Technical Report
Number: NU-CS-2021-01

June, 2021

Intrusion Response via Graph-based Low-level System Event Analysis

Authors

Xutong Chen

Abstract

This report presents the evolution of the endpoint security system and discusses novel challenges in the state-of-the-art endpoint security system, i.e., endpoint detection response (EDR) based on graph-based low-level system event analysis, when its deployment is required under different platform/environment/task setups. To elaborate and answer this problem, two research projects will be introduced, i.e., RATScope for Windows/Operating System/Detection and CLARION Linux/Container/Forensics.

Keywords

System security, Malicious behavior analysis, Forensics, Provenance tracking, Microservice and container

NORTHWESTERN UNIVERSITY

Intrusion Response via Graph-based Low-level System Event Analysis

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Science

By

Xutong Chen

EVANSTON, ILLINOIS

June 2021

ABSTRACT

Intrusion Response via Graph-based Low-level System Event Analysis

Xutong Chen

From cyber theft of personal financial information to Advanced Persistent Threat (APT) attacks, nowadays *endpoint* devices suffer from various intrusions which cause inestimable property and privacy loss. To protect the security on endpoints, *endpoint detection and response (EDR)* systems have been developed to serve as the powerful solution against those intrusions. Among numerous EDR systems, those based on graph-based low-level system event analysis generally benefit from their higher detection accuracy and they are also less likely to be compromised or evaded. However, the *effectiveness* and *efficiency* of those systems could vary on different *platform/environment/task* setups.

This dissertation focuses on exploring how effectiveness and efficiency could be achieved differently when platform/environment/tasks are changed and it proposes solutions for specific technical problems in different setups. To elaborate and answer this problem, two research projects will be introduced, i.e., RATScope for Windows/Operating System/Detection and CLARION Linux/Container/Forensics.

Acknowledgements

First, I would like to thank my advisor, Prof. Yan Chen, who gave me the opportunity to join Northwestern university and a busy but fulfilling Ph.D. career. I will certainly benefit from the research and engineering training I received here throughout my life.

Then, I would like to express my sincere appreciation to Dr. Vinod Yegneswaran and Dr. Ashish Gehani from SRI International. They mentor my internship and provide tremendous comprehensive help including high-level idea discussion, hands-on detail guidance and paper writing in CLARION-series projects. I will never be able to finish my first published top-tier conference paper without their help.

Next, I would like to thank Prof. Peter Dinda and Prof. Simone Campanoni from Northwestern university for being my thesis defense committee members and qualifying exam committee members. They provide critical comments during my presentations so that I can have better content in my dissertation.

In addition, I would like to thank Prof. Yinzhi Cao from John Hopkins University and Dr. Yueqiang Cheng from Security Research in NIO. Prof. Cao helps me in conducting a blockchain-related project. Dr. Cheng teaches me how to do a good academic paper writing in RATScope project.

Also, I want to thank my colleagues and friends in both Northwestern University and Zhejiang University, specifically Dr. Xiang Pan, Prof. Haitao Xu, Kaiyu Hou, Mohammad Kavousi, Guannan Zhao, Xiangming Shen from Northwestern University and Dr. Runqing

Yang, Dr. Xue Leng, Lin Jin, Xing Li, Xuechao Du, Chunlin Xiong, Zhengyuan Li from Zhejiang University. I have a fairly good time with them in my Ph.D. career.

Finally, I am grateful to my parents, my girlfriend and other people who concern about me during my Ph.D. career. They really accompany me through the toughest moments.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

With the rapid growth of the Internet, it has become an indispensable part of our daily life. As the Internet keeps penetrating every aspect of the normal functioning of the society, it has been attracting tremendous hackers to launch intrusions from the Internet, steal valuable resources and cause damages for individuals, enterprises, and governments.

According to an investigation about the distribution of data leakage[**29**], one of the most severe but common breaches on today's network, over 70% of this kind of breach begins at network *endpoints* on the Internet, including personal hosts, servers, and cloud services. This reveals that *endpoint security systems* are under pressing need on various network endpoints. Fortunately, endpoint security systems have evolved swiftly over the last decade. The evolution of endpoint security systems consist of three generations.

The first generation endpoint security system is the anti-virus software. Anti-virus (AV) software determines that a file represent a malware if this file contains some specific *signatures*, i.e., sequences of bytes extracted from malwares. It is a static approach and performs well in containing known malware outbreaks. However, it heavily relies on a state-of-the-art signature library to ensure detection capability and suffer from vital evasion problems caused by code obfuscation.

The second generation endpoint security system is anti-virus plus host-based intrusion detection system (AV + HIDS). Compared with the first generation endpoint security system, it incorporates the support from the HIDS which introduces the *static scanning*, *sandbox* and *API hooking/drivers* low-level event auditing technique as Figure 1.1 shows. It is a dynamic approach whose detection is based on heuristic patterns on low-level logs. This method helps the endpoint security system not only to get rid of a signature library, but also to receive limited capability of detecting unknown threats.

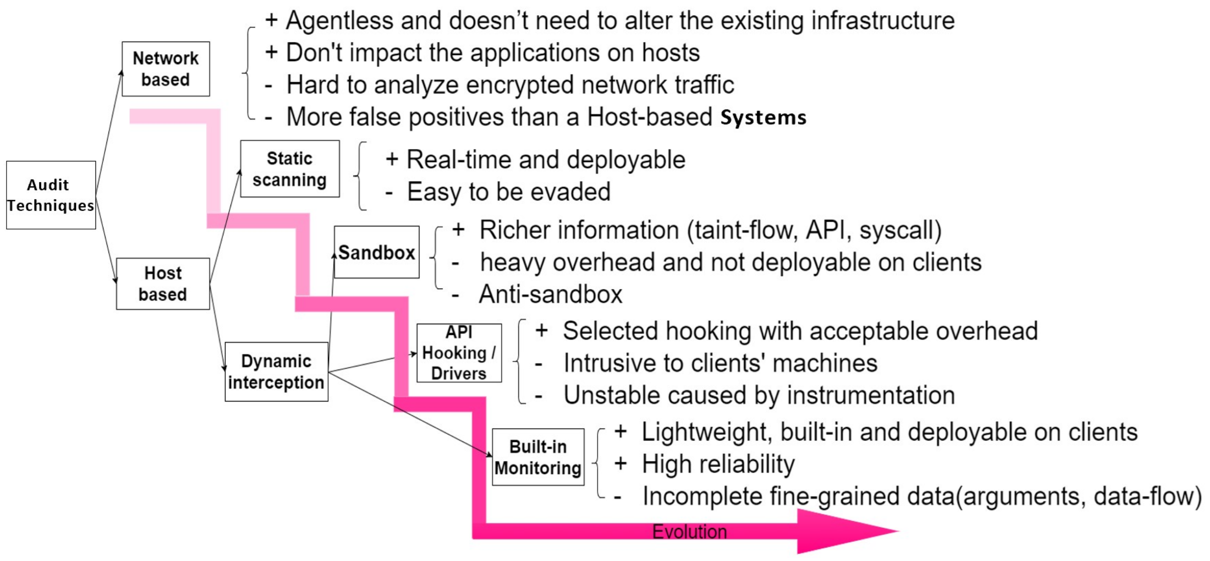

Audit Techniques
Network based
+ Agentless and doesn't need to alter the existing infrastructure
+ Don't impact the applications on hosts
- Hard to analyze encrypted network traffic
- More false positives than a Host-based Systems
Host based
Static scanning
+ Real-time and deployable
- Easy to be evaded
Dynamic interception
Sandbox
+ Richer information (taint-flow, API, syscall)
- heavy overhead and not deployable on clients
- Anti-sandbox
API Hooking / Drivers
+ Selected hooking with acceptable overhead
- Intrusive to clients' machines
- Unstable caused by instrumentation
Built-in Monitoring
+ Lightweight, built-in and deployable on clients
+ High reliability
- Incomplete fine-grained data(arguments, data-flow)
Evolution

Figure 1.1. Evolution of Low-level Event Auditing

Though the second generation endpoint security system, to a large extent, improves endpoint security, low-level logs they use are still too coarse-grained to help to detect many advanced malicious behaviors. Therefore, AV + HIDS fails to handle some complicated intrusions. Representative examples include advanced persistent threats (APT) shown in Figure 1.2. Sophisticated APTs remain concealed for months and they are usually equipped with progressive evasion techniques, e.g., being file-less and code polymorphic. Given this situation, a comprehensive upgrade on both low-level event auditing techniques and detection techniques is required.

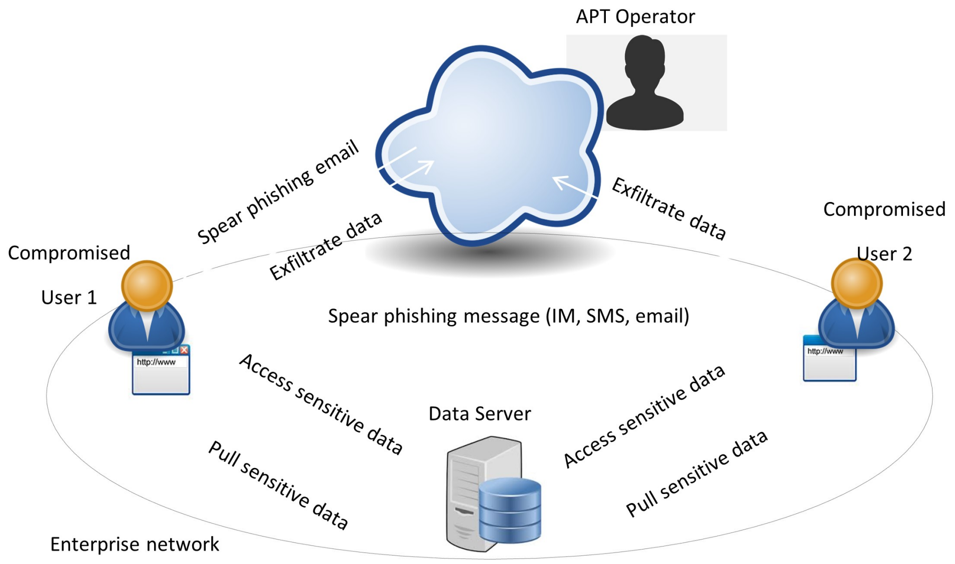


Figure 1.2. The Work Flow of APT attack

The above challenges lead to the third generation endpoint security system, i.e., the endpoint detection and response (EDR) system, which is based on *graph-based low-level system event analysis*. Compared to the second generation endpoint security system, it emphasizes the understanding of program/system behaviors, attack provenance tracking, real-time response, and even more capability in detecting unknown threats, facilitating effective (high accuracy) and efficient (low overhead) intrusion response.

