MrKIP: Rootkit Recognition with Kernel Function Invocation Pattern

CHI-WEI WANG, CHONG-KUAN CHEN, CHIA-WEI WANG, SHIUHPYNG WINSTON SHIEH
Network Security Laboratory of Department of Computer Science
National Chiao Tung University
Hsinchu, Taiwan

E-mail: cwwangabc@gmail.com, ckchen@cs.nctu.edu.tw, glacier314@gmail.com,

ssp@cs.nctu.edu.tw

Existing mechanisms tracing user-level activities such as system calls and APIs can be circumvented by the kernel-level rootkits. In this paper, a novel system, MrKIP, is proposed to recognize rootkits based on their kernel-level activities. Our scheme semi-automatically generates suitable locations for analysts to implement checkpoints, which are used to profile kernel-space activities. Then, collected rootkits are executed in an emulator with these checkpoints for behavior profiling. The collected behaviors are clustered and used for model construction. The constructed model can be used to recognize new variants of rootkit families. Our scheme differs from conventional tracers due to its ability to cover kernel-space malware and the whole-system scope. In addition, monitoring at the kernel level raises high barrier for malware to evade, since all tasks are eventually executed through the basic kernel functions.

Keywords: **Rootkit Recognition, Malware Analysis, Virtual Machine Introspection, Data Mining**

1. INTRODUCTION

To efficiently develop cure and detection for modern rapid-growing malware, automatic behavior analysis is an important measure in the whole analysis process. Through behavior analysis, malware instances with similar behaviors can be classified into clusters, which dramatically reduce the effort for human analysts [3]. The effectiveness is led by the following two facts. First of all, not all malware (as a matter of fact, only a few) are written from scratch. Most variants are simply modifications or upgrades of the original ones. Secondly, advanced polymorphism or metamorphism techniques deform the code appearance but not the program behavior. In fact, techniques for malware pattern extraction and recognition had been discussed in previous research [12][13][18][19][22][25][31]. 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. Although virtualization-based inspection [39][40][41] may be used to resolve the aforementioned issues of privilege-escalation, they still have to face the following challenges. A Kernel-level program directly invokes kernel-level functions to accomplish its tasks, relying on neither system calls nor user-level APIs. A representative example will be Trojan.Srizbi, which is responsible for 40% of all the spam on the Internet in 2008 [32]. Trojan.Srizbi executes all its functionalities such as hiding files and sending botnet traffic in the kernel space. Without monitoring mechanism for kernel-space behaviors, such malware can never be profiled accurately. Although recent studies advanced their interests at kernel-level rootkits [33][34][35], they devote to revealing hooking or DKOM behaviors and provide limited functionality for general behavior profiling.

To cope with problems above, in this paper we propose a novel system to recognize malware with invocation pattern of kernel functions. The proposed system is named MrKIP. Through hooking internal functions of the OS kernel, MrKIP can monitor kernel-level activities. This step requires massive reversing effort since we target on the Windows operating system, which is a closed-source OS. A method is devised to semi-automatically identify possible locations to place checkpoints. The discovered hooks are implemented into a PC emulator equipped with system-level taint tracking capability. As long as any arguments of the hooked 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.

The major features of MrKIP can be summarized as follows.

- A novel approach is proposed to profiling program behaviors with invocation of in-kernel function instead of user space APIs or system calls. This feature ensures that our method is able to deal with malware existing purely in kernel-space.
- To distinguish those events actually originated from the subject program from normal background activities, MrKIP adopts a system-wide taint tracker to capture tainted arguments. This feature refines the collected data, producing more an accurate pattern for recognition.
- A semi-automatic module is devised to identify suitable locations to place hooks. This feature eases the pain for hook developers.
- 351 kernel-level rootkit instances are fed into our pattern generator, and then another 185 rootkit instances are matched against the constructed patterns. The experiment shows that 80% of 185 test subjects can be efficiently classified into the correctly families.

