as the name and the content of a file in file system, and its path can be therefore conceived intuitively. Therefore, most malware create new items or modify existing settings at certain paths in the registry database, such as start-up application list or file extension associations. Monitoring modification on the registry is no less important than file system monitoring because locations modified actually give the rich semantics about the intention of the target.

Most registry operation profiling tools do their jobs by hooking specific functions in Windows API or kernel since registry database are usually accessed through them. It is also possible to follow a similar design so that taint-based checks could be performed when these APIs or functions are invoked. However, we do not want to implement this feature with any hooking techniques because they can be easily circumvented once the modification does not rely on hooked functions. Besides, hooking is so system-dependent that a small update for Windows can easily invalidate our system. These considerations lead us to the following approach.

Modified or newly created keys in registry database files will be extracted in the end of analysis. With taint tags of hard disk sectors in hands, tainted fragments in registry database files can be therefore identified. Note that the taint analysis assures that these fragments are actually injected due to the behavior of the target file. If there is any registry key newly created or modified by the target, its key value will be therefore included in one of these fragments. With the key value located, we parse the database file and traverse the registry tree reversely to infer the name and the path of the key. Although the format of the registry database has never been full explained by Microsoft, yet previous papers had already demonstrated its internal data structure, which benefits implementation of our traversing algorithm.

To verify correctness and usability of system-wide DIFT and taint analysis we designed, a malware behavior profiling system is implemented based on them. In this section we evaluate its effectiveness by profiling real malware spreading in the Internet. The report is presented on four dimensions: tainted file system objects, tainted processes, tainted registry keys and values, and tainted network traffics. In the meantime, we also submit the sample to ThreatExpert, which is a powerful online dynamic malware analysis tool, to check if our system gives matched reports. In the following paragraphs we discuss analysis results by case study.

Case Study 1: TR/Dldr.FraudLoad

SWIFT generates the following report for TR/Dldr.FraudLoad.

```
              Taint Analysis Report #1 : TR/Dldr.FraudLoad
                    ===============  Files ===============
//$Bitmap
/Documents and Settings/All Users/Application
Data/boost_interprocess/20101231030705.500000/GoogleImpl
/Documents and Settings/dsns/Local Settings/Temporary Internet
Files/Content.IE5/PH91KL45/flash3[1].exe
/Documents and Settings/dsns/NTUSER.DAT
/Documents and Settings/dsns/NTUSER.DAT.LOG
/Documents and Settings/dsns/Desktop/sample1.exe
//$LogFile
//$MFT
//$Secure:$SDH
```

```
//$Secure:$SDS
//$Secure:$SII
/WINDOWS/system32/config/software
/WINDOWS/system32/config/software.LOG
/WINDOWS/system32/config/system
                   ===============  Process ===============
sample1.exe, PID : 1112
                   ===============  Registry ===============
HKLM/software/Microsoft/Windows/CurrentVersion/Run
  Key: SmartIndex
  Value: C:\Documents and Settings\dsns\Desktop\sample1.exe
HKLM/software/Microsoft/Cryptography/RNG
  Key: Seed
  Value: (binary)
HKCU/Software/Google
  Key: ID3
  Value: (binary)
HKCU/Software/Google
  Key: AppID
  Value:
  DFlQqm+e4GDgQ0G+5GTaW1+CtBpxNgwhPn5mcHoouaFMzYkUfw26cM0mM4OYMuKCxg==
HKCU/Software/Google
  Key: ID2
  Value: (binary)
                   ===============  Packet ===============
-> 188.229.90.5 , TCP 1035 -> 80
  GET /flash3.exe HTTP/1.1 0x0d 0x0a User-Agent: Mozilla/4.0 (compatible;
  MSIE 8.0; Windows NT 6.1; Trident/4.0) 0x0d 0x0a Host: 188.229.90.5 0x0d
  0x0a Cache-Control: no-cache 0x0d 0x0a 0x0d 0x0a
-> 76.101.129.197 , TCP 1034 -> 80
  GET /NS3/.htm HTTP/1.1 0x0d 0x0a Host: 76.101.129.197 0x0d 0x0a Content-
  Length: 242 0x0d 0x0a User-Agent: Mozilla/4.0 (compatible; MSIE 8.0;
  Windows NT 6.1; Trident/4.0) 0x0d 0x0a 0x0d 0x0a
-> 95.168.185.46 , TCP 1031 -> 80
  GET /Gnodu8Ir.htm HTTP/1.1 0x0d 0x0a Host: 95.168.185.46 0x0d 0x0a
  Content-Length: 242 0x0d 0x0a User-Agent: Mozilla/4.0 (compatible; MSIE
  8.0; Windows NT 6.1; Trident/4.0) 0x0d 0x0a 0x0d 0x0a
```

According to the report generated by ThreatExpert, the Trojan program establishes network connections to remote hosts and several registry keys are implanted. As shown in the report, several tainted packets and registry keys are captured by our system. The tainted payload indicates that it tries to download several files from remote servers. For the registry part, the captured tainted value shows a start-up item is registered. Items above agree with report reported by ThreatExpert. However, in the file system our report shows that the target program contaminates `//$Secure`, which is used to store NTFS security descriptor table for files and directories. The report generated by ThreatExpert gives no information about this.

One odd phenomenon can be observed here. According to our report, three HTTP requests are issued to download different files, yet only flash3.exe is shown in the tainted file section. The reason causing this is that NTFS stores files with content less than 1Kbytes in the Master File Table (`//MFT`) directly to save disk space. Therefore, disk blocks occupied by the two small HTML files are actually possessed by `//MFT`.

Case Study 2: Crypt.NSPM.Gen

SWIFT generates the following report for Crypt.NSPM.Gen.

```
              Taint Analysis Report #2 : TR/Crypt.NSPM.Gen
                 ===============  Files ===============
/
* /Documents and Settings/dsns/Local Settings/History/History.IE5/index.dat
/Documents and Settings/dsns/Local Settings/Temp/wdagnb7.dll
/Documents and Settings/dsns/NTUSER.DAT
/Documents and Settings/dsns/NTUSER.DAT.LOG
/Documents and Settings/dsns/Desktop/sample2.exe
//$LogFile
//$MFT
//ntdelect.com
/WINDOWS/system32/config/software
/WINDOWS/system32/config/software.LOG
/WINDOWS/system32/config/SysEvent.Evt
/WINDOWS/system32/config/system
/WINDOWS/system32/config/system.LOG
/WINDOWS/system32/kavo0.dll
/WINDOWS/system32/kavo.exe
```

