RATSCOPE: Recording and Reconstructing Missing RAT Semantic Behaviors for Forensic Analysis on Windows

Runqing Yang *, Xutong Chen *, Haitao Xu, Yueqiang Cheng, Chunlin Xiong, Linqi Ruan, Mohammad Kavousi, Zhenyuan Li, Liheng Xu, and Yan Chen, *Fellow, IEEE*

Abstract—Remote Access Trojan (RAT) attacks have become an extensively prevailing and serious threat to enterprise security. A forensic system targeting RAT attacks is needed to record and reconstruct fine-grained semantic behaviors of RATs. However, existing forensic systems suffer from various issues such as intrusive instrumentation, nontrivial recording overhead, and RAT behavior blindness. In this paper, we first conduct a large-scale study of a representative set of real-world RAT families active from 1999 to 2016. This is the first study to understand the landscape of RATs in the literature. Based on the study, we then propose RATSCOPE, an instrumentation-free RAT forensic system targeting Windows platform. Specifically, RATSCOPE offers an audit logging module to efficiently record system logs by leveraging Event Tracing for Windows (ETW), and provides a novel program behavior modeling technique to reconstruct semantic behaviors of RATs accurately. We implement a prototype of RATSCOPE and evaluate the recording overhead and the behavior identification accuracy. The results show that the audit logging module only incurs 3.7% runtime overhead on average. Our system can achieve around 90% true positive rate in the cross-family experiment, around 80% true positive rate in the two-year spanning temporal experiment, and near *zero* false positive rate.

Index Terms—Remote Access Trojan, Forensics, Windows

1 INTRODUCTION

FROM cyber theft of personal financial information to Advanced Persistent Threat (APT) attacks aiming at intellectual properties or critical infrastructures, nowadays RATs cause a wide range of damage to individual users, corporations, and governments [1], [2], [3]. However, there is still a lack of studies that attempt to understand the landscape of RATs and a lack of forensic systems targeting RAT attacks.

To understand RAT attacks, we conduct a large-scale study (see Section 2) of 53 RAT families starting active from 1999 to 2016 in terms of their workflow, functions equipped, and how their functions are implemented. To the best of our knowledge, this is the largest corpus of RAT families ever studied in academia, and we have made our collected RAT samples public on the GitHub.

Based on our study, we learn that one main difference between other malware and RATs is their operating mode after gaining a foothold on the target host. *A RAT is mainly working in an interactive mode.* Each of its action is synchronously controlled by remote attackers. For example, a RAT can be activated to start audio recording of the victim's surrounding environment or deactivated immediately after the remote attacker switches the button "Audio Record" on or off on the RAT's GUI control panel. In addition, unlike most other malware, the functionalities of a RAT can be clearly split up into tens of standalone functions, termed by us as Potential Harmful Functions (PHFs), e.g., Remote Camera and Audio Record. *Each PHF can be triggered at will by the attacker, depending on his/her intention at that specific time.* Furthermore, based on our comprehensive study, we learn that *90% of RATs only target Windows.*

RATs definitely represent a singular category of malware given that they are so significantly different from most other malware in terms of their operating mode and clear-cut functionalities. *Therefore, it is highly crucial and necessary to have an efficient forensics system on Windows to help understand the intent and ramification of RAT attacks, and make a quick incident response.*

In this paper, we present such a forensic system targeting RATs on Windows which leverages two categories of audit logs, i.e., system calls and call-stacks, to recover PHFs performed by remote attackers with high accuracy and reasonable overhead. Note that the general audit data we use and the way we build the system makes our system quite easy to be extended to cover other malware. To build such a practical RAT forensic system on Windows, it should satisfy the following requirements:

R1: Instrumentation-free System. By the term instrumentation-free, we mean that a system leverages existing built-in event logging systems and does not require

- * *These authors contribute equally to this work.*
- *R. Yang, C. Xiong, L. Ruan, Z. Li, and L. Xu are with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China. E-mail: rainkin1993@zju.edu.cn, chunlinxiong94@zju.edu.cn, ruanlinqi@zju.edu.cn, lizhenyuan@zju.edu.cn, and zd_xlh@zju.edu.cn.*
- *H. Xu is with School of Cyber Science and Technology, Zhejiang University, Hangzhou 310027, China. E-mail: haitao.wm@gmail.com.*
- *Y. Chen, X. Chen, and M. Kavousi are with Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60208 USA. E-mail: ychen@northwestern.edu, xutongchen2021@u.northwestern.edu, and kavousi@u.northwestern.edu.*
- *Y. Cheng (the corresponding author) is a security scientist at Baidu Security. E-mail: chengyueqiang@baidu.com.*

extra instrumentation on the end-user systems. Instrumentation techniques are usually prohibitive in the enterprise environment because they can make applications and the operating system unstable [4], [5], [6]. Furthermore, patching the kernel has never been supported by Microsoft. Microsoft even integrated Kernel Patch Protection (KPP) into Windows to prevent patching the kernel [7]. Thus, the previous instrumentation-based forensic schemes, like [8], [9], [10], cannot be applied.

R2: Low Recording Overhead. The log recording module always runs on end hosts, and thus it should not cause high performance overhead. Some existing techniques [11], [12], [13] do not rely on instrumentation, but they introduce an unacceptably high system overhead at runtime (e.g., memory forensic, sandbox, and taint techniques), which is not practical and cannot be accepted by users.

R3: Accurate Fine-grained Semantic Behaviors Reconstruction. The behaviors of RATs are composed of many PHFs (e.g., Key Logging). It is required to identify those fine-grained semantic behaviors in order to understand the intent and ramification of RAT attacks [14]. However, most of existing forensic systems [8], [9], [15], [16], [17], [18], [19], [20], [21], [22] rely on audit logs which consist of a limited number of security-related objects like processes, files, and sockets, to diagnose attacks so that they are blind to PHFs. Furthermore, identifying PHFs on Windows is challenging due to the serious *Semantic Collision* problem (Section 3.3.1). The *Semantic Collision* problem renders the state-of-the-art behavior identification approaches [23], [24] cannot work on Windows directly.

Our solution. Our work aims to take the first step towards building a practical RAT forensic system on Windows. Specifically, we propose an instrumentation-free RAT forensic system, RATSCOPE, in which an audit logging module is built for efficient audit recording, and a novel program behavior modeling technique is developed to reconstruct fine-grained semantic behaviors of RATs accurately. To the best of our knowledge, RATSCOPE is the *first* RAT forensic system on Windows. Figure 1 illustrates how RATSCOPE is distinguished from traditional forensic systems. Traditional forensic systems typically provide human users with a list of obscure processes and files/sockets associated with a malicious process as well as ambiguous speculations about the attack intent and damages caused. By contrast, RATSCOPE can offer much clearer visibility into the different functions (independent semantic units, such as *Key Logging* and *Audio Record*) performed by the malicious process.

We choose to build the audit logging module upon the Windows built-in instrumentation-free ETW (**R1**). We improve the recording efficiency by filtering out application-specific events, picking up forensic-related fields from the selected events, and creating parsing shortcuts for the picked fields (**R2**). We address the *Semantic Collision* issue by proposing a novel behavior model which skillfully combines the information from low-level system calls and higher-level call-stacks to represent RAT behaviors accurately, then generate behavior models for PHFs of known RAT families, and match generated models against audit logs at runtime. This allows us to identify PHFs of unknown RAT families whose PHFs have similar implementations of known RAT families (**R3**).

We build a prototype system of RATSCOPE and perform both performance and behavior identification accuracy evaluation with 53 RAT families and 90 popular benign applications. The performance results show that the audit logging module only incurs 3.7% runtime overhead on average. The accuracy evaluation results show that our system can achieve around 90% TPR in the cross family experiment, around 80% TPR in the two-year spanning temporal experiment, and near zero FPR.

Our contributions are summarized as follows:

- We are the first to conduct a large-scale study of 53 real-world RAT families active from 1999 to 2016.
- We are the first to propose an instrumentation-free RAT forensic system capable of efficiently recording and accurately reconstructing missing RAT semantic behaviors for forensic analysis on Windows.
- We build a prototype system and perform both performance and behavior identification accuracy evaluation of it thoroughly. The evaluation results show that RATSCOPE is able to identify fine-grained behaviors of RAT accurately with reasonable overhead.

2 A Large-Scale Study of Real-World RATs

Although APT attacks [25], [26], [27], [28], [29] involving RATs have caused tremendous damage to public and private sectors worldwide, there is still a lack of studies that attempt to understand the landscape of RATs. In this section, we report a large-scale study of real-world RATs in terms of their workflow, functions equipped, and how their functions are implemented. In particular, we first briefly describe the workflow of a typical RAT attack in Section 2.1. Then we describe how we manage to collect a representative corpus of RAT samples for our study in Section 2.2. We report our major findings based on dynamic and static analysis of those RAT samples in Section 2.3. This study not only fills the gap between attackers and defenders but also motivates our system design described in section 3.

2.1 Workflow of A Typical RAT Attack

A RAT toolkit consists of two main components: a *RAT stub* and a *RAT controller*. After the RAT stub is delivered and executed on victim hosts (e.g., via phishing emails), attackers gain full control over the victim hosts. The RAT controller, residing on the attacker side, has a control panel which provides a graphical user interface (GUI) for an attacker to perform any attack function (e.g., *Key Logging*) with simple mouse clicks and keystrokes. Consequently, the RAT stub on the victim host would perform the corresponding malicious activities stealthily. *Note that, before triggered remotely by RAT controllers, RAT stubs would remain dormant.*

2.2 RAT Sample Collection

Since a RAT stub gaining a foothold on a victim host can only be triggered by its corresponding RAT controller owned by a remote attacker, we have to collect both the stub and the controller components of a RAT. However, victims usually report only the RAT stubs installed on their hosts (not the corresponding RAT controller) to public malware

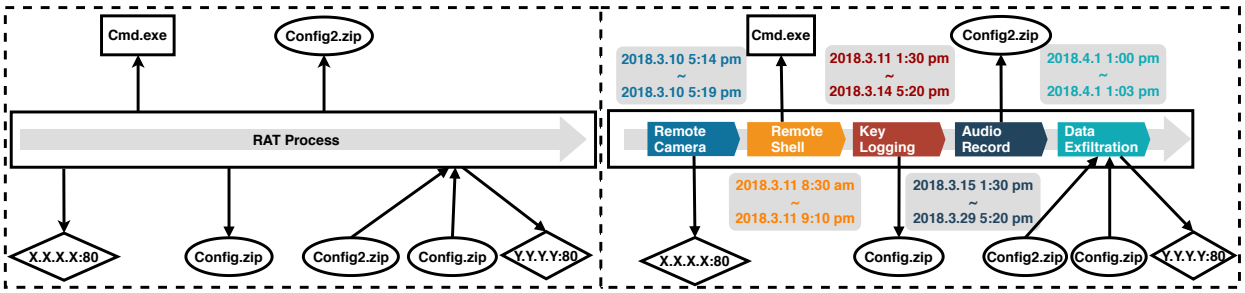


Fig. 1. Comparison between traditional forensic systems and our RATSCOPE on a simplified attack scenario. Rectangles denote processes. Ovals and diamonds denote files and sockets, respectively.

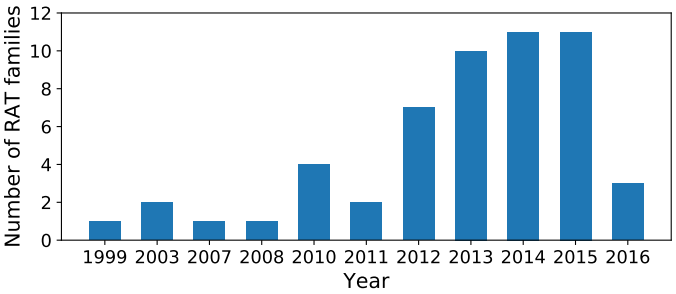


Fig. 2. Distribution of RAT families based on debut years.

repositories (e.g., VirusTotal) [14]. *Therefore, it is possible to collect RAT stubs for accessible controllers on VirusTotal; it is, however, nearly impossible to find RAT controller software on VirusTotal.*

To address the issue, we spend significant effort searching in underground hacker forums [30], [31], [32] where RAT controllers are sold or cracked. As a result, we find a total of 53 well-known and RATs families listed in Table 1. Most of them are notorious and involved in recent famous security incidents. For example, Poison Ivy RAT active since 2006 was involved in the RSA SecurID attack [33] and the Nitro attacks on chemical firms [34]; DarkComet active since 2008 was used in the Syrian activists attack [35] and leveraged in the Charlie Hebdo shooting incident for malware spreading [36]; XtremeRAT active since 2010 was responsible for the attacks on US, UK, Israel and other Middle East governments [37], [38], [39]; Adwind RAT active since 2012 was used in aerospace enterprises attacks [25] and attacks targeting Danish companies [26].

Furthermore, we identify the debut year of each RAT family using various information sources including blogs, white papers, and file creation time of each RAT family. A distribution of RAT families based on the year is shown in Figure 2. We provide more details about the debut year of each RAT family and how we determine it in the appendix.