In general, graph-based low-level event analysis tries to describe program/system behavior as graphs using data collected by *built-in monitoring* auditing techniques in Figure 1.1. Then it performs graph analyses which can help detection and forensics in intrusion response.

When the graph is used for describing program behavior, which is usually the case for detection, the corresponding graph is *behavior graph* as shown in Figure 1.3 and it consists of vertices representing low-level events and edges representing control/data dependency

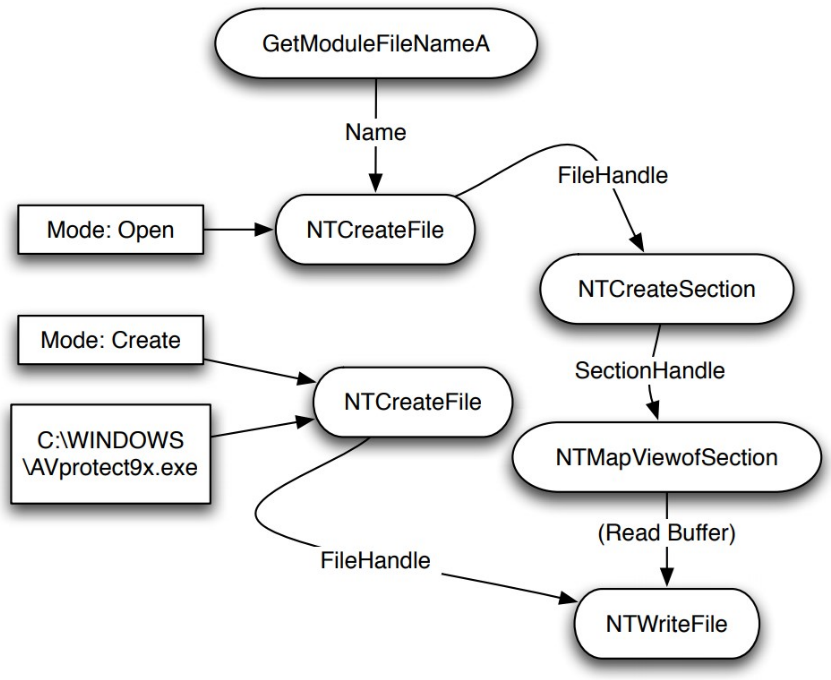


Figure 1.3. Behavior Graph Example: Ovals stand for system call vertices and each edge is marked with the dependent data between two system calls. Rectangles are just initialization attributes of system calls, which are neither vertices nor edges.

between those events. This kind of graph can be used to accurately model semantic behavior performed by programs/processes.

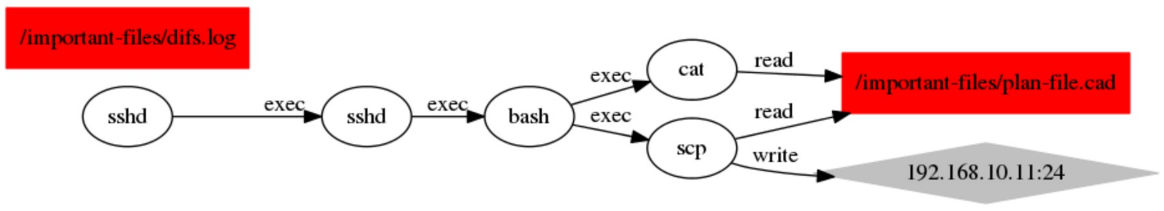


Figure 1.4. Provenance Graph Example: Ovals are processes, rectangles are files and diamonds are sockets. Edges are marked with corresponding system calls.

When the graph is used for describing system behavior, which is usually the case for forensics, the corresponding graph is *provenance graph* as shown in Figure 1.4 and it consists of vertices representing low-level system objects, e.g., processes, files, sockets, and edges representing control/data dependency carried by corresponding low-level events.

However, applying graph-based low-level event analysis technique in EDR systems still requires further research because when different platforms/environment/task setups are used, we encounter specific problems. In this dissertation, we focus on the following question: **How can we *effectively* and *efficiently* leverage graph-based low-level system event analysis to perform intrusion response for *different platform/environment/tasks*?**

This dissertation tackles this question by addressing specific problems that we encounter in two projects with different platform/environment/task setups: Windows/operating system/detection for RATScope project and Linux/container/forensics for CLARION project. The brief answer can be summarized as follows.

- With the Windows/operating system/detection setup, RATScope uncovers the problem of missing arguments in Windows low-level events, i.e., ETW events, which must be addressed so that effectiveness can be achieved. Given a detection task, a meticulous graph pattern matching algorithm is also needed for providing efficiency.
- With the Linux/container/forensics setup, CLARION exposes the influence introduced by container virtualization in Linux low-level events, i.e., Linux Audit events, which usually cannot be realized by people. Given a forensic task, efforts

on selecting system calls and system implementation optimization must be paid to reach efficiency.

The dissertation will consist of two above projects in the order of RATScope and CLARION. The connection between two projects is shown in Figure 1.5 from an architecture view of an EDR system.

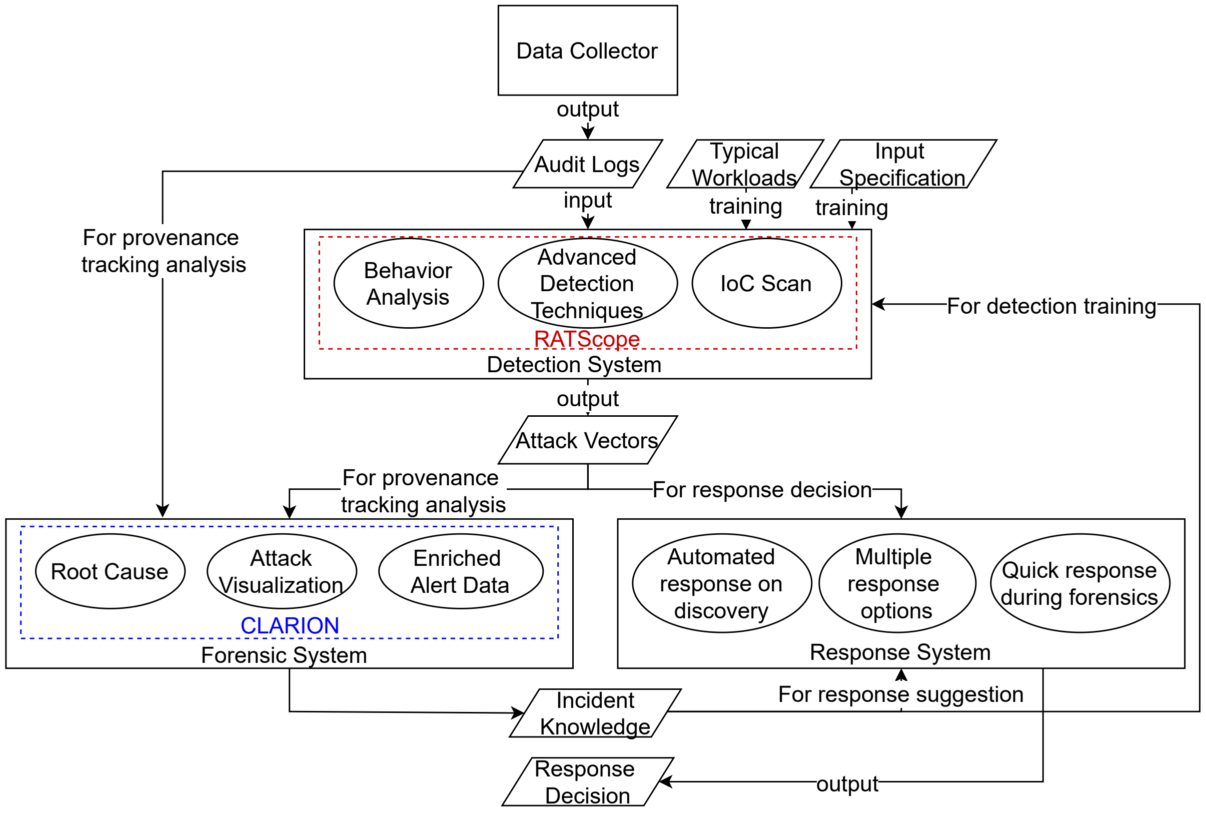


Figure 1.5. An architecture view of RATScope and CLARION

In Chapter 2 and 3, the dissertation will show what we have done to answer the above-mentioned question, investigating effectiveness and efficiency of graph-based low-level event analysis in different cases accordingly. At last, Chapter 4 will summarize the dissertation.

CHAPTER 2

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

2.1. 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 [**58, 60, 64**]. 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 2.2) 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 project, 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 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 [**50, 34, 9**]. Furthermore, patching the kernel has never been supported by Microsoft. Microsoft even integrated Kernel Patch Protection (KPP) into Windows to prevent patching the kernel [**32**]. Thus, the previous instrumentation-based forensic schemes, like [**96, 104, 103**], 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 [**63, 67, 49**] 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.