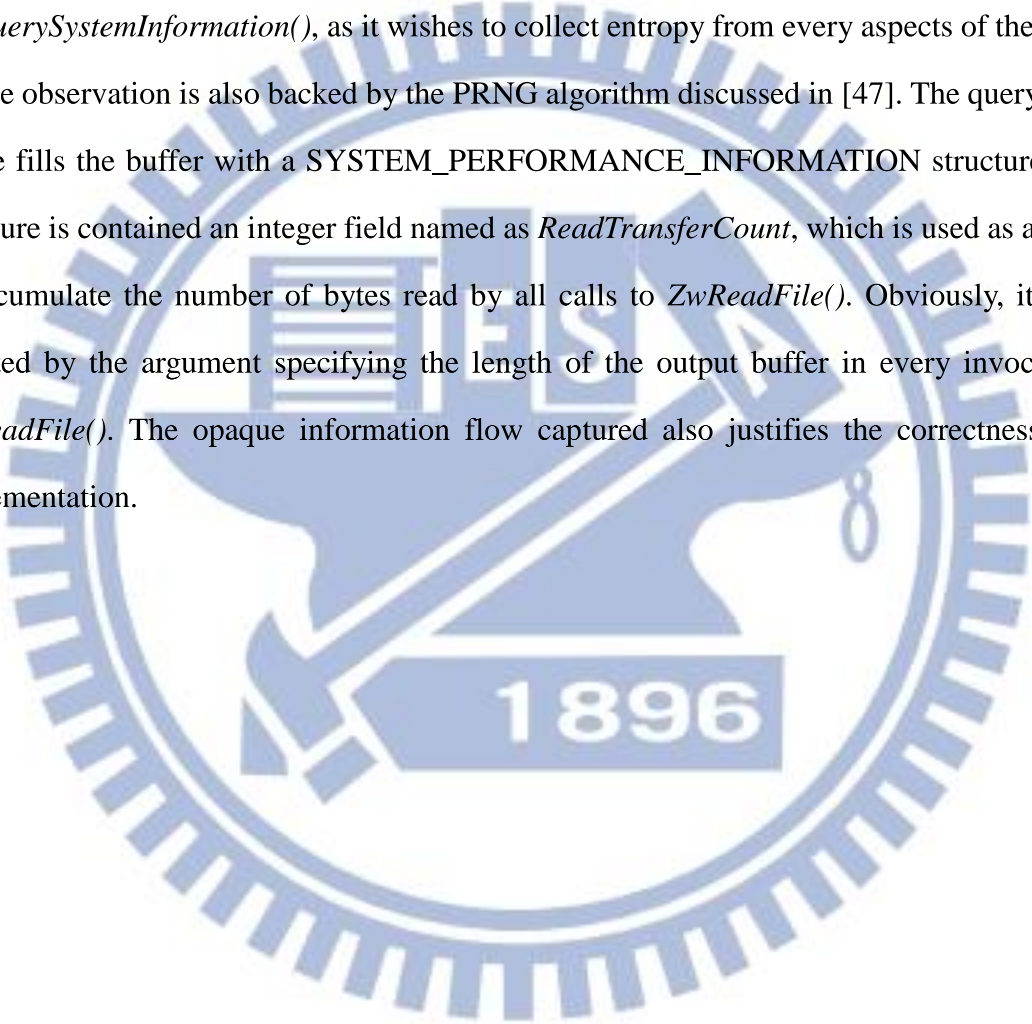
```
                  ===============   Process ===============
System, PID : 4
lsass.exe, PID : 544
svchost.exe, PID : 696
explorer.exe, PID : 1268
IEXPLORE.EXE, PID : 1108
IEXPLORE.EXE, PID : 1136
sample2.exe, PID : 1116
                  ===============   Registry ===============
HKLM/SYSTEM/ControlSet001/Enum/Root/LEGACY_GHTRFDCXDSWEA/0000
  Key: Service
  Value: ghtrfdcxdswea
HKLM/SYSTEM/ControlSet001/Enum/Root/LEGACY_GHTRFDCXDSWEA/0000
  Key: DeviceDesc
  Value: ghtrfdcxdswea
HKCU/Software/Microsoft/Windows/CurrentVersion/Run
  Key: kava
  Value: C:\WINDOWS\system32\kavo.exe
HKLM/software/Microsoft/Cryptography/RNG
  Key: Seed
  Value: (binary)
                  ===============   Packet ===============
168.95.1.1 , UDP 1027 -> 53
    0xa7 c 0x01 0x00 0x00 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x03 www 0x0b
microsofttw 0x03 com 0x00 0x00 0x01 0x00 0x01
```

As shown in the second report, several processes are tainted by the target program. The power of whole-system information flow tracking is demonstrated in this example. Based on the condition we label processes as tainted, we can be assured that dirty code is executed in in user-level processes and even kernel space (System, PID: 4). The tainted network packet also shows the target program tries to look up the domain name `www.microsofttw.com`, but no additional tainted packets is founded later. It turns out that the query for the domain name is responded with NXDOMAIN, and the site is considered as malicious by Google Web Search.

In nearly every report generated by SWIFT we discovered that the key "`Seed`" in the registry entry `HKLM/software/Microsoft/Cryptography/RNG/` is tainted. As a matter of fact, our system reports it as tainted even if profiling a simple dummy program which

exits immediately. It is quite intriguing because the registry is used as the seed of the pseudo-random number generator for various Windows CryptoAPI functions, of which our dummy program does not invoke any.

To better understand the consequence between the taint source and the registry key, the information flow is dumped and traced backward manually with the help of IDAPro. It turns out that the pseudo-random number generator calculates a SHA-1 hash value on outputs of *ZwQuerySystemInformation()*, as it wishes to collect entropy from every aspects of the system. Above observation is also backed by the PRNG algorithm discussed in [47]. The query routine above fills the buffer with a SYSTEM_PERFORMANCE_INFORMATION structure. In the structure is contained an integer field named as *ReadTransferCount*, which is used as a counter to accumulate the number of bytes read by all calls to *ZwReadFile()*. Obviously, it will be affected by the argument specifying the length of the output buffer in every invocation of *ZwReadFile()*. The opaque information flow captured also justifies the correctness of our implementation.



Chapter 6

Applications – Kernel Rootkit Recognition

Techniques for malware pattern extraction and recognition had been discussed in previous research [48][49][50][51][52][53][54]. Among these studies, a very common characteristic taken into consideration is the invocation on Application Programming Interface, API, or system calls. Since malware are designed to carry out certain malicious tasks, they inevitably interact with the running environment through these interfaces. In addition, semantics of program behaviors are actually embedded in invocations on those functions since one important designing principle for API is descriptiveness.

However, existing API-trace-based behavior analysis systems lose their advantages when facing advanced malware equipped with kernel-level rootkit. Successfully invading the OS kernel implies the acquisition of the privilege of system administrator, which is able to circumvent or to sabotage any other programs in the system. As aforementioned, Trojan.Srizbi, which is responsible for 40% of all the spam on the Internet in 2008 [55], executes all its functionalities such as hiding files and sending botnet traffic in the kernel space.

To cope with problems above, in this dissertation a novel system, MrKIP, is proposed to recognize malware with invocation pattern of kernel functions. With the assistance of ProbeBuilder, MrKIP implements the generated probes to monitor kernel-level activities. These probes are implemented into SWIFT, which is capable of system-level taint tracking. As long as any arguments of the probed function are tainted, the invocation and the associated arguments are recorded.

MrKIP performs rootkit behavior recognition in two phases: pattern training and recognition. In the training phase, MrKIP executes variants known belonging to the same rootkit family and collects invocations of important in-kernel functions with the associated arguments. The collected invocation sequences and the arguments together are used to construct a behavior-based pattern for that malware family. In the recognition phase, we again execute the given suspicious program inside our profiling emulator to collect its in-kernel function invocations as its behavior profile, which will be matched against patterns of those known rootkit families.

6.1 MrKIP Internals

In the section, we describe the methodology and design of MrKIP. Its task is to test whether the behavior of a suspicious program follows the pattern of a certain malware family. In Figure 17, an overview on the flow chart is given. As shown in the figure, the system operation can be separated into the pattern training phase and recognition phase. In both phases, we rely on the **BehaviorProfiler** to dynamically execute programs and collect the in-kernel function invocation traces. In pattern training phase, traces of instances from the same malware family are fed into the **PatternGenerator** to construct a pattern for later recognition. The pattern consists of an HMM to recognize the temporal pattern hidden in the invocation sequence of in-kernel functions and normalized string patterns for argument similarity measurement. In recognition phase the profiler is again used to record the behaviors in kernel space led by the testing subject. The **PatternRecognizer** is responsible for evaluating the deviation of the collected trace from patterns of families.

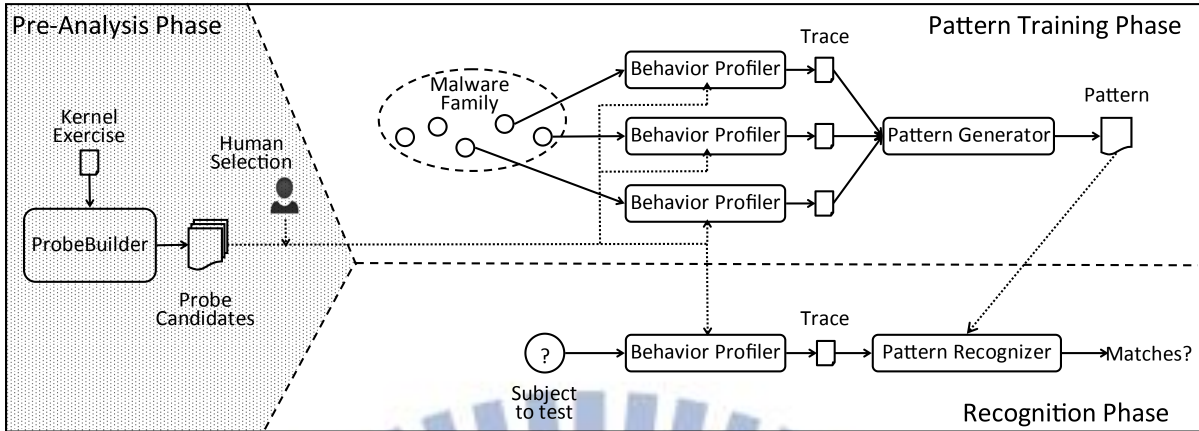


Figure 17 : MrKIP Architecture

To avoid being detected by malware, the monitoring functionality should be realized with the so-called "out-of-box" hooking [2]. Namely, the code to capture in-kernel function invocation and their arguments should be implemented in the VMM to be unobservable for program running inside the guest operating system. In addition, to mitigate the possibility being detected by VM-aware malware, **BehaviorProfiler** renames the devices and wipe out VM-related flags in memory, both of which are widely used to detect VM environment. The pre-analysis phase shown in Figure 17 demonstrates that MrKIP utilizes ProbeBuilder to automatically generate probes for kernel functions to monitor.