In short, we collect a large corpus of RAT families active from 1999 to 2016 which we believe are representative of real-world RATs. We have made them public on the GitHub[1]*, which will be beneficial to other security researchers.*

2.3 Key Findings

We perform static and dynamic program analysis on collected RAT samples, which results in four key findings regarding the characteristics of RATs. Next, we describe how we derive those findings.

1. https://bit.ly/35Z0ksm

TABLE 1
Programming language usage of 53 RAT families.

Programming Language	RAT Families			Number
.NET based (C#.NET and VB.NET)	Back Connect	Mega	ctOs	24
	BXRAT	MLRAT	KilerRat	
	Cloud Net	MQ5	L6-RAT	
	Coringa	NanoCore	xRAT	
	Imperium	NingaliNET	Vantom	
	Imminent Monitor	NjRAT	SpyGate	
	Quasar	Proton	Revenge	
	Virus RAT	Comet RAT	D-RAT	
Delphi	Alusinus	Greame	Pandora	17
	CyberGate	NovaLite	Spycronic	
	Dark Comet	Nuclear	Spy-Net	
	Turkojan	Orion	Sub-7	
	Xena	Rabbit-Hole	Bozok	
	DHRat	Xtreme	-	
Java	Crimson	jSpy	Maus	5
	Frutas	Adwind	-	
Visual Basic (Native)	SkyWyder	HAKOPS	njworm	3
C++	Babylon	ucuL	-	2
Python	pupy	-	-	1
MASM	Poison Ivy	-	-	1

F1: High-level programming languages are preferred by RAT developers to write RAT stubs: We leverage `Detect-It-Easy` [40], a sophisticated file type detection tool, to identify the programming languages used by attackers to write RAT stubs. Specifically, for each RAT family, we generate a RAT stub from its RAT controller and feed it to Detect-It-Easy. The tool conducts static analysis of the stub and reports the programming languages (e.g., C++, Delphi, and Java) used to write the stub.

As shown in Table 1, `.NET` based language (`C#` and `VB.NET`) and `Delphi` are the two most popular programming languages. This is expected because (1) those programming languages either require few runtime dependencies (e.g., `Delphi` requires no dependencies) or are installed on Windows by default (e.g., `.NET` is installed on most Windows platforms). In this way, RAT developers ensure that their RAT stubs are executable on most Windows computers; (2) `.NET` and `Delphi` have vast amounts of ready-to-use libraries available online, which allow RAT developers to develop sophisticated RATs equipped with tens of rich functions easily and rapidly. For instance, we find that `Vantom` and `Mega` RATs implement the *Audio Record* function by directly invoking a well-known third-party library, `DirectX.Capture` [41]. In contrast, `Java`, `Python`, and `C++` are rarely used because they either require heavy runtime dependencies or go against rapid development.

F2: RATs are commonly equipped with tens of Potential

```
1  int main (){
2     /* register a keyboard hook */
3     SetWindowsHookEx (WH_KEYBOARD_LL , callback ,...)
4  }
5  /* receive and parse user input and  details of keys are stored in lParam*/
6  void callback (WPARAM wParam , LPARAM lParam){
7     if (wParam == WM_KEYDOWN){
8        PressKey()  // press the key
9     } else if (wParam == WM_KEYUP){
10       ReleaseKey()  // release the key
11    }
12    CallNextHook() // next hook procedure in the hook chain
13 }
```

K1: *Key Logging* implemented by hooking

```
1  int main (){
2     while (true){
3        /* get the state of each key */
4        for (each key in the keyboard){
5           NtUserGetAsyncKeyState (key);
6           ParseState()  // parse the state
7        }
8     Sleep ()
9     }
10 }
```

K2: *Key Logging* implemented by polling

```
1  LRESULT CALLBACK WndProc(UINT message , ...){
2     if (message == WM_CREATE){
3        /* config and register a raw input device */
4        RAWINPUTDEVICE rid
5        rid.dwFlags = RIDEV_INPUTSINK
6        NtUserRegisterRawInputDevices (rid , ...)
7     } else if (message == WM_INPUT){
8        /* receive user inputs into buffer */
9        RAWINPUT *buffer
10       NtUserGetRawInputData (... , buffer , ...)
11       ParseUserInput()  // parse user inputs
12    }
13 }
```

K3: *Key Logging* implemented by raw input

Fig. 3. Pseudocode of all three implementations of *Key Logging*.

Harmful Functions (PHFs): In this study, we obtain a list of functions with the occurring frequency in 53 RAT families. Specifically, as mentioned in Section 2.1, each RAT family has a GUI control panel which clearly lists available PHFs. Thus by traversing the control panel of all 53 RAT families, we collect a complete list of PHFs and calculate the occurring frequency of each PHF in 53 RAT families. Note that we never find a PHF which can be invoked without being triggered explicitly on the control panel from white papers, blogs, and our experience. Thus we believe the result obtained by analyzing RAT control panels is accurate.

TABLE 2
Popular RAT potential harmful functions (PHFs).

PHF	Description	Frequency
Key Logging	log all the keys pressed down by a victim	81.13%
Remote Shell	remotely open a console and execute commands	81.13%
Download and Execute	download a file and execute it automatically	84.90%
Remote Camera	remotely enable and access victims' camera	66.03%
Audio Record	capture audios with victims' microphone	49.05%

Table 2 provides the list of popular PHFs with brief descriptions. A PHF's occurring frequency in all RAT families is given in the third column. We can see that most (43% to 84%) RAT families are equipped with those PHFs. A complete list of PHFs is provided in the appendix.

F3: Different RAT families active from 1999 to 2016 implement the same PHF using similar methods: In this study, we identify possible implementation methods of 5 popular PHFs listed in Table 2. Those 5 PHFs are quite representative considering their prevalence among available RAT samples. Our current system requires us to manually analyze how a PHF is implemented in a RAT, which is quite labor intensive given that there are tens of RATs and tens of PHFs. Therefore, we plan to study all the remaining PHFs in our future work. Specifically, we have full control of a RAT (i.e., both of its stub and controller components), and therefore we are able to collect execution traces (e.g., system calls and Windows APIs) of each PHF by triggering it each time from the control panel of the RAT controller. We analyze the execution traces and learn what system calls and Windows APIs are invoked by each PHF. Once new system calls or APIs are invoked to perform a PHF, we consider that a new implementation method is probably identified and then we double check it by referring to the document of new APIs; otherwise, we attribute the execution trace to an existing method. Once done, we find that the ways of implementing a PHF are limited, and different families implement the same PHF quite similarly Take *Key Logging* as an example. All 53 RAT families implement the PHF only in 3 different ways, listed in Table 3. The third column represents what percentage of *Key Logging*-available RAT families takes a specific implementation way. The implementation methods of other PHFs studied are provided in the appendix. The finding that there only exists quite limited ways of implementing a PHF makes sense since operating systems do not provide a number of methods to perform a certain function.

TABLE 3
Implementation methods of `Key Logging` in RATs.

Method	Descriptions & Key Syscalls	Frequency
K1	RATs invoke `NtUserSetWindowsHookEx` to register a callback function into a message hook chain of Windows. The callback function will receive a virtual key code when victims press the key.	53.65%
K2	RATs invoke `NtUserGetAsyncKeyState` in an endless loop to poll every key state.	39.02%
K3	RawInput is another channel to get user input. RATs invoke `NtUserGetRawInputData` to get input when a WM_INPUT message occurs.	7.33%

Understanding how a PHF is implemented is essential for red teams to simulate real-world RAT attacks. However, due to a lack of study of RATs, even state-of-the-art red-team tools [42], [43], [44], [45], [46] either cannot simulate RAT attacks or can simulate only one way of RAT PHF implementation. To fill the gap, we provide an actionable executable file for each implementation method. Figure 3 provides the pseudo code for all three means of implementing the *Key Logging* PHF. That intelligence could be incorporated into existing red-team tools [42] in the future and would thus benefit other RAT researchers.

F4: Around 90% RATs only target Windows: We make this finding based on the observation that around 90% RAT stubs, produced by compiling the corresponding RAT controllers, take the file formats exclusive to Windows platforms, such as Windows PE executable files and Windows batch files. This implies that most RATs can only compromise Windows platforms, which is reasonable, considering that Windows is still the most popular operating system, especially in the enterprise environment [47].

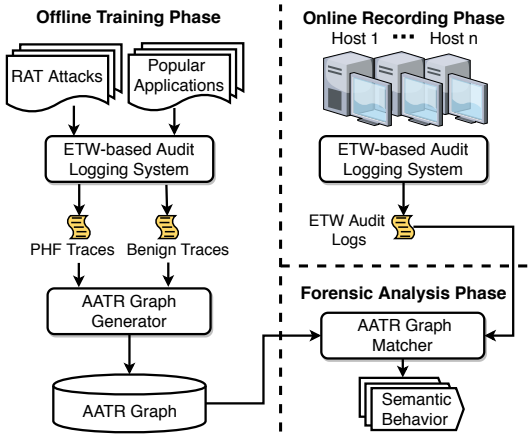


Fig. 4. System Architecture.

3 System Design

3.1 Threat Model and Design Overview

Threat Model: In this paper, we consider the OS kernel and auditing system (i.e., ETW) as part of the trust computing base (TCB). We assume that OS kernel is well protected by existing techniques [48], [49]. In our work, we consider a RAT attack that performs PHFs in a user space process. Our threat model is as reasonable and practical as the models of previous forensic works [17], [18], [50], [51].

Design Overview: Our goal is to develop a RAT forensic system suitable in an enterprise environment. Figure 4 illustrates our system architecture. The workflow of our system proceeds in three phases: *offline training*, *online recording*, and *forensic analysis*. The goal of the offline training phase is to model each RAT PHF based on both positive data (i.e., execution traces of PHFs) and negative data (i.e., execution traces corresponding to normal usage of benign applications). Specifically, we first introduce the *Semantic Collision* problem to explain why existing works fail, and then propose a novel behavior graph model, i.e., *Aggregated API Tree Record (AATR) Graph* (Section 3.2). Then we build an enhanced version of ETW (Section 3.3) to collect log data, which is then inputted to the *AATR Graph Generator* (Section 3.4) to generate AATR graphs, which characterize the internal implementation mechanism of PHFs. In the online recording phase, our system utilizes the enhanced ETW deployed on each Windows host for audit logging. Audit logs are then transferred to a specialized server. In the final forensic analysis phase, *AATR Graph Matcher* (Section 3.5) running on the server takes in both the collected audit logs and AATR graphs obtained in the offline training phase for PHF identification.

Our approach is able to identify PHFs of unknown RATs as long as corresponding PHFs with similar implementation is observed in the training phase. We believe this is a necessary prerequisite for a system that aims to identify behaviors of previously unknown RATs. Furthermore, our approach is effective in practice because our RAT study (Section 2) shows that although RAT families active from 1999 to 2016 were written by different programming languages and involved in different security incidents, PHFs of such RAT families have similar implementation methods.

3.2 Aggregated API Tree Record Graph

We choose to build RATScope upon ETW, considering ETW is the only instrumentation-free audit logging framework on Windows. However, unlike native audit systems on other platforms (e.g., `Linux Audit` and `DTrace`), **`ETW` does not provide input arguments for any low-level data including system call and API**, which would cause a serious *Semantic Collision* problem for behavior graph model approaches. To resolve this problem, we propose a novel behavior model, *Aggregated API Tree Record(AATR) Graph*.

3.2.1 Definitions

We first provide formal definitions of the *Semantic Collision* problem and *AATR Graph* for clarity.

Call Stack. A call stack CS_s on a system call s is an API calling stack. The entry of each call stack is a caller function with its direct lower entry being its callee function. From top to bottom, a call stack starts at an API in the application binary, then APIs in system libraries[2], and ends at the triggered system call s.

Top-layer API. A Top-layer API TA_s on a system call s is the first system library API invoked by the application binary.

Library call stack and Application call stack. A Library call stack $LibCS_s$ on a system call s is a subsequence of its call stack CS_s starting from TA_s to the bottom of CS_s while Application call stack $AppCS_s$ is a subsequence starting from the top of CS_s to the API directly calling TA_s.

Call Stack Tree. A call stack tree $CSTree_a$ of an API a is a tree where the root is the API a and each tree path, starting from the root to a leaf, represents a call stack starting from the API a.

Figure 6 provides a concrete ETW event example to explain the above concepts.

3.2.2 Semantic Collision Problem

Semantic Collision refers to the cases that two *different* program behaviors end up being represented as the *same* behavior graph, i.e., different program behavior semantics collide on a graph, which would cast doubt on the accuracy of the detection or forensic systems. *Semantic Collision* results from the fact that many program behaviors cannot be exactly reflected in low-level data without arguments, and hence crucial causality information is missing within the behavior graph model.