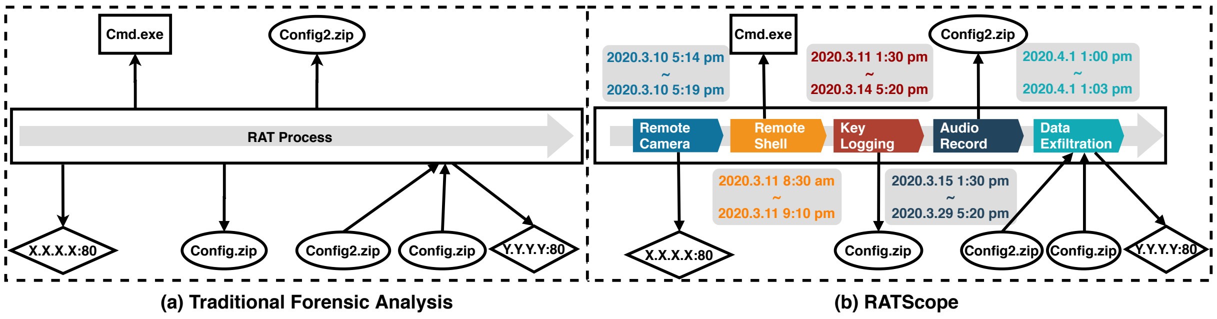


Figure 2.1. Comparison between traditional forensic systems and our RATScope on a simplified attack scenario. Rectangles denote processes. Ovals and diamonds denote files and sockets, respectively.

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 [**75**]. However, most of existing forensic systems [**99, 82, 77, 83, 111, 104, 102, 95, 96, 91**] 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 2.3.3.1). The *Semantic Collision* problem renders the state-of-the-art behavior identification approaches [**93, 90**] 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 2.1 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.

2.2. Background and A Large-Scale Study of Real-World RATs

Although APT attacks [**1, 2, 5, 59, 52**] 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 describe how we manage to collect a representative corpus of RAT samples for our study in Section 2.2.2. We report our major findings based on dynamic and static analysis of those RAT samples in Section 2.2.3. This study not only fills the gap between attackers and defenders but also motivates our system design described in section 2.3.

2.2.1. Workflow of RAT Attack

RAT is the abbreviation of Remote Access Trojan. It is a prevailing type of malware widely used in severe end-host attacks like Advanced Persistent Threat (APT) 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.*

2.2.2. 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 repositories (e.g., VirusTotal) [**75**]. *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 [**25, 27, 48**] where RAT controllers are sold or cracked. As a result, we find a total of 53 well-known and RATs families listed in Table 2.1. 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 [**54**] and the Nitro attacks on chemical firms [**24**]; DarkComet active since 2008 was used in the Syrian activists attack [**15**] and leveraged in the Charlie Hebdo shooting incident for malware spreading [**28**]; XtremeRAT active since 2010 was responsible for the attacks on US, UK, Israel and other Middle

East governments [**46, 39, 69**]; Adwind RAT active since 2012 was used in aerospace enterprises attacks [**1**] and attacks targeting Danish companies [**2**].

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 2.2. We provide more details about the debut year of each RAT family and how we determine it in the appendix of the TDSC paper of this project.

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.*

Table 2.1. 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

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

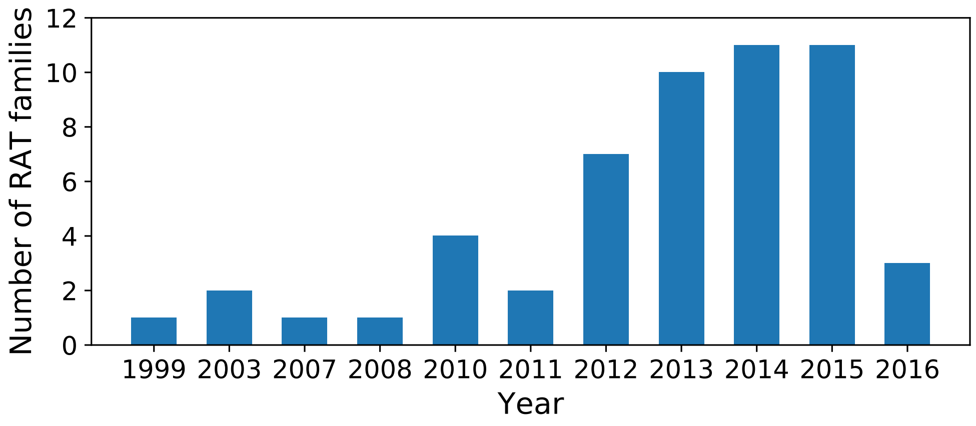
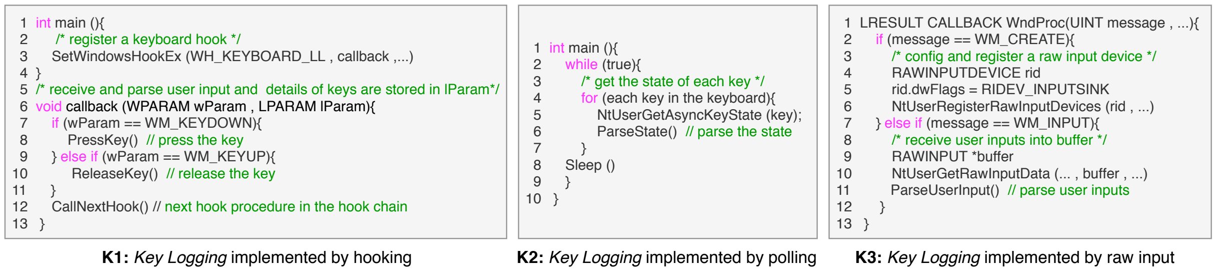


Figure 2.2. Distribution of RAT families based on debut years.



```
int main (){
    /* register a keyboard hook */
   SetWindowsHookEx (WH_KEYBOARD_LL , callback ,...)
}
/* receive and parse user input and  details of keys are stored in lParam*/
void callback (WPARAM wParam , LPARAM lParam){
    if (wParam == WM_KEYDOWN){
        PressKey()  // press the key
    } else if (wParam == WM_KEYUP){
        ReleaseKey()  // release the key
    }
    CallNextHook() // next hook procedure in the hook chain
}
```

K1: *Key Logging* implemented by hooking

```
int main (){
    while (true){
        /* get the state of each key */
        for (each key in the keyboard){
            NtUserGetAsyncKeyState (key);
            ParseState()  // parse the state
        }
    Sleep ()
    }
}
```

K2: *Key Logging* implemented by polling

```
LRESULT CALLBACK WndProc(UINT message , ...){
    if (message == WM_CREATE){
        /* config and register a raw input device */
        RAWINPUTDEVICE rid
        rid.dwFlags = RIDEV_INPUTSINK
        NtUserRegisterRawInputDevices (rid , ...)
    } else if (message == WM_INPUT){
        /* receive user inputs into buffer */
        RAWINPUT *buffer
        NtUserGetRawInputData (... , buffer , ...)
        ParseUserInput()  // parse user inputs
    }
}
```

K3: *Key Logging* implemented by raw input

Figure 2.3. Pseudocode of all three implementations of *Key Logging*.

2.2.3. 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.

F1: High-level programming languages are preferred by RAT developers to write RAT stubs We leverage `Detect-It-Easy` [**51**], 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 2.1, `.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` [17]. 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 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 **??**, 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 2.2. 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 2.2 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 of the TDSC paper of this project.

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 2.2. 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 2.3. 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 of the TDSC paper of this project. 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 2.3. 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 [**6, 8, 53, 40, 20**] 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 2.3 provides the pseudo code for all three means of implementing the *Key Logging* PHF. That intelligence could be incorporated into existing red-team tools [**6**] 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 [**16**].

2.3. System Design

2.3.1. Threat Model and Design Overview

Threat Model In this project, 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 [**68, 26**]. 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 [**76, 77, 83, 81**].

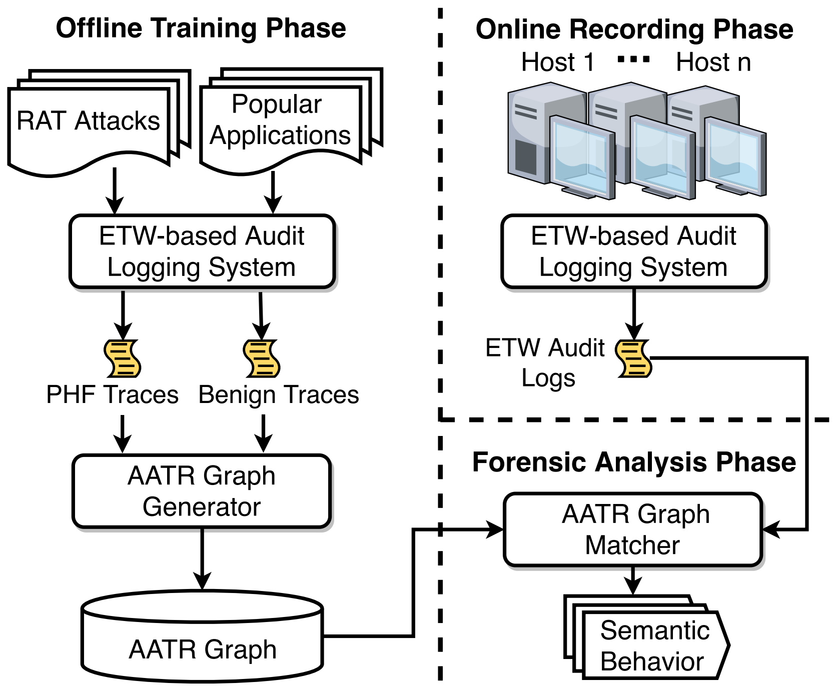


Figure 2.4. System Architecture.