6.1.1 Behavior Profiler

The **BehaviorProfiler** serves to generate the behavior trace of the subject program. It is constructed by installing the probes that ProbeBuilder generated in the pre-analysis phase. The profiling process begins with importing the subject program into the hard-disks of the guest system. Then, these sectors occupied by that imported file are tainted as the source of contamination. Then, the subject is executed and hence the tainted information will be propagated all over the system. Once a probe is hit by the execution, its dereference is walked through to check whether it reaches tainted information.

A probe H is defined as a 3-tuple $H = (n, loc_H, IT_H)$, where loc_H is a memory address and IT_H is an ordered sequence of 2-tuple (I^H_k, T^H_k) for $1 \leq k \leq n$. Each I^H_k: $States \rightarrow \{0,1\}^*$ is a function maps a machine state S to a binary string ω. Namely, it extracts certain information

Table 12 : Excerpts from the profiled behaviors of ad.zenosearch.

loc	data	type
1	0x0a00020f	u32
	0xa85f0101	u32
	0x0402	u16
	0x0035	u16
	"… 0x010x000x000x000x000x000x000x03 www 0x09 think-adz 0x03 com 0x000x000x01 …"	raw
2	"C:\WINDOWS\system32\dbglogfolder\n_inst_05_01_11.log"	path
3	"REGISTRY/USER/S-1-…1003/SOFTWARE/MICROSOFT/WINDOWS/CURRENTVERSION/INTERNET SETTINGS"	path
	"EnableAutodial"	text
	"0x060xae0xc8"	raw
1	0x0a00020f	u32
	0x48378cb8	u32
	0x0403	u16
	0x0050	u16
	"GET /instreport8_2.asp?uid=0 HTTP/1.1Ox0dOx0aUser-Agent: [ELT001]52-54-00-12-34-56:99-9E-E4-4C-:0:..."	raw

according to the register, memory, and taint status of state S. Each $T^H{}_k$ is an element in the type collection T specifying how the output of $I^H{}_k$ should be interpreted in later analysis. Therefore, when a probe $H = (n, loc_H, IT_H)$ is triggered, a sequence of binary strings $(I^H{}_1(S), I^H{}_2(S), \ldots, I^H{}_n(S))$ will be generated respectively out of the machine state S. We define a *behavior B* profiled by a probe H at state S as a 3-tuple $(loc_B, type_B, data_B)$, where $loc_B = loc_H$, $type_B = (T^H{}_1, T^H{}_2, \ldots, T^H{}_n)$ and $data_B = (I^H{}_1(S), I^H{}_2(S), \ldots, I^H{}_n(S))$. For simplicity, from now on we denote the k-th element in $type_B$ and $data_B$ as $type_B[k]$ and $data_B[k]$ respectively.

The type collection T is a finite set defined heuristically. To achieve generality, elements of T should be platform-independent while preserving maximal semantics since it provides clues for later data-processing. In our design we defined T as

{bitmap,u8,u16,u32,text,path,raw,random}

The `bitmap` type indicates that the data should be viewed as a vector of individual bits, which are generally used in simultaneously expressing states of multiple Boolean variables. The type `u8`, `u16`, and `u32` stand for unsigned integers of 8-bit, 16-bit, and 32-bit data size respectively. A `text` is a sequence of readable characters with length less than 128. To capture

all those data used to express a `path` in tree-like structures such as file paths or Windows registry are classified into the type path. Entries which cannot be classified into any categories listed above are treated as `raw` data.

The type `random` is a special attribute attached to those data are generated or mixed with random numbers. For example, the source port of a TCP or UDP connection is usually picked randomly by the system. Due to their randomness, they have negative impact on the learned behavior model. Therefore, it is necessary for us to filter out these meaningless data. To achieve this, MrKIP locates the pseudo random number generator in the system, and taints its output with this special tag. The taint propagation ensures that data calculated out of random numbers can be distinguished from ordinary ones.

Table 12 shows an excerpt from the profiled behaviors of the adware zenosearch. Each behavior *B* is presented in three columns: loc_B, $data_B$ and $type_B$. Note that the *loc* is substituted with merely a unique ID since the address is not meaningful. However, it is quite easy for us to "guess" which kind of information is processed by the codes near loc_B by observing $data_B$ only. The adware performs DNS lookup for the name "`www.think-adz.com`", tries to access a file named as "`n_inst_05_01_11.log`", writes data into registry, and then issues an HTTP query. Note that all these information are acquired in kernel space while zenosearch is executed as a user-level application.

6.1.2 Pattern Generator

Given a set of malware known in the same family, **PatternGenerator** tries to build a model for that family. The control flow transition and the data characteristics are both powerful metrics for recognizing program behavior, and hence they should be both captured. In addition, the model should be able to associate a probability to a subject program so that the model predicts the probability with which a subject belongs to that family. In our design, **PatternGenerator** attempts to group behaviors with similar arguments together. By viewing each cluster as a state, the original sequence of profiled behaviors can be transformed to a sequence of transition between states, and a Markov chain can be hence learned.

To capture the execution context, we use a Markov chain to model the transferring probability between the hooks. An intuitive idea is to associate each loc_H with a state of the Markov chain. In this way, a sequence of behaviors can be viewed as a path in the chain by consequently walking through those states specified by loc_B, and the transition probability matrix states can be therefore trained. However, the approach stated above neglects the fact that even a single function may provide different functionalities when different arguments are given. Therefore, associating a loc_H with only a single state may lead to a rough model.

To provide better recognition rate, we further partition those behaviors with the same loc_B into smaller groups based on their argument similarity. The partitioning is done with the agglomerative, complete linkage clustering algorithm, which progressively groups elements in a bottom-up way. We will describe the algorithm briefly but skip its details since it is a well-known technique for data clustering. Before applying the algorithm the distance function d to measuring the similarity between any two objects in the group, a threshold δ specifying the stopping criterion must be determined. The clustering begins with forming a singleton for each element. The distance between any two clusters X and Y are given by $max(d(x, y))$ where $x \in X$ and $y \in Y$. The process continuously joins two clusters if the distance between them is less than δ. The value of δ determines how similar the elements in the same cluster will be, and it hence affects the quality of the learned model. We evaluate the effectiveness with different δ value in our experiments.

It is clear that the characteristic of distance function has a direct influence on the quality of partition. We use the following formula to measure the distance between the two behaviors B_1 and B_2.

$$\mathrm{D}(B_1, B_2) = \sum_{k=1}^{n} w_k \times d_{(type[k])}\left(data_{B_1}[k], data_{B_2}[k]\right)$$

where w_k are weighting constants defined heuristically in the range (0, 1), and satisfy $w_1 + w_2 + \ldots + w_k = 1$. Note that we do not discuss the distance between different types because our goal is to partition those behaviors with the same loc_B into smaller groups. Since given a loc_B its $type_B = (T^H{}_1, T^H{}_2, \ldots, T^H{}_n)$ is uniquely determined, the distance will be measured only

between two elements with the same type. As previously stated, there are seven possible attributes for $data_B$: bitmap, u8, u16, u32, text, path, and raw. For each of them, we define the distance function $d_{(type)}$, which measures the distance between behaviors B_1 and B_2. Note that we do not discuss the distance between different $type_B$ because our goal is to partition those behaviors with the same loc_B into smaller groups.

`bitmap`: Data labeled with this attribute are used as flags or attributes such as file opening modes. The purpose and meaning are assigned to each bit in the sequence. In addition, the bit sequence usually has fixed length. Therefore, the hamming distance function is a good metric

$$d_{\text{bitmap}} = \frac{\text{The hamming distance}}{\text{The length of the bitmap}}$$

for measuring the number of bits varying in the two binary strings. To normalize it, the hamming distance is divided by the length of the bit sequence.

`u8`, `u16`, and `u32`: Numeric values in which an ordering relation is maintained are attributed with these types. Since they can be viewed as points residing on the line of real

$$d_{u_n}(x, y) = \frac{|x - y|}{2^n}$$

numbers, the most natural way to define their distance would be:

Although numeric values may not be used as real numbers, in most cases they still preserves certain ordering relations and justify the meaning of above formula. For example, the distance between the IP address 64.233.171.18 (Google web server) and 64.233.179.19 (another Google web server) is intuitively smaller than the distance between 64.233.171.18 and 220.181.6.6 (a web server of Baidu search engine in China) due to the geographical difference between the machines holding on to these addresses. Another important kind of data, which possess good characteristic of real numbers, is time-related values. To ensure the consistency among profiling, we always adjust the system time of the emulator to a fixed instant every time before profiling a subject. Note that those data originating (even partially) from random number generator had been filtered out by the profiler as stated previously.