Figure 5 presents one practical example of *Semantic Collision* at the system call level. Specifically, the top half (enclosed by the red dotted box) represents a *call-stack* tree of the *Audio Record* behavior of a RAT. The bottom half (enclosed by the blue solid box) represents a *call-stack* tree of the normal *website browsing* behavior of `Chrome`. Although the two program behaviors are quite different and they trigger different call-stack data, their triggered system call sequences are exactly the same. Note that *Semantic Collision* could happen with any low-level data which lack input arguments, not just at the system call level. Furthermore, it results in the universal failure of previous works [23], [24], [52] which heavily rely on input arguments.

2. *System Library* here refers to Windows system libraries, including `ntdll.dll`, `kernel32.dll`, `user32.dll` and so on.

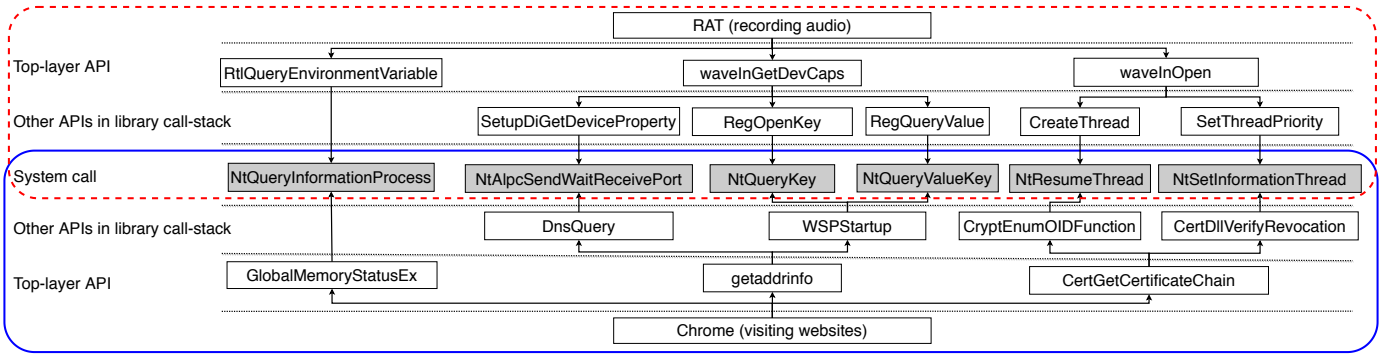


Fig. 5. An example of *Semantic Collision* on system call level data (shaded boxes). Two different program behaviors trigger exactly the same system call sequences though different library call-stacks.

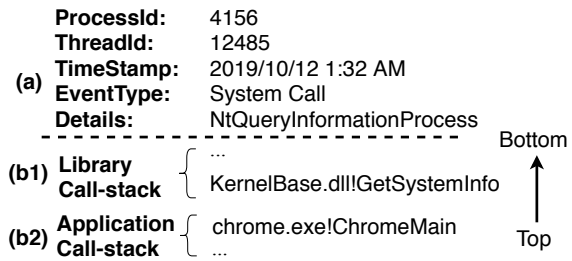


Fig. 6. An example of ETW event generated by our system: (a) system call event, (b1) library call-stack, and (b2) application call-stack. `GetSystemInfo` is the Top-layer API.

3.2.3 Collision Avoidance by AATR Graph

Motivated by the example illustrated in Figure 5, two different behaviors may look exactly the same on the system call level but apparently different on the *call-stack* level, so it could help solve the *Semantic Collision* if we model program behaviors on multiple-level of audit logs.

Further, we make three other vital observations:

- System calls and library call-stack are more generic and stable than application call-stacks which is specific to certain applications.
- Different call-stack trees of an API execution implies the API taking in different arguments. Conversely, different input arguments of an API typically result in different call-stack trees.
- System calls and their corresponding library call-stack invoked by a top-layer API typically present the characteristics of space-time clustering. That is, they usually appear adjacent in `ETW` traces.

These observations suggest that i) system calls and library call-stacks are suitable for general program behavior modeling. ii) call-stack tree could be used as an alternative to approximate the missing input arguments in differentiating the program behavior semantics, and iii) attribution of system calls and corresponding library call-stacks triggered by an API are adjacent in `ETW` traces and thus that the reconstruction of call-stack trees would be easy. This leads to our core program behavior model design, termed as *Aggregated API Tree Record (AATR) Graph*.

AATR. An Aggregated API Tree Record $AATR_a$ is a call-stack tree of a top-layer API a.

AATR Graph. An Aggregated API Tree Record Graph G=(V, E) is a directed graph where the vertex set V is a set of AATRs and the edge set E represents the causality dependency between AATRs. An AATR $AATR_a$ is dependent on $AATR_b$ if API a must be invoked after API b.

Compared with traditional behavior graph models, AATR graph adopts an effective data fusion method and takes advantage of both low-level system calls and higher-level call-stack trees to establish fine-grained program behavior semantics.

3.3 `ETW`-based Audit Logging System

`ETW` is a Windows built-in audit logging system. It consists of two components: a *recorder* and a *parser*. The native recorder records low-level data events (e.g., system calls and call-stacks) and stores them in memory buffers or as log files in binary format. The native parser [53], [54], [55] parses the binary audit logs into human-understandable events for further usage. `ETW` has two intrinsic features: *instrumentation-free* and *low-overhead*, which makes it an ideal audit logging subsystem in our proposed RAT forensic system RATSCOPE.

3.3.1 Limitations with `ETW` for RAT Forensic Purpose

`ETW` has been used in existing forensics and detection works [17], [50], [51] for audit logging. Specifically, `ETW` in existing works is used for collecting only the events related to OS-level objects, e.g., process creation and file write. However, addressing the *Semantic Collision* problem through AATR Graph requires to collect additional levels of events, e.g., system calls and call-stacks. Native `ETW` when collecting those extra events suffers from data parsing and data quality issues, which have not been reported before. We summarize the limitations with `ETW` as follows.

Performance issues with native `ETW` parser. Native `ETW` produces a huge amount of redundant data and would waste system resources when parsing. Furthermore, system call and call-stack traces are typically much larger than OS-level event traces, which aggravates the parsing problem.

Flawed `ETW` data. Crucial fields of `ETW` events (e.g., process id) are assigned meaningless value -1, which makes it hard to attribute a system call to its belonging process. More details and examples can be found in Section 4.1.2.

3.3.2 Our Solutions

Efficient Parsing. We propose a technique to improve the performance of parsing `ETW` data by filtering out application-specific events, identifying and focusing on the fields of the events helpful for forensics, and creating parsing shortcuts for those fields (described in Section 4.1.1).

Semantic Recovering. We propose to address the data quality issue of ETW events by recovering the missing crucial field value, resolving a system call's entry-point address to its name, and resolving the return address of call stack to library functions (see Section 4.1.2).

3.4 AATR Graph Generator

Our AATR Graph Generator is developed in the offline training phase to produce AATR graphs, which model the PHF behavior based on ETW logs.

Semantics Redundancy Problem. Typical behavior graph model generation [23], [24] usually involves as much low-level data as possible, which is collected by monitoring the targeted hosts for a long time to ensure that the program behaviors would not be cut off. Based on our observation, low-level traces (e.g., system calls) suffer from tremendous semantic redundancy, a large portion of which could be represented as loops in a long-time execution (Section 4.2.1).

Redundancy-reduction based method. To generate AATR graph from the low-level traces with the aforementioned issues, we developed a *redundancy-reduction-based* algorithm (Section 4.2.2) to extract the essential parts in traces to represent the PHF semantics. Specifically, we leverage call stack to precisely identify loop bodies and then select representative loop body to represent the PHF semantics.

3.5 AATR Graph Matcher

Our AATR Graph Matcher is developed in the forensic phase to identify PHFs of RATs from the audit logs.

Noise in low-level data traces. Performing the same program behavior at a different time could invoke different low-level data traces, due to the different runtime system context. We consider such instability within collected low-level traces as the *noise*. Such *noise* could exist in every level of audit log events collected in our system. Since the noise is irrelevant to the core program behaviors, our AATR graph matcher should be designed to be more robust to tolerate the noise when performing graph matching on the collected data traces.

Optimal partial graph matching method. To tolerate the noise, we propose an *optimal partial graph matching algorithm*. The main idea is that different from previous works that usually perform exact matching between predefined behavior graphs and collected data traces, we design optimized matching functions to perform the mapping between the AATR graphs already generated offline and the collected ETW trace to evaluate how well the collected traces match against the AATR graphs. We detail how we implement the AATR Graph matcher in Section 4.3.

4 Implementation

In this section, we present the details about how we implement the three most important components of RATScope, which are ETW-based audit logging system, AATR graph generator, and AATR graph matcher.

4.1 ETW-based Audit Logging System

Fig. 7 illustrates how our parser works on raw ETW data. In the following, we present how we perform efficient parsing and semantic recovering in our audit logging system.

4.1.1 Efficient Parsing

This step aims to improve the performance of parsing ETW data. It consists of three steps: **i) Filtering out application-specific events.** ETW provides over one thousand groups of log events. Most of them are specific to certain applications such as `Internet Explorer` and `Word`, while our system depends on general events (e.g., system calls and call-stack) to represent malware behaviors. Therefore, we reduce the audit logs by filtering out application-specific events and also focusing on the remaining system call and call-stack events. **ii) Identifying forensic-related fields from the selected events.** All ETW events share 18 common fields [56]. However, not all those common fields are useful for forensics. Based on previous works [17], [57] and our domain knowledge of forensics, we identify 3 fields out of the 18 common ones: *process id*, *thread id*, and *time stamp*, which are necessary for forensics. **iii) Creating parsing shortcuts for the selected fields.** The ETW recorder stores events in binary format. In order to correctly extract values of fields from binary, complex parsing steps are necessary, such as checking Windows version, analyzing data structures of event fields, and locating fields in ETW binary data. We propose an optimized parsing method. The main idea is to create and cache shortcuts for reusing the results of complex parsing steps. Specifically, each field is stored in a certain offset of ETW binary data with a certain data size. In the offline phase, in order to create shortcuts, we perform those complex parsing steps for each interested field to obtain offset and data size, and store this information in a cache file. Note that the process of generating shortcuts is automatically performed without manual intervention. In the online phase, the cache file is loaded into memory. When a new ETW event occurs, we retrieve the cached offset and data size of a field from memory, directly jump to the offset of ETW binary data, and extract values of the field using data size without going through those complex parsing steps.

4.1.2 Semantic Recovering

In addition to the parsing performance issues, default ETW data suffer from data quality issues. Specifically, i) Some crucial fields (e.g., process id) are assigned with -1, the default missing value. And Microsoft does not provide any guidelines on how to address the issue. ii) ETW only provides the memory address of system call and call-stack rather than their symbolic names which are necessary to our AATR model.

We address the above two issues in two steps: **i) Recovering the missing crucial field values.** We leverage the reverse engineering technique to find a solution successfully. Specifically, ETW always returns -1 for the thread id field of system calls but returns meaningful thread id value for the context switch events. Then we analyze the semantics of the context switch event to determine whether context switch and system calls can be combined. In particular, an operating system offers time slices of CPU processor to the threads eligible to run. Once a time slice is completed, a context switch event occurs and the CPU processor is switched from one thread to another. Thus by tracking context switch events, we can obtain which thread is running

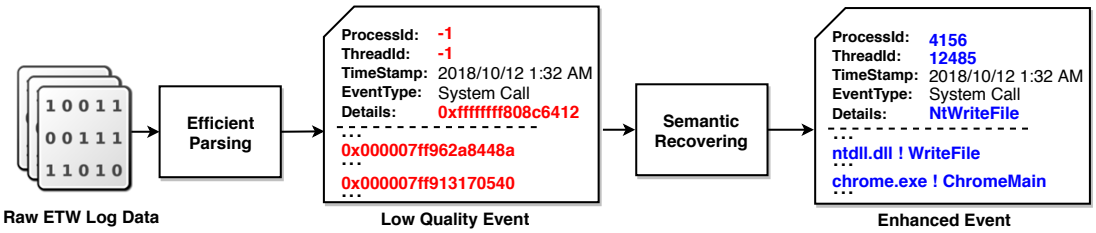


Fig. 7. How our audit logging system works on the raw ETW log data.

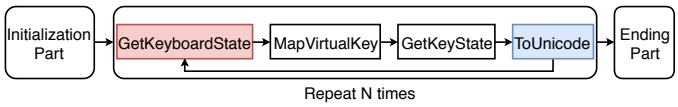


Fig. 8. A breakdown of the execution traces of the *Key Logging* PHF: initialization, loop body, and ending.

under a certain CPU processor. At the same time, a system call event provides which CPU processor the system call is related to. Thus by correlating the CPU processor between context switch events and system call events, we can map a system call to a thread. With the thread id, we can easily get the belonging process id. **ii) Resolving system call and call-stack.** The process of resolving memory addresses to symbolic names involves two steps. First, our system locates a module which a memory address belongs to, and converts the raw memory address to an offset of the module. Note that Windows loads a module (e.g., DLL library) in a random memory space when operating system restarts. Our system leverages an ETW event called `ImageLoad` to obtain dynamic mapping relationships between memory addresses and modules. Second, our system maps an offset of a module to a symbolic name. Mapping relationships between offsets of a module between symbolic names can be obtained by debugging symbol files [58].

