RATScope: Recording and Reconstructing Missing RAT Attacks for Forensic Analysis with Semantics on Windows

Abstract

Remote Access Trojan (RAT)-based attacks have become prevalent, which represent a serious threat to enterprise security. To understand ramification of RAT attacks and discloses attackers' actions, forensic systems need first to record sufficient audit logs for a long period of time and then to reconstruct accurate fine-grained semantic RAT behaviors. However, existing forensic systems either are blind to RAT behaviors or incur high overhead.

To address these problems above, we propose RATScope, an instrumentation-free semantic-aware forensic system for RAT attacks on Windows. Specifically, we build a more comprehensive audit system upon the Windows built-in Event Tracing for Windows (ETW), and propose a novel technique to reconstruct semantic behaviors of RATs on recorded data. Evaluation results on a representative set of RAT attacks from 2006 to 2017 show that our system can accurately identify fine-grained semantic behaviors of unknown RAT with trivial overhead.

1 Introduction

From cyber theft of personal financial information to Advanced Persistent Threat (APT) attacks aiming at intellectual properties or critical infrastructures, today's Remote Access Trojans (RATs) cause a wide range of damage to individual users, corporations and governments [22, 25, 26]. Unfortunately, RAT attacks are characterized by long-term persistence and devastating consequences, which render targeted efficient forensic system a pressing requirement to security practitioners.

To understand RAT attacks, we conduct a measurement study of more than 500 white papers about the real-world RAT attacks from 2008 to 2018 [3], and collect and analyze 82 workable RATs belonging to 51 RAT families from 2006 to 2017. To the best of our knowledge, this is the largest collection of RAT families in academics. We already make collected RAT samples public on the Internet. And we highlight three major observations:

O1 RATs are commonly equipped with tens (10 - 40) of Potential Harmful Functionalities (PHFs), such as key logging, screen grabbing and audio recording. Identifying these fine-grained semantic behaviors can help analysts understand intent and ramification of attacks [32].

O2 A RAT attack usually is not reported until more than 3 months after the breach happens [33]. What is worse, the involved malware and its triggered intermediate traces have already been self-deleted by then.

O3 Around 90% RAT families we collected only target at Windows platforms.

According to the observations above, it will be greatly helpful to reconstruct the vast majority of RAT attacks if we can *store sufficient audit logs for long periods of time* and *accurately identify PHFs of RATs* by a forensic system *on Windows platform*. Plenty of existing research works have focused on forensics. However, most of them are facing the following challenges specific to Windows platforms:

1) Blind to fine-grained semantic behaviors of RATs. According to the observation O1, it is essential to identify PHFs of RATs. However, most of existing works [36–38, 43, 45–48, 50, 53] rely on audit logs consisting of a limited amount of security-related objects like processes, files and sockets, to diagnose attacks, and these audit logs are usually provided by native audit systems, and can capture enough semantics of the attack in some cases. For example, in a drive-by download attack [38], it is enough to understand the attack by knowing what file was downloaded from what IP address. However, when it comes to RAT attacks [32], the attack semantics represented by PHFs are not reflected in these objects. Failing to capture attack semantics slows down the progress of forensic processes and reduces the quality of forensic results.

2) Requiring instrumentation or incurring high overhead. Some previous works rely on instrumentation to obtain fruitful audit logs of applications. Among various instrumentation techniques, hooking acts as the most popular choice. It alters or augments the dynamic behavior of applications by intercepting function calls, which intrusively patch applica-

tions or operating system. Instrumentation indeed provides fruitful audit logs for identifying fine-grained semantics of application behaviors [46, 49, 51].

However, instrumentation techniques are prohibitive in the enterprise environment because intrusive patching could make applications and operating system unstable and even introduce new vulnerabilities which can be leveraged by malware [4, 14, 19]. Furthermore, patching the kernel has never been supported by Microsoft, since it can greatly compromises system security, reliability, and performance. Microsoft even integrated Kernel Patch Protection (KPP) into Windows to prevent patching the kernel [13].

And for those previous works which don't rely on instrumentation, they either simply introduce a unacceptable system overhead (sandbox and taint techniques) at runtime, meaning that they cannot monitor the system for a long time where RAT attacks often persist, or fail to fetch enough data for analyses (static analysis, memory forensic), since malwares and intermediate data have been removed, according to the observation O2.

3) Incomplete audit data peculiar to Windows. Given two challenges above, leveraging native audit systems becomes popular in recent related works [35, 36, 44, 48]. Since the majority of RATs only work on Windows according to the observation O3, we need to focus on Windows. Serving as the only native audit system on Windows, Event Tracing for Windows (ETW) [7] is an instrumentation-free and low-overhead tool providing low-level events (e.g. system call and API) of applications on Windows. However, unlike native audit systems on other platforms, ETW cannot provide input parameters for any low-level data like system call and API, which cause a serious *Semantic Collision* problem (see Sec. 3.2). This incompleteness causes that most state-of-the-art forensic approaches cannot be applied to ETW data directly since their works rely on input parameters in various ways.

Summarizing all discussion above, severe and fundamental challenges exist for recording and reconstructing RAT attacks for forensic analysis on its fine-grained semantics on Windows platform.

Our solution Our work is the first step towards solving these fundamental challenges focusing on RAT attacks. We propose RATScope, an instrumentation-free semantic-aware forensic system for RAT attacks, which maps incomplete ETW data to semantic behaviors and assists security practitioners to understand attack ramification. To the best of our knowledge, RATScope is the first forensic system on Windows that can record and accurately reconstruct RAT attack with high-level semantics described by PHFs. A comparison is shown in Figure 1.

Specifically, we solve the challenges above by first improving the performance and data quality of native ETW audit system so that we can efficiently provide sufficient audit logs for all applications, including system call, API and call-stack, and then proposing a novel multi-dimensional behavior model called Aggregated API Tree Record (AATR) Graph model where we combine system call data and library call-stack data together to describe semantic behaviors of RATs accurately.

Our contributions are summarized as follows:

- We are the first one to conduct a comprehensive study on 500+ white papers and 50+ RAT families in the last 11 years (the study of the largest scale to the best of our knowledge).
- We propose a novel method to efficiently record and effectively reconstruct RAT attacks with fine-grained semantics. We not only provide an enhanced data collector with low system overhead but also accurately identify fine-grained semantic behaviors, i.e. PHFs of RATs, using defective native data.
- We build and evaluate our system thoroughly. The evaluation results show our method has high accuracy (over 90% true positive rates and near zero false positive rates) with reasonable runtime and space overhead. More importantly, evaluation results also show our system is capable of identifying behaviors of unknown RATs, where an amazing finding is functionalities of RATs in past 11 years are implemented with a limited number of methods.

2 Key Observations and Problem Statement

In this section, we first summarize and describe the work flow of a typical RAT attack as well as some interesting observations from more than 500 white papers, and then we present the threat model and our system goals.

2.1 Anatomy of A Typical RAT Attack

The RAT toolkit consists of two main components: a RAT controller and a RAT stub. The RAT stub is installed on victim hosts while the RAT controller is owned by the attacker.