2. RELATED WORK

Testing whether a suspicious program resembles known malware is an extremely important procedure in malware research and defense. Various techniques have been

proposed and developed. Depending on the attributes used to measure the similarity between programs, these techniques can be classified into static analysis and dynamic analysis. In static analysis, subjects are examined, and features are extracted without actual execution. The approach proposed by Bilar et al. [9] distinguishes malware and benign programs with statistical analysis on op-code distributions. Tian et al. [10] measure the code length of each function in the program and use their frequency as the feature for malware classification. Another automated malware classification system [11] is proposed which is based on the function length and the printable strings extracted in the executable file. Sathyanarayan et al. [12] suggested generating signatures and for malware families with their imported API. Their system scans the executable file to extract the frequency of invocations of critical APIs to evaluate the likelihood of malice. More advanced static analysis [20] utilizes function call graph to perform malware classification. The advantage of static analysis is its efficiency since no execution is needed. However, the lack of runtime information also makes them ineffective against advanced malware crafting techniques such as polymorphism or metamorphism. Although methods [26][30] are proposed to identify the presence of such engines in programs, the behavior profiles cannot be correctly extracted if the executable images are packed or encrypted.

Dai et al. [5] proposed a method executing the program inside a debugger of single-step running mode to collect its instruction trace. The instruction trace is then decomposed into basic blocks of abstract op-codes, which are processed by data mining techniques to discover common patterns in malware. Recognizing malware with instruction patterns can still suffer from metamorphism. Although various techniques [13] are used to solve these problems, advanced metamorphism such as instruction replacement or reordered memory access can still disable them. There are also systems identifying malware with patterns extracted in network traffics such as HTTP requests [14], DNS queries [15], and other communication channels [16]. Since these network packets cannot be sent without invoking the functions provided by the driver, these functionalities are automatically included in our scheme.

Identifying malware by their interactions with the execution environment is not a novel idea. System calls invoked by the monitored process can be collected to build a graph-based behavior pattern [18][19]. By comparing similarity between graphs, malware can be clustered and identified. Higher accuracy can be obtained with patterns capturing more details. In other approaches [4][22][23][24][25], not only the system calls but also the arguments passed into are considered. With different modeling techniques (text reports, graphs, automata, etc.), the collected information are transformed to identifying patterns for recognition. However, all the works above confine their discussion to the monitored user-level process. Compared with previous research, our scheme provides the advantage of comprehensiveness, which leads to higher accuracy, by utilizing machine-level system-wide taint tracking.

Due to the appearance of kernel-level rootkits, there are also studies extending their coverage to kernel space. To detect the presence of such rootkits, techniques proposed in PoKeR [33], HookMap [36], and K-Tracer [35] are quite effective. However, these methods are devoted to reveal information hiding and system data manipulation instead of a general and comprehensive behavior profile. A partial solution to this problem is given by Rkprofiler [34]. Based on QEMU emulator, Rkprofiler determines whether code to be executed in kernel space is malicious by matching them against pre-generated

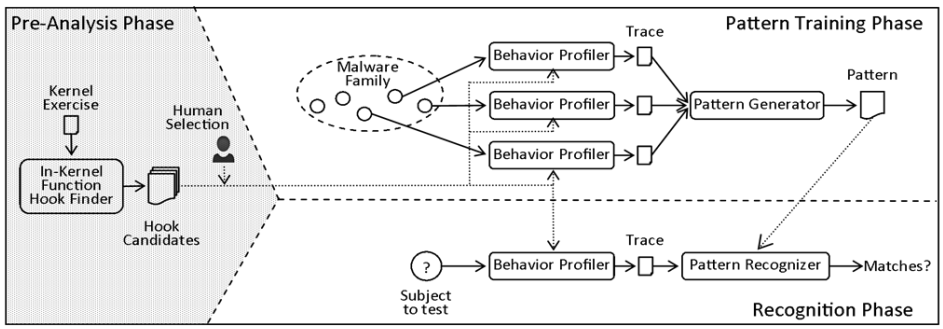

Figure 1 An overview of the system architecture of MrKIP.

hash values of trusted code. By recording all control flow transferring between trusted and untrusted code, the interaction between the suspicious module and the rest of the kernel can be profiled. However, Rkprofiler differs from MrKIP in several aspects. First of all, our system is a generic approach to capturing the behaviors of both user- and kernel-space malware, while Rkprofiler is not. Secondly, neglecting the arguments passed into, Rkprofiler records only which kernel functions are called. Thirdly, MrKIP consists of not only the profiling mechanism but also the pattern building and recognizing parts. All these reasons make the functionality of Rkprofiler a proper subset of MrKIP.

3. PROPOSED SCHEME

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 1, 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.