`text` and `path`: To compare the difference between two human-readable strings the

Levenshtein distance, which is usually referred as the edit distance, is widely adopted. The distance is defined as the minimum number of edit operations needed to transform the original string to another. However, it is obvious that two strings of length 10 differing in 1 bit show more difference than two strings of length 1000 differing in only 3 bits. Therefore, it is necessary to normalize the edit distance by taking the string length into account. To this end we adopted the normalized edit distance proposed by Marzal et al [56]. In their work, the normalized edit distance is acquired by minimalizing the average cost spent by each step in the edit path. Also, their algorithm works in $O(m \cdot n^2)$, where m and n stand of the length of strings, and $n \leqq m$. Since only those data of lengths less than 128 should be attributed to the type Text, the computation is still acceptable. However, data labeled as Path such a pathname or a registry entry could be too long to compute the distance efficiently. To accelerate the distance computation, we substitute the substring of each level in the path with a 32-bit CRC value computed out of them. For example, the string "`/Program Files/Microsoft Office`" will be transformed to "`/\x10\x97\xE8\x4A/\xC2\xDC\xC7\x7E`" before being fed into the normalized edit distance calculator.

`raw`: Due to performance issue we do not consider content-aware method to compute the distance between data larger than 128 bytes. In our approach, these data are compared with a conventional but effective metric, which is the Jensen–Shannon divergence [57]. With the occurrence frequency of each of the 256 possible byte patterns computed, the Jensen–Shannon divergence measures similarity between two probabilistic distributions. Due to its many desirable characteristics such as symmetry, non-negativity, and boundedness, the metric had been widely adopted in bioinformatics, genomic comparison, and various data mining techniques.

The algorithm converting behaviors to state transitions is listed in Algorithm 5. A set of sequences of behaviors, which are acquired by profiling executions of malware known to be the same family, is fed into the Behavior-To-State procedure as input. Line 4-6 groups all behaviors with the same loc_b together so that the clustering is done on behaviors profiled by the same hook. With the distance functions defined in previous paragraphs, the complete-linkage

Algorithm 5 : Behavior to HMM state.

Input:
BS : A set of sequences of behaviors.

Output:
n : The number of states.
SS : A set of sequences of states
R : $<r_1, r_2, \ldots, r_n>$ is a list of representative behaviors.

```
Behavior_To_State(F)
1  H ← A key-value map (mapping an address to a set of behaviors)
2  C ← A key-value map (mapping a behavior to an integer)
3  n ← 0, R ← <>, SS ← {}
4  foreach sequence s ∈ BS
5  |  foreach behavior b ∈ s
6  |  |  H[loc_b] ← H[loc_b] ∪ b          // Grouping behaviors by loc_b
7  foreach value h ∈ H
8  |  cl ← Complete_Link_Cluster(h)       // cl contains sets of behaviors
9  |  foreach c ∈ cl
10 |  |  foreach b ∈ c
11 |  |  |  C[b] ← n
12 |  |  r ← pick b out of c  and  b minimizes Σ_{b'∈c} D(b, b')
13 |  |  R ← R || r
14 |  |  n ← n + 1
15 foreach sequence s : <b_1,b_2,…,b_k> ∈ BS
16 |  SS ← SS ∪ <C[b_1], C[b_2], …, C[b_k]>
17 return n, SS, R
```

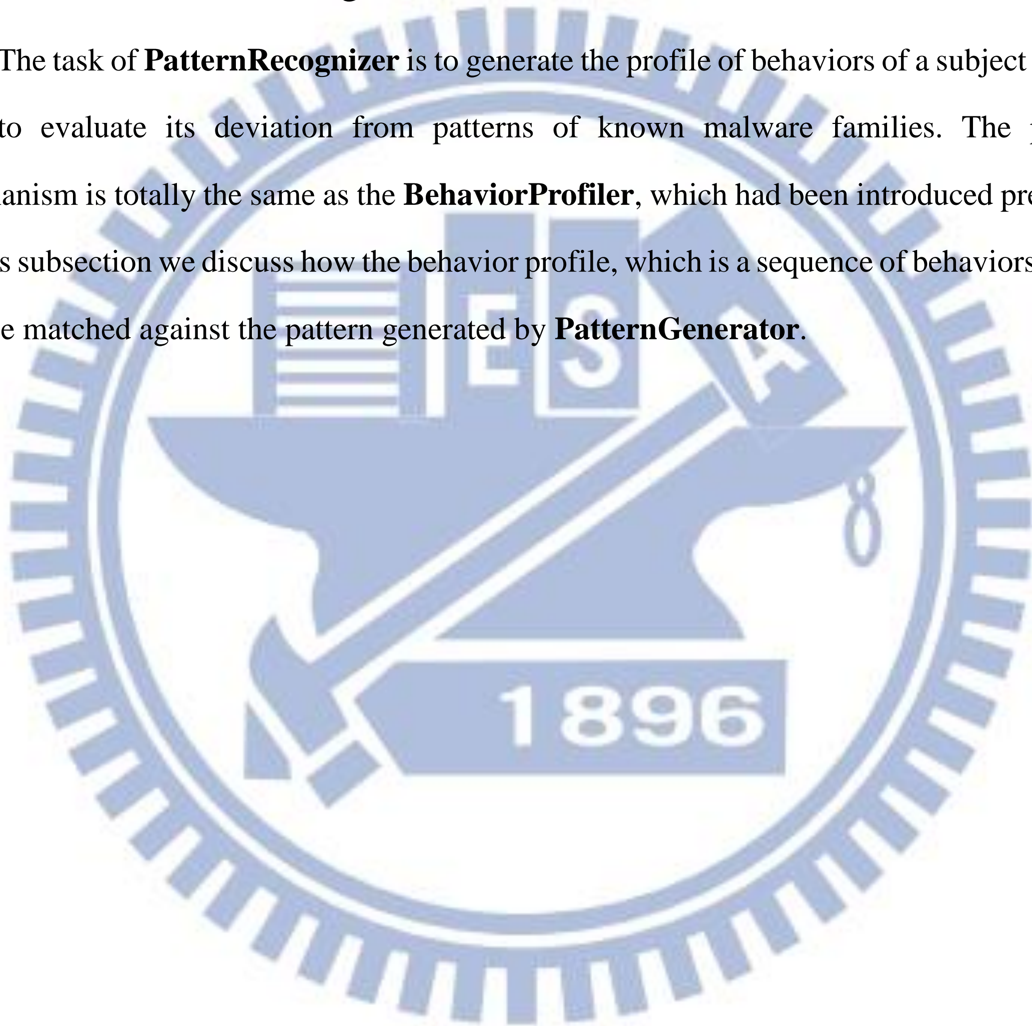
clustering algorithm invoked in line 7-8 further cluster behaviors according to their argument similarity. In line 9-14, the algorithm assigns an integer n to each group acquired by clustering as its state number. In addition, behaviors belonging to the same group are labeled with that number. The mapping is recorded in the key-value map C. Lines 15 and 16 convert each sequence of behaviors $s \in BS$ to a sequence of states by replacing each $b_k \in s$ with $C[b_k]$.

With sequences of states SS in hands, it is trivial to learn the Markov transition probability matrix from them. However, only the Markov chain itself is not enough. Let's consider what tasks the **PatternRecognizer** should perform. In recognition phase, a behavior profile, which is simply a sequence of behaviors, say bp, will be matched against a Markov chain learned by **PatternGenerator**. Therefore, the matching can only be performed after each behavior in bp

has been mapped to states in the chain. To this end, we generate a centroid for each group acquired in the clustering by picking the element which minimizes its distance summation to all other elements in that group. Behaviors in *bp* are compared with these centroids, and appropriate states can be therefore found. Line 12 and 13 are responsible for the task above, and the resulted centroids are preserved in *R*.

6.1.3 Pattern Recognizer

The task of **PatternRecognizer** is to generate the profile of behaviors of a subject program, and to evaluate its deviation from patterns of known malware families. The profiling mechanism is totally the same as the **BehaviorProfiler**, which had been introduced previously. In this subsection we discuss how the behavior profile, which is a sequence of behaviors actually, can be matched against the pattern generated by **PatternGenerator**.



Algorithm 6 : Calculation of Matching Degree

Input:

B : A sequence of behaviors.

m : A 3-tuple (n, M, R), where

n is the number of states,

M is the Markov transition matrix, where $M_{0,k}$ preserves initial probability of state k.

$R = \langle r_1, r_2, \ldots, rn \rangle$ is the list of centroids.