As depicted in Figure 2, the work flow of a typical RAT attack is described as follows. ❶ Reconnaissance phase. An attacker stealthily performs passive or active reconnaissance to search and select the targets with identified vulnerabilities. ❷ Delivery phase. Once the targets are determined, attackers use controllers to create dedicated RAT stub loader tailored to several selected vulnerabilities. Next, the attacker delivers the RAT stub loader to the targets through spear phishing or insiders. ❸ Installation phase. The delivered RAT stub loader is eventually triggered and executed by the user of the target host or certain auto execution services. The loader installs a RAT stub connecting back to the RAT controller on the C&C server so that the attacker obtains the privilege to directly access target environment. ❹ Action phase. The RAT controller usually has a control panel with a graphical user interface (GUI), with which the attacker could select an action with simple mouse clicks and consequently the RAT stub on the victim host will perform actions accordingly. According to

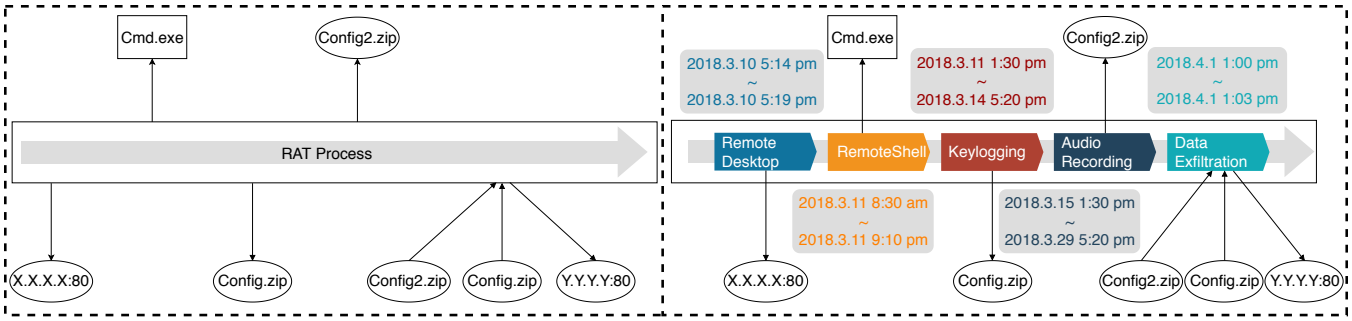

Figure 1: Reduced Version of Attack Scenario Graph. Rectangles represent processes; ovals denote files and sockets; Figure(a) shows the result of traditional forensic methods; Figure(b) shows that our approach can provide more visibility into the RAT process

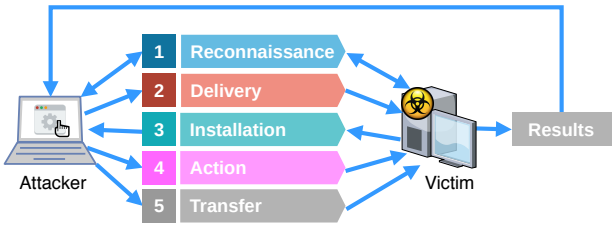

Figure 2: Work Flow of A Typical RAT Attack.

our observation, there are 5 popular potential harmful functionalities, i.e., Key Logging, Remote Desktop, Remote Shell, Audio Record, and Download and Execute, used by RATs. We summarize them into Table 1. ❺ Transfer phase. In the final phase, the attacker has collected all needed information and transfer data back to the RAT controller or any other identities controlled by the attacker. The time interval between action phase and transfer phase may be several months or even years. This phase is optional for certain RATs that have no purpose of collecting information.

Table 1: Top 5 Popular RAT Potential Harmful Functionalities

Top-5 Popular Functionalities
Key Logging: log all the keys pressed down by a victim
Remote Desktop: remotely capture and control the victim's desktop screen
Remote Shell: remotely open a console and execute arbitrary commands
Audio Record: capture audios with victim's microphone
Download and Execute: download a file and automatically execute it

2.2 Measurement of A Large RAT Data Set

RAT samples collection. In order to understand PHFs of RATs comprehensively, we need to analyze both RAT controller and RAT stub. However, RAT controllers are owned by attackers and never installed on victim's computer, so they are quite difficult to get. To address this issue, we spent lots of time searching in underground hacker forums [9, 11, 18]. After searching and availability checking, we find 82 workable RATs controllers along with their stubs as a result. Furthermore, we classify 82 RATs controllers into 51 RAT families according to GUIs, functionalities and authors. Most of them are quite notorious and involved in famous security incidents in recent years (Table 2). We are confident that these RATs are good representatives of real-world RATs.

Table 2: A Subset of RATs and Security Incidents Involved

RAT Family	Year of Activity	Security Incidents Involved
Poison Ivy	since 2006	RSA SecurID attack, 2011 [20]; Nitro attacks on chemical firms, 2011 [8]
DarkComet	since 2008	Attack on Syrian activists, 2012 [6]; leveraging the Charlie Hebdo shooting incident for malware spreading, 2015 [12]
XtremeRAT	since 2010	Attack on US, UK, Israel and other Middle East governments, 2012 [15, 17, 27]
Adwind RAT	since 2012	Attack on aerospace enterprises worldwide, 2017 [1]; Targeting Danish companies, 2016 [2]
njRAT	since 2012	Attack on India, 2014 [10]; Middle East cybercrimes, 2014 [24]; Syrian surveillance campaigns, 2013 [34]

Observations. After collecting RAT samples, we execute and analyze the functionalities of RATs. **An observation is that around 90% RATs only target at Windows.** This observation derives from a finding where around 90% RAT stub can only be generated as files specific to Windows, such as Windows PE executable files, Windows batch files and VBScript files, which means that they can only compromise Windows platforms. **Another observation is that attackers prefer writing RAT stubs in popular Windows built-in programming languages.** As we can see in Table 3, the most popular language is C# and Delphi. This is expected because (1) required runtime dependence libraries are few or none (Delphi) or installed by default (C#, Net framework installed on most Windows versions). Thus RAT written in C# or Delphi can be executed easily on most Windows computers; (2) vast amounts of ready-to-use open-source functionalities of C# and Delphi are available on Internet, helping attackers implement a RAT. In contrast, Java and Python are rarely used because performing the opposite way in two aspects above.

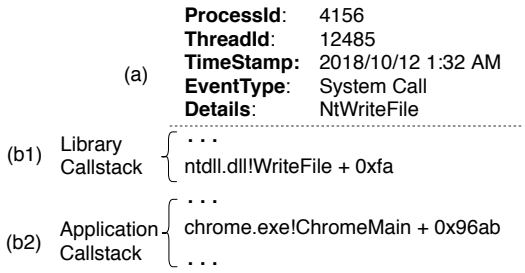

Figure 4: Data format of enhanced ETW event.

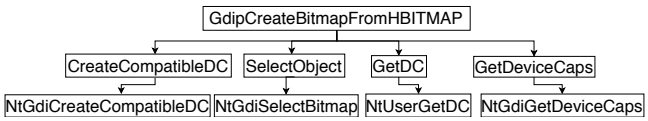

Figure 5: An AATR of GdipCreateBitmapFromHBITMAP.

ing return address of call-stack to library function. According to our evaluation, we confirm that our parser is about 6 times faster than the native parser.

A sample of enhanced ETW event can be found in Figure 4. Naturally the ETW trace stored in log files will be a sequence of ETW events arranged in chronological order (see section 4). Note that ETW does not provide any input parameters for system calls and APIs. The lacking of parameters could make identifying fine-grained semantic behaviors of RATs challenging (see Section 3.2).

3.2 Aggregated API Tree Record Graph

To recover the fine-grained semantics of RAT attacks, we must provide an accurate program behavior model to describe PHFs of RATs.

Semantic Collision Problem. The most effective state-of-the-art approach to identify program behaviors is to use behavior graphs built on system audit traces of low-level data like system call and API. Behavior-graph based approaches gained solid success in previous works [42, 44] but their success relies heavily on the input parameters because parameters not only provide essential behavior information but also are used for reconstructing causality among low-level data in behavior graphs. However, ETW lacks these parameters.

Without essential behavior information and causality reconstruction in behavior graph model, a severe problem called *Semantics Collision* will come up. *Semantics Collision* refers to the ambiguity of behavior graphs where two different program behaviors have the same behavior graph, i.e. different program behavior semantics collide on a single graph, which will lead to a ridiculous accuracy reduction. A real-world example is shown in Figure 6. *Semantics Collision* ultimately results in the failure of all previous works.