4.1.3 AATR-based Log Reducing

To further reduce storage usage without affecting the final forensic analysis, we propose an AATR-based log reducer to 1) eliminate the application call-stack of every ETW event which is not needed in our system, and 2) remove redundant data. In particular, our preliminary result shows that a long-running execution trace only contains around 0.06% unique call stacks in the trace. It indicates the existence of tremendous duplicated call-stacks. We rearrange ETW audit logs into AATR format to compress those duplications by folding shared parts.

4.2 AATR Graph Generator

4.2.1 Semantics Redundancy Problem

Low-level data traces are collected for a long enough time so that traces can include a complete life cycle for program behavior. Although the whole execution is long, core parts tend to be short and self-repeated. For instance, in *key logging* PHF, RAT stub keeps recording keyboard strokes. The conceptual ETW trace is shown in Figure 8. It clearly shows that a program behavior could be divided into three parts semantically: *initialization*, *main loop body*, and *ending*. As the major semantics lies inside the repeated main loop body, semantics redundancy explodes with the execution and collection time.

4.2.2 Redundancy-reduction-based Generation

Enlightened by the example in Figure 8, as long as we extract the representative loop body, we can describe PHFs of RATs in a compact way. Our redundancy-reduction (Algorithm 1) is to use call-stack on each system call to conduct the reduction starting from top-level loops in input traces. **The key is that two system call events represent 2 invocations of the same system call in a loop if and only if their call-stack information is exactly the same**. This helps confirm a revisit of a system call in the loop and mark the correct border of initialization (Line 2 to Line 3 in Algorithm 1), ending (Line 4 to Line 7 in Algorithm 1) and main loop body (Line 8 to Line 10 in Algorithm 1). After we get all loop bodies identified, apply redundancy-reduction recursively to deal with nested loops, and ultimately incorporate compacted initialization and ending as our redundancy reduction result (Line 11 to Line 13 in Algorithm 1). With this algorithm, the long-time execution `ETW` traces can be reduced to be a compacted behavior sequence representing essential semantics in loop bodies.

Algorithm 2 is the overall logic of Aggregated API Record graph generation. After we get a semantics-redundancy-free behavior sequence of long-time execution ETW traces from Algorithm 1, we first get rid of application call-stack (Line 5 in Algorithm 2) which can easily cause evasion problem if application authors are malicious, then we add causality with an existing causality engine [13], which is able to check the causal relationship between APIs, to turn this AATR sequence into an AATR graph (Line 6 in Algorithm 2) and filter out every AATR graph which matches a benign or other-PHF trace (Line 7 to Line 12 in Algorithm 2). Notice that during the training, we use causality to build the behavior graph.

In addition, when we perform the matching between AATR behavior graphs and real-world ETW traces, we cannot get the corresponding causality because ETW cannot provide that, i.e., the real-world ETW traces cannot be transformed into a corresponding AATR graph for matching. As a trade-off between slight accuracy loss and excessive overhead increment, we choose to satisfy the chronological order of those causal related AATRs in ETW traces when we perform AATR graph matching. Details are provided in Section 4.3.

4.2.3 Explanatory Example

In this section, we give an example in Fig. 9 to show the workflow of our redundancy-reduction AATR graph generation algorithm in which an ETW trace is transformed into an AATR graph. The character sequence in the figure represents the original ETW trace where each character represents a system call along with its call-stack. Two same

characters mean that those two system calls are the same and their call-stacks are the same.

Redundancy reduction on the given ETW trace consists of 3 steps. In Step 1a, we identify the initialization, ending, and main loop bodies of raw ETW trace by locating loop body separators. Specifically, we scan the trace to locate the loop body starting (lbs) separator (θ_{lbs} = Ⓒ) that splits the initialization and the first main loop body, and thus we identify the initialization (ϕ_{init} = Ⓐ-Ⓑ) (Line 2 to 3 in Algorithm 1). Then we scan the trace again to locate the loop body ending (lbe) separator (θ_{lbe} = Ⓔ) that splits the last main loop body and the ending, and thus we identify the ending (ϕ_{end} = Ⓕ-Ⓖ-Ⓗ-Ⓘ) (Line 4 to 7 in Algorithm 1). After that, we just shrink main loop bodies in Step 1b and the redundancy reduction on the top-level is done (Line 8 to 10 in Algorithm 1). Then we repeat the above process to recursively handle the nested loops enclosed by a red dotted box in Step 2 and get a compacted trace (Line 11 to 13 in Algorithm 1). In the final step 3, we eliminate the application call-stack of each character (e.g., it turns A to A'), organize the event sequence to be an AATR sequence (Line 5 in Algorithm 2), and then use an existing causality engine [13] to replace the original temporal relationship between AATRs with causality dependency indicated by solid arrows (Line 6 in Algorithm 2). For instance, considering the trace related to file download, the AATR whose Top-layer API is `CreateFile` must be invoked to create a file handle before the AATR whose Top-layer API is `WriteFile`, and we will create a causality dependency pointing from the former AATR to the latter AATR. We perform those 3 steps for each trace in the dataset.

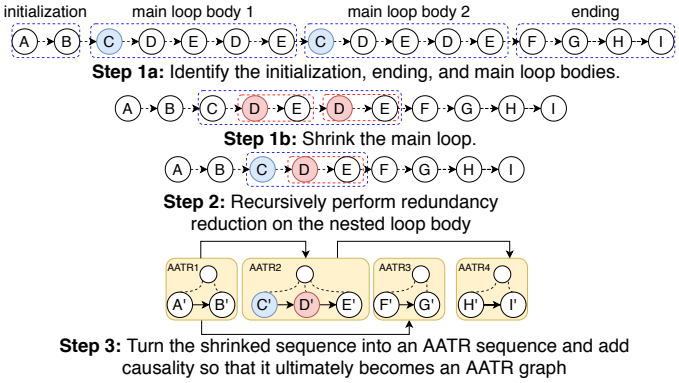


Fig. 9. The workflow of AATR graph generation on a given trace.

Algorithm 1 Redundancy-reduction Algorithm

Input: An unfolded AATR trace ϕ_{input}={θ_j=(syscall, library call-stack, application call-stack) $|j$=1 . . . m}
Output: An AATR sequence without causality $\phi_{reducted} \subseteq \phi_{input}$
Initialize: $\phi_{reducted} \leftarrow \emptyset$

1: **function** REDUNDANCY-REDUCTION(ϕ_{input})
2: scan from θ_m to θ_1, find the last $\theta_{lbs} \in \phi_{input}$ where $\exists k$, θ_{lbs}=θ_k, $lbs + 1 \leq k \leq m$
3: $\phi_{init} \leftarrow \theta_1 \ldots \theta_{lbs-1}$
4: get all loop body separator $\alpha_{lbs} \leftarrow \{j_k | \theta_{j_k}$=$\theta_{lbs}\}$
5: $llbs \leftarrow max(\alpha_{lbs})$
6: scan from θ_1 to θ_m, find the last $\theta_{lbe} \in \phi_{input}$ where $\exists k$, θ_{lbe}=θ_k, $lbs \leq k \leq llbs$
7: ϕ_{end}=$\theta_{lbe+1} \ldots \theta_m$
8: $\alpha_{lbs} \leftarrow \alpha_{lbs} \cup \{lbe + 1\}$
9: get the maximum gap index j_s=arg max$_{j_s}$ $j_{s+1} - j_s$
10: get selected loop body ϕ_{slb}=$\theta_{j_s} \ldots \theta_{j_{s+1}}$
11: get nested loop recursively reduced result $\phi_{nesrec} \leftarrow$ Redundancy-Reduction(ϕ_{slb})
12: $\phi_{reducted} \leftarrow$ Concatenate(ϕ_{init}, $\phi_{nestrec}$, ϕ_{end})
13: return $\phi_{reducted}$

Algorithm 2 Aggregated API Tree Record Graph Generation

Input: (1) n unfolded AATR traces collected by the audit logging system, corresponding to one selected PHF: Φ_{phf}={$\phi_i | i$=1 . . . n}, where trace ϕ_i={θ_j=(syscall, library call-stack, application call-stack)$|j$=1 . . . m}; (2) a causality analysis engine $CausalityEngine$ which take an AATR sequence to output an AATR graph; (3) traces about all other PHFs: Φ_{other}; (4) traces of benign program normal operation: Φ_{benign};
Output: n AATR graphs, each represented as direct acyclic graph of AATRs which defines the PHF semantically: Ψ={ψ}, where ψ=(∪AATR, ∪causality)
Initialize: $\Psi \leftarrow \emptyset$.

1: **procedure** AGGREGATED API TREE RECORD GRAPH GENERATION
2: Preprocess each trace $\phi_i \in \Phi_{phf}$.
3: **for** each trace $\phi_i \in \Phi_{phf}$ **do**:
4: $\psi_i \leftarrow$ REDUNDANCY-REDUCTION(ϕ_i)
5: Eliminate application call-stack in ψ_i and organize ψ_i to be an AATR behavior sequence
6: $\psi_i \leftarrow$ CausalityEngine(ψ_i)
7: $eligible \leftarrow true$
8: **for** each trace $\phi_{i2} \in (\Phi_{other} \cup \Phi_{benign})$ **do**
9: **if** ψ_i matched ϕ_{i2} **then**
10: $eligible \leftarrow false$
11: **if** $eligible$ **then**
12: $\Psi \leftarrow \Psi \cup \{\psi_i\}$

4.3 AATR Graph Matcher

In this section, we explain our design for AATR graph matcher. We formalize the optimal partial matching problem and explain the *optimal partial graph matching algorithm* we propose to address the noise problem without introducing extra false positives.

4.3.1 Optimal Partial Matching Problem

Definition 1 (Optimal Partial Matching Problem) *Given an AATR graph (a labelled DAG) G=(V={$v_i | 1 \leq i \leq m$},$E \subseteq V \times V$) and an enhanced ETW trace (a labelled sequence) ϕ={$\theta_j | 1 \leq j \leq n$}, find a one-to-one mapping $f : V' \leftrightarrow \phi'$, where $V' \subseteq V$ and $\phi' \subseteq \phi$, so that (1):f can maximize a matching rate function $t : \wp(V \times \phi) \rightarrow [0, 1]$; (2) for $\forall v_x, v_y \in V'$, let θ_p=$f(v_x)$ and θ_q=$f(v_y)$ if there is a path from v_x to v_y in G, then $p < q$ must be satisfied.*

Here $\wp(V \times \phi)$ denotes the power set of $V \times \phi$ and N denotes the set of natural numbers. Function t is the matching rate function which is designed to reflect how well the graph is matched, defined in Section 4.3.2.

4.3.2 Matching Rate Function

Definition 2 (Matching Rate Function) *Given a labeled direct acyclic simple graph G=(V={$v_i | 1 \leq i \leq m$}, $E \subseteq V \times V$), a labeled sequence ϕ={$\theta_i | 1 \leq i \leq n$} and a one-to-one mapping $f : V' \leftrightarrow \phi'$, where $V' \subseteq V$ and $\phi' \subseteq \phi$, a matching rate function $t : P(V \times \phi) \rightarrow N$ is*

defined as $t=\frac{\sum_{\forall(v,\theta)\in f} bonus(v,\theta)}{\sum_{\forall v\in V} bonus(v,v)}$, *where bonus is a matching score function, defined as:*

$$bonus(v,\theta) = \begin{cases} 1+\Delta(lib\_stack_v, lib\_stack_\theta) & sysc_v = sysc_\theta \\ 0 & otherwise \end{cases}$$

Here $sysc_v$ is the system call of v and lib_stack_v is the corresponding library call-stack. $\Delta(lib\_stack_v, lib\_stack_\theta)$ is the longest common part in library call-stacks of v and θ starting from the bottom (system call) to top (top-layer API).

The intuitive explanation of matching rate function is that we sum up the similarity of every matched pair (v,θ) and get the normalized matching rate by computing the ratio of the sum score to the score of the optimal matching, i.e., $\sum_{\forall v\in V} bonus(v,v)$. Evidently, the higher value of the matching rate function indicates a better AATR graph matching.

4.3.3 Dynamic Programming based Matching

This section describes how we build the optimal mapping f. When we map a vertex v_i to an event θ_j, all ancestor vertices of v_i (the vertices that have a path to v_i in the DAG G) can only be mapped to events prior to θ_j due to the constraint (2) in Definition 1, so that the optimal mapping f' for all ancestors of v_i must be determined. This observation motivates us that the *optimal partial matching problem* has the optimal substructure property so that it can be solved using a dynamic programming method on the AATR graph. A clearer case is that when G happens to be a sequence, our matching problem becomes the *Maximal Weighted Common Sequence* problem, a generalized version of the *Longest Common Sequence* problem which is a matching problem between **two sequences**. From this perspective, the general case of our problem is a matching problem between **a sequence** and a **DAG**.