Output:

p: A value in (0,1) indicating the matching degree between b and m .

```
Calculate_Matching_Degree(B, m)
1   p ← 0, j ← 0, l ← 0
2   foreach b ∈ B
3   |  k ← 0, s ← δ
4   |  for i ← 1 to n
5   |  |  d ← D(b, r_i)
6   |  |  if d < s then
7   |  |_ |_ s ← d, k ← i
8   |   p ← p · M_j,k
9   |   j ← k
10  |_  l ← l + 1
11  return p^(1/l)
```

With the state sequences SS acquired in the last subsection, a Markov chain M can be immediately learned from them. Together with the centroids R, the 3-tuple (n, M, R) can give a matching score to a given sequence of behaviors. The procedure is listed in Algorithm 6. The probability calculation basically follows the procedure of evaluating the probability of a state sequence. For each behavior, we calculate the distance between it and every centroid in R to figure out which state gives birth to that behavior in line 3-8 of the algorithm. In line 9 the transition probability is cumulated. In the end, the geometric mean of the cumulated product is returned as the output.

Note that the distance between the behavior and its closest centroid could be still larger than the threshold δ. Since no appropriate state can be found for such behaviors in the matched

pattern, they should be considered as total deviation from that model. In such a case, the index k remains as its initial value 0, which is assigned at line 3. Therefore, before the algorithm starts we search for the smallest value in the matrix M and replace all elements $M_{k,0}$ with it for all $1 \leqq k \leqq n$. Behaviors whose arguments present huge deviation from that model will attenuate the value of the final output.

6.2 Experiments

In order to evaluate performance and precision of MrKIP, we conduct three sets of experiments. In the first experiment, the trojan Srizbi is used to demonstrate MrKIP's profiling mechanism against pure kernel-level rootkits. The second experiment measures the performance of MrKIP, showing its capability to recognize rootkits in a reasonable time. In the last experiments, we cluster 536 kernel-level rootkit instances with VirusTotal, and divide them into training set and testing set randomly. Then, we evaluate the effectiveness of recognition. All our experiments are conducted on an Intel i7 machine with Windows 7 OS. Samples used in experiments are collected from offensive computing, a public sample sharing forum. Please note that MrKIP can be also applied on the recognition of ordinary user-level malware, since their behaviors are eventually executed through kernel-level functions. However, our experiments focus on the evaluation of the effectiveness of MrKIP against advanced, kernel-level trojans.

6.2.1 Case Study : Srizbi

Srizbi is one of world's largest botnet. With the capability to hide itself from both user and system level, it is difficult to remove and detect. Since Srizbi is executing totally in kernel mode, it can make its files and network traffic invisible to bypass detection. With these advanced rootkit technique, Srizbi is considered one of sophisticate rootkits. In order to demonstrate correctness of **BehaviorProfiler**, we use this famous rootkit family as a case study. We use two variants, Trojan.Win32.Srizbi.ah and Trojan.Win32.Srizbi.x, labeled by Kaspersky, to evaluate correctness of the extracted behavior profiles.

Our tool records 116 behaviors in both sample's profiles. We can observe that both Srizbi

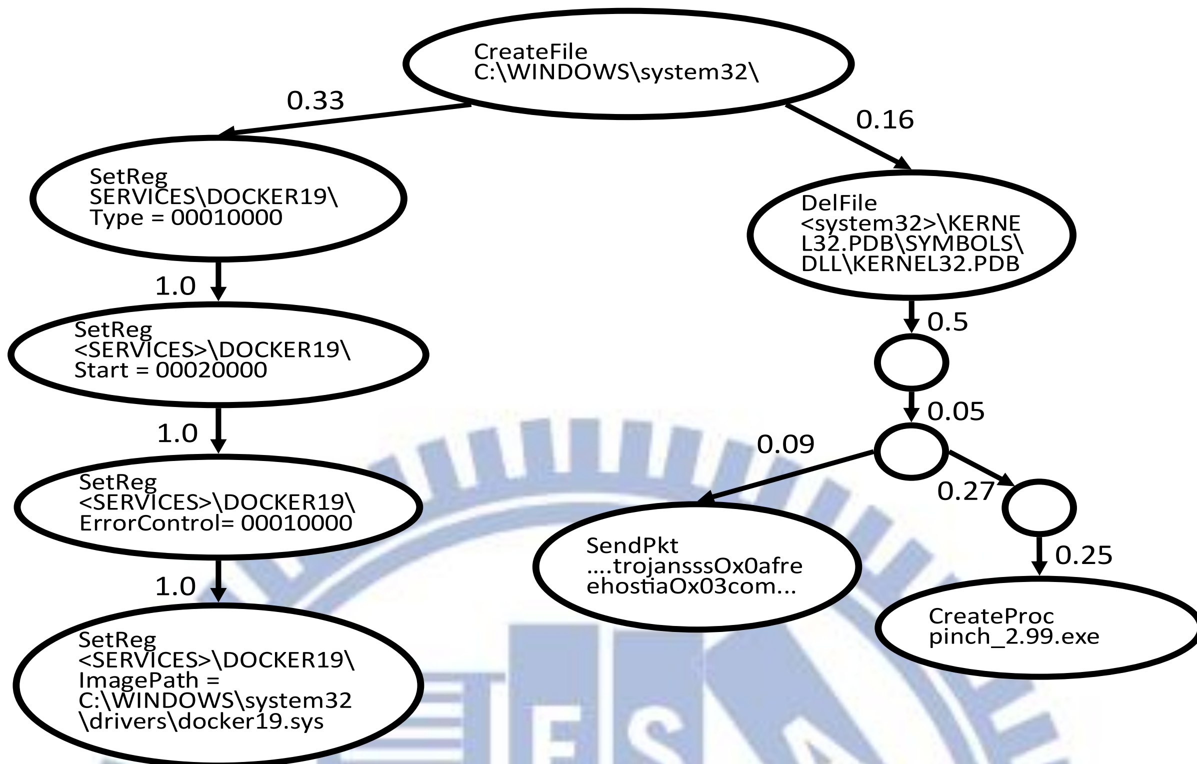


Figure 18 : Constructed model for Srizbi.

samples first delete some system files and then do some file manipulation to driver files. It also registers itself as a system service. We also uploaded the Srizbi trojan instances onto two famous online malware analysis systems, Threat Expert and Anubis, for comparison. It turns out that Anubis does not generate information at all about it. Threat Expert captured certain registry modification behaviors, which form merely a subset of our profiling result. This comparison shows that our kernel-level behavior profiling is more effective than conventional approaches.

The whole HMM model generated by **PatternGenerator** for Srizbi contains more than a hundred states, which are difficult to present in the article. To illustrate the idea, we show in Figure 18 a portion of the generated pattern. Each node is a clustered state, and the string inside the node is the data selected as the centroid for that cluster. On each edge the transition probability is also listed. In the model we can observe the three major types of captured behaviors: registry modification, packet transfer, and process creation. As shown, the transition probabilities between the sequential registry modifications are 1. This matches the convention that registering a program as a system service requires setting up multiple registry entries.

6.2.2 Effectiveness of Recognition

To evaluate the effectiveness of **PatternRecognizer** of MrKIP, the next experiment compares the clustering result of MrKIP with the clustering done by commercial an-ti-virus software. The comparison is performed as follows. For each collected rootkit instance, we upload it onto VirusTotal, which is a website providing simultaneously the analysis results of dozens of anti-virus software. Two instances which reported by any different anti-virus software as the same family will be grouped together. This is used as the ground truth, and our recognition result will be compared with it. The 536 rootkit in-stances are then separated into the training set and the testing set. We divide one family into two partitions with equal sizes, intending to keep the total size of the training set equal to that of the testing set. Yet, certain family contains so few variants that we have to maintain an enough amount of instances for training, leading to a slightly imbalanced partition. In the end, we have 351 rootkit samples in the training set and 185 samples in the testing set.

For each instance in the testing set, our **PatternRecognizer** compares it with each constructed model and generates a matching score. Thereby, through sorting we can observe in which place the correct group (the right answer) gets among all other families. We refer to the index of the correct group in the sequence of families (sorted with the similarity score, from high to low) as the rank of that instance. For instance, if the similarity score of family Trojan.Win32.Delf takes the fourth place among other families when we recognize Trojan.Win32.Delf.cit, which is confirmed a variant of Trojan.Win32.Delf, the rank of Trojan.Win32. Delf.cit is 4, which means Trojan.Win32.Delf is fourth similar to Trojan.Win32.Delf.cit.