To avoid being detected by malware, the monitoring functionality should be realized with the so-called “out-of-box” hooking [4]. 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. To ease the pain of finding and maintaining these out-of-box hooks, in this study a semi-automatic method is developed to discover hook candidates satisfying the two constraints. As indicated in Figure 1, the method is implemented in the in-kernel function **HookFinder**, which is executed in the pre-analysis phase. Repetitively commanding the system to perform certain tasks such as packet sending and file opening, the **HookFinder** automatically identifies the instructions accessing memory regions with the argument data (e.g. the packet

```
;START OF FUNCTION CHUNK FOR _NtQueryValueKey

loc_4ABF29:
mov     eax, [ebp+0xC]
mov     ecx, [eax]
mov     dword ptr [ebp+String1.Length], ecx
mov     eax, [eax+4]
mov     [ebp+String1.Buffer], eax
mov     edi, [ebp+ResultLength]
jmp     loc_49C049
; END OF FUNCTION CHUNK FOR _NtQueryValueKey
```

EBP, +0xC, +0x0 → string length
EBP, +0xC, +0x4 → string buffer

Figure 2 Path from stack pointer to member field of a data structure.

content or the file I/O buffer). For each identified instruction, a backward-slicing procedure is applied to the pointer variable to check whether a stable sequence of pointer-dereferencing steps exists. If so, the location of head instruction in the backward slice and the associated pointer structure is collected. Note that the final decision on usability of the collected checkpoints still requires human assistance. Yet, tremendous efforts on reversing the kernel are already saved in the development of MrKIP. In the following, we introduce each component of MrKIP.

3.1 Hook Finder

As stated in previously, the goal of the **HookFinder** is to discover suitable kernel functions to place checkpoints to profile certain behaviors. To achieve this goal, we exercise the kernel to record the execution paths. For instance, to place a hook which monitors file system I/O operation, we execute a program to write and read a file in user space. By recording the consequent control flow transfer, the execution path can be therefore established. Since we are building a profiling environment for both kernel and user-level programs, it is reasonable to refine the scope to the nodes of the recorded path. To further refine the execution path, the process can be repeated multiple times to eliminate the nodes with random occurrence. However, the execution path will still contain too many code blocks for choosing.

Algorithm 1 Hook-Finding

Requires:
A string pattern S
Depth limit D

```
In the guest OS in the emulator, invoke a user-level API f(..,S,…)
The emulator collects a set E =
{(e,i) | the instruction, i, on e accesses memory region containing S}
foreach (e,i) ∈ E
    d ← 0 , p ← <.>
    while d < D
        if i matches format "mov rd, [rs + offset]"
            p ← <offset> || p
            if rs ∈ {ESP, EBP}
                p ← <rs> || p
                break
        else
            d ← d + 1
        i ← Perform backward-slicing on source operands of i
        if i = Nil
            break
    if d < D ∧ p[0] ∈ {ESP, EBP}
        output p
```

In addition, not all nodes in the execution path collected in the above procedure are suitable for hooking. Although obtaining such a checkpoint avails behavior detection, to construct a complete behavior profiling system requires more than that. Let's use the file system I/O as an example again. In a file I/O operation, in addition to its occurrence we are more interested in determining the file being accessed. For a writing operation, the data being written are also of interest. Capturing the information on an upper abstraction layer such as API and system call is not an issue, however it is difficult to do so in the kernel of a closed-source OS kernel. Figure 2 shows an example of data path from the stack frame pointer to a member field of a data structure. The member is not located in the stack, but in a data object allocated on the heap. Yet, the pointer to the data object is stored on the stack, and it should be referenced by [ebp+0xc]. This is merely a trivial case illustrating the idea of data path. In a compounded or chained data structures, the data path to important data can be much more complicated. Figuring out these data paths is necessary since they are valuable information for behavior profiling.

Algorithm 1 describes the procedure of **HookFinder**. The procedure starts with executing an upper-layer API call, say f, in the guest system. In addition, the argument interests us is intentionally given with a special string S. For instance, the file path of the OpenFile() API will be the ideal argument to adopt. Or, the data buffer of the WriteFile() API is another suitable usage. After the API is successfully invoked, the guest OS kernel will execute the task, and our emulator will examine each executed instruction, searching for those accessing memory containing the string pattern S. The above two procedures are described on lines 1 and 2 of Algorithm 1. The string pattern S is devised by us, and hence it can be so unique that no other data in the guest can match with it. The above two steps can effectively eliminate instructions irrelevant to the subject behavior, quickly reducing the problem size.