Our Solution. In summary, it's risen to critical that we need to solve semantics collision on ETW data which lacks parameters to perform effective fine-grained semantics recovery. Motivated by the example in Figure 5, two different behaviors may collide on a graph built on one single level of data (system call), but apparent difference can be found on other levels (library call-stack level). It suggests that combining multiple dimensions of low-level data is a key solution. In this work, we decide to use system call and library call-stack to build our behavior model. The reason is that for a specific program behavior, system call and library call-stack are generic and stable across different applications while application call-stack varies drastically. For the ease of our further explanation, we specify the top layer of library call-stack, which is the API invoked directly by applications, as top-layer API.

Furthermore, three vital observations are found as follows:

- System calls invoked by a top-layer API and corresponding library call-stack appear adjacently (aggregated) in ETW traces.
- Different parameters of an API basically cause different call-stack trees.
- Different call-stack trees of an API execution basically imply different parameters inputed to that API.

These observations suggest that the call-stack tree can be used for approximating parameters to mitigate the parameter lack issue and call-stack correponding to one API can be found adjacently in ETW traces. It leads to our core behavior model design proposed as Aggregated API Tree Record Graph Model (AATR Graph Model), a behavior graph built on multi-dimensional data, achieving a balance between performance and overhead. Compared to traditional behavior graph models, AATR Graph Model turns single-dimensional low-level data on its vertices to multi-dimensional data, specifically an AATR, and correspondingly adjust the edges to be the causality between AATRs.

Our design novelty surrounds the design of Aggregated API Tree Record (AATR). An AATR is the API tree structured by library call-stack invoked by a root top-layer API with the system call as leaves. AATR can be regarded as a multi-dimensional data of two different data levels. According to the observations we mentioned above, we can mitigate the defect as much as possible by leveraging call-stack. An example of AATR in practice is shown in Figure 5.

3.3 AATR Based Log Reducer

When performing online recording, after parsing the ETW audit logs collected by our collector, we find stored parsed data are usually ridiculously huge. A key reason is that each system call has a corresponding call-stack which contains average 30 layers library functions.

To reduce logs, the basic idea of our solution is to eliminate useless data and to compress duplicated data. According to this basic idea, the jobs done by our log reducer can be shown as follows:

- Eliminate application call-stack for every ETW event.
- Rearrange ETW audit logs into AATR traces.

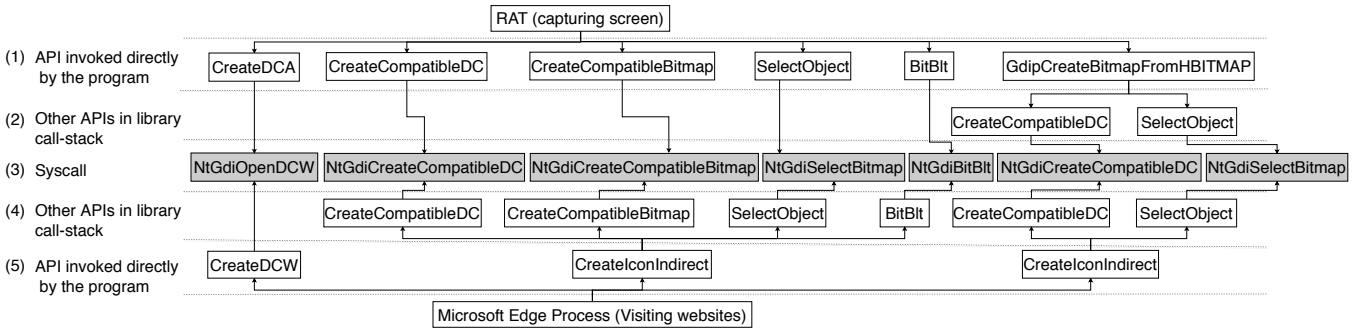

Figure 6: An example of Semantics Collion on system call level data. Two different program behaviors, visiting website and capturing screen, look like exactly the same from the perspective of traces of system calls (Block (3) in the figure). But the difference between these two program behaviors can be easily found on call-stack information.

The firs job is straightforward for these data are not needed by our AATR graph model. The intuition is that library call-stack of an AATR occurs repeatedly in corresponding adjacent ETW events. And the structure of AATR will compress these duplication so by folding duplicated parts.

3.4 AATR Graph Generator

Our Aggregated API Tree Record Graph Generator is used in offline training phase to produce AATR graphs presenting PHFs in collected PHF traces.

Typical behavior graph model generation usually generates too huge behavior graphs because they include as much as information they have from low-level data traces which are basically collected for a long enough time to ensure that collected traces don't cut off a complete program behavior. Apparently, too huge behavior graphs will lead to troubles for storing them and performing an efficient matching at runtime.

According to our observation, tremendous semantic redundancy exists in our collected ETW audit logs and a large portion of this redundancy is represented as loops in a long-time execution. It implies that if we can reduce those repeated parts, we will be able to find a compact representation for fine-grained semantics, i.e. PHFs.

Therefore, our generation process is accomplished by a redundancy-reduction-based algorithm and a causality plug-in. The basic idea of redundancy-reduction-based algorithm to leverage call-stack to precisely identify loop bodies and for causality plug-in, we just use some existing techniques to build causality for AATRs. since it is not our main contribution, we won't dive much into it in this paper.

3.5 AATR Graph Matcher

Our Aggregated API Tree Record Graph matcher is used in forensic phase to model the fine-grained semantics of reported RAT attacks.

The design of our AATR graph matcher is motivated by solving the noise problem in low-level data traces. Noise refers to some unexpected system routine due to different runtime system context, which could lead to instability where we observe differences between low-level data traces of the same program behavior. Noices are unrelated to program behaviors so matching should not be tampered by them. Since we are using multi-dimensional data (ETW event), we need care about noises in every dimension, which demands a relatively loose matching algorithm to identify PHFs represented by the graphs and meanwhile, without any loss of false-positive concern since false-positive will be introduced by a loose matching.

To address the noise problem without any loss of false-positive concern, we propose a optimal partial graph matching algorithm. The idea of it is different from finding exact occurrences of behavior graphs in audit traces as previous works do, we find a mapping between vertices of an AATR graph and an ETW trace fulfilling the causality constraint and optimizing a matching rate function which reflects how well an AATR graph is matched on the input. This method works well for a gap is found on the matching rate between a true match and other cases so that we can set a universal threshold for each PHF for triggering the match.

4 Enhanced Data Collector

Our enhanced data collector reuses the native ETW recorder which efficiently record and store low-level events in log files, and build our own parser which can efficiently parses binary ETW data, recover semantics of events and finally aggregate parsed ETW logs into enhanced events. The architecture is depicted in figure 7.