Design Overview Our goal is to develop a RAT forensic system suitable in an enterprise environment. Figure 2.4 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 2.3.2). Then we build an enhanced version of `ETW` (Section 2.3.3) to collect log data, which is then inputted to the *AATR Graph Generator* (Section 2.3.4) to generate AATR graphs, which characterize the internal implementation mechanism of PHFs. In the online

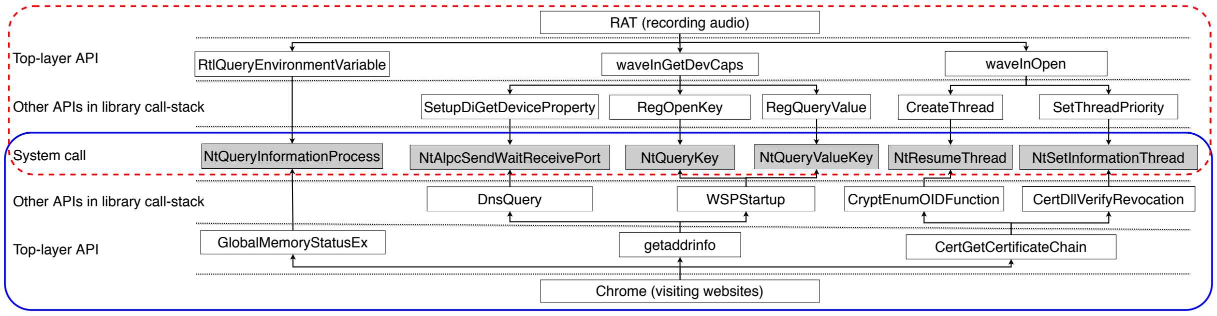


Figure 2.5. An example of *Semantic Collision* on system call level data (shaded boxes). Two different program behaviors trigger exactly the same system call sequences though different library call-stacks.

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 2.3.5) 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 2.2) 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.

2.3.2. 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.*

2.3.2.1. 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 2.6 provides a concrete ETW event example to explain the above concepts.

2.3.2.2. Semantic Collision Problem. *Semantic Collision* refers to the cases that two *different* program behaviors end up being represented as the *same* behavior graph,

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

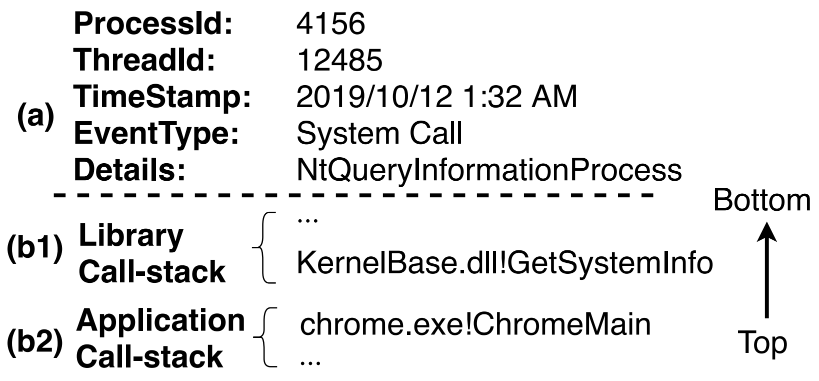

(a)
ProcessId: 4156
ThreadId: 12485
TimeStamp: 2019/10/12 1:32 AM
EventType: System Call
Details: NtQueryInformationProcess
(b1) Library Call-stack
...
KernelBase.dll!GetSystemInfo
(b2) Application Call-stack
chrome.exe!ChromeMain
...
Bottom
Top

Figure 2.6. 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.

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 2.5 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 [**88, 90, 93**] which heavily rely on input arguments.

2.3.2.3. Collision Avoidance by AATR Graph. Motivated by the example illustrated in Figure 2.5, 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.

2.3.3. `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 [**55, 107, 71**] 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.

2.3.3.1. Limitations with `ETW` for RAT Forensic Purpose. `ETW` has been used in existing forensics and detection works [**77, 76, 81**] 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 2.4.1.2.

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 2.4.1.1).

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 2.4.1.2).

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 [**90, 93**] usually involves as much low-level data as possible, which is collected by monitoring

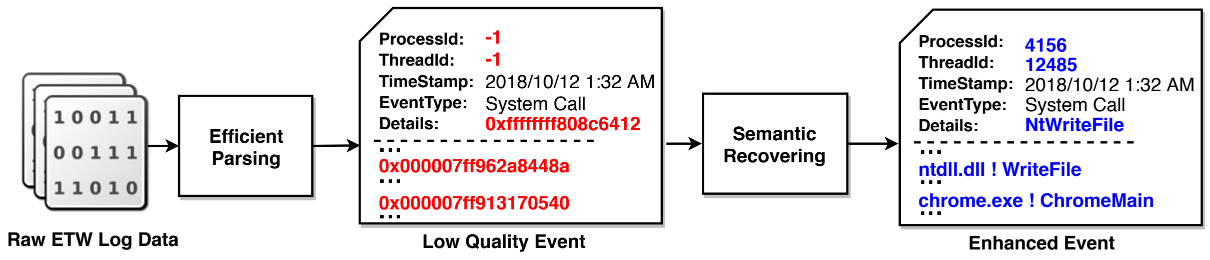


Figure 2.7. How our audit logging system works on the raw `ETW` log data.

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 2.4.2.1).

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 2.4.2.2) 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.

2.3.5. 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 2.4.3.

2.4. 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.

2.4.1. `ETW`-based Audit Logging System

Fig. 2.7 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.

2.4.1.1. 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 [**10**]. However, not all those common fields are useful for forensics. Based on previous works [**77, 86**] 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.

2.4.1.2. 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 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 [**70**].

2.4.1.3. 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.

2.4.2. AATR Graph Generator

2.4.2.1. 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 2.8. 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.

2.4.2.2. Redundancy-reduction-based Generation. Enlightened by the example in Figure 2.8, 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.

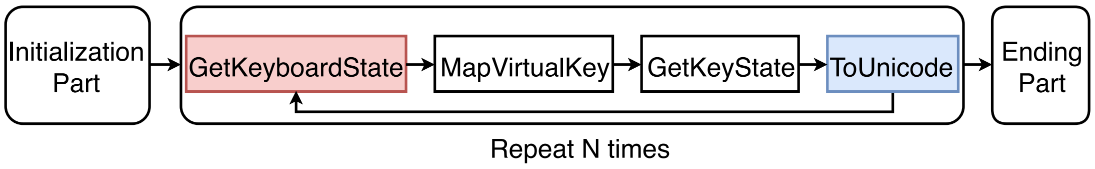


Figure 2.8. A breakdown of the execution traces of the *Key Logging* PHF: initialization, loop body, and ending.

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 [**49**], 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 2.4.3.

2.4.2.3. Explanatory Example. In this section, we give an example in Fig. 2.9 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 ($\theta_{lbs} =$ Ⓒ) that splits the initialization and the first main loop body, and thus we identify the initialization ($\phi_{init} =$ Ⓐ-Ⓑ) (Line 2 to 3 in Algorithm 1). Then we scan the trace again to locate the loop body ending (lbe) separator ($\theta_{lbe} =$ Ⓔ) that splits the last main loop body and the ending, and thus we identify the ending ($\phi_{end} =$ Ⓕ-Ⓖ-Ⓗ-Ⓘ) (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 [**49**] 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.

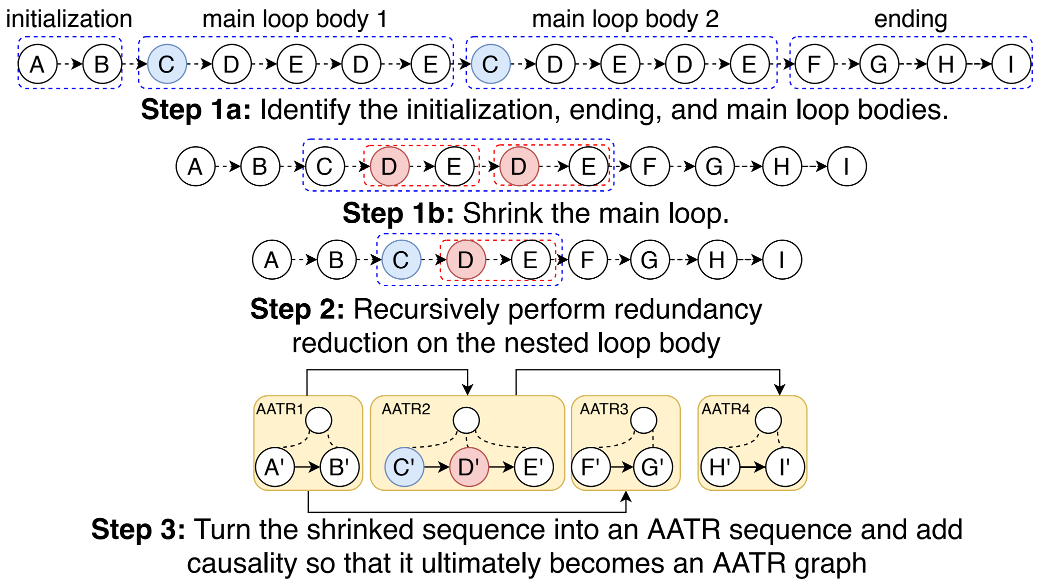

initialization
main loop body 1
main loop body 2
ending
A
B
C
D
E
F
G
H
I
Step 1a: Identify the initialization, ending, and main loop bodies.
Step 1b: Shrink the main loop.
Step 2: Recursively perform redundancy reduction on the nested loop body
AATR1
AATR2
AATR3
AATR4
A'
B'
C'
D'
E'
F'
G'
H'
I'
Step 3: Turn the shrinked sequence into an AATR sequence and add causality so that it ultimately becomes an AATR graph

Figure 2.9. The workflow of AATR graph generation on a given trace.

Algorithm 1 Redundancy-reduction Algorithm

```
Input: An unfolded AATR trace $\phi_{input}$=$\{\theta_j$=(syscall, library call-stack, application call-stack) $|j$=$1 \ldots m\}$
Output: An AATR sequence without causality $\phi_{reducted} \subseteq \phi_{input}$
Initialize: $\phi_{reducted} \leftarrow \emptyset$
1:  function REDUNDANCY-REDUCTION($\phi_{input}$)
2:      scan from $\theta_m$ to $\theta_1$, find the last $\theta_{lbs} \in \phi_{input}$ where $\exists k$, $\theta_{lbs}$=$\theta_k$, $lbs + 1 \leq k \leq m$
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 $\theta_1$ to $\theta_m$, find the last $\theta_{lbe} \in \phi_{input}$ where $\exists k$, $\theta_{lbe}$=$\theta_k$, $lbs \leq k \leq llbs$
7:      $\phi_{end}$=$\theta_{lbe+1} \ldots \theta_m$
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 $\phi_{slb}$=$\theta_{j_s} \ldots \theta_{j_{s+1}}$
11:     get nested loop recursively reduced result $\phi_{nesrec} \leftarrow$ Redundancy-Reduction($\phi_{slb}$)
12:     $\phi_{reducted} \leftarrow$ Concatenate($\phi_{init}$, $\phi_{nestrec}$, $\phi_{end}$)
13:     return $\phi_{reducted}$
```

2.4.3. 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.