To perform a dynamic programming (DP) based graph matching on the AATR graph, we define a *graph state* (Definition 3) to represent each matching subproblem.

Definition 3 (Graph State)
Given a direct acyclic graph $G=(V,E)$, a graph state gs can be defined as a mapping $gs: V \to \{0,1\}$ where $gs(v)=0$ means that v is not matched and $gs(v)$=1 means v is matched.

One DP transition step acts on a graph state gs by selecting the next vertex to be matched. This also means that the direct descendant graph state gs_{new} of a graph state is built by adding **only one more matched vertex** to gs (Line 5 to Line 13 in Algorithm 3). A vertex v can be selected if and only for its every direct/indirect ancestor v', $gs(v')=1$ so that it does not violate the constraint (2) in Definition 1.

The whole matching algorithm consists of two algorithms. Algorithm 3 serves an important initialization part in the whole behavior graph matching process. It transforms a direct acyclic graph into a *graph state* transition graph by simulating the process of a topological sort on the AATR graph. Algorithm 4 provides the main logic of our AATR matching algorithm. We follow the problem formalization given in Definition 2 and try to maximize the target function by dynamic programming.

4.3.4 Explanatory Example

In this section, we provide examples to show how AATR graph matcher works. Specifically, the example in Fig. 10 shows the input and output of the matcher from a high-level view. Then examples in Fig. 11 and Fig. 12 show details how the matcher obtain the output from the input step by step.

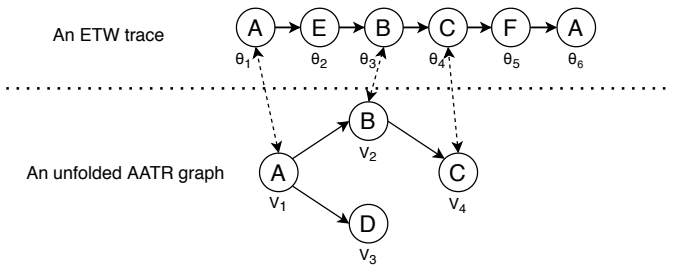


Fig. 10. An example of a match mapping between a given AATR graph and a given ETW trace.

In Fig. 10, θ_n represents the nth event in the ETW trace; v_m represents a vertex of the AATR graph; a capital character inside a circle represents a concrete system call with its call-stack. For ease of explanation, we just assume that each AATR v_m only has one node (i.e., character), and two same characters mean that they have the same system call and call-stack. That is, if θ_n and v_m have the same character, the matching score $bonus(v_m,\theta_n)$ defined in Definition 2 will be 1; otherwise, it will be just 0.

As shown in Fig. 10, the matcher receives an ETW trace and an AATR graph as input and outputs an optimal matching rate (i.e., 0.75) between them. Specifically, v_1, v_2, and v_4 in the AATR graph are matched with θ_1, θ_3, and θ_4 in the ETW trace respectively, which means $bonus(v_1,\theta_1)$, $bonus(v_2,\theta_3)$, and $bonus(v_4,\theta_4)$ are 1. Thus, the optimal matching score is 3 and the matching rate of the AATR graph that has 4 vertices is 3/4 (i.e., 0.75).

Fig. 11 and Fig. 12 show how our DP-based algorithm obtains the optimal match score in Fig. 10 in detail. Algorithm 3 first builds a state transition graph (the right graph in Fig. 11) to enumerate all possible graph states of the AATR graph (the left graph in Fig. 11). Specifically, the initial graph state gs_{init} is an empty set and it means no vertex in the AATR graph is matched. Because v_1 must be matched before any other vertices in the graph, the next graph state $gs_1=\{v_1\}$ is generated to represent v_1 is matched in this state. For one further step, based on gs_1, once v_1 is matched, $v2$ or v_3 can be selected to be matched. Thus two new graph states $gs_2=\{v_1,v_3\}$ and $gs_3=\{v_1,v_2\}$ are added with correspondingly selected vertex matched. This process will be repeated until all possible graph states are generated.

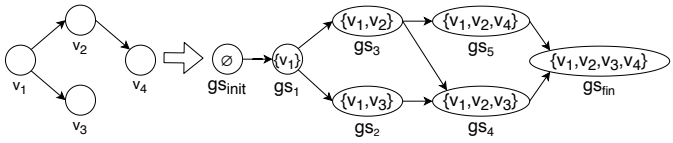


Fig. 11. Convert an AATR graph into a graph state transition graph.

Lastly, Algorithm 4 does DP optimization and ultimately find the optimal paths in the transition graph gen-

erated in Fig. 11. Fig. 12 shows one of the optimal transition path gs_{init}→gs$_1$→gs$_3$→gs$_4$→gs$_{fin}$. For each transition step, score changes are shown in the figure accordingly. Specifically, considering the transition from gs_{init} to gs_1, the algorithm ultimately decides to match v_1 with θ_1 rather than other events in the ETW trace because this local matching helps to build the overall optimal matching. Importantly, helping to build the overall matching is the only reason why Algorithm 4 takes those specific steps on the transition graph. Then the algorithm decides to match v_2 with θ_3 from gs_1 to gs_3 and the score becomes 2. From gs_3 to gs_4, the bonus of v_3 with all ETW events θ_n is 0, and thus the algorithm skips v_3 and the score is still 2. Finally, the algorithm matches v_4 with θ_4 and the final score is 3. The algorithm eventually gets the score of the optimal matching got from the above process and then calculates the matching rate as the output of our AATR graph matcher shown in Fig. 10.

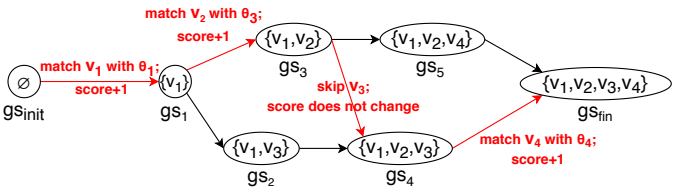


Fig. 12. One DP transition path for getting the optimal matching.

Algorithm 3 Transition Graph Building

Input: An unfolded AATR graph ψ=(V=∪AATR, E=∪causality);
Output: A graph state transition graph TG=(V_{tr}=$\{\forall$graph-state$\}$, E_{tr}=$\{\forall$graph-state-transition$\}$ $\subseteq$ $(V_{tr} \times V_{tr})$,$label$: $E_{tr} \rightarrow V$)
Initialize: $gs_{init} \leftarrow \{gs(v)=1|\forall v \in V\}$; $addedGs \leftarrow \{gs_{init}\}$; $TG \leftarrow (\{gs_{init}\}, \emptyset, \emptyset)$; $tpsortQueue \leftarrow \{gs_{init}\}$;

procedure TRANSITION-GRAPH-BUILDING(ψ)
  while $tpsortQueue \neq \emptyset$ **do**
    $gs_{head} \leftarrow tpsortQueue.pop()$
    $V_{frontier} \leftarrow$ GetFrontier(gs_{head}, ψ)
    for each $v_{itr} \in V_{frontier}$ **do**
      $gs_{new} \leftarrow gs_{head}$
      $gs_{new}(v_{itr}) \leftarrow 0$
      $V_{tr} \leftarrow V_{tr} \cup \{gs_{new}\}$
      $E_{tr} \leftarrow E_{tr} \cup \{(gs_{head}, gs_{new})\}$
      $label((gs_{head}, gs_{new})) \leftarrow v_{itr}$
      if $gs_{new} \notin addedGs$ **then**
        $tpsortQueue.push(gs_{new})$
        $addedGs \leftarrow addedGs \cup \{gs_{new}\}$
  return TG

Algorithm 4 AATR Graph Matching

Input: (1) A unfolded input ETW trace ϕ=$\{\theta_j$=(syscall, library call-stack) $|j$=$1 \ldots m\}$; (2) A unfolded AATR graph ψ=(V=∪AATR, E=∪causality);
Output: A matching rate $\in [0, 1]$
Initialize: TG=(V_{tr},E_{tr},$label$) ← Transition-Graph-Building(ψ); $gs_{init} \leftarrow \{gs(v)=1|\forall v \in V\}$; $gs_{fin} \leftarrow \{gs(v)=0|\forall v \in V\}$; $\forall gs \in V_{tr}$, $score(gs, \theta_0) \leftarrow 0$, where $score : V_{tr} \times \phi_{input} \rightarrow N$;

procedure GRAPH-MATCHING(ϕ_{input}, ψ)
  for $j = 1 \rightarrow m$ **do**
    $dpQueue \leftarrow \{gs_{init}\}$
    $score(gs_{init}, \theta_j) \leftarrow 0$
    while $dpQueue \neq \emptyset$ **do**
      $gs_{head} \leftarrow dpQueue.pop()$
      $score(gs_{head}, \theta_j) \leftarrow score(gs_{head}, \theta_{j-1})$
      for each gs_{pred} where e_1=$(gs_{pred}, gs_{head}) \in E_{tr}$ **do**
        if $score(gs_{pred}, \theta_j) > score(gs_{head}, \theta_j)$ **then**
          $score(gs_{head}, \theta_j) \leftarrow score(gs_{pred}, \theta_j)$
        $delta \leftarrow score(gs_{pred}, \theta_{j-1})$+$bonus(label_{e_1}, \theta_j))$
        if $delta > score(gs_{head}, \theta_j)$ **then**
          $score(gs_{head}, \theta_j) \leftarrow delta$
      for each gs_{succ} where e_2=$(gs_{head}, gs_{succ} \in E_{tr})$ **do**
        if $gs_{succ} \notin dpQueue$ **then**
          $dpQueue.push(gs_{succ})$
    $finalScore \leftarrow score(gs_{fin}, \theta_m)$
    return $finalScore/maxScore$

5 EVALUATION

In this section, we evaluate RATSCOPE by answering the following questions.

- **Q1.** How *effective* is RATSCOPE in dealing with a real-world RAT attack? (Section 5.1)
- **Q2.** How *robust* is RATSCOPE against different (and new) RAT families? (Sections 5.2 and 5.3)
- **Q3.** What is the *overhead* of RATSCOPE? (Section 5.4)

To deploy RATSCOPE in an enterprise environment, each end host is required to enable the built-in ETW equipped with our own parser for everyday audit logging; a sophisticated machine is needed to receive audit logging from end hosts, aggregate the log data, and perform AATR graph matching. In our experiments, each end host has the configuration of i5-4590 CPU and 8 GB RAM, and the sophisticated machine is configured with Xeon(R) E5-2650 CPU and 252 GB RAM.

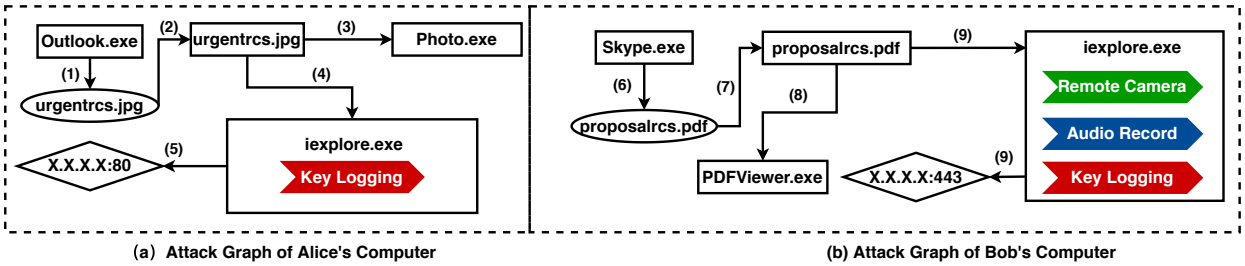
5.1 Effectiveness of RATSCOPE on Simulated Real-World RAT Attacks

In this experiment, we act as a red team and simulate a real-world RAT attack, i.e., one of recent RAT attacks related to Syria, described in white papers and reports [29], [59], [60], [61], by utilizing the same adversarial tactics in terms of infection vectors, involved RAT families, and anti-detection techniques, to achieve the same attack goals. Note that, existing white papers or reports cannot provide fine-grained semantics about the Syria attacks at the PHF level as we do.

Real-World Syrian RAT Attack. The general goal of recent RAT attacks related to Syria is to steal the intelligence of Syria activists (e.g., credentials, chat record and audio). Specifically, attackers leverage sophisticated social engineering techniques (e.g., phishing emails [29], and Skype messages sent from trusted people whose credentials have been stolen [60]) with visual spoofing techniques (e.g., file extension spoofing [59]) to trick victims into the execution of a malicious file, which is actually a remote access trojan [60] (e.g., DarkComet, NjRAT, and Xtreme), protected using anti-detection techniques (e.g., obfuscation [29] and process injection [61]). Attackers then remotely control the implanted RAT to obtain victims' intelligence by performing PHFs, such as *Audio Record*, *Key Logging*, and *Remote Camera*.

TABLE 4
Experiment details of our simulation of the Syria RAT attack.