The cumulative distribution of classification ranking is shown in Figure 19. The X-axis represents the rank and the Y-axis indicates the cumulative percentage of instances. A coordinate (x,y) in the figure indicates that y% instances of the whole testing have rank numbers less than x, which indicate the correct family of y% instance can be found in top x similar families. A steeper curve indicates more instances have lower rank numbers, which implies that

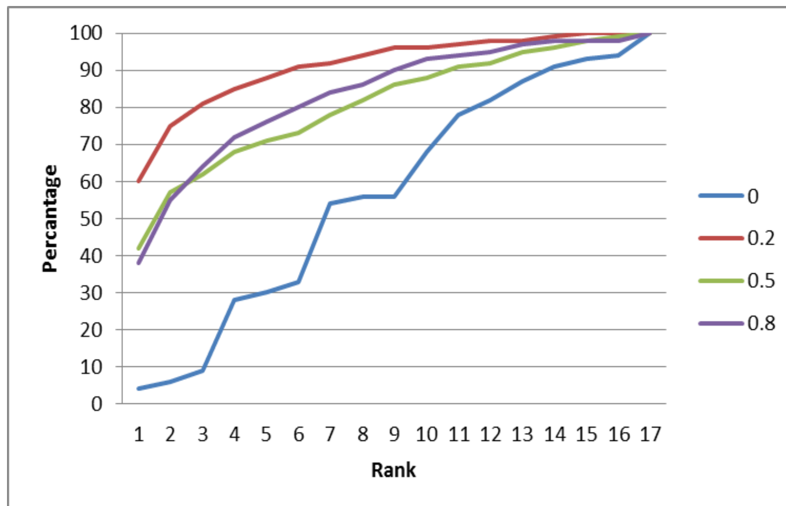


Figure 19 : Cumulative Ranking.

the correct family gets a higher score from our **PatternRecognizer**.

Meanwhile, since there is a parameter δ in our algorithms, we repeat this experiment multiple times with different threshold values. When δ is set to 0, each behavior will form its own behavior group, even if their arguments are similar. As shown, without grouping similar behaviors, the classification result is poor. After raising the threshold value to 0.2, 60% of the instances in the testing set have rank number 1, indicating MrKIP finds correct answer. Note that the cumulative curve also indicates that 80% instances have rank less than 4. Namely, MrKIP can successfully sort the correct answer in the top three places for 80% test instances. The cumulative percentage even increases to 90% when rank reaches to 5. If we further raise the threshold value, unrelated behavior may be group together. Therefore the classification rate will decrease. As our experiments shows, the appropriate threshold is around 0.2.

Chapter 7

Limitation and Discussion

Although the effectiveness of ProbeBuilder has been demonstrated in the experiments, it is neither complete nor sound in the perspective of code coverage. However, in this application, completeness is not required since ProbeBuilder only attempts to discover probable locations for VMI implementations. Finding all of them is not mandatory. ProbeBuilder tries to approach soundness by repetitively exercising the subject behavior. However, the validity of generated data dereferences is still not guaranteed either. The experiment in section 5.3 showed that the collected probes captured a small amount of irrelevant data, namely 0.21% of the total data. This ratio is competitively low as a human-implemented probe can get, considering the tremendous effort saved by ProbeBuilder.

The scope of dereference analysis in current implementation only covers data originating from dedicated data sources. The API arguments demonstrated in the experiment fit in this category. However, it is sometimes desirable in a probe implementation to record data spontaneously generated by the operating system. For instance, a probe on process creation is often used to capture process ID or page directory base address of the created process. Since these data do not originate from a finite set of dedicated data sources, the predicate *P* implemented with taint analysis becomes less effective. To cope with the limitation, there exists a possible solution. Through virtual machine recording and replaying, the evolution of the system state can be faithfully reconstructed at bit-precision [52][58]. Since these methods are able to completely remove non-determinism of the machine state, the targeted data (e.g. process

ID, page directory base address, etc.) will possess the same values in the replaying phase. Therefore, implementing the predicate *P* with fixed pattern matching may solve the problem. However, combining ProbeBuilder with the replaying framework requires further implementation and is out of the scope of this study. We leave it as a future work.

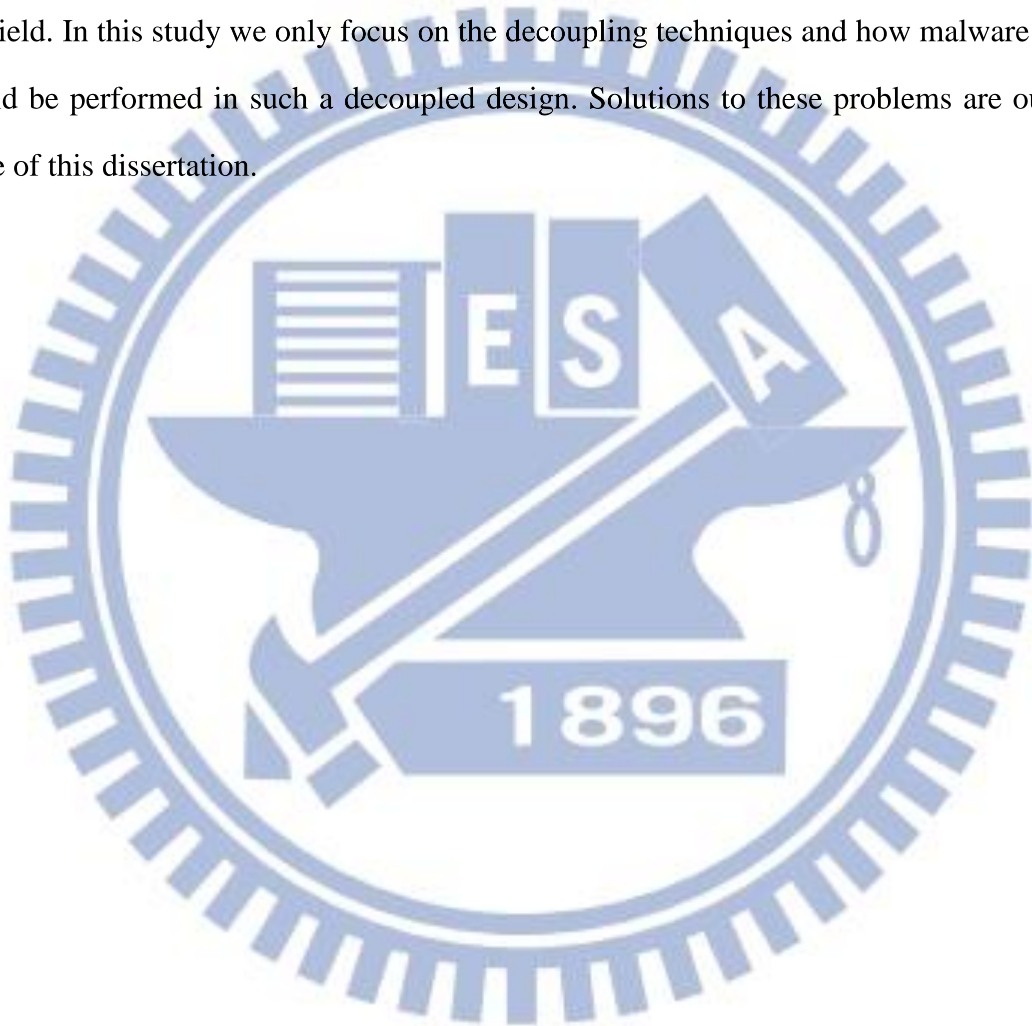
Another potential problem in current implementation is that the location (the EIP column in Table 5) of discovered probes may not be applicable directly to the target system since it may vary due to non-deterministic memory layout. This issue can be resolved by implementing the probe trigger with code-pattern matching instead of EIP-matching. It can be also optimized back to EIP-matching through scanning for the code patterns beforehand.

The collected probes provide locations suitable for placing triggers to activate the corresponding VMI monitor. Theoretically, the generated probe candidates are applicable not only to this emulator, but also to all systems with the identical OS kernel installed, as they are supposed to share the same control flows. The inferred data dereferencing steps should be also applicable as long as the target system uses the same kernel. In addition, their applicability is not interfered by the type of virtualization (emulated, virtualized, etc), either. In practice, however, even with the identical OS kernel, two machines with different hardware configurations can still lead to partially inconsistent control flows due to the divergence of their device drivers. Currently, ProbeBuilder does not differentiate between the probe candidates located in driver modules and those in the main kernel body. It simply attempts to discover as many probe candidates as possible.

On the other hand, taint analysis has its own limitation. Although information flow tracking is a powerful technique for malware analysis, it suffers from certain well-known problems [59], namely the under-tainting and overtainting issues. To prevent SWIFT from being laundered by the circumvention, we taint all memory-writing operations performed by a code block existing in a tainted memory region as stated in the "Dirty Code Execution" paragraph in section 6. The over-tainting effects caused by the approach above was compensated by the conservative index-register tainting proposed in subsection 4.1.3, which only propagate information flow for 1-byte or 2-bytes indirect memory reading from index

register to the destination register. However, a crafty malware may still circumvent the approach above with return-to-libc or return-oriented programing techniques, since they do not introduce any dirty code blocks.