2.4.3.1. Optimal Partial Matching Problem.

Definition 2.1 (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 $t : \wp(V \times \phi) \to [0, 1]$; (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.*

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: $\varPhi_{phf}$=$\{\phi_i|i$=$1\ldots n\}$, where trace ϕ_i=$\{\theta_j$=(syscall, library call-stack, application call-stack)$|j$=$1\ldots m\}$; (2) a causality analysis engine $CausalityEngine$ which take an AATR sequence to output an AATR graph; (3) traces about all other PHFs: $\varPhi_{other}$; (4) traces of benign program normal operation: $\varPhi_{benign}$;

Output: n AATR graphs, each represented as direct acyclic graph of AATRs which defines the PHF semantically: $\varPsi$=$\{\psi\}$, where ψ=($\cup$AATR, $\cup$causality)

Initialize: $\varPsi \leftarrow \emptyset$.

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

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 2.4.3.2.

2.4.3.2. Matching Rate Function.

Definition 2.2 (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:*

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

Here $sysc_v$ is the system call of v and lib_stack_v is the corresponding library call-stack. $\mathit{\Delta}(lib\_stack_v, lib\_stack_\theta)$ 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.

2.4.3.3. 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 2.1, 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 2.3) to represent each matching subproblem.

Definition 2.3 (Graph State). *Given a direct acyclic graph $G = (V, E)$, a graph state gs can be defined as a mapping $gs : V \rightarrow \{0, 1\}$ 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 2.1.

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 2.2 and try to maximize the target function by dynamic programming.

2.4.3.4. Explanatory Example. In this section, we provide examples to show how AATR graph matcher works. Specifically, the example in Fig. 2.10 shows the input and output of the matcher from a high-level view. Then examples in Fig. 2.11 and Fig. 2.12 show details how the matcher obtain the output from the input step by step.

In Fig. 2.10, 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

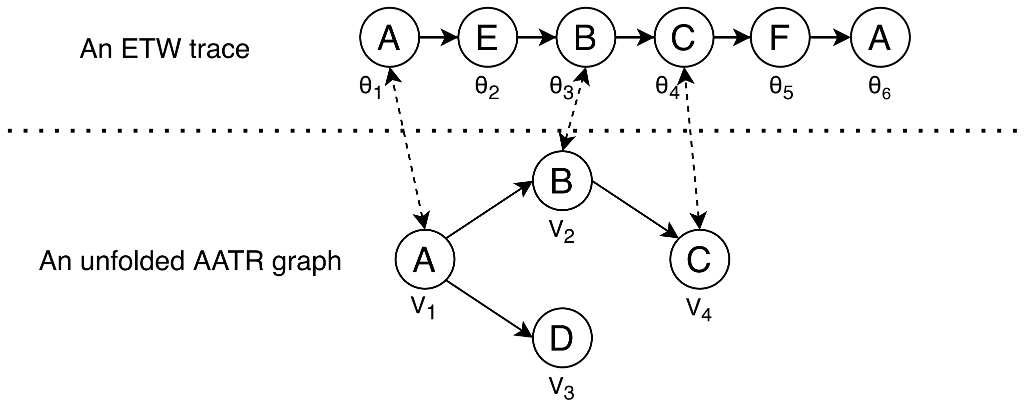


Figure 2.10. An example of a match mapping between a given AATR graph and a given ETW trace.

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 2.2 will be 1; otherwise, it will be just 0.

As shown in Fig. 2.10, 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. 2.11 and Fig. 2.12 show how our DP-based algorithm obtains the optimal match score in Fig. 2.10 in detail. Algorithm 3 first builds a state transition graph (the right graph in Fig. 2.11) to enumerate all possible graph states of the AATR graph (the left graph in Fig. 2.11). 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.

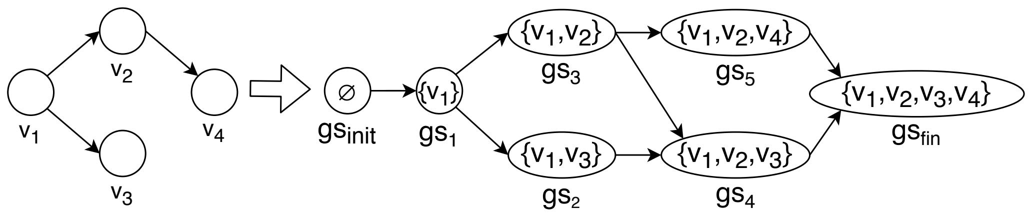


Figure 2.11. Convert an AATR graph into a graph state transition graph.

Lastly, Algorithm 4 does DP optimization and ultimately find the optimal paths in the transition graph generated in Fig. 2.11. Fig. 2.12 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. 2.10.

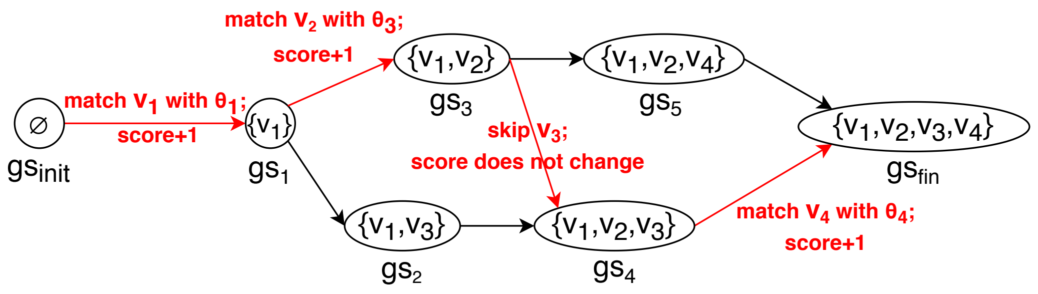


Figure 2.12. One DP transition path for getting the optimal matching.

Algorithm 3 Transition Graph Building

```
Input: An unfolded AATR graph $\psi$=($V$=∪AATR, $E$=∪causality);
Output: A graph state transition graph $TG$=($V_{tr}$={∀graph-state},
    $E_{tr}$={∀graph-state-transition} $\subseteq (V_{tr} \times V_{tr})$,$label : E_{tr} \rightarrow V$)
Initialize: $gs_{init} \leftarrow \{gs(v)=1|\forall v \in V\}$; $addedGs \leftarrow \{gs_{init}\}$; $TG \leftarrow (\{gs_{init}\}, \emptyset, \emptyset)$;
    $tpsortQueue \leftarrow \{gs_{init}\}$;
1: procedure TRANSITION-GRAPH-BUILDING($\psi$)
2:     while $tpsortQueue \neq \emptyset$ do
3:         $gs_{head} \leftarrow tpsortQueue.pop()$
4:         $V_{frontier} \leftarrow$ GetFrontier($gs_{head}$, $\psi$)
5:         for each $v_{itr} \in V_{frontier}$ do
6:             $gs_{new} \leftarrow gs_{head}$
7:             $gs_{new}(v_{itr}) \leftarrow 0$
8:             $V_{tr} \leftarrow V_{tr} \cup \{gs_{new}\}$
9:             $E_{tr} \leftarrow E_{tr} \cup \{(gs_{head}, gs_{new})\}$
10:            $label((gs_{head}, gs_{new})) \leftarrow v_{itr}$
11:            if $gs_{new} \notin addedGs$ then
12:                $tpsortQueue.push(gs_{new})$
13:                $addedGs \leftarrow addedGs \cup \{gs_{new}\}$
14:    return $TG$
```

2.5. 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 2.5.1)
- **Q2.** How *robust* is RATScope against different (and new) RAT families? (Sections 2.5.2 and 2.5.3)

Algorithm 4 AATR Graph Matching

Input: (1) A unfolded input ETW trace ϕ=\{θ_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 $score : V_{tr} \times \phi_{input} \rightarrow N$;

```
1:  procedure GRAPH-MATCHING($\phi_{input}$, $\psi$)
2:      for $j = 1 \rightarrow m$ do
3:          $dpQueue \leftarrow \{gs_{init}\}$
4:          $score(gs_{init}, \theta_j) \leftarrow 0$
5:          while $dpQueue \neq \emptyset$ do
6:              $gs_{head} \leftarrow dpQueue.pop()$
7:              $score(gs_{head}, \theta_j) \leftarrow score(gs_{head}, \theta_{j-1})$
8:              for each $gs_{pred}$ where $e_1$=$(gs_{pred}, gs_{head}) \in E_{tr}$ do
9:                  if $score(gs_{pred}, \theta_j) > score(gs_{head}, \theta_j)$ then
10:                     $score(gs_{head}, \theta_j) \leftarrow score(gs_{pred}, \theta_j)$
11:                 $delta \leftarrow score(gs_{pred}, \theta_{j-1})$+$bonus(label_{e_1}, \theta_j))$
12:                 if $delta > score(gs_{head}, \theta_j)$ then
13:                     $score(gs_{head}, \theta_j) \leftarrow delta$
14:             for each $gs_{succ}$ where $e_2$=$(gs_{head}, gs_{succ} \in E_{tr})$ do
15:                 if $gs_{succ} \notin dpQueue$ then
16:                     $dpQueue.push(gs_{succ})$
17:     $finalScore \leftarrow score(gs_{fin}, \theta_m)$
18:     return $finalScore/maxScore$
```

- **Q3.** What is the *overhead* of RATScope? (Section 2.5.4)

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.