Victim	Infection Vector	RAT	Anti-Detection	PHFs	Duration (HH:MM:SS)	Storage (MB)	TPR	FPR
Alice	Phishing Email [29] & File Extension Spoofing [59]	DarkComet	UPX Packer [60] & Process Injection [61]	Key Logging	8:32:11	184	100%	1.25%
Bob	Skype Message [60] & File Extension Spoofing [59]	NjRAT	DeepSea Obfuscator [29] & Process Injection [61]	Key Logging Remote Camera Audio Record	9:03:25	235	100%	0.5%



(a) Attack Graph of Alice's Computer (b) Attack Graph of Bob's Computer

Fig. 13. Simplified attack graph generated by RATSCOPE. Rectangles represent processes; ovals and diamonds denote files and sockets, respectively; Figure (a) shows the graph of Alice's computer; Figure (b) shows the graph of Bob's computer.

Our Simulation of the Syrian RAT Attack. In our simulation, we utilize the same adversarial tactics as the real-world Syrian RAT attacks and attempt to achieve the same attack goals. Table 4 provides the details of the simulation, including the involved victims, exploited infection vectors, utilized RATs, anti-detection techniques of RATs, and the PHFs performed. The table also shows the experiment duration, audit log size, and the accuracy of identification of PHF-level semantics, which will be discussed soon later.

Specifically, Alice and Bob are employees of a company. The attacker aims to target Bob, who owns information of the attacker's interest, including emails, chats, contacts, and audios. However, Bob has a strong security awareness. The attacker then launches the attack in two steps. In step 1, the attacker attempts to compromise Alice's computer and uses it as a stepping stone. The attacker sends a spear phishing email to Alice, in which the malicious executable attachment `urgentgpj.scr` is disguised as a JPG file `urgentrcs.jpg` using a file extension spoofing technique called *Right To Left Override (RTLO)*. Alice blindly double clicks the fake JPG file, which invokes the malicious file, later opens a JPG file in the foreground, and meanwhile releases an obfuscated RAT `DarkComet` in the background. The RAT then injects itself into the benign `Internet Explorer` process to bypass firewall, and connect to the attacker at `X.X.X.X:80`, which allows the attacker to remotely trigger *Key Logging* of `DarkComet` and finally to steal the credentials of Alice's Skype account. In step 2, the attacker generates a malicious fake PDF file `proposalrcs.pdf` using the same technique and sends it to Bob through Alice's Skype. Bob opens the fake PDF file without a doubt. Consequently, the RAT `NjRAT` is implanted and executed. Later on, the RAT connects to the attacker at `X.X.X.X:443`. And the attacker triggers a series of PHFs including *Remote Camera*, *Audio Record* and *Key Logging* of `NjRAT` to steal his interested information about Bob. During the whole attack, Alice and Bob behaves normally, such as visiting web pages with `Chrome`, sending/receiving emails with `Outlook`, reading documents with `Adobe Reader`, and communicating with `Skype`.

Attack Investigation with RATSCOPE. Our simulation attack is designed to happen on a working day. Our ETW-based auditing system deployed on both Alice and Bob's machines performed audit logging for about 8.5 hours and 9 hours, respectively, which resulted in 184 MB and 235 MB logs, respectively. We assume that a third-party threat intelligence product is also utilized in the company network, and after the attack happens, the product reports an alarm and correctly labels the IP address `X.X.X.X` as a threat alert. Using the alert as a starting point, existing forensic systems [15], [17], [51] can only track the files and processes related to the attack. It is hard for them to provide PHF-level semantics which could be comprehensible to non-expert users like Alice and Bob. By contrast, RATSCOPE successfully identifies all PHFs performed by attackers with very low FP rates (1.25% and 0.5%, respectively). We will discuss those FPs in Section 5.2.2. Figure 13 provides a simplified attack graph generated by RATSCOPE combined with a causality tracking approach [57]. In addition to the involved processes, files, sockets, and other expert-friendly but obscure information, RATSCOPE provides users extra PHF-level semantics (highlighted in color), which are necessary to better understand the attack tactics and intent and make informed remediation decisions. For example, based on the timestamp and the identified *Remote Camera* PHF, Alice and Bob could recall who might be captured via Remote Camera, and such information is quite important to post-attack response and remediation.

5.2 Robustness Across RAT Families

In this experiment, we evaluate how robust RATSCOPE is in identifying the PHFs of the RAT families whose traces were not involved in training RATSCOPE.

5.2.1 Experiment Setup

Family-split PHF Dataset. We first randomly select one RAT sample for each of the 53 representative RAT families (listed in Table 1). Then, for each of the 53 RAT samples, we execute the 5 popular PHFs (listed in Table 2) one by one,

TABLE 5
Comparison of identification accuracy.

PHF	Model	TPR	FPR
Key Logging	**AATR Graph**	**87.7%**	**1.1%**
	API-only Graph	87.7%	33.3%
	Syscall-only Graph	87.7%	34.4%
Remote Shell	**AATR Graph**	**85.7%**	**0%**
	API-only Graph	85.7%	6.6%
	Syscall-only Graph	85.7%	26.6%
Download & Exec	**AATR Graph**	**92.8%**	**2.2%**
	API-only Graph	92.8%	4.4%
	Syscall-only Graph	92.8%	40%
Audio Record	**AATR Graph**	**93.7%**	**2.2%**
	API-only Graph	93.7%	11.1%
	Syscall-only Graph	93.7%	16.6%
Remote Camera	**AATR Graph**	**90.3%**	**0%**
	API-only Graph	90.3%	0%
	Syscall-only Graph	90.3%	27.7%

including *Key Logging*, *Remote Shell*, *Download and Execute*, *Remote Camera*, and *Audio Record*. We collect traces for every execution using our ETW-based audit logging system. We then group traces based on the PHF which each execution trace invokes, and assign each group a label with the same name as the PHF. In this way, we collect 5 PHF subdatasets. *No two traces in a PHF subdataset are collected from the same one RAT family.*

Benign Dataset. We select 90 benign applications which are widely and daily used in a typical enterprise environment. By category, the selected applications include editor software (e.g., `Notepad++`, `GIMP`, and `Word`), communication software (e.g., `Skype`, `Foxmail`, and `Outlook`), browsers (e.g., `Chrome` and `IE`), file transfer software (e.g., `WinSCP`, `FileZilla`, and `FreeFileSync`), and long-running system processes (e.g., `explorer` and `dllhost`), and so on. Besides, we select several benign applications which had the similar functionality to the 5 PHFs we focused on, such as audio-related applications (`QuicktimePlayer` and `Audiorecorder`), shell-related (`CMD`), and download-related (`FreeDownloadManager`). We install each selected application in normal users' computers in which normal users operate those applications (e.g., visiting web pages with `Chrome`) during a working day, and collect traces as our benign dataset.

Evaluation Method. We evaluate RATSCOPE on family-split PHF dataset with the 10-fold cross-validation method. Specifically, for each tested PHF, we split the benign dataset and its corresponding family-split PHF dataset into 10 random folds, respectively, and iteratively select one fold as the testing set and the rest part as the training set for the accuracy evaluation. *Note that in this evaluation, one RAT family would never occur in both of the training set and the testing set, which avoids the bias [62] introduced by a completely randomized cross-validation method in which the training set and testing set might contain samples from the same family. The performance of previous works was commonly inflated because due to this bias.*

Baseline Approach. APIs and system calls represent the two most popular categories of data used for modeling program behavior. Our AATR model fuses both APIs and system calls to provide more accurate and fine-grained semantics and addresses the problems such as *Semantic Collision* due to limitations with the available ETW log data. For a comparative evaluation, we generate two baseline models, *API-only* graph model and *syscall-only* graph model to approximate the previous work [24] by replacing AATR with API and syscall. In our AATR model, each node denotes either a library call API or a system call, the root node is a top-layer API, and the leaf nodes are system calls. We replace each node of our generated AATR graph with either the root top-layer API or the leaf system calls to generate the two baseline graph models, i.e., the API-only graph, and syscall-only graph models, respectively. Note that those graph models are similar to the previous model [24] except for lack of arguments.

5.2.2 Result Analysis

Table 5 details the comparative evaluation results. In the context of this paper, the True Positive (TP) measures the total number of RAT families whose certain PHF is correctly identified by RATSCOPE, and the False Positive (FP) measures the total number of benign applications whose behaviors are mistakenly identified as PHFs by RATSCOPE.

RATScope (or the AATR model) is capable of accurately identifying PHFs across RAT families, as good as *API-only* graph model and *syscall-only* graph model. We can see in Table 5 that for each PHF, AATR model can identify most of the RAT families that are not presented in the training set (85.7% to 93.7%), which conforms to one key finding of our RAT study (in Section 2) that the implementation methods of one PHF tend to be quite similar across various RAT families. For example, the AATR graph generated for the *Remote Shell* PHF of the `SpyNet` RAT can match that of 28 other RAT families.

AATR model causes much fewer false positives than either *API-only* graph model or *syscall-only* Graph model. Table 5, shows that AATR model has a much lower FPR (0% to 2.2%) than the other two models. This is because AATR model is proposed to address the *Semantic Collision* problem and precisely capture the PHF semantics and thus could achieve higher identification accuracy, while the other two models suffer from the problem and thus could only introduce more identification inaccuracy. Furthermore, we find that APIs directly invoked by programs contain more semantic information than system calls, which is why the FPR of *API-only* graph model is always less than the *syscall-only* graph model. We also observe that when identifying the PHF *Remote Camera*, the *API-only* graph model achieves the same accuracy as our AATR graph model, because the API (e.g., `VFW (Video For Windows)`) is sufficient to model the PHF alone even though ETW provides no input arguments.

AATR model introduces very few false positives on benign traces. We also collect benign traces by performing similar but benign behavior to PHFs, e.g., *Audio Record*. Our AATR model introduces few FPs on those benign traces generated by those "malware-like" benign programs, such as the built-in Windows program `Audio Recorder`. Close scrutiny of those slight FPs reveals that in such cases, it is hard to differentiate a "malware-like" program behavior from a really malicious behavior because the implementation methods are similar. Also, malware could abuse benign applications to perform malicious actions [63]. That is the reason why we name RAT functions as Potential Harmful Functions (PHFs). However, in these cases, we believe it

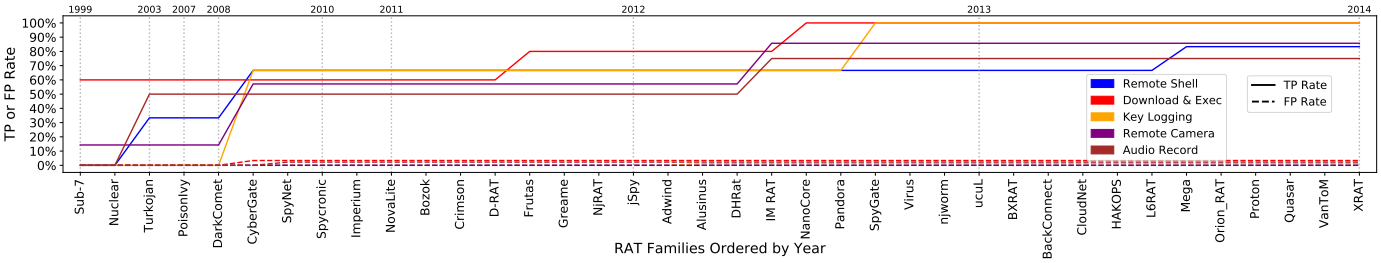


Fig. 14. Temporal evaluation. For a RAT family R_i listed in the X axis, we fixed the testing set, which included the RAT families from 2014 to 2016; the training set is dynamic, which included all RAT families occurring before the RAT family R_i.

is still worth reporting those FPs by a forensic system in the first place and performing triage with other information later. Actually, both academia [51], [64] and industry [65] have realized that relying on a single suspicious behavior to triage an alert is not sufficient in practice. NODOZE [51] proposes an automatic alert triage approach by leveraging contextual information of the generated alert, e.g., the chain of events that lead to an alert event and the ramifications of the alert event. Similarly, in the future, RATSCOPE could take into consideration the contextual information to automatically triage PHF alerts. For instance, it is hard to diagnose the browser `iexplorer.exe` in Figure 13 as malicious only due to accessing a camera. However, the contextual information of the browser (e.g., it is spawned by a suspicious process `proposalrcs.pdf` downloaded from Skype, and it performs other two PHFs after accessing camera) differentiates its behavior from the normal behavior of a browser which is spawned by a system process.

5.3 Temporal Evaluation

In this evaluation, considering the evolution of RAT over time, RATSCOPE is trained on RAT samples observed prior to a certain date and tested on newer samples.

5.3.1 Experiment Setup

Temporally-sorted PHF Dataset. We follow the same dataset generation process as the Family-split PHF dataset in Section 5.2.1. Then we sort the traces in every PHF subdataset by the debut year of the corresponding RAT families (shown in Figure 2). In the end, we obtain 5 temporally-sorted PHF subdataset.