For each instruction collected into E, lines 4-15 perform backward slicing on the register addressing the source memory operand. On line 4, the depth-indicating variable d is initiated with zero and the list structure p is emptied. Line 5 checks if the current depth has exceeded the depth limit D. This limit is based on the assumption that a usable data path should have limited lengths. It is due to the programming convention that only necessary data should be passed into functions as input variables. Note that the assumption above does not deny the existence of long data referencing paths since walking a linked list can easily lead to a quite long series of referencing steps. However, such paths are not static and hence cannot be used in our application. Line 6 checks if the instruction uses an index pointer to address its memory source operand. If so, the immediate offset is inserted into p. The process stops when the stack pointer or the frame pointer is reached, since they usually point to the base address of the activation record of routines. Line 13 performs backward-slicing on the instruction, looking for the last instruction modifying the source operand of instruction i.

Surely our approach does have its limitation. First of all, human assistance is still required to select the proper string pattern S. Secondly, the current implementation only considers data paths originating from the stack. Thirdly, the proposed approach provides little benefit for information which does not originate from user space. For example, the PID of a process being created cannot be acquired with the approach because it is generated by the operating system itself and we predict the string pattern S in advance. In spite of the limitation listed above, the approach indeed saved tremendous effort of reversing

the kernel in the development process of MrKIP.

3.2 Behavior Profiler

The **BehaviorProfiler** is a major component deployed in both the training phase and the recognition phase. It serves to generate the behavior trace of the subject program. Below its work flow is given.

The profiler is constructed by installing the hooks collected in the pre-analysis phase in the whole-system taint tracker, which propagates taint across CPU registers, memory, and hard-disks. 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 hook is hit by the execution, its data path is walked through to check whether it reaches tainted information.

A hook 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 $1 \leqq k \leqq n$. 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 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 hook $H = (n, loc_H, IT_H)$ 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 hook 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}

Figure 3. 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:WinXP Pro Service Pack 3mj_5_mn_1Ox0dOx0aHost: www.think-adz.comOx0dOx0a..."	raw

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 very 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.

Figure 3 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.

In the framework above, the hook H serves the most important purpose of extracting information from the machine state. Although the **HookFinder** introduced in the previous subsection provides useful clues (namely the table *paths*), the gap between *paths*[*pc*] acquired and a hook $H = (n, loc_H, IT_H)$ must be filled by human. First of all, the data referencing sequence given by *paths*[*pc*] only specifies the beginning address of the $data_B$. How to infer the length of $data_B$, which is necessary to retrieve the full $data_B$, is out of the capability of the **HookFinder**. In addition, which type T^H_k should be assigned cannot be decided by *paths*[*pc*] either. In our current design, this step is still achieved with human effort to reverse engineer the binary code locating near loc_H. However, it is quite easy for human to "guess" the meaning of data listed in the *data* column without knowing its *type*. Therefore, we can view the **HookFinder** as a powerful assisting tool for reverse engineering.

3.3 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.

3.3.1 Pattern Generator

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 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.

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

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

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

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

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 [37]. 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 [38]. 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 2. 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 clustering algorithm invoked in line 7-8 further cluster behaviors according to their argument similarity. In line 9-14, the algorithm assigns an in-

teger 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 $C[b_k]$.

Algorithm 2 Behavior-To-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 $H \leftarrow$ A key-value map (mapping an address to a set of behaviors)
2 $C \leftarrow$ A key-value map (mapping a behavior to an integer)
3 $n \leftarrow 0$, $R \leftarrow \{\}$, $SS \leftarrow \{\}$
4 **foreach** sequence $s \in BS$
5 **foreach** behavior $b \in S$
6 $H[loc_b] \leftarrow H[loc_b] \cup b$ // Grouping behaviors by loc_b
7 **foreach** value $h \in H$
8 $cl \leftarrow$ Complete-Link-Cluster(h) // cl contains sets of behaviors
9 **foreach** $c \in cl$
10 **foreach** $b \in c$
11 $C[b] \leftarrow n$
12 $r \leftarrow$ pick b out of c and b minin $\sum_{b' \in c} \mathrm{D}(b, b')$
13 append r to R
14 $n \leftarrow n + 1$
15 **foreach** sequence $s : <b_1, b_2, \ldots, b_k> \in BS$
16 append $<C[b_1], C[b_2], \ldots, C[b_k]>$ to SS
17 **return** n, SS, R

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 P**atternRecognizer** 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.