Table 2.4. Experiment details of our simulation of the Syria RAT attack.

Victim	**Infection Vector**	**RAT**	**Anti-Detection**	**PHFs**	**Duration (H:M:S)**	**Storage (MB)**	**TPR**	**FPR**
Alice	Phishing Email [52] & File Extension Spoofing [66]	Dark Comet	UPX Packer [61] & Process Injection [65]	Key Logging	8:32:11	184	100%	1.25%
Bob	Skype Message [61] & File Extension Spoofing [66]	NjRAT	DeepSea Obfuscator [52] & Process Injection [65]	Key Logging; Remote Camera; Audio Record	9:03:25	235	100%	0.5%

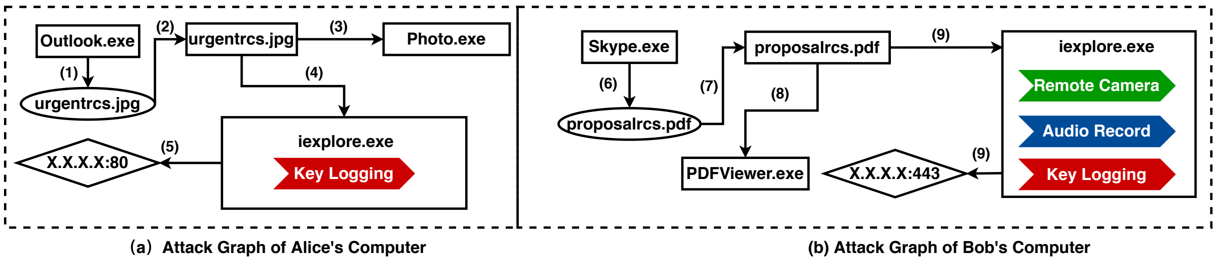


Figure 2.13. Simplified attack graph generated by RATScope. Rectangles represent processes; ovals and diamonds denote files and sockets, respectively; Figure (a) shows the graph of Alice's computer; Figure (b) shows the graph of Bob's computer.

2.5.1. 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 [**52**, **61**, **65**, **66**], 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 [**52**], and Skype messages sent from trusted people whose credentials have been stolen [**61**]) with visual spoofing techniques (e.g., file extension spoofing [**66**]) to trick victims into the execution of a malicious file, which is actually a remote access trojan [**61**] (e.g., DarkComet, NjRAT, and Xtreme), protected using anti-detection techniques (e.g., obfuscation [**52**] and process injection [**65**]). Attackers then remotely control the implanted RAT to obtain victims' intelligence by performing PHFs, such as *Audio Record*, *Key Logging*, and *Remote Camera*.

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 2.4 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 [**99, 77, 81**] 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 2.5.2.2. Figure 2.13 provides a simplified attack graph generated by RATScope combined with a causality tracking approach [**86**]. 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.

2.5.2. 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.

2.5.2.1. Experiment Setup. Family-split PHF Dataset. We first randomly select one RAT sample for each of the 53 representative RAT families (listed in Table 2.1). Then, for each of the 53 RAT samples, we execute the 5 popular PHFs (listed in Table 2.2) one by one, 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

Table 2.5. 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%

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* [**112**] *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 [**90**] 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 [**90**] except for lack of arguments.

2.5.2.2. Result Analysis. Table 2.5 details the comparative evaluation results. In the context of this project, the True Positive (TP) measures the total number of RAT families

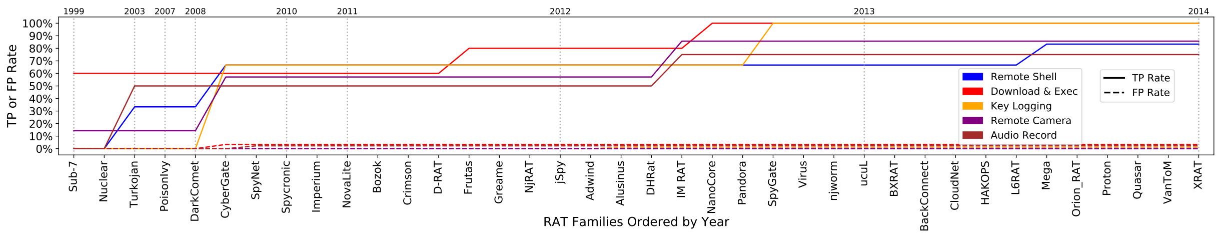


Figure 2.14. Temporal evaluation. For a RAT family R_i listed in the X axis, we fixed the testing set, which included the RAT families from 2014 to 2016; the training set is dynamic, which included all RAT families occurring before the RAT family R_i.

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 2.5 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 2.2) 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 2.5, 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 [**45**]. That is the reason why we name RAT functions as Potential Harmful Functions (PHFs). However, in these cases, we believe it is still worth reporting those FPs by a forensic system in the first place and performing triage with other information later. Actually, both academia [**108, 81**] and industry [**7**] have realized that relying on a single suspicious behavior to triage an alert is not sufficient in practice. NoDoze [**81**] 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 2.13 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.

2.5.3. 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.

2.5.3.1. Experiment Setup. Temporally-sorted PHF Dataset. We follow the same dataset generation process as the Family-split PHF dataset in Section 2.5.2.1. Then we sort the traces in every PHF subdataset by the debut year of the corresponding RAT families (shown in Figure 2.2). 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 2.5.2. 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 [**112**] 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 2.14. 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.

2.5.3.2. Result Analysis. Figure 2.14 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** [**47**]**.** For instance, `NjRAT` is one of far-reaching RATs whose source code was leaked in 2013 [**31**]. 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` [**3**], `Crimson` and `jSpy` implement *Key Logging* using a Java library `JNativeHook` [**22**], and `Quasar` implements *Key Logging* using a `C#` library `GlobalMouseKeyHook` [**21**]. 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 2.3.

2.5.4. 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 2.15(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 2.15(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 2.15(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%.

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 2.4.1. 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 2.6 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 2.6. Comparison of parsing speed.

	# of Parsed Event / Sec
TDH [**55**]	93,254
TraceEvent [**71**]	242,716
RATScope	552,090

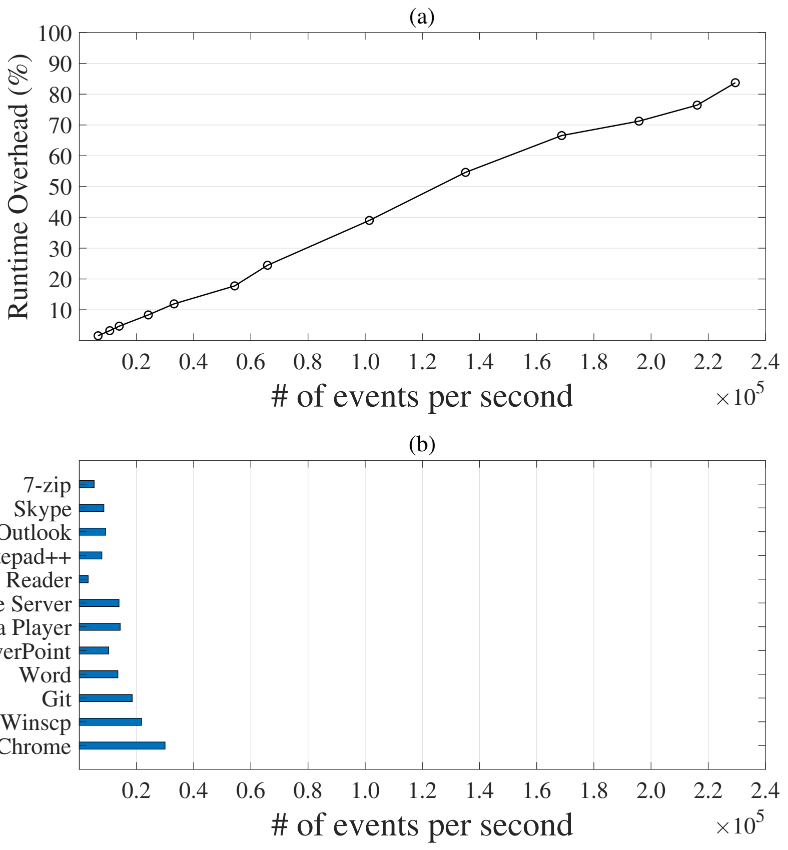


Figure 2.15. Figure (a) shows the runtime overhead with different numbers of events per second. Figure (b) shows the number of events per second generated by different programs.

Storage Overhead. We deploy our system on two machines (i.e., Alice and Bob) under the typical real-world scenario mentioned in Section 2.5.1 and collect data for one day on each machine. Table 2.7 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 2.7. 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%

2.6. 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 [**86, 88**]. 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 [**86, 88**]. Many works have been proposed aiming at improving the causality graph framework. Some works [**93, 104, 102, 96, 91**] mitigated the dependency explosion problem by fine-grained causality tracking to reduce more unrelated data. Meanwhile, some other works [**99, 82, 77, 83**] 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 [**83**], it is sufficient to understand the attack by knowing what file was downloaded from what IP address. However, when it comes to RAT attacks [**75**], 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 [**90, 74, 93**] 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. 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 [**84, 72**]. 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. [**75**] and Rezaeirad et al. [**113**] 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 project 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.

CHAPTER 3

CLARION: Sound and Clear ProvenanceTracking for Microservice Deployments

3.1. Introduction

Linux container technology has seen a rapid rise in adoption due to the miniaturized application footprints and improved resource utilization that are crucial in contemporary microservice architectures[**41**] and serverless computing environments[**57**]. The performance boost realized in containerized environments stems from their use of light-weight virtualization techniques whereby a single Linux operating system (OS) kernel is used to manage an array of virtualized containers. However, a side effect of this design choice is that an attack initiated inside a container may affect the shared host Linux OS kernel. Compared to the traditional virtual machine (VM) model, in which the guest VM OS is completely isolated from the host, this provides a much greater target surface to the attacker. Hence, comprehensive security tracking and analysis are vital in container networks.