Benign Dataset. For benign applications, we reuse 90 applications selected for the cross-family evaluation in Section 5.2. Then we randomly divide those applications into two sets. For each application in one set, we obtain the old version released between 2012 and 2013. For applications in the other set, we obtain the release versions between 2015 and 2016. Finally, following the same dataset generation process in the cross family evaluation, we obtain two benign datasets: `OLD` benign dataset containing execution traces of benign applications released between 2012 and 2013, and `NEW` benign dataset containing execution traces of benign applications released between 2015 and 2016.

Evaluation Method. In this evaluation, our temporal splitting between the training and testing sets follows the best practices recommended by TESSERACT [62] about how to avoid the temporal experimental bias. That is, training on data from the past and testing on data from the future. Specifically, for tested PHF, we use the traces of RAT families from 2015 to 2016 (around 27% RAT families) and the `NEW` benign dataset as a fixed testing dataset. Then we evaluate how matching accuracy changes with the varying training set, which gradually involves more and more RAT families starting from 1999 to 2014. The `OLD` benign dataset is used in training. The result is shown in Figure 14. For a RAT family R_i listed in the X axis, the training set is dynamic, which includes all RAT families occurring before the RAT family R_i; the testing set is fixed, which includes the RAT families from 2014 to 2016. The debut year of RAT families are given on top of the figure.

5.3.2 Result Analysis

Figure 14 shows that RATSCOPE has the capability of identifying new RAT samples although we use old samples in the training set. For most of PHFs, the AATR model trained using 8 RAT samples which appear before 2010 can identify over 50% RATs from 2015 to 2016 with slight FPs (lower than 3%). When the training set includes RAT families from 2012 to 2014, the TPR goes up to more than 80%. To further understand the result, we manually analyze those RATs by reverse engineering. We find that **i) RAT developers tend to reuse other RAT families' codebase which was leaked or cracked [66].** For instance, `NjRAT` is one of far-reaching RATs whose source code was leaked in 2013 [67]. We found that the code architecture of `Kiler RAT` appearing in 2015 is highly similar to `NjRAT`, and the format of configuration files of `Coringa RAT` appearing in 2015 is also the same as `NjRAT`. Other RATs such as `CyberGate` and `NanoCore` are also found to be sharing codes. **ii) RAT developers implement PHFs based on public libraries.** For instance, we find that `ctOS` and `Imminent Monitor` RATs implement the *Remote Camera* by directly invoking a well-known third-party library `AForge` [68], `Crimson` and `jSpy` implement *Key Logging* using a Java library `JNativeHook` [69], and `Quasar` implements *Key Logging* using a `C#` library `GlobalMouseKeyHook` [70]. This is reasonable because developing a stable and complete PHF from scratch is nontrivial. Furthermore, the implementation methods of different libraries are also similar, e.g., `JNativeHook` and `GlobalMouseKeyHook` implement *Key Logging* using the `K1` method mentioned in Figure 3.

5.4 Performance and Storage Overhead

Runtime Overhead. To evaluate runtime overhead of our ETW-based auditing system, we measure the overhead under different workloads. Specifically, we conduct two experiments to evaluate the performance of ETW.

First, we build a benchmark to generate diversified types of ETW events and control the speed of event generation (number per second), and then perform measurement upon it. As Figure 15(a) depicted, the overhead stays around 7%

when generating 20,000 events per second while it can go up to 83.72% when generating 229,475 events per second, the maximal event generation speed that our test machine can afford. This indicates that the runtime overhead hinges on the speed of event generation.

Second, we test how RATSCOPE performs for applications under heavy workload and under real-world scenarios. We first test applications under heavy workload. Specifically, we select 12 popular applications including browsers, messaging applications, editors, media players, server-side software, and development tools. We then leverage a GUI testing tool to automatically trigger the typical functions of those applications repeatedly, such as opening webpages, sending emails, editing documents, playing videos, online voice chatting, and compressing files. Meanwhile, we repeatedly collect traces of them and calculated the average number of events generated per second. The results are shown in Figure 15(b). Provided the average number of events generated per second is smaller than 20,000 for most applications, the runtime overhead under heavy workload would be lower than 7% according to Figure 15(a). Then we test how RATSCOPE performs under real-world scenarios where real-world users usually operate the applications and then stop to read contents displayed in the GUI without any operations. Thus we simulate real-world user behaviors on these applications by sleeping 5 seconds before performing the next mouse click using the same GUI testing tool. The simulation experiment lasts 30 minutes and the average runtime overhead is 3.7%.

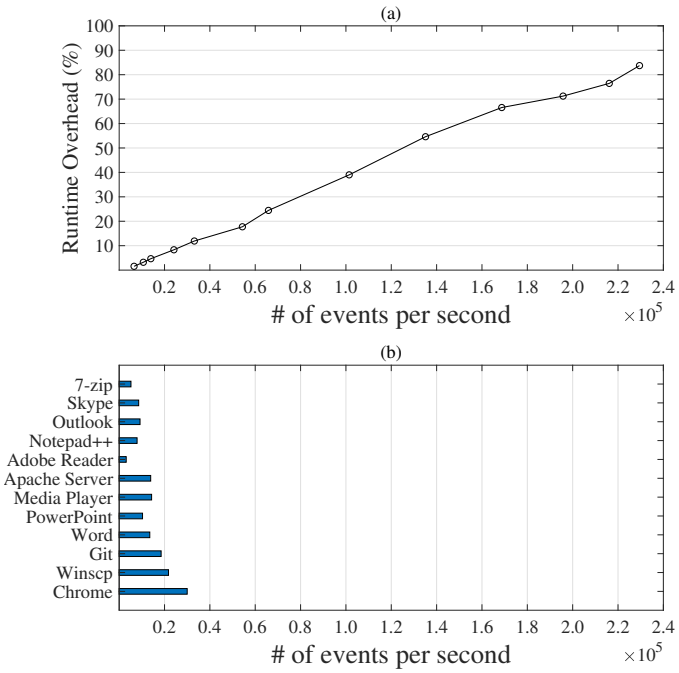


Fig. 15. Figure (a) shows the runtime overhead with different numbers of events per second. Figure (b) shows the number of events per second generated by different programs.

ETW Parsing Overhead. To evaluate the efficiency of our proposed parsing techniques, we conduct a comparative experiment between our system and other built-in parsing libraries mentioned in Section 4.1. Specifically, we parse the same set of raw ETW data using each parsing tool on the same Windows machine. The results are listed in Table 6 and it shows that RATScope achieves much faster parsing speed than the state-of-the-art libraries, that is, nearly 6 times faster than `TDH` and 2 times faster than `TraceEvent`. Faster parsing speed enables our system to surpass other systems by saving system resources and responding to attackers more quickly.

TABLE 6
Comparison of parsing speed.

	# of Parsed Event / Sec
TDH [53]	93,254
TraceEvent [55]	242,716
RATScope	552,090

Storage Overhead. We deploy our system on two machines (i.e., Alice and Bob) under the typical real-world scenario mentioned in Section 5.1 and collect data for one day on each machine. Table 7 lists compressed storage sizes before and after applying our AATR-based log reducing technique. It shows that our technique could reduce original logs by 97.61%, which confirms that there indeed exists tremendous semantic redundancy issue with the log data. Furthermore, we manually check the data after reduction to ensure that data related to RAT attacks is not removed. Overall, RATSCOPE generates 0.2 GB log data per day. Considering an enterprise environment with 30 machines, RATSCOPE would generate around 2TB log data per year. With the market price for a 2TB hard drive being around 60 US dollars, we believe that the storage cost is reasonable and affordable.

TABLE 7
Storage overhead of running RATSCOPE for 1 day.

	Alice	Bob	Average
Before Reduction	7.5 G	9.4 G	8.4 G
After Reduction	0.18 G	0.23 G	0.2 G
Reduction Ratio	2.4%	2.44%	2.38%

6 DISCUSSION

Evasion. Our system is more robust than the state-of-the-art technique, i.e., traditional signature-based techniques [71], which can easily be evaded by static obfuscation techniques. Besides, our system is even resilient to dynamic obfuscation techniques like reordering independent APIs and inserting irrelevant APIs [72] due to the design of the AATR graph model. However, our system has the same evasion risk as any forensic or detection systems. If attackers know our design, they can change the implementations of PHFs to evade our system. However, we argue that 1) this is a prerequisite for all behavior-based works that aim to identify previously unknown behaviors. 2) the number of implementations of PHFs is much less than malware variants. When an unknown implementation of a PHF occurs, as long as our system can generate a behavior graph for this implementation, that PHF of any unknown RAT based on this implementation will be captured by our system still. 3) our RAT study and experiments show that although RAT families active from 1999 to 2016 are written by different programming languages and involved in different security incidents, PHFs of such RAT families have similar implementation methods. This demonstrates that our approach is effective in practice. For previous systems which do not consider the input arguments of system calls, the mimicry attack [72] can evade those system by replacing system call

arguments to mimic a benign behavior, but with a malicious twist. This attack is consistent with the *Semantic Collision* problem in Fig. 5 in which a malicious behavior and a benign behavior have the same system call trace. Our AATR graph model can solve this problem by combining both low-level system calls and higher-level call-stack trees.

Extension to other types of malware. Basically, RATSCOPE studies the general problem of inferring high-level semantics of malicious behaviors from low-level events (i.e., the Semantic Collision problem), in a sense using RAT on Windows as a case study. To solve this problem, we propose a general solution, the AATR model, which combines two different data levels, i.e., system call and library call-stack, to distinguish program behaviors which collide on a single data level. Thus we believe our system can also extend to other types of malware.

7 Related Works

Attack Causality Analysis plays a vital role in today's forensics area. The basic idea is to build causal graphs by connecting system objects like processes, files, and registries using low-level events like file IO and network operations [52], [57]. Given a detected attack point, forward and backward tracking along causal graphs will be used to find attack-related events so that it can extract a blueprint of the attack from tremendous system data [52], [57]. Many works have been proposed aiming at improving the causality graph framework. Some works [8], [9], [20], [22], [23] mitigated the dependency explosion problem by fine-grained causality tracking to reduce more unrelated data. Meanwhile, some other works [15], [16], [17], [18] focus on real-time and scalability by prioritizing the tracking process and proposing efficient data storage model. Those works can capture enough semantics of the attack in some cases. For example, in a drive-by download attack [18], it is sufficient to understand the attack by knowing what file was downloaded from what IP address. However, when it comes to RAT attacks [14], they cannot identify fine-grained behaviors (i.e., PHFs) of RATs when performing forensic analyses so that their result is too coarse-grained to be used for understanding RAT attacks.

Malware behavior modeling is a mature security research topic where researchers propose behavior models [23], [24], [73] to describe the semantics of malware behavior rather than easily changed artifacts. However, those models rely heavily on input arguments of system call which is not provided on Windows without instrumentation, it leads to the Semantic Collision problem and the failure for previous works. RATSCOPE proposes a novel AATR model to solve this problem.

Remote Access Trojan. The increasingly prevalent RAT attacks draw more attention. A few previous works focus on RAT detection [74], [75]. They rely on network-based features to detect RATs in the early stage. However, network-based methods cannot identify fine-grained semantic behaviors of RATs. Farinholt et al. [14] and Rezaeirad et al. [76] tries to understand the motivations, intentions, and behaviors of RAT. However, they only focus on two RAT families (DarkComet and NjRAT) and do not propose any approach to identify RAT behaviors. Our paper conducts a large-scale study of 53 real-world RAT families active from 1999 to 2016 and proposes a system to accurately identify RAT semantic behaviors.

8 Conclusion

In this paper, we are the first to conduct a large-scale study of real-world RAT families active from 1999 to 2016. Based on the study, we then propose RATSCOPE, an instrumentation-free RAT forensic system which can efficiently record and accurately reconstruct RAT attack with fine-grained semantics on Windows. Specifically, we improve the performance and data quality of native ETW and propose a novel AATR graph model for accurately characterizing PHFs of RATs. Evaluation results show that RATSCOPE outperformed existing forensic systems in simulated real-world RAT attack scenarios, and can achieve around 90% TPR in the cross-family evaluation and around 80% TPR on the two-year spanning temporal evaluation with near zero FPR while incurring as low as 3.7% runtime overhead.