4.1 Quick Parsing

The key idea of quick parsing is to build a RAT-oriented parser. It consists of three steps: **1) filtering out RAT-unrelated events.** ETW provides over one thousand groups of log events, and collecting all these events could easily produce gigabytes of data per minute. However, after collecting

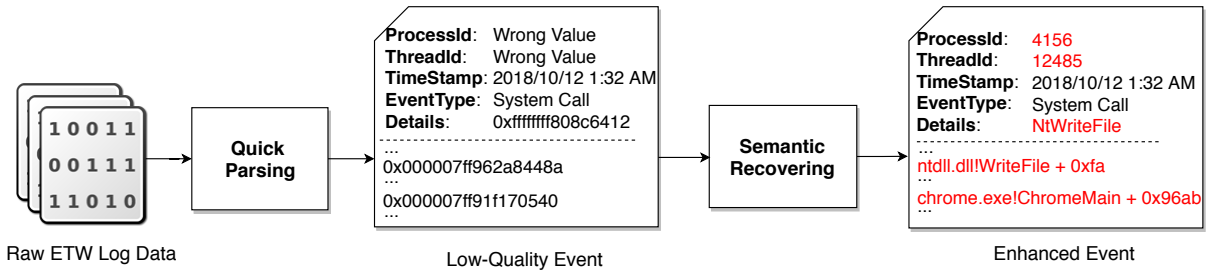

Figure 7: Architecture of Enhanced Data Collector.

and looking inside traces of behaviors of all 82 RAT samples, we observe that most events are totally unrelated to RAT (e.g. events related to Internet Explorer, Windows Defender, and Word) while behaviors of RAT can be effectively reflected in low-level events (e.g. system call and call-stack). Thus we filter out totally unrelated events and focus on system call and call-stack events. **2) picking forensic-related fields from the selected events.** Except fields specific to certain types of events, each Windows kernel level event has 18 common fields [5]. However, not all common fields are useful in RAT forensic. For example, a common field in kernel events, elapsed execution time for kernel-mode instructions, is designed to debug performance issues of applications, but useless in RAT forensic. According to previous forensic system [36, 40] and our domain knowledge on RATs, we select and parse 3 fields from 18 common fields: *process id*, *thread id* and *time stamp*. **3) creating parsing shortcuts for picked fields.** The ETW recorder stores events in a binary file. In order to correctly extract values of fields from binary, complex parsing steps are performed one by one, such as checking Windows version, analyzing data structures of field, and locating fields in ETW binary data. Our idea of optimization is to create and cache shortcuts for reusing 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 complex parsing steps for each picked field to obtain offset and data size, and store these information in a cache file. Note that the process of generating shortcuts is automatically performed without human efforts. In the online phase, the file is loaded into memory. When a new ETW event occurs, we retrieve offset and data size of a field from the memory and directly jump to the offset of ETW binary data and extract values of the field using data size without complex parsing steps.

4.2 Semantic Recovering

In addition to the parsing performance problem, some fields provided by ETW lack semantics. Specifically, 1) values of certain significant fields provided by ETW are wrong. For example, the value of process id of system call is wrong, which could cause a benign process detected as malware. 2) ETW only provides memory address of system call and call-stack rather than symbolic names. It is practical to resolve system call and call-stack to symbolic names which contain more semantics than memory address.

We address the above two problems in two steps: **1) Correcting field values.** ETW does not provide any documents describing how to correct field values. By reverse engineering, we find that field values of a event can be corrected from other events. Specifically, given a event with a wrong value, we first locate another event which have a correct value, then determine whether two events can be combined by correlating semantics of two events. For example, the value of process id of system call event is wrong. We found that the context switch event has correct value of process id. Then we analyze semantics of context switch event to determine whether context switch and system call can be combined and how to combine. Specifically, the operating system offers time slices of CPU processor to threads that are eligible to run. After a time slice is completed, a context switch event occurs which switch the CPU processor from one thread to another. Thus by tracking context switch events, we can obtain which thread is running 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. After getting the thread id, we can easily get the process id by searching which process create the thread. Finally, we correct the process id and thread id value in the original system call event. **2) resolving system call and call-stack.** The process of resolving memory address to symbolic name has two steps. First, our system locates a module which a memory address belong to, and converts raw memory address to an offset of the module. Specifically, Windows loads a module (e.g. DLL library) in a random memory space when operating system restarts. Our system leverage an event called ImageLoad from ETW to obtain dynamic mapping relationships between memory address 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 from debugging symbol files [28]. In order to resolve memory address correctly after updates or patches, our system checks and updates the mapping relationships automatically when our system is started.

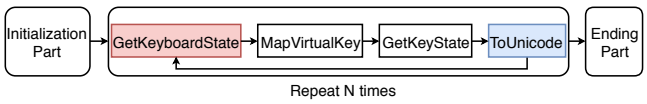

Figure 8: The collected ETW trace. The red block is the entry of the main loop body and the blue one is the exit.

4.3 Enhanced Event Format

With the above two techniques, raw ETW log data are eventually transformed into enhanced events. Figure 4 shows an example enhanced event of our logging system. Block (a) is the basic information of an ETW event include the process id (ProcessId) and the thread id (ThreadId) of the thread which invoke the event, the event type (EventType), and the details of the event (Details). In this example, EventType is system call, and Details is the syscall name. Note that EventType could also be kernel security-related events, such as FileIORead, FileIOWrite and IPV4Connect, and the Details of these events are the target object, such as the absolute path of a file, the IP address with port. Block (b) is the callstack of the event, which can be simplified into three parts: (b1) library callstack and (b2) application callstack. Each entry of the callstack represents a callee function with its lower entry being its caller function.

5 AATR Graph Generator

In this section, we explain our Aggregated API Tree Record Graph Generator design with insights detailedly.

5.1 Semantics Redundancy Problem

Low-level data traces are usually collected for a long enough time so that traces can include a complete life cycle for a program behavior. Although the whole execution can be long, core parts often repeat themselves naturally. As a concrete instance, in key logging PHF, RAT stub keeps recording keyboard strokes. The corresponding ETW trace is shown in Figure 8.

Obviously, the major semantics lies inside the main loop body while the collected trace still contains a tremendous amount of duplicated events, which leads to a severe semantics redundancy problem.

Existing works focus more on how to build and reconstruct accurate causality in low-level data but ignore semantics redundancy, which result in a huge behavior graph result, making storing of behavior graphs demanding unnecessary space and graph matching inefficient. What's more, redundant parts in a trace also contain more noise which graph matcher has to tolerate.

5.2 Redundancy-reduction-based Generation

Enlightened by the example in Figure 8, it seems that most of semantics redundancy occurs as loops. We further confirm this conjecture by conducting trace analyses, finding that most RAT execution can be divided into 3 phases.

1. (Prologue) The starting phase of the program execution. It performs initialization like reading configuration files and allocating memory for latter execution.
2. (Loops) The main part of the program execution. It contains the major functionality in several loops.
3. (Epilogue) The ending phase of the program execution. It releases all acquired system resources and finishes the execution in the end.

This observation is further confirmed in [48], a forensics paper also using ETW call-stack information.

These observations motivate us that as long as we can reduce the repeated loops, we can extract information we need to describe fine-grained semantic behavior, i.e. PHFs of RATs. Our method is to use application call-stack and library call-stack on a system call to conduct the reduction on input traces. These call-stack information accurately locates every system call in program code. And it helps distinguish whether two system call events of the same system call type refers to a revisit of the same system call in the code by the program counter. Once we confirm a revisit, we are confident that all system call events in the trace between these two events forms a loop body. After we get all loop bodies identified, we select the most representative one, apply redundancy-reduction recursively so that we can deal with nested loops not only in application code but also in library code, along with the parts of prologue and epilogue as our redundancy reduction result.

The above paragraph describes the major logics of our redundancy-based graph generation. Two more technical details are as follows. First, we unfold the input traces received from our log reducer, which resumes the data format to be multi-level data so that it is convenient for generation algorithm to manipulate. Second, we organize the redundancy-reduction result into an AATR sequence and then add the causality result on it to turn it into an AATR graph.

We give our AATR graph generation as the following pseudocodes in Algorithm 1 and Algorithm 2.

We implement our redundancy-reduction by a depth-first recursion where we first deal with top-level loops and dive into nested loops at once. This implementation follows the real work flow of the program execution and also provides ease for implementation code writing. The result shows our generated graph is basically 10k times smaller than the traditional behavior graph.

6 AATR Graph Matcher

In this section, we explain our design for Aggregated API Tree Record graph matcher. We will formalize the optimal partial matching problem and explain our solution to noise problem without loss of false-positive concern.