3.4 Pattern Recognizer

The task of P**atternRecognizer** 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**.

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 3. 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 geomet-

Algorithm 3 Calculate-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.
$R = <r_1, r_2, \ldots, r_n>$ 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)
```

ric 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.

4. EVALUATION

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.

4.1 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

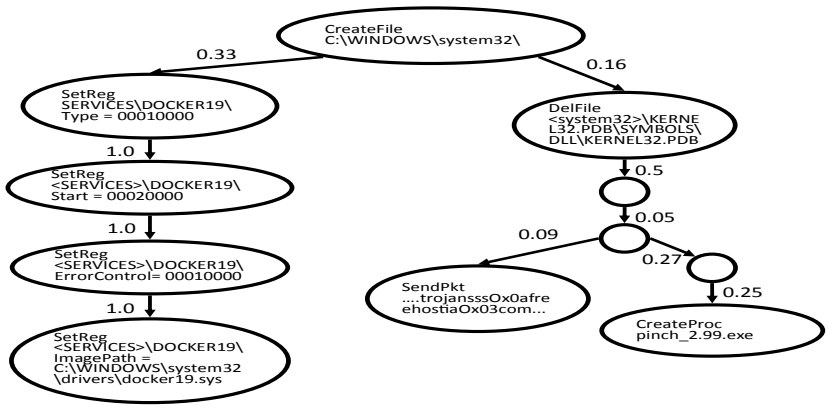

Figure 4. Constructed model for Srizbi.

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. The profile is listed in the Appendix. We can observe that both 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 4 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.

4.2 Performance

In this experiment we measure the time needed by MrKIP to construct model for rootkit families. The result is shown in Figure 5. Each point in the figure indicates a model constructed from a certain rootkit family by **PatternGenerator**, and the its coordinate (x,y) represents that there are x events (or behaviors) recorded in this rootkit family and it took **PatternGenerator** y seconds to analyze and form the HMM model**.** As predicted, a larger size of recorded behaviors will increase the time spent on the pattern construction. However, Figure 5(a) shows that families with the record size less than 4000 can be processed in less than one second. Even the family with the largest amount of behaviors (~20000) can be resolved in 20 seconds, as shown in Figure 5(b). This result shows that our method can be efficiently executed in a reasonable performance.

4.3 Effectiveness of Recognition

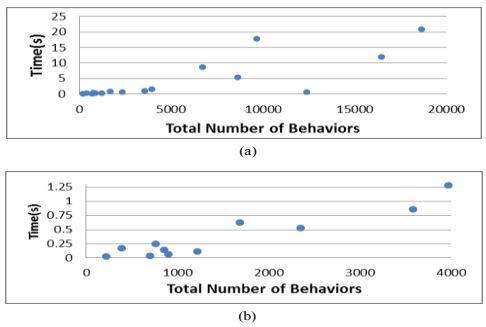

Figure 6. Time needed for model construction versus behavior size. (a) Size 0~4000 (b) Size 0~20000

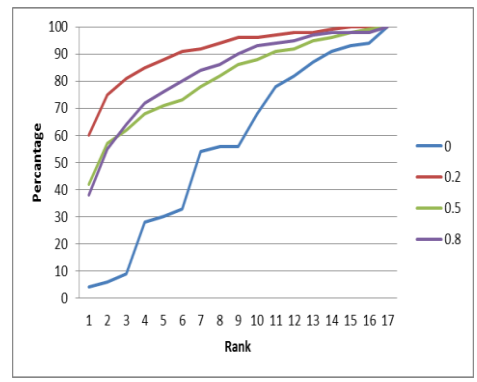

Figure 5. Cumulative ranking

To evaluate the effectiveness of **PatternRecognizer** of MrKIP, the next experiment compares the clustering result of MrKIP with the clustering done by commercial anti-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 instances 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.

The cumulative distribution of classification ranking is shown in Figure 6. 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. A steeper curve indicates more instances have lower rank numbers, which implies that 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% in-

stances 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.

5. CONCLUSION

In this paper, we proposed MrKIP, a rootkit recognition system based on kernel-level function invocation patterns. This system can semi-automatically assist analysts to identify suitable functions in the OS kernel for hook insertion, greatly reducing the effort to develop analysis systems. Unlike conventional user-level tracing tools, the generated kernel-level checkpoints can be used to profile kernel-level activities. We also proposed a HMM-based method to construct behavior models for rootkit families. The constructed model can be used to recognize other variants in the future. Our experiment shows that the profiling tool can accurately record activities of pure kernel-level trojan programs such as Srizbi. The pattern construction and recognition are also effective weapons for analysts to deal with new variants of rootkit families.

REFERENCES

1. G Data Malware Report Half-year report January-June 2010.
2. D. Perry, “Here comes the flood or end of the pattern file,” In *Virus Bulletin*, Ottawa, 2008.
3. E. Stinson, and J. C. Mitchell, ”Characterizing Bots' Remote Control Behavior,” in *Proc. of the 4th international conference on Detection of Intrusions and Malware and Vulnerability*, Lucerne, Switzerland, 2007.