Another possible way to elude information flow tracking is through control-flow. The issue has been discussed [10]. In addition, malware can hide their intentions with time-bomb or trigger-based behaviors. Revealing such behaviors is a well-known and very hard problem in this field. In this study we only focus on the decoupling techniques and how malware analysis should be performed in such a decoupled design. Solutions to these problems are out of the scope of this dissertation.



Chapter 8

Conclusion

This dissertation introduces ProbeBuilder, a powerful framework to automate the probe construction process in the implementation of any VMI-based systems. To our knowledge, ProbeBuilder is the first system proposed to automatically uncover the opaque chaining relations between undocumented kernel data structures. Through recursively walking through the pointers in the guest memory, potential probe locations and data dereference are collected during the emulation process. With the control flow extracted from the kernel image, the proposed refinement algorithms eliminate non-dedicated probe candidates, producing probes of good quality. Our experiment shows that ProbeBuilder only needs 7 minutes for simpler behaviors, and 167 minutes for a complicated behavior like creating probe candidates for process creation. Although ProbeBuilder is based on a heuristic multi-run refining process, the experiment shows that the probes generated by ProbeBuilder can capture all events that conventional monitors captured. Only a small number of false positives and irrelevant data (0.21%) are generated. Furthermore, the generated probes can be practically applied to behaviors profiling for both user-space and kernel-space activities. ProbeBuilder works in a black-box paradigm, automatically generating code snippets of probes for KVM, Xen, and QEMU. Considering the effort and time spent in the conventional reverse-engineering way, ProbeBuilder is very effective in automatically generating probes. Developers of VM-based analysis tools can directly benefit from the deployment of ProbeBuilder.

ProbeBuilder utilizes dynamic taint tracking to label data of interest. In order to provide

practical speed during analysis, a decoupled design of system-wide information flow tracking, SWIFT, is presented to shift the heavy overhead imposed by the analysis process onto another processor core. Unlike previous DIFT-capable system emulators injecting analysis routines directly into generated code blocks, our design extracts information flows only at translation phase, and therefore analysis to be performed on extracted information flows can be carried out by different threads. To further improve the analysis performance, two optimization techniques are proposed herein to eliminate unnecessary message exchanges between the emulator and the helper executing analysis routines. Compared with conventional interleaved design, SWIFT operates 1.82~3.22 times faster on common workloads. It runs 2.74~7.48 times faster than the interleaved design does in PassMark Performance Test 6.0.

To evaluate the effectiveness, a malware behavior analysis platform is implemented based on SWIFT. Due to the feature of system-wide tracking, it successfully detected contaminated information flows spreading into file systems, processes, registry, and network interfaces. Based on the decoupled design and optimizations, the system can generate comprehensive reports on malware analysis at a much higher speed than previous research.

Bibliographies

[1] C. Willems, T. Holz and F. Freiling, "Toward Automated Dynamic Malware Analysis Using CWSandbox," *Security & Privacy, IEEE* , vol.5, no.2, pp.32,39, March-April 2007 doi: 10.1109/MSP.2007.45

[2] U. Bayer, C. Kruegel and E. Kirda. “TTanalyze: A Tool for Analyzing Malware,” *In Proc. of the 15th Annual Conference European Institute for Computer Antivirus Research, EICAR,* 2006.

[3] S.P. Shieh and V. D. Gligor, “On a Pattern-Oriented Model for Intrusion Detection,” *IEEE Transactions on Data and Knowledge Engineering*, Vol. 9, No. 4, pp. 661–667, August 1997.

[4] S.P. Shieh and V.D. Gligor, “Detecting Illicit Information Leakage in Operating Systems,” *Journal of Computer Security*, pp. 123–148, April 1997.

[5] S.P. Shieh, and V.D. Gligor, “A Pattern-Oriented Intrusion Detection Model and Its Applications," *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, California, (May 7-9, 1991), pp 327-342.

[6] S.P. Shieh, and V.D. Gligor, “Auditing the Use of Covert Storage Channels in Secure Systems," *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, California, (May 1990), pp 285-295.

[7] F. Bellard, “QEMU, a fast and portable dynamic translator,” *Proc. annual USENIX Annual Technical Conference (ATEC '05)*

[8] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, “PIN: a binary instrumentation tool for computer architecture research and education,” *Proc. the ACM workshop on Computer architecture education*: held in conjunction with *the 31st International Symposium on Computer Architecture (WCAE '04)*, article 22, 2004, doi: 10.1145/1275571.1275600.

[9] N. Nethercote and J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” *Proc. ACM SIGPLAN conference on Programming language design and implementation (PLDI '07)*, pp. 89-100, 2007, doi: 10.1145/1250734.1250746.

[10] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, “Panorama: capturing system-wide information flow for malware detection and analysis,” *Proc. the 14th ACM conference on Computer and communications security (CCS '07)*, pp. 116-127, 2007, doi:10.1145/1315245.1315261.

[11] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham., " Vigilante: End-to-end containment of Internet worm epidemics," *ACM Trans. Computer Systems*, vol. 26, issue 4, Dec 2008, doi: 10.1145/1455258.1455259

[12] J. Newsome and D. Song, “Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software,” *Proc. ISOC conference on Network and Distributed System Security Symposium (NDSS '05).*

[13] G. Portokalidis, A. Slowinska, and H. BosH, “Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation,” *Proc. the ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '06)*, pp. 15-27, 2006, doi: 10.1145/1217935.1217938.

[14] H. Kannan, M. Dalton, and C. Kozyrakis, "Decoupling Dynamic Information Flow Tracking with a dedicated coprocessor," *IEEE/IFIP International Conference on Dependable Systems & Networks (DSN '09)*, pp.105-114, June 29 2009-July 2 2009, doi: 10.1109/DSN.2009.5270347.

[15] S. Chen, M. Kozuch, P. B. Gibbons, M. Ryan, T. Strigkos, T. C. Mowry, O. Ruwase, E. Vlachos, B. Falsafi, and V. Ramachandran, “Flexible Hardware Acceleration for Instruction-Grain Lifeguards,” *J. IEEE Micro*, vol 29, issue 1, pp. 62-72, 2009, doi: 10.1109/MM.2009.6.

[16] E. Vlachos, M. L. Goodstein, M. A. Kozuch, S. Chen, B. Falsafi, P. B. Gibbons, and T. C. Mowry, "ParaLog: enabling and accelerating online parallel monitoring of multithreaded applications," *Proc. ACM the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems (ASPLOS '10)*, 2010, pp. 271-284, doi:10.1145/1736020.1736051.

[17] O. Ruwase, S. Chen, P. B. Gibbons, and T. C. Mowry, "Decoupled lifeguards: enabling path optimizations for dynamic correctness checking tools," *Proc. ACM SIGPLAN conference on Programming language design and implementation (PLDI '10)*, pp. 25-35, doi:10.1145/1806596.1806600

[18] P. Saxena, R. Sekar, and V. Puranik, "Efficient fine-grained binary instrumentation with applications to taint-tracking," *Proc. the 6th annual IEEE/ACM international symposium on Code generation and optimization (CGO '08)*, pp. 74-83, doi:10.1145/1356058.1356069

[19] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. "LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks," *Proc. the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*, pp. 135-148, doi:10.1109/MICRO.2006.29.

[20] V. Nagarajan, H-S.Kim, Y.Wu and R. Gupta, "Dynamic Information Flow Tracking on Multicores," *Workshop on Interaction between Compilers and Computer Architectures*, Feb. 2008.

[21] P. Bravo and D. F. García. "Rootkits Survey: A concealment story,“ http://www.pablobravo.com/files/survey.pdf

[22] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi. "Secure in-VM monitoring using hardware virtualization," In *Proc. of the 16th ACM conference on Computer and communications security (CCS '09)*. ACM, New York, NY, USA, 477-487. DOI=10.1145/1653662.1653720

[23] B.D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An Architecture for Secure Active Monitoring Using Virtualization," *IEEE Symposium on Security and Privacy, S&P*, 2008, pp.233,247, 18-22 May 2008. doi: 10.1109/SP.2008.24

[24] F. Baiardi, D. Maggiari, D. Sgandurra, and F. Tamberi. "PsycoTrace: Virtual and Transparent Monitoring of a Process Self," In *Proc. of the 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2009, pp.393,397, 18-20 Feb. 2009. doi: 10.1109/PDP.2009.45

[25] Z. Gu, Z. Deng, D. Xu, and X. Jiang, "Process Implanting: A New Active Introspection Framework for Virtualization," *IEEE Symposium on the 30th Reliable Distributed Systems, SRDS*, 2011, pp.147,156, 4-7 Oct. 2011 doi: 10.1109/SRDS.2011.26

[26] L Martignoni, R Paleari, and D Bruschi, "A Framework for Be-havior-Based Malware Analysis in the Cloud," In *Proc. of the 5th International Conference on Information Systems Security (ICISS '09)*, Springer-Verlag, Berlin, Heidelberg, 178-192. DOI=10.1007/978-3-642-10772-6_14