Algorithm 1 Redundancy-reduction Algorithm

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

function REDUNDANCY-REDUCTION(ϕ_{input})
  scan from θ_m to θ_1, find the last $\theta_{flbs} \in \phi_{input}$ where $\exists k$, θ_{flbs}=θ_k, $flbs+1 \leq k \leq m$
  $\phi_{pro} \leftarrow \theta_1 \dots \theta_{flbs-1}$
  get all loop body separator $\alpha_{lbs} \leftarrow \{j_k | \theta_{j_k}$=$\theta_{flbs}\}$
  $llbs \leftarrow max(\alpha_{lbs})$
  scan from θ_1 to θ_m, find the last $\theta_{lbe} \in \phi_{input}$ where $\exists k$, θ_{lbe}=θ_k, $flbs \leq k \leq llbs$
  ϕ_{epi}=$\theta_{lbe+1} \dots \theta_m$
  $\alpha_{lbs} \leftarrow \alpha_{lbs} \cup \{lbe+1\}$
  get the maximum gap index j_s=$\arg\max_{j_s}\ j_{s+1} - j_s$
  get selected loop body ϕ_{slb}=$\theta_{j_s} \dots \theta_{j_{s+1}}$
  get nested loop recursively reduced result $\phi_{nesrec} \leftarrow$ Redundancy-Reduction(ϕ_{slb})
  get next top-level loop recursively reduced result $\phi_{nextrec} \leftarrow$ Redundancy-Reduction(ϕ_{epi})
  $\phi_{reducted} \leftarrow$ Concatenate(ϕ_{pro}, $\phi_{nestrec}$, $\phi_{nextrec}$)
  return $\phi_{reducted}$
end function

Algorithm 2 Aggregated API Tree Record Graph Generation

Input: (1) n unfolded AATR traces collected by enhanced data collector, corresponding to one selected PHF: Φ_{phf}=$\{\phi_i | i$=$1 \dots n\}$, where trace ϕ_i=$\{\theta_j$=(syscall, library call-stack, application call-stack)$| j$=$1 \dots m\}$; (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=$\{\psi\}$, where ψ=($\cup$AATR, $\cup$causality)
Initialize: $\Psi \leftarrow \emptyset$.

procedure AGGREGATED API TREE RECORD GRAPH GENERATION
  Preprocess each trace $\phi_i \in \Phi_{phf}$.
  for each trace $\phi_i \in \Phi_{phf}$ **do**:
    $\psi_i \leftarrow$ REDUNDANCY-REDUCTION(ϕ_i)
    Eliminate application call-stack in ψ_i and organize ψ_i to be an AAR behavior sequence
    $\psi_i \leftarrow$ CausalityEngine(ψ_i)
    $eligible \leftarrow true$
    for each trace $\phi_{i2} \in (\Phi_{other} \cup \Phi_{benign})$ **do**
      if ψ_i matched ϕ_{i2} **then**
        $eligible \leftarrow false$
      end if
    end for
    if *eligible* **then**
      $\Psi \leftarrow \Psi \cup \{\psi_i\}$
    end if
  end for
end procedure

6.1 Optimal Partial Matching Problem

As mentioned in system overview, our task is to find the mapping between vertices of an AATR graph and ETW trace event fulfilling the causality constraint, i.e. the chronological order between AATRs and meanwhile maximize the matching rate function which is designed to reflect how well an AATR graph is matched. For ease of our further explanation, we formalize optimal partial matching program as follows.

Definition 1 (Optimal Partial Matching Problem) *Given a labeled direct acyclic simple graph G=(V=$\{v_i | 1 \leq i \leq m\}$,$E \subseteq V \times V$) and a labeled sequence S=$\{s_j | 1 \leq j \leq n\}$, find an one-to-one mapping $f : V' \leftrightarrow S'$, where $V' \subseteq V$ and $S' \subseteq S$, so that (1):f can maximize a matching rate function $t : P(V \times S) \rightarrow [0,1]$; (2) for $\forall v_x, v_y \in V'$, if there is a path from v_x to v_y in G, s_x=$f(v_x) < s_y$=$f(v_y)$.*

Here $P(V \times S)$ represents the power set of $V \times S$ and N represents the set of natural number. Function t is the matching rate function which is designed to reflect how well the graph is matched. Its definition will be given in 6.2.

Consider how we build the optimal mapping f, when we decide to map a vertex v_i to an event s_j, all previous vertices of v_i can only be matched to events prior to s_j due to the contraint (2). This observation is more clear in a special case of our matching problem where G is a sequence. In this case our matching problem become the Maximal Weighted Common Sequence problem (a generalized version of Longest Common Sequence problem) where the optimal substructure property is clear of matching starting i vertices in G to starting j events in S. This finally motivate our dynamic-programming-based matching algorithm on AATR graph matching and it will be introduced in 6.3.

6.2 Matching Rate Function

To reflect how well an AATR graph is matched, our matching rate function is given as follows:

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 S=$\{s_i | 1 \leq i \leq n\}$ and an one-to-one mapping $f : V' \leftrightarrow S'$, where $V' \subseteq V$ and $S' \subseteq S$, a matching rate function $t : P(V \times S) \rightarrow N$ is defined as t=$\frac{\sum_{\forall (v,s) \in f} bonus(v,s)}{\sum_{\forall v \in V} bonus(v,v)}$, where bonus is a mapping :$V \times S \rightarrow N$ representing a matching score between a pair of v and s.*

Further, given a pair of (v,s) where v=$(sysc_v, lib_stack_v)$ and s=$(sysc_s, lib_stack_s)$. Here $sysc_v$ is the system call of v and lib_stack_v is the corresponding library call-stack. And the score function can be given as follows:

$$bonus(v,s) = \begin{cases} 1+\Delta(lib_stack_v, lib_stack_s) & sysc_v = sysc_s \\ 0 & otherwise \end{cases}$$

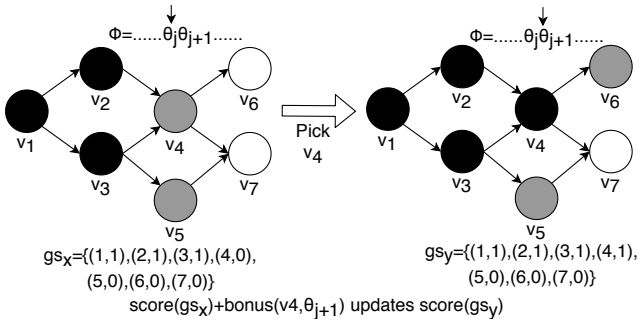

Figure 9: An example of dynamic programming transition on graph states. The gray vertices stand for Frontier Vertex Set in each step. They are vertices which can be picked for next step because all of their previous vertices are covered.

Here $\Delta(lib_stack_v, lib_stack_s)$ is the longest common part in library call-stacks of v and s starting from bottom-level to top-level.

The intuitive explanation of matching rate function is that we sum up the similarity of every matched (v,s) pair and use the rate of this summed score over the score of a perfect matching, i.e. $\sum_{\forall v \in V} bonus(v,v)$ to get our normalized matching rate. Obviously, the higher value of the matching rate function reflects a better match of the AATR graph.

6.3 Dynamic Programming based Matching

To perform a dynamic programming (DP) based graph matching for AATR graph, we need to define the transition state representing every subproblem so that DP algorithm can do its transition. In this work, we define it as Graph State as follows:

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 in the subgraph represented by gs and $gs(v)$=1 means it has not.

Our DP transition will be performed to calculate matching rate function t for subproblems on the transition graph built on graph states. One transition step will act on a graph state by picking next vertex to be included. Note that we can only pick the vertices in Frontier vertex set to not violate constraint (2) in Definition 1. A concrete example of a step of DP transition on a graph state is shown in Figure 9.

Our algorithm pseudocode is given as Algorithm 3 and 4.