The application of data provenance analysis techniques[**87, 79, 85, 89, 92, 110, 73, 100, 97, 105**] for host and enterprise security monitoring has been well studied. However, extending such capabilities to container-based microservice environments raises some unique research challenges. At a cursory glance, the shared-host OS kernel substrate provides a centralized monitoring platform for observing events across containers and implementing security policy. In fact, the use of Linux namespaces introduces *fragmentation* and *ambiguities* in data streams used by provenance tracking systems, such as those based on the Linux Audit subsystem. Here, fragmentation refers to abnormal vertex splitting leading to false disconnections in the provenance graph. Conversely, ambiguity refers to vertex merging where a single vertex incorrectly represents multiple distinct objects in the correct provenance graph. Both fragmentation and ambiguities lead to false or missing dependencies. We refer to these as *soundness* challenges for container provenance analysis.

Namespaces[**36**] are a fundamental feature in the Linux kernel that facilitate efficient partitioning of kernel resources across process groups. This is the key feature exploited by popular containerization technologies such as Docker [**18**]. While processes within the same namespaces will share OS resources, those in different namespaces have isolated instances of corresponding operating system resources. For example, files in the same mount namespace have the same root directory so they must have different path names. Conversely, two files in different mount namespaces can appear to have exactly the same path names within but can still be distinguished by the root directory of their respective mount namespaces – i.e., their path names are virtualized (*containerized)* by the mount namespace. Unfortunately, it is the virtualized path names that will be recorded and reported by the kernel's audit subsystems, making those two files indistinguishable, which leads to falsely conflated elements in inferred provenance graphs.

Furthermore, mishandling the effect of namespaces can prevent a provenance tracking system from correctly characterizing essential aspects, such as the boundary of containers.

Here the boundary of containers refer to the delineation of a provenance subgraph that represents the behavior within a container. It includes the processes running inside the container, the files manipulated by them, the sockets they create, etc. Without a proper understanding of container semantics (i.e., ability to define boundary of containers and activity patterns of container engines corresponding to initialization, termination etc.), it will be impossible for security analysts to reason about *how*, *when*, and *what* containers are affected by attacks. We refer to these as *clarity* challenges for container provenance analysis.

CLARION Solution. To resolve the aforementioned soundness and clarity challenges, we propose CLARION, a namespace- and container-aware provenance tracking solution for Linux microservice environments. For soundness, we first provide an in-depth analysis of how the virtualization provided by each relevant namespace causes fragmentation and ambiguity in the inferred provenance. For each relevant namespace, we then propose a corresponding technical solution to resolve both issues. To improve clarity, we first define essential container-specific semantics including boundary of containers and initialization of containers. Next, we propose summarization techniques for each semantics to automatically mark the corresponding provenance subgraphs.

We show that soundness and clarity challenges are not specific – i.e., they exist in a range of monitoring approaches, including Linux Audit[**56**], Sysdig[**62**] and LTTng[**37**]. We describe a prototype implementation based on SPADE[**78**], an open source state-of-the-art provenance tracking system and comprehensively evaluate the effectiveness, efficiency, and generality of our solution. We studied the effectiveness and utility of our system using container-specific attacks. We also empirically evaluated system efficiency by running our solution on desktop computers as well as in an enterprise-level microservice environment. To assess generality, we collected provenance graphs for various state-of-the-art container engines including Docker, rkt[**11**], LXC[**38**] and Mesos [**4**]. These results show our solution works across container engines and outperforms the traditional provenance tracking technique by producing superior provenance graphs with an acceptable increase in system overhead (smaller than 5%).

Contributions. In summary, our project makes the following contributions:

- We thoroughly analyze the ways namespace virtualization can affect provenance tracking. To the best of our knowledge, this is the first in-depth analysis of the implications of namespaces on microservice provenance tracking.
- Based on these insights, we designed and implemented a namespace- and container-aware provenance tracking solution – i.e., CLARION – that holistically addresses the soundness and clarity challenges.
- We conducted a comprehensive evaluation of the effectiveness, efficiency, and generality of our solution. The results show our solution produces sound and clear provenance in container-based microservice environments with low system overhead.

3.2. Background

3.2.1. Linux Namespace

Linux Namespaces[**36**] provide a foundational mechanism leveraged by containerization technologies to enable system-level virtualization. They are advertised as a Linux

Table 3.1. Supported Linux Namespaces

Namespace	**Isolated System Resource**
Cgroup	Cgroup root directory
IPC	System V IPC, POSIX message queues
Network	Network devices, stacks, ports, etc.
Mount	Mount points
PID	Process IDs
Time	Boot and monotonic clocks
User	User and group IDs
UTS	Hostname and NIS domain name

kernel feature that supports isolating instances of critical operating system resources including process identifiers, filesystem, and network stack across groups of processes. Internally, namespaces are implemented as an attribute of each process, such that only those processes with the same namespaces attribute value can access corresponding instances of containerized system resources. Currently, eight namespaces are supported by the Linux kernel as listed in Table 3.1.

Consider the mount namespace as an example. On a Linux operating system that has just been booted, every process runs in an initial mount namespace, accesses the same set of mount points, and has the same view of the filesystem. Once a new mount namespace is created, the processes inside the new mount namespace can mount and alter the filesystems on its mount points without affecting the filesystem in other mount namespaces.

3.2.2. Linux Container

Linux Containers may be viewed as a set of running processes that collectively share common namespaces and system setup. In practice, containers are usually created by a container engine using its container runtime. The container runtime will specify the namespace to be shared among processes running inside the container. As a concrete example, the Docker container engine specifies five namespaces (PID, Mount, Network, IPC and UTS) to be shared, initializes several system components including rootfs `/`, hostname, `/proc` pseudo-filesystem, and finally executes the target application as the first process inside the container.

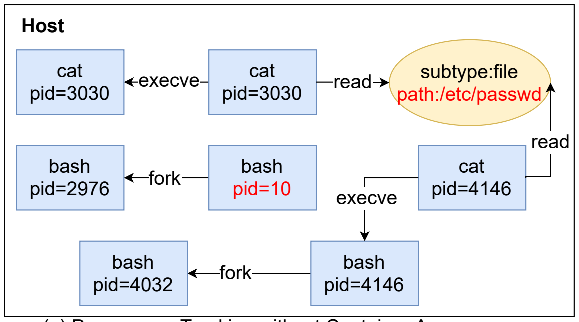
3.3. Motivation and Threat Model

3.3.1. Motivating Example

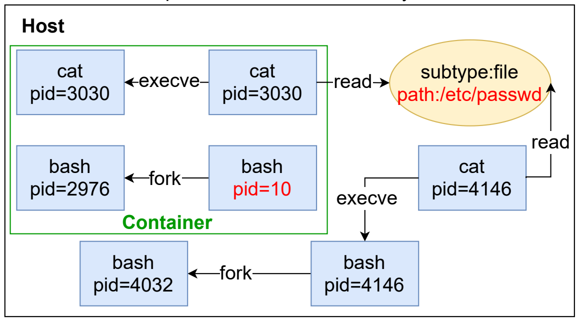
We motivate our solution by investigating the performance of three classes of state-of-the-art provenance tracking solutions against a trivial credential theft insider attack[1]. Notably, during this attack, the attacker touches the `/etc/passwd` file in both a container and the host system.

First, as shown in Figure 3.1(a), traditional solutions that lack both container and namespace awareness, e.g., SPADE, are unable to deliver a sound and clear illustration of this attack step. To illustrate soundness challenges, we explain how *fragmentation* and *ambiguity* occur in the figure. For fragmentation, when `bash (2976)` forks a child process `bash (10)` with the global PID 3030 to execute the `cat` command, the virtualized PID 10 will be reported and used in building this process creation provenance so `bash (3030)` splits into two vertices, `bash (10)` and `cat (3030)`, which build incorrect `fork` and `execve` edges correspondingly. For ambiguity, consider the file `/etc/passwd`. Since the file path is virtualized, *ambiguity* occurs on the vertex representing two `/etc/passwd` files (inside and outside the container respectively) simultaneously. The correct graph should contain two separate `/etc/passwd` file artifact vertices. With respect to clarity, it

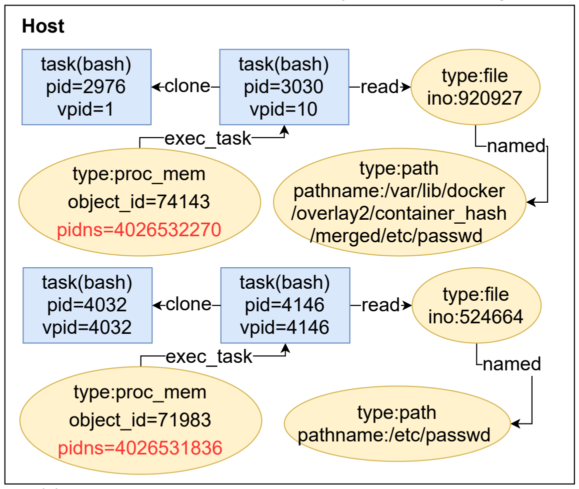
[1]A complete attack description can be found in the Appendix, but is not required to illustrate the challenges faced by container provenance systems.



(a) Provenance Tracking without Container Awareness or Namespace Awareness
Red labels represent the errors caused by this solution.



(b) Provenance Tracking with only Container Awareness
Red labels represent the errors caused by container labeling solution.


Host
task(bash) pid=2976 vpid=1
clone
task(bash) pid=3030 vpid=10
read
type:file ino:920927
exec_task
named
type:proc_mem object_id=74143 pidns=4026532270
type:path pathname:/var/lib/docker /overlay2/container_hash /merged/etc/passwd
task(bash) pid=4032 vpid=4032
clone
task(bash) pid=4146 vpid=4146
read
type:file ino:524664
exec_task
named
type:proc_mem object_id=71983 pidns=4026531836
type:path pathname:/etc/passwd

(c) Provenance Tracking with only Namespace Awareness
Red labels represent the information which could be used for identifying containers.