4. U. Bayer, C. Kruegel, and E. Kirda , “TTAnalyze: A Tool for Analyzing Malware,” In *Proc. of the 15th European Institute for Computer Antivirus Research Annual Conference (EICAR)*, April 2006.
5. J. Dai, R. Guha, and J. Lee, “Efficient Virus Detection Using Dynamic Instruction Sequences,” *Journal of computers* 4(5), 405–414, May 2009.
6. ThreatExpert http://www.threatexpert.com/
7. M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song, “Dynamic spyware analysis,” in *Proc. of the USENIX Annual Technical Conference (ATC)*, June 2007.
8. A. Moser, C. Kruegel, and E. Kirda, “Exploring multiple execution paths for malware analysis,” in *Proc. of the IEEE Symposium on Security and Privacy*, p.231-245, May 20-23, 2007.
9. D. Bilar, “Fingerprinting malicious code through statistical opcode analysis,” in *Proc. of the 3rd International Conference on Global E-Security (ICGeS)*, London, UK, April 2007.
10. R. Tian, L. M. Batten, S. C. Versteeg, “Function Length as a Tool for Malware Classification,” in *Proc. of the 3rd International Conference on Malicious and Unwanted Soft-ware (MALWARE)*, pp.69-76, 2008.
11. R. Islam, R. Tian , L. Batten, and S. Versteeg. “Classification of Malware Based on String and Function Feature Selection,” in *Proc. of the 2010 Cybercrime and Trustworthy Computing Workshop*, July 2010.
12. 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.
13. 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.
14. R. Perdisci, W. Lee, and N. Feamster, “Behavioral clustering of http-based malware and signature generation using malicious network traces,” in *Proc. of the 7th USENIX conference on Networked Systems Design and Implementation (NSDI), April 2010.*
15. D. Plonka and P. Barford, “Context-Aware Clustering of DNS Query Traffic,” in *Proc. of the 8th ACM SIGCOMM conference on Internet Measurement*, October 2008.
16. G. Gu, R. Perdisci, J. Zhang, and W. Lee, “BotMiner: Clustering Analysis of Network Traffic for Protocol and Structure Independent Botnet Detection,” in *Proc. of the 17th USENIX Security Symposium*, 2008.
17. Y. Park, and D. Reeves, “Deriving Common Malware Behavior through Graph Clustering,” in *Proc. of the 6th ACM Symposium on Information, Computer and Communications Security (CCS)*, NY, USA, 2011.

18. 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.
19. 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.
20. X. Hu, T. C. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proc. of the 16th ACM conference on Computer and communications security (CCS)*, pages 611–620, 2009.
21. C. Willems, T. Holz, and F. Freiling, "Toward Automated Dynamic Malware Analysis Using CWSandbox,"in *IEEE Security and Privacy*, 5(2), 2007.
22. 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.
23. 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.
24. Y. Park, D. Reeves, V. Mulukutla, and B. Sundaravel, "Fast malware classification by automated behavioral graph matching," in *Proc. of the Workshop on Cyber Security and Information Intelligence Research*, 2010.
25. G. Jacob, H. Debar, and E. Filiol, "Malware detection using attribute-automata to parse abstract behavioral descriptions," CoRR, abs/0902.0322, 2009.
26. C. Krugel, E. Kirda, D. Mutz, W. K. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," In *RAID*, pages 207–226, 2005.
27. F. Bellard, "Qemu, a Fast and Portable Dynamic Translator," in *Usenix Annual Technical Conference*, 2005.
28. Jahmm - An HMM library http://www.run.montefiore.ulg.ac.be/~francois/software/jahmm/jahmmViz/
29. ClamAV http://www.clamav.net/lang/en/
30. Q. Zhang and D. S. Reeves, "Metaaware: Identifying meta-morphic malware," in *Proc. of the 23rd Annual Computer Security Applications Conference (ACSAC)*.
31. 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.
32. Srizbi botnet. http://en.wikipedia.org/wiki/Srizbi_botnet
33. R. Riley, X. Jiang, and D. Xu, "Multi-Aspect Profiling of Kernel Rootkit Behavior," in *Proc. of the Fourth ACM European Conference on Computer Systems.*