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$\} \subseteq (V_{tr} \times V_{tr})$,$label : E_{tr} \to V$)
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 ≠ ∅ do
        gs_head ← tpsortQueue.pop()
        V_frontier ← GetFrontier(gs_head, ψ)
        for each v_itr ∈ V_frontier do
            gs_new ← gs_head
            gs_new(v_itr) ← 0
            V_tr ← V_tr ∪ {gs_new}
            E_tr ← E_tr ∪ {(gs_head, gs_new)}
            label((gs_head, gs_new)) ← v_itr
            if gs_new ∉ addedGs then
                tpsortQueue.push(gs_new)
                addedGs ← addedGs ∪ {gs_new}
            end if
        end for
    end while
    return TG
end procedure
```

7 Evaluation

In this section, we evaluate RATScope with a large set of real-world RAT samples and benign applications by answering the following questions.

- **Q1.** Can RATScope effectively identify fine-grained semantics of attacks caused by existing real-world RATs as well as previously unknown RATs?
- **Q2.** Can RATScope be deployed on real-world systems with acceptable performance and storage overhead?

All machines used in our experiments have the same hardware configuration which is 3.3 GHz Intel i5-4590 CPU and 4 GB RAM.

7.1 Matching Accuracy

7.1.1 Experiment Setup

PHF Trace Collection. We trigger 5 PHFs in Table 1 for all 82 workable RATs from 51 families in Table 3 and collect traces for every PHF of every RAT sample.

Benign Trace Collection. We select tens of popular applications of various utilities. Particularly, we select several benign applications who have functionalities similar to the selected PHFs, such as audio-related (QuicktimePlayer and Audiorecorder), shell-related (CMD and Remote Utility), screen-related (SnippingTool, Teamviewer), and download-related (FreeDownloadManager) for a more strict false-positive test.

True-Positive Experiment. We apply 10-fold cross validation method on collected traces for each PHF. Specifically, we split traces of each PHF into 10 folds randomly and pick one fold as testing set iteratively and the rest part as training set. Then we generate behavior patterns for each trace in training set, and match these patterns on testing set. To demonstrate the strength of our AATR design, we generate three kinds of patterns which are AATR model, TopAPI model where only top-layer APIs are used to describe PHFs, and Syscall model

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=$\cup$AATR, E=$\cup$causality);
Output: A matching rate $\in [0,1]$
Initialize: TG=(V_{tr},E_{tr},$label$) $\leftarrow$ 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**
          $score(gs_{head}, \theta_j) \leftarrow score(gs_{pred}, \theta_j)$
        end if
        $delta \leftarrow score(gs_{pred}, \theta_{j-1})$+$bonus(label_{e_1}, \theta_j))$
        if $delta > score(gs_{head}, \theta_j)$ **then**
          $score(gs_{head}, \theta_j) \leftarrow delta$
        end if
      end for
      for each gs_{succ} where e_2=$(gs_{head}, gs_{succ} \in E_{tr})$ **do**
        if $gs_{succ} \notin dpQueue$ **then**
          $dpQueue.push(gs_{succ})$
        end if
      end for
    end while
  end for
  $finalScore \leftarrow score(gs_{fin}, \theta_m)$
  return $finalScore/maxScore$
end procedure

where only system calls are used. The results are shown in the `TP Rate` column in Table 4.

False-Positive Experiment. Our false-positive experiment consists of two parts: 1) Non-PHF test. For each PHF, we use traces of this PHF as the training set and use traces of the other 4 PHFs as the testing set. 2) Benign test. For each PHF, traces of this PHF are considered as training set and benign traces as testing set. The results are listed in the `FP Rate` column of Table 4, respectively.

7.1.2 Results Analysis

RATScope is capable of identifying PHFs of unknown RATs accurately. From the TP testing result in Table 4, we can see that for every PHF, almost all unknwon RATs can be matched with AATR graphs generated by known RATs. To further explore the reason, we manually reverse engineer all RAT samples and analyze implementation methods of each PHF. Our finding is the implementation methods for one PHF are homogeneous across various RATs in the past 11 years. For example, the AATR graph of remote shell of SpyNet RAT which is active since 2008 can match that of 28 other RATs, meaning they implement remote shell using the same method.

AATR model has much less false-positve than TopAPI model and System call model. From Table 4, AATR model has a high TP rate close to TopAPI and Syscall model, while it has a much lower FP rate than the other two models. The results are consistent with our observations about *Semantic Collision*. Note that the FP rate of AATR model on the non-phf test is still relatively high. After manually checking the traces, we find that RATs of these traces enable some PHFs like key logging by default, which means these traces actually perform these PHFs and should not be considered as False-positive. After correcting these results, all FP rates of AATR model on the non-phf test reduce to 0%.

AATR model has some slight false-positive on benign traces. Still from Table 4, AATR model succeeds in identifying audio record, remote shell, and download&exec perfectly, but some slight FPs on the benign test occur on remote desktop and key logging. Two major reasons are 1) benign applications truly perform actions similar to RATs. For example, some RATs implement remote desktop by capturing screens continuously, and the built-in `snipping tool` on Windows uses the similar method to implement capturing screen. 2) some RATs functionalities are implemented by very low-level APIs so that the call-stack tree is not deep enough to differentiate AATR model from Syscall model and TopAPI model. For example, GetKeyboardState, a core API in key logging, invokes the system call NtUserGetKeyboardState directly. We suppose these behaviors should be reported as it is exactly what RAT can do, especially in the first case, but still we think we can build a white list for a small number of benign applications to avoid both kinds of false-positives if a harsh requirement exists.

Evasion Discussion. The key reason why our system can detect previously unknown RATs is that our system model semantic behaviors of RATs and different RAT families use the same implementations of PHFs. If attackers know our design, they can change the implementations of PHFs to evade our system. However, we argue that 1) Our system is more robust than state-of-the-art technique, i.e. traditional signature-based techniques [23], which can easily be evaded by static obfuscation techniques. What's more, our system is even resilient to dynamic obfuscation techniques like reordering APIs and API Injection. 2) The number of implementations of PHFs is limited and much less than malware variants. When an unknown implementation of PHFs occurs, as long as our system can generate a behavior graph for this implementation, any unknown RAT which is based on this implementation will be detected by our system. 3) Our work can increase needed efforts from attackers. In order to evade our system, they have to find an unknown implementaion of each PHF, which generally cannot be obtained easily.

Table 4: Detection accuracy of three models on 5 PHFs.

PHF	Model	TP Rate	FP Rate	
			NonPHF	Benign
Audio Record	**AATR**	**100%**	**0%**	**0%**
	TopAPI	100%	10.51%	10.20%
	Syscall	100%	10.51%	48.97%
Remote Shell	**AATR**	**95.00%**	**4.16%**	**0%**
	TopAPI	86.30%	11.10%	2.04%
	Syscall	96.11%	18.05%	32.65%
Download & Exec	**AATR**	**89.40%**	**0%**	**0%**
	TopAPI	91.90%	17.46%	14.28%
	Syscall	93.99%	9.32%	38.77%
Remote Desktop	**AATR**	**92.30%**	**4.36%**	**4.08%**
	TopAPI	94.03%	17.06%	20.41%
	Syscall	96.33%	19.64%	42.85%