[27] X. Jiang, X. Wang, and D. Xu. "Stealthy malware detection and monitoring through VMM-based "out-of-the-box" semantic view reconstruction," *ACM Trans. Inf. Syst. Secur.* 13, 2, Article 12 (March 2010), 28 pages. DOI=10.1145/1698750.1698752

[28] G. Xiang, H. Jin, D. Zou, X. Zhang, S. Wen, and F. Zhao, "VMDriver: A Driver-Based Monitoring Mechanism for Virtualization," *IEEE Symposium on the 29th Reliable Distributed Systems, RDS*, 2010, pp.72,81, Oct. 31 2010-Nov. 3 2010 doi: 10.1109/SRDS.2010.38

[29] B. Li, J. Li; T. Wo, C. Hu; L. Zhong, "A VMM-Based System Call Interposition Framework for Program Monitoring," *IEEE International Conference on the 16th Parallel and Distributed Systems, ICPADS*, 2010, pp.706,711, 8-10 Dec. 2010. doi: 10.1109/ICPADS.2010.53

[30] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection," *IEEE Symposium on Security and Privacy, S&P, 2011*, pp.297,312, 22-25 May 2011 doi: 10.1109/SP.2011.11

[31] T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," In *Proc. Network and Distributed Systems Security Symposium, NDSS*, 2003, pp 191-206. doi: 10.1.1.11.8367

[32] Y. Fu and Z. Lin, "Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection," *IEEE Symposium on Security and Privacy, S&P*, 2012, pp.586,600, 20-23 May 2012. doi: 10.1109/SP.2012.40

[33] K. Nance, M. Bishop, and B. Hay, "Virtual Machine Introspection: Observation or Interference?," *IEEE Security & Privacy*, vol.6, no.5, pp.32,37, Sept.-Oct. 2008. doi: 10.1109/MSP.2008.134

[34] B. Hay and K. Nance, "Forensics examination of volatile system data using virtual introspection," *SIGOPS Oper. Syst*. Rev. 42, 3 (April 2008), 74-82. DOI=10.1145/1368506.1368517

[35] Z. Wang, X. Jiang, W. Cui, and X. Wang, "Countering Persistent Kernel Rootkits through Systematic Hook Discovery," In *Proc. of the 11th international symposium on Recent Advances in Intrusion Detection (RAID '08)*, pp 21-38. DOI=10.1007/978-3-540-87403-4_2

[36] Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering kernel rootkits with lightweight hook protection," In *Proc. of the 16th ACM conference on Computer and communications security, CCS*, 2009. ACM, New York, NY, USA, 545-554. DOI=10.1145/1653662.1653728

[37] B. Dolan-Gavitt, T. Leek, J. Hodosh, and W. Lee, "Tappan Zee (North) Bridge: Mining Memory Accesses for Introspection," In *Proc. of the 20th ACM conference on Computer*

and communications security, CCS, 2013. ACM, New York, NY, USA, 839-850. DOI=10.1145/2508859.2516697

[38] J. R. Crandall, S. F. Wu, and F. T. Chong, “Minos: Architectural support for protecting control data,” *ACM Trans. Architecture and Code Optimization,* vol. 3, issue 4, pp. 359-389, Dec 2006, doi:10.1145/1187976.1187977.

[39] P. P. Bungale and C. K. Luk, “PinOS: a programmable framework for whole-system dynamic instrumentation,” *Proc. the 3rd ACM international conference on virtual execution environments (VEE '07)*, pp. 137-147, doi:10.1145/1254810.1254830.

[40] J. Chow, T. Garfinkel, and P. M. Chen, “Decoupling dynamic program analysis from execution in virtual environments,” *USENIX Annual Technical Conference (USENIX '08)*, 2008

[41] M. Dalton, H. Kannan, and C. Kozyrakis, “Raksha: a flexible information flow architecture for software security,” *Proc. The 34th ACM annual international symposium on Computer architecture (ISCA '07)*, pp. 482-493, doi:10.1145/1273440.1250722.

[42] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "FlexiTaint: A programmable accelerator for dynamic taint propagation," *IEEE 14th International Symposium on High Performance Computer Architecture (HPCA ’08)*, 2008, pp.173-184, 16-20 Feb 2008, doi:10.1109/HPCA.2008.4658637.

[43] E. Bosman, A. Slowinska, and H. Bos, “Minemu: The World’s Fastest Taint Tracker,” *Symposium on Recent Advances in Intrusion Detection (RAID ’11)*, 2011

[44] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand, “Practical Taint-Based Protection using Demand Emulation,” *Proc. the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06),* pp. 29-41, doi:10.1145/1217935.1217939

[45] A. Ermolinskiy, S. Katti, S. Shenker, L. Fowler, and M. McCauley, “Towards Practical Taint Tracking,” Technical Report No. UCB/EECS-2010-92, 2010

[46] T. N. Bui and C. Jones. 1992. Finding good approximate vertex and edge partitions is NP-hard. *Inf. Process. Lett.* 42, 3 (May 1992), 153-159. DOI=10.1016/0020-0190(92)90140-Q http://dx.doi.org/10.1016/0020-0190(92)90140-Q

[47] M. Howard, D. LeBlanc, *Writing Secure Code (Second Edition)*, Microsoft Press, pp. 262-265, 2003.

[48] S. Sathyanarayan, P. Kohli, and B. Bruhadeshwar, "Signature Generation and Detection of Malware Families," in *Proc. of the 13th Australasian conference on Information Security and Privacy*, pp. 336-349.

[49] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in *Proc. of the 2005 IEEE Symposium on Security and Privacy*, Oakland, May 2005.

[50] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *Proc. of the ACM SIGSOFT symposium on The foundations of software engineering (FSE)*, pages 5–14, 2007.

[51] X. Wang, W. Yu, A. Champion, X. Fu, and D. Xuan, "Detecting Worms via Mining Dynamic Program Execution," in *Proc. of the Third International Conference on Security and Privacy in Communication Networks and the Workshops, SecureComm*, pages412-421, Nice, 2007.

[52] F. Maggi, M. Matteucci, and S. Zanero, "Detecting intrusions through system call sequence and argument analysis," in *IEEE Transactions on Dependable and Secure Computing (TDSC)*, October 2010.

[53] G. Jacob, H. Debar, and E. Filiol, "Malware detection using attribute-automata to parse abstract behavioral descriptions," *CoRR*, abs/0902.0322, 2009.

[54] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Effective and Efficient Malware Detection at the End Host," in *Proc. of the 18th conference on USENIX security symposium,* 2009.

[55] Srizbi botnet. http://en.wikipedia.org/wiki/Srizbi_botnet

[56] A. Marzal and E. Vidal, "Computation of Normalized Edit Distance and Applications," in *IEEE Transactions on Pattern Analysis and Machine Intelligence,* Volume 15 Issue 9, September 1993.

[57] J. Lin, "Divergence measures based on the Shannon entropy," in *IEEE Transactions on Information theory*. Volume 37. p145-151. 1991.

[58] U. Bayer, P. M. Comparetti, C. Hlauscheck, C. Kruegel, and E. Kirda, "Scalable, Behavior-Based Malware Clustering," in *Proc. of the 16th Symposium on Network and Distributed System Security*, 2009.

[59] A. Slowinska and H. Bos, "Pointless Tainting? Evaluating the Practicality of Pointer Tainting," *Proc. the ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '09),* April 2009, Nuremberg, Germany

Autobiography

Chi-Wei Wang (王繼偉) received his B.S. in the Department of Computer Science and Information Engineering, National Chiao Tung University, Taiwan. In his college time, he was recommended by the CS department to participate the exchange student program, which is held by Electrical Engineering and Computer Science Undergraduate Honors Program, NCTU, and studied in the CS department of University of Illinois at Urbana-Champaign for one semester. He was also recommended to participate the visiting scholar program, which is sponsored by iCAST, greatly contributing to the development of SWOON, a testbed for secure wireless overlay networks in UC Berkeley. He has been very active in the malicious software analysis community, and has received many awards. In 2007, he and his team developed a technique, combining the big data provided by Microsoft Research Asia and browsing history, to personalize the search engine result for an individual user. This technique earned them the 1st place in the Microsoft Cross-Strait Innovation Contest among more than 50 teams from Taiwan and China. He also led his team and achieved 1st place in the Wargame Contest held by Hacks in Taiwan Conference (HITCON) in both 2010 and 2011. In 2008, 2011, and 2013, he and his team achieved the 2nd, 2nd, and 3rd places respectively in Interuniversity Information Security Technical Ability Contest, held by Institute for Information Industry, Taiwan. His research interests include network security, software security, and operating systems.