Acknowledgment

The authors would like to thank anonymous reviewers for their feedback in finalizing this paper. We would also like to thank Dr. Xue Leng for informative discussions on the submitted manuscript. This work is supported by NSFC under U1936215. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

[1] "Shocking New Reveals From Sony Hack," https://goo.gl/18RnK9, accessed on 2020-2-10.
[2] "Sony Pictures hack," https://goo.gl/t6oJcp, accessed on 2020-2-10.
[3] "Target's Data Breach: The Commercialization of APT," https://goo.gl/cDYXCG, accessed on 2020-2-10.
[4] "Plague in (security) software drivers," https://goo.gl/kmycvb, accessed on 2020-2-10.
[5] "KHOBE – 8.0 earthquake for Windows desktop security software," https://goo.gl/5UhzpQ, accessed on 2020-2-10.
[6] "Captain Hook:Pirating AVS to Bypass Exploit Mitigations," https://goo.gl/zVyuAL, accessed on 2020-2-10.
[7] "Kernel Patch Protection — Wikipedia, The Free Encyclopedia," https://goo.gl/S4idr7, accessed on 2020-2-10.
[8] K. H. Lee, X. Zhang, and D. Xu, "High Accuracy Attack Provenance via Binary-based Execution Partition." in *NDSS*, 2013.
[9] S. Ma, X. Zhang, and D. Xu, "ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting." in *NDSS*, 2016.
[10] S. Ma, J. Zhai, F. Wang, K. H. Lee, X. Zhang, and D. Xu, "MPI: Multiple Perspective Attack Investigation with Semantic Aware Execution Partitioning," in *USENIX Security*, 2017.
[11] "Taintgrind: a Valgrind taint analysis tool," http://bit.ly/2KlOb5q, accessed on 2020-2-10.
[12] "VOLATILITY," http://bit.ly/2rJFtat, accessed on 2020-2-10.
[13] "PANDA," http://bit.ly/2OhOyiv, accessed on 2020-2-10.
[14] B. Farinholt, M. Rezaeirad, P. Pearce, H. Dharmdasani, H. Yin, S. Le Blond, D. McCoy, and K. Levchenko, "To catch a ratter: Monitoring the behavior of amateur darkcomet rat operators in the wild," in *S&P*, 2017.
[15] Y. Liu, M. Zhang, D. Li, K. Jee, Z. Li, Z. Wu, J. Rhee, and P. Mittal, "Towards a Timely Causality Analysis for Enterprise Security," in *NDSS*, 2018.
[16] W. U. Hassan, M. Lemay, N. Aguse, A. Bates, and T. Moyer, "Towards Scalable Cluster Auditing through Grammatical Inference over Provenance Graphs," in *NDSS*, 2018.

[17] P. Gao, X. Xiao, Z. Li, F. Xu, S. R. Kulkarni, and P. Mittal, "AIQL: Enabling Efficient Attack Investigation from System Monitoring Data," in *USENIX ATC*, 2018.

[18] M. N. Hossain, S. M. Milajerdi, J. Wang, B. Eshete, R. Gjomemo, R. Sekar, S. D. Stoller, and V. Venkatakrishnan, "SLEUTH: Real-time attack scenario reconstruction from COTS audit data," in *USENIX Security*, 2017.

[19] K. Pei, S. Gu, Zhongshu, and et al., "HERCULE: attack story reconstruction via community discovery on correlated log graph," in *ACSAC*, 2016.

[20] S. Ma, K. H. Lee, C. H. Kim, J. Rhee, X. Zhang, and D. Xu, "Accurate, low cost and instrumentation-free security audit logging for windows," in *ACSAC*, 2015.

[21] L. Kyu Hyung, Z. Xiangyu, and X. Dongyan, "LogGC: garbage collecting audit log," in *SIGSAC*, 2013.

[22] S. Krishnan, K. Z. Snow, and F. Monrose, "Trail of bytes: efficient support for forensic analysis," in *CCS*, 2010.

[23] Y. Kwon, F. Wang, W. Wang, K. H. Lee, W.-C. Lee, S. Ma, X. Zhang, D. Xu, S. Jha, G. Ciocarlie *et al.*, "MCI: Modeling-based Causality Inference in Audit Logging for Attack Investigation," in *NDSS*, 2018.

[24] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X.-y. Zhou, and X. Wang, "Effective and Efficient Malware Detection at the End Host." in *USENIX Security*, 2009.

[25] "Adwind has resurfaced targeting enterprises in the Aerospace industries worldwide," https://goo.gl/uAidwD, accessed on 2020-2-10.

[26] "Adwind resurfaces, targeting Danish companies," https://goo.gl/aJjE8J, accessed on 2020-2-10.

[27] "APT-C-27 (Goldmouse): Suspected Target Attack against the Middle East with WinRAR Exploit," http://bit.ly/2NP3yoY, accessed on 2020-2-10.

[28] "Simple njRAT Fuels Nascent Middle East Cybercrime Scene," https://goo.gl/esSXeb, accessed on 2020-2-10.

[29] E. Galperin, M. Marquis-Boire, and J. Scott-Railton, "Quantum of Surveillance: Familiar Actors and Possible False Flags in Syrian Malware Campaigns," *Electronic Frontier Foundation*, 2013.

[30] "HackForums.net," https://goo.gl/dHGFKU, accessed on 2020-2-10.

[31] "Hellbound Hackers," https://goo.gl/3Xi1zg, accessed on 2020-2-10.

[32] "Offensive Community," https://goo.gl/jiFd3A, accessed on 2020-2-10.

[33] "Researchers Uncover RSA Phishing Attack, hding in plain sight," https://goo.gl/5EjYUZ, accessed on 2020-2-10.

[34] "Hackers used "Poison Ivy" malware to steal chemical, defense secrets," https://goo.gl/JF3ZrE, accessed on 2020-2-10.

[35] "DarkComet RAT Used in New Attack on Syrian Activists," https://goo.gl/HPCZmv, accessed on 2020-2-10.

[36] "How Hackers Are Using JeSuisCharlie To Spread Malware," https://goo.gl/8Yjg1N, accessed on 2020-2-10.

[37] "New Xtreme RAT Attacks US, Israel, and Other Foreign Governments," https://goo.gl/MgmKm5, accessed on 2020-2-10.

[38] "Malware Spy Network Targeted Israelis, Palestinians," https://goo.gl/BdqSPm, accessed on 2020-2-10.

[39] "XtremeRAT: Nuisance or Threat?" https://goo.gl/MvhNEB, accessed on 2020-2-10.

[40] "Program for determining types of files for Windows, Linux and MacOS," http://bit.ly/2NNM2RV, accessed on 2020-2-10.

[41] "DirectX.Capture Class Library," http://bit.ly/2XiePRD, accessed on 2020-2-10.

[42] "Atomic Red Team," http://bit.ly/32K9tA1, accessed on 2020-2-10.

[43] "CALDERA plugin: Adversary," http://bit.ly/354WJFK, accessed on 2020-2-10.

[44] "Red Team Automation (RTA)," http://bit.ly/2q3s50B, accessed on 2020-2-10.

[45] "Metta," http://bit.ly/2rJDVxb, accessed on 2020-2-10.

[46] "Empire: a PowerShell and Python post-exploitation agent," http://bit.ly/2NLVtBn, accessed on 2020-2-10.

[47] "Desktop Operating System Market Share Worldwide," http://bit.ly/3410xHC, accessed on 2020-2-10.

[48] "Windows-10-Mitigation-Improvement," https://ubm.io/2IIVwtn, accessed on 2020-2-10.

[49] "Hardening Windows 10 with zero-day exploit mitigations," https://bit.ly/2KdiTiv, accessed on 2020-2-10.

[50] P. Gao, X. Xiao, D. Li, Z. Li, K. Jee, Z. Wu, C. H. Kim, S. R. Kulkarni, and P. Mittal, "SAQL: A Stream-based Query System for Real-Time Abnormal System Behavior Detection," in *USENIX Security*, 2018.

[51] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, "NODOZE: Combatting Threat Alert Fatigue with Automated Provenance Triage," in *NDSS*, 2019.

[52] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen, "Enriching Intrusion Alerts Through Multi-Host Causality." in *NDSS*, 2005.

[53] "Retrieving event data using TDH," https://bit.ly/2AXsKBS, accessed on 2020-2-10.

[54] Microsoft, "KrabsETW: a modern C++ wrapper and a .NET wrapper around the low-level ETW trace consumption functions." https://github.com/Microsoft/krabsetw, 2018, accessed on 2020-2-10.

[55] "TraceEvent: a library designed to make controlling and parsing Event Tracing for Windows (ETW) events easy." https://bit.ly/2Ekvk9q, 2018, accessed on 2020-2-10.

[56] "Common fields in ETW events," https://bit.ly/2zvJLDr, accessed on 2020-2-10.

[57] S. T. King and P. M. Chen, "Backtracking intrusions," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 223–236.

[58] "Debugging with Symbols," https://bit.ly/2RIYOkl, 2018, accessed on 2020-2-10.

[59] "Trojan Hidden in Fake Revolutionary Documents Targets Syrian Activists," http://bit.ly/2pgmbsm, accessed on 2020-2-10.

[60] "Syrian Malware, the ever-evolving threat," http://bit.ly/2NQC96e, accessed on 2020-2-10.

[61] "The Syrian spyware to target the opposition activists," http://bit.ly/2KnNPuW, accessed on 2020-2-10.

[62] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time," in *USENIX Security*, 2019.

[63] "New Mac backdoor using antiquated code," http://bit.ly/3587ff5, accessed on 2020-2-10.

[64] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "HOLMES: real-time APT detection through correlation of suspicious information flows," in *S&P*, 2019.

[65] "ATT&CK in Practice Primer to Improve Your Cyber Defense," https://bit.ly/32vCG1c, accessed on 2020-2-10.

[66] "NjRAT source code," http://bit.ly/2OkknHA, accessed on 2020-2-10.

[67] "Is There A New njRAT Out There," http://bit.ly/32Qj7Ro, accessed on 2020-2-10.

[68] "AForge.NET Framework," https://bit.ly/3jG8g6n, accessed on 2020-2-10.

[69] "Global keyboard and mouse listeners for Java." http://bit.ly/32KaaJD, accessed on 2020-2-10.

[70] "Global keyboard and mouse listeners for C#," http://bit.ly/33OSk9n, accessed on 2020-2-10.

[71] "Signatures of RATs," http://bit.ly/2QnEaIC, accessed on 2020-2-10.

[72] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, 2002, pp. 255–264.

[73] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in *S&P*, 2005.

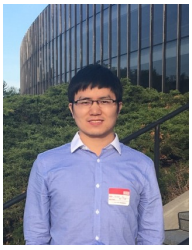
[74] D. Jiang and K. Omote, "An approach to detect remote access trojan in the early stage of communication," in *AINA*, 2015.

[75] D. Adachi and K. Omote, "A host-based detection method of remote access trojan in the early stage," in *ISPEC*, 2016.

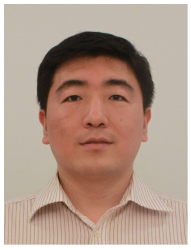
[76] M. Rezaeirad, B. Farinholt, H. Dharmdasani, P. Pearce, K. Levchenko, and D. McCoy, "Schrödinger's RAT: Profiling the Stakeholders in the Remote Access Trojan Ecosystem," in *USENIX Security*, 2018.

Runqing Yang received his B.Eng. degree in software engineering from HeFei University of Technology, China, in 2015. He is currently pursuing the Ph.D. degree with the college of computer science and technology, Zhejiang University, Hangzhou, China. His research interests include intrusion detection and attack investigation.

Xutong Chen received his B. Eng. degree from Shanghai Jiao Tong University, Shanghai, China, in 2016. He is currently a graduate student of Northwestern University, IL, USA. His research interests include system security, blockchain security and forensic analysis.



Haitao Xu is a Professor in the School of Cyber Science and Technology, Zhejiang University. His research focuses on the intersection of Cyber Security, User Privacy, and Data Analytics. His work on underground cybercrime investigation has been covered by multiple medias, including the Wall Street Journal and Forbes.



Yueqiang Cheng is experienced researcher with a demonstrated history of working in the research industry. Skilled in Xen, SGX, Linux Kernel and data privacy. Strong research professional with a PhD focused on System Security, software security, data privacy and hardware security.

Chunlin Xiong received his B.Eng. on computer science from Xian Jiaotong University, Xian, China, in 2015. He is currently a candidate of Ph.D. with the college of cyber security, Zhejiang University, China. His research interests include system security, software security and forensic analysis.

Linqi Ruan is a graduate student of Zhejiang university. She mainly interested in endpoint security and reverse engineering.

Mohammad Kavousi received his BSc degree from Sharif University of Technology, Tehran, Iran. He is currently working towards his PhD at Northwestern. His research interests include forensics, system security and operating systems.

Zhenyuan Li received the B.E. from Xidian University, Xi'an, China, in 2017. He is currently a Ph.D. candidate at Zhejiang University, Hangzhou, China. His research interests include systems security, threat detection and forensic.

Liheng Xu received his B.Eng. from Zhejiang university, Hangzhou, China, in 2016. He is currently a graduated student of Zhejiang university, China. His research interests include system security and cloud native security.

Yan Chen received the Ph.D. degree in computer science from the University of California, Berkeley, CA, USA, in 2003. He is a Professor with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL, USA. Based on Google Scholar, his papers have been cited over 7000 times and his h-index is 34. His research interests include network security, measurement, and diagnosis for large-scale networks and distributed systems. Prof. Chen won the Department of Energy (DoE) Early CAREER Award in 2005, the Department of Defense (DoD) Young Investigator Award in 2007, and the Best Paper nomination in ACM SIGCOMM 2010.