Key Logging	**AATR**	**83.50%**	**2.3%**	**2.04%**
	TopAPI	90.01%	16.26%	10.20%
	Syscall	90.50%	22.42%	20.41%

7.2 Performance and Storage Overhead

ETW Recording Overhead. To evaluate ETW recording overhead, we need to measure the overhead under different workloads. Therefore, we conduct two experiments to exhibit the performance of ETW.

Firstly, we build a benchmark to generate diversified types of ETW events and control the number of events generated per second and show a measurement on it. As Figure 10 (a) depicted, the overhead stays lower than 7% under 20,000 events per second while it can go up to 83.72% under 229,475 events per second, which is maximal events the machine can generate per second. This indicates that the runtime overhead of the ETW recorder hinges on the number of ETW events generated per second.

Secondly, we observe how ETW performs in real-world system environments. Specifically, we select 12 popular applications including browsers, messaging applications, editors, media players, server-side software and development tools. Then we leverage a GUI testing tool to automatically trigger functionalities of these applications repeatedly, such as visiting websites, receiving and sending emails, editing documents, playing videos, online voice chatting and compressing files. Meanwhile, we repeatably collect traces for them and calculate average number of generated events per second. The results are shown in Figure 10 (b). Provided the average number of generated events per second of most applications is smaller than 20,000, the ETW runtime overhead will be basically lower than 7% according to Figure 10 (a).

To further investigate the results, we look into the details of traces. We find that the number of events per second goes up to 30,000 $\sim$ 40,000 events per second only when applications start up, which could incur 10% $\sim$ 14% overhead. The reason is that applications usually invoke lots of APIs to initialize themselves by loading libraries, reading configuration files and checking machine states. Since the duration of start up

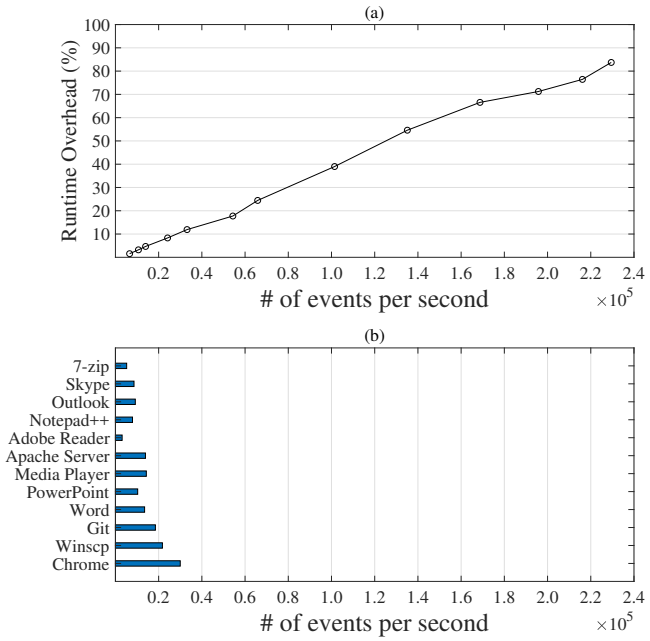

Figure 10: Figure (a) shows the runtime overhead with different numbers of events per second. Figure (b) shows the number of events per second generated by different kinds of popular applications.

is short and users seldom restart applications frequently, this won't be a big trouble for deploying our system. For some other exceptions in the figure, Chrome and Windows generates over 11% overhead due to their program complexity while Winscp and Git incur around 7% overhead because they were always downloading files from Internet, which leads into a generation of many IO-related ETW events.

ETW Parsing Overhead. To show the efficiency of our proposed parsing techniques, we conduct a comparing experiment between our system and other built-in parsing libraries mentioned in Section 4. Given the same raw ETW log file, we parse this file with every parsing tool on the same Windows platform in the same machine. The final results are listed in Table 5 and it shows that RATScope is nearly 6 times faster than TDH and 2 times faster than TraceEvent (state-of-the-art library). Note that another library called Krabsetw does not support parsing from log files so we don't compare with it. Faster parsing speed helps our system to surpass other systems by saving system resources and responding to attackers more quickly.

Table 5: Comparing Parsing speed of different ETW parsers.

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

Matching Overhead. Our matching overhead consists of

preprocessing overhead and graph matching overhead. To evaluate it, we collect traces of three representative real-world scenarios and apply preprocessing and graph matching respectively. The user on the scenario A installs a tiny Web server, a FTP server and a SSH server for internal communications. The user on the scenario B is an officer who usually visits web pages and watches videos with `Chrome`, sends or receives emails with `Outlook`, and edits documents with `PowerPoint`. The user on the scenario C is a software development engineer who usually writes codes with `Visual Studio`, visits web pages with `Chrome`, and reads programming manuals with `Adobe Reader`. The result are listed in Table 6. Note that the preprocessing overhead is very heavy which could slow download the process of forensic. We argue that 1) traces are sent to servers and processed in servers, and users would not be affected by the preprocessing. 2) Each trace only need to be preprocessed one time. 3) Current preprocessing program is single-thread and we haven't optimize it. In the future, we can leverage multiple CPU cores to preprocess traces in parallel to reduce preprocessing time.

Table 6: Matching Overhead under different scenarios.

Scenario	A	B	C
Recording Time	297s	301s	286s
Preprocessing Time	306s	446s	353s
Matching Time	152s	278s	195s

Storage Overhead. We evaluate the storage overhead under the same three typical real-world scenarios mentioned in Section 7.2. We deploy our system under these three scenarios and collect data for one day respectively. Table 7 lists storage sizes before and after applying our AATR-based log reducing technique. This technique can get a good compression ratio because most of call-stacks of neighbor events are the same.

Table 7: Storage overhead of one day under three scenarios.

Scenario	A	B	C	Avg.
Storage Size Before Reduction	0.45 GB	0.97 GB	0.61 GB	0.68 GB
Storage Size After Reduction	0.09 GB	0.24 GB	0.15 GB	0.16 GB
Compression Ratio	20%	24.7%	24.6%	23.1%

8 Related Works

Attack Causality Analysis plays a vital role in today's forensics area. The basic idea of original causality analysis is to build causal graphs by connecting system objects like processes, files and registries using low-level events like file IO and network operations. 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 [40, 41]. Many works have been proposed aiming at improving causality graph framework. Some works [43, 44, 46, 48, 50] mitigated the dependency explosion problem by fine-grained causality tracking to reduce more unrelated data. Meanwhile, some other works [36–38, 47] focus on real-time and scalability by prioritizing the tracking process and proposing effiecient data storage model. However, previous works don't concentrate on the attacker's intention when performing forensic analyses so that their result are too coarse-grained to be used for understanding RAT attacks.

Malware behavior modelling is a mature security research topic where researchers propose and evolve behavior models to describe semantics of malware behavior, keeping pace with the increasing complexity of malware. This topic derives from the need for solving evasion problems caused by static obfuscation in malware detection area. The key reason for static obfuscation to be effective is that static features used for detection are easily changed artifacts in malware binaries [54, 55]. Behavior modelling solves this evasion problem by capturing the information flows among system calls essential to malwares' task, which cannot be easily evaded by static obfuscation [31, 42]. However, these models focus on the behavior of the entire malware, which is not suitable for our PHF modelling since PHFs are just some core parts of the whole program behavior. Furthermore, most behavior models rely heavily on the parameter of system call which is not provided, it leads to the final failure for previous works.