Figure 3.1. Comparison between different provenance tracking solutions. The traditional provenance solution graph illustrated in (a) lacks namespace awareness and container awareness. The container-aware graph shown in (b), produced by systems such as Winnower, performs slightly better because it can distinguish processes from different containers, but lacks namespace awareness, leading to disconnected intra-container provenance graphs. The namespace-aware graph, illustrated in (c), produced by CamFlow lacks container-awareness. While this graph does not suffer from the soundness issue, it cannot effectively capture essential container semantics (e.g., the boundary of containers).

is not intuitive which processes are inside the container because *container boundaries* are not marked in the graph.

Second, solutions that only provide container awareness, e.g., Winnower, also suffer from the soundness challenge. Though they can distinguish the processes inside the container in Figure 3.1(b), the ambiguity and graph fragmentation issues persist. This is also the case for other simple container labeling solutions, e.g., using a cgroup prefix or a SELinux label for every provenance artifact.

Third, solutions that only provide namespace awareness, e.g., CamFlow, still suffer from the clarity challenge. As we can see in Figure 3.1(c), they do not capture essential container semantics, such as the boundary of containers, complicating security analysis. As CamFlow provides a more fine-grained and complex provenance graph[2], non-trivial additional graph analysis will be required to design and apply similar semantic patterns in CamFlow to provide clarity. For instance, to support the boundary of containers, it is necessary in CamFlow to (1) put the PID namespace identifier on every task vertex to group processes inside a container by aggregating PID namespace information; (2) get the namespace-virtualized pathname and the mount namespace identifier for each file to reveal whether the file is inside a container by complementing mount namespace information.

For (1), we need to find the process memory vertex assigned to each task vertex and use its PID namespace identifier. Figure 3.1(c) illustrates a simple case. In practice, the graph analysis required is more complex. Because CamFlow uses versions to avoid cycles or to record any object state change for a provenance artifact, a path traversal is needed to find

[2]The provenance graph of CamFlow is framed over fine-grained kernel objects, e.g., task, process memory, inode, path, packet.

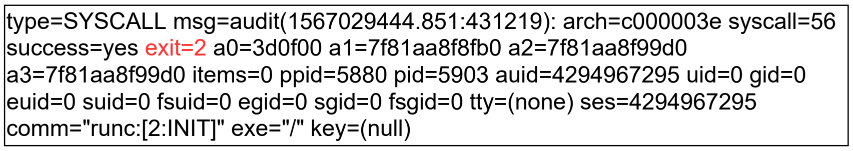
the correct version of the task vertex, i.e., where a clone tries to assign the process memory. For (2), CamFlow does not provide virtualized paths and mount namespace identifier for file vertices natively. The same state management implemented on CLARION (See Section 3.5.1.2) to track `pivot_root` and `chroot` calls and path traversal analogous to what was described above for (1) will need to be implemented within CamFlow.

3.3.2. Threat Model

We consider the OS kernel and audit subsystem, i.e., Linux Audit, to be part of the trust computing base (TCB). We assume that the OS kernel is well protected by existing kernel-protection techniques[**109, 33**]. The integrity of Linux Audit assures the ability to observe all system calls associated with malicious activity in user space. If the attackers succeed in compromising the kernel, they can disrupt the normal operation of Linux Audit and the kernel module used by CLARION. To address such attacks, the security of the TCB can be bolstered using TPM-based approaches as used by prior provenance-tracking systems[**73, 101**].

We further assume adversaries can only have limited or no *a priori* privileges. Thus, we only consider a threat model where attacks are launched from user space. This threat model is based on what was used in prior provenance tracking systems, and it is reasonable because the container virtualization does not mitigate the effort required for attackers to compromise the kernel. Implementing provenance tracking for containers and addressing namespace virtualization problems shown in Section 3.4 do not require additional information beyond what is provided in the kernel, as described in Section 3.5. Finally, the system may be subject to resource exhaustion attacks, leading to missed events. We believe that the defense against such attacks is outside the scope of this paper.

3.4. Container Provenance Challenges



```
type=SYSCALL msg=audit(1567029444.851:431219): arch=c000003e syscall=56
success=yes exit=2 a0=3d0f00 a1=7f81aa8f8fb0 a2=7f81aa8f99d0
a3=7f81aa8f99d0 items=0 ppid=5880 pid=5903 auid=4294967295 uid=0 gid=0
euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 tty=(none) ses=4294967295
comm="runc:[2:INIT]" exe="/" key=(null)
```

Figure 3.2. Problematic Linux Audit Record (PID)

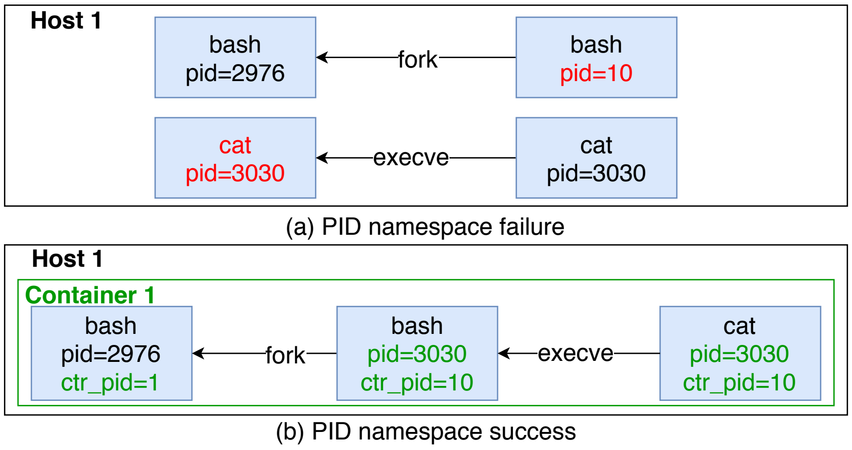
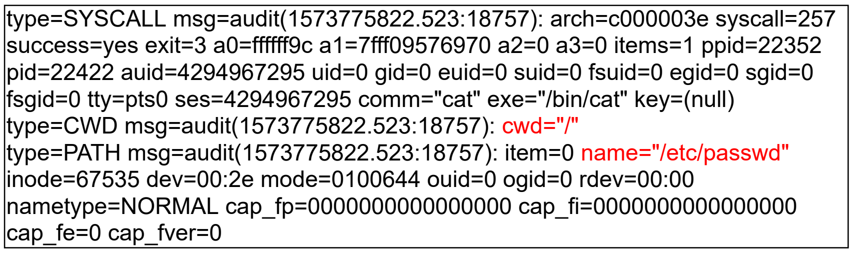


Figure 3.3. PID Namespace: Failure and Success



```
type=SYSCALL msg=audit(1573775822.523:18757): arch=c000003e syscall=257
success=yes exit=3 a0=ffffff9c a1=7fff09576970 a2=0 a3=0 items=1 ppid=22352
pid=22422 auid=4294967295 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0
fsgid=0 tty=pts0 ses=4294967295 comm="cat" exe="/bin/cat" key=(null)
type=CWD msg=audit(1573775822.523:18757): cwd="/"
type=PATH msg=audit(1573775822.523:18757): item=0 name="/etc/passwd"
inode=67535 dev=00:2e mode=0100644 ouid=0 ogid=0 rdev=00:00
nametype=NORMAL cap_fp=0000000000000000 cap_fi=0000000000000000
cap_fe=0 cap_fver=0
```

Figure 3.4. Problematic Linux Audit Record (Mount)

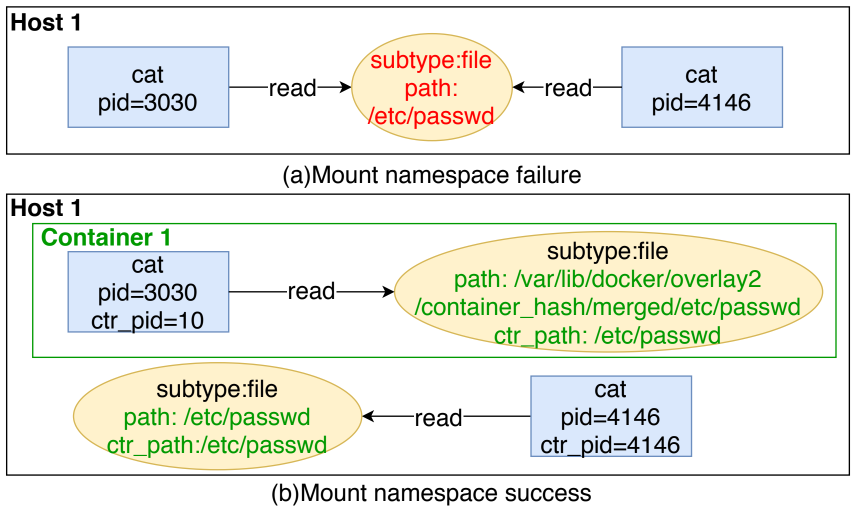

Host 1
cat pid=3030
read
subtype:file path: /etc/passwd
read
cat pid=4146
(a)Mount namespace failure
Host 1
Container 1
cat pid=3030 ctr_pid=10
read
subtype:file path: /var/lib/docker/overlay2 /container_hash/merged/etc/passwd ctr_path: /etc/passwd
subtype:file path: /etc/passwd ctr_path:/etc/passwd
read
cat pid=4146 ctr_pid=4146
(b)Mount namespace success

Figure 3.5. Mount Namespace: Failure and Success

We elaborate on the soundness and clarity challenges introduced by mishandling the effect of container virtualization in each namespace. Through the analysis in this section, we also ensure that the technique we propose in Section 3.5 covers all the needed namespace interactions.

3.4.1. Soundness: Namespace Virtualization

As illustrated in the motivating example, *fragmentation* and *ambiguity* are soundness issues caused by namespace virtualization in provenance tracking. However, not every namespace triggers either or both issues. In Table 3.2, we provide a deeper analysis about *how* each namespace impacts provenance tracking and *what* events will be affected. In addition, we use audit records from Linux Audit to demonstrate the problem and show how the soundness challenge can extend to other monitoring techniques such as Sysdig and LTTng.