34. C. Xuan, J. Copeland, and R Beyah, "Toward Revealing Kernel Malware Behavior in Virtual Execution Environments," in *Proc. of the 12th International Symposium on Recent Advances in Intrusion Detection, RAID '09*.
35. A. Lanzi1, M. Sharif1, and W. Lee, "K-Tracer: A System for Extracting Kernel Malware Behavior," in *Proc. of the 16th Annual Network and Distributed System Security Symposium (NDSS'09)*, February 2009.
36. 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,2008.*
37. 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.
38. J. Lin, "Divergence measures based on the Shannon entropy," in *IEEE Transactions on Information theory*. Volume 37. p145-151. 1991.
39. X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection and monitoring through VMM-based "out-of-the-box" semantic view reconstruction," in *ACM Trans. Inf. Syst. Secur. 13, 2, Article 12*.
40. G. Xiang, H. Jin, D. Zou, X. Zhang, S. Wen, and F. Zhao, "VMDriver: A Driver-Based Monitoring Mechanism for Virtualization," in *Reliable Distributed Systems, 2010 29th IEEE Symposium on*, pp.72,81, Oct. 31 2010-Nov. 3 2010.
41. B. Li, J. Li; T. Wo, C. Hu; L. Zhong, "A VMM-Based System Call Interposition Framework for Program Monitoring," in *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pp.706,711, 8-10 Dec. 2010.

Chi-Wei Wang (王繼偉) is currently a Ph.D candidate in the Department of Computer Science and Information Engineering, National Chiao Tung University, Taiwan. He has been very active in the malicious software analysis community, and has received many awards. Recently, he led his team and achieved 1st place in the Wargame Contest held by Hacks in Taiwan Conference 2010 and 2011. He also received the 1st place in the Microsoft Cross-Strait Innovation Contest in 2007. His research interests include network security, software security, and operating systems.

Chong-Kuan Chen(陳仲寬) is a Ph.D student in Institute of Computer Science and Engineering at National Chiao Tung University, Taiwan. He graduated from Department of Computer Engineering at National Chiao Tung University in 2011. His research focuses on network security, malware analysis, reverse engineering and digital forensics.

Chia-Wei Wang(王嘉偉) is a PhD student in the Laboratory for Distributed System and Network Security at the National Chiao-Tung University, Taiwan. His research interests include the virtual machine security and the malware research, Win32 especially. Chiawei has a bachelor degree in computer science from National Sun-Yat Sen University, Taiwan. Contact him at cwwang.cs98g@g2.nctu.edu.tw

Shiuhpyng Winston Shieh (謝續平) received the M.S. and Ph.D. degrees in electrical and computer engineering from the University of Maryland, College Park, respectively. He is a Professor of the Department of Computer Science, National Chiao Tung University (NCTU), and the Director of Taiwan Information Security Center at NCTU. Dr. Shieh currently serves as the Chair of IEEE Reliability Society Taipei and Tainan Chapter, and an ACM SIGSAC Awards Committee member. He is also the Editor-in-Chief of IEEE Reliability Society Newsletter, an Associate Editor of IEEE Transactions on Reliability, IEEE Transactions on Dependable and Secure Computing, former editor of ACM Transactions on Information and System Security, Journal of Computer Security, Journal of Information Science and Engineering, Journal of Computers, and guest editor of IEEE Internet Computing, respectively. He was on the organizing committees of numerous conferences, such as Steering Committee Chair and Program Chair of ACM Symposium on Information, Computer and Communications Security. Recently he received the ACM Distinguished Scientist Award among the 41 recipients worldwide in 2010. He also received ACM Service Award for his contribution to ACM, and Distinguished Information Technology Award (presented by Vice President of Taiwan). His research interest includes reliability and security mechanisms, network and system security.