Remote Access Trojan. The increasingly prevalent RAT attacks draw more attention. A few previous works focused on RAT detection [30, 39]. They rely on network-based features to detect RATs in early stage. However, RATs tend to be stealthy and low-traffic so that they cannot accurately detect RAT in practice. Farinholt et al. [32] tried to understand the motives, intentions, and procedures of RAT. However, they only focus on one RAT family and did not propose any approach to mitigate RAT attacks.

9 Conclusion

In this paper, we propose RATScope, an instrumentation-free semantic-aware forensic system which can efficiently record and reconstruct RAT attack with fine-grained semantics on Windows. We propose novel methods on ETW collector to achieve a fast parsing and a low space overhead along with a novel AATR graph model for describing PHFs of RATs. Evaluation results show that RATScope can record low-level events including syscalls with call-stack for long periods of time with reasonable runtime and space overhead, achieving great accuracy of identifying semantics of RAT attacks.

References

[1] Adwind has resurfaced targeting enterprises in the

Aerospace industries worldwide. https://goo.gl/uAidwD. Accessed on 2019-02-14.

[2] Adwind resurfaces, targeting Danish companies. https://goo.gl/aJjE8J. Accessed on 2019-02-14.

[3] APTnotes data. https://goo.gl/jgkCsS. Accessed on 2019-02-14.

[4] Captain hook:pirating avs to bypass exploit mitigations. https://goo.gl/zVyuAL. Accessed on 2019-02-14.

[5] Common fields in etw events. https://bit.ly/2zvJLDr. accessed on 2019-02-14.

[6] DarkComet RAT Used in New Attack on Syrian Activists. https://goo.gl/HPCZmv. Accessed on 2019-02-14.

[7] Event tracing for windows. https://goo.gl/EnQpPq. Accessed on 2019-02-14.

[8] Hackers used "Poison Ivy" malware to steal chemical, defense secrets. https://goo.gl/JF3ZrE. Accessed on 2019-02-14.

[9] HackForums.net. https://goo.gl/dHGFKU. Accessed on 2019-02-14.

[10] Hacking virus 'Bladabindi' targets Windows users in India. https://goo.gl/ABKPFX. Accessed on 2019-02-14.

[11] Hellbound Hackers. https://goo.gl/3Xi1zg. Accessed on 2019-02-14.

[12] How Hackers Are Using JeSuisCharlie To Spread Malware. https://goo.gl/8Yjg1N. Accessed on 2019-02-14.

[13] Kernel Patch Protection — Wikipedia, The Free Encyclopedia. https://goo.gl/S4idr7. Accessed on 2019-02-14.

[14] Khobe – 8.0 earthquake for windows desktop security software. https://goo.gl/5UhzpQ. Accessed on 2019-02-14.

[15] Malware Spy Network Targeted Israelis, Palestinians. https://goo.gl/BdqSPm. Accessed on 2019-02-14.

[16] Missing events in etw. https://bit.ly/2E5BUPY. accessed on 2019-02-14.

[17] New Xtreme RAT Attacks US, Israel, and Other Foreign Governments. https://goo.gl/MgmKm5. Accessed on 2019-02-14.

[18] Offensive Community. https://goo.gl/jiFd3A. Accessed on 2019-02-14.

[19] Plague in (security) software drivers. https://goo.gl/kmycvb. Accessed on 2019-02-14.

[20] Researchers Uncover RSA Phishing Attack, hding in plain sight. https://goo.gl/5EjYUZ. Accessed on 2019-02-14.

[21] Retrieving event data using TDH. https://bit.ly/2AXsKBS. Accessed on 2019-02-14.

[22] Shocking New Reveals From Sony Hack. https://goo.gl/18RnK9. Accessed on 2019-02-14.

[23] Signatures of rats. https://malwareconfig.com/yara/. accessed on 2019-02-14.

[24] Simple njRAT Fuels Nascent Middle East Cybercrime Scene. https://goo.gl/esSXeb. Accessed on 2019-02-14.

[25] Sony Pictures hack. https://goo.gl/t6oJcp. Accessed on 2019-02-14.

[26] Target's Data Breach: The Commercialization of APT. https://goo.gl/cDYXCG. Accessed on 2019-02-14.

[27] XtremeRAT: Nuisance or Threat? https://goo.gl/MvhNEB. Accessed on 2019-02-14.

[28] Debugging with symbols. https://bit.ly/2RIYOkl, 2018. accessed on 2019-02-14.

[29] Traceevent: a library designed to make controlling and parsing event tracing for windows (etw) events easy. https://bit.ly/2Ekvk9q, 2018. accessed on 2019-02-14.

[30] Daichi Adachi and Kazumasa Omote. A host-based detection method of remote access trojan in the early stage. In *ISPEC*, 2016.

[31] Mihai Christodorescu, Somesh Jha, Sanjit A Seshia, Dawn Song, and Randal E Bryant. Semantics-aware malware detection. In *S&P*, 2005.

[32] Brown Farinholt, Mohammad Rezaeirad, Paul Pearce, Hitesh Dharmdasani, Haikuo Yin, Stevens Le Blond, Damon McCoy, and Kirill Levchenko. To catch a ratter: Monitoring the behavior of amateur darkcomet rat operators in the wild. In *S&P*. IEEE, 2017.

[33] FireEye. M-trends 2017: A view from the front lines. https://bit.ly/2RdnXDl, 2018. accessed on 2019-02-14.

[34] Eva Galperin, M Marquis-Boire, and John Scott-Railton. Quantum of surveillance: Familiar actors and possible false flags in syrian malware campaigns. *Electronic Frontier Foundation*, 2013.

[35] Peng Gao, Xusheng Xiao, Ding Li, Zhichun Li, Kangkook Jee, Zhenyu Wu, Chung Hwan Kim, Sanjeev R Kulkarni, and Prateek Mittal. Saql: a stream-based query system for real-time abnormal system behavior detection. *Usenix Security*, 2018.

[36] Peng Gao, Xusheng Xiao, Zhichun Li, Fengyuan Xu, Sanjeev R. Kulkarni, and Prateek Mittal. AIQL: Enabling efficient attack investigation from system monitoring data. In *USENIX ATC*, 2018.

[37] Wajih Ul Hassan, Mark Lemay, Nuraini Aguse, Adam Bates, and Thomas Moyer. Towards scalable cluster auditing through grammatical inference over provenance graphs. In *NDSS*, 2018.

[38] Md Nahid Hossain, Sadegh M Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R Sekar, Scott D Stoller, and VN Venkatakrishnan. Sleuth: Real-time attack scenario reconstruction from cots audit data. In *USENIX Security*, 2017.

[39] Dan Jiang and Kazumasa Omote. An approach to detect remote access trojan in the early stage of communication. In *AINA*, 2015.

[40] Samuel T King and Peter M Chen. Backtracking intrusions. *SIGOPS*, 2003.

[41] Samuel T King, Zhuoqing Morley Mao, Dominic G Lucchetti, and Peter M Chen. Enriching intrusion alerts through multi-host causality. In *NDSS*, 2005.

[42] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao-yong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *USENIX*, 2009.

[43] Srinivas Krishnan, Kevin Z Snow, and Fabian Monrose. Trail of bytes: efficient support for forensic analysis. In *CCS*, 2010.

[44] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela Ciocarlie, et al. Mci: Modeling-based causality inference in audit logging for attack investigation. In *NDSS*, 2018.

[45] Lee Kyu Hyung, Zhang Xiangyu, and Xu Dongyan. Loggc: garbage collecting audit log. In *SIGSAC*, 2013.

[46] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High accuracy attack provenance via binary-based execution partition. In *NDSS*, 2013.

[47] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. Towards a timely causality analysis for enterprise security. In *NDSS*, 2018.

[48] Shiqing Ma, Kyu Hyung Lee, Chung Hwan Kim, Junghwan Rhee, Xiangyu Zhang, and Dongyan Xu. Accurate, low cost and instrumentation-free security audit logging for windows. In *ACSAC*, 2015.

[49] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. MPI: Multiple perspective attack investigation with semantic aware execution partitioning. 2017.

[50] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *NDSS*, 2016.

[51] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. Protracer: Towards practical provenance tracing by alternating between logging and tainting. In *NDSS*, 2016.

[52] 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 2019-02-14.

[53] Kexin Pei, Zhongshu Gu, Brendan Saltaformaggio, Shiqing Ma, Fei Wang, Zhiwei Zhang, Luo Si, Xiangyu Zhang, and Dongyan Xu. HERCULE: attack story reconstruction via community discovery on correlated log graph. In *ACSAC*. ACM, 2016.

[54] Matthew G Schultz, Eleazar Eskin, F Zadok, and Salvatore J Stolfo. Data mining methods for detection of new malicious executables. In *S&P*, 2001.

[55] Andrew H Sung, Jianyun Xu, Patrick Chavez, and Srinivas Mukkamala. Static analyzer of vicious executables (save). In *ACSAC*. IEEE, 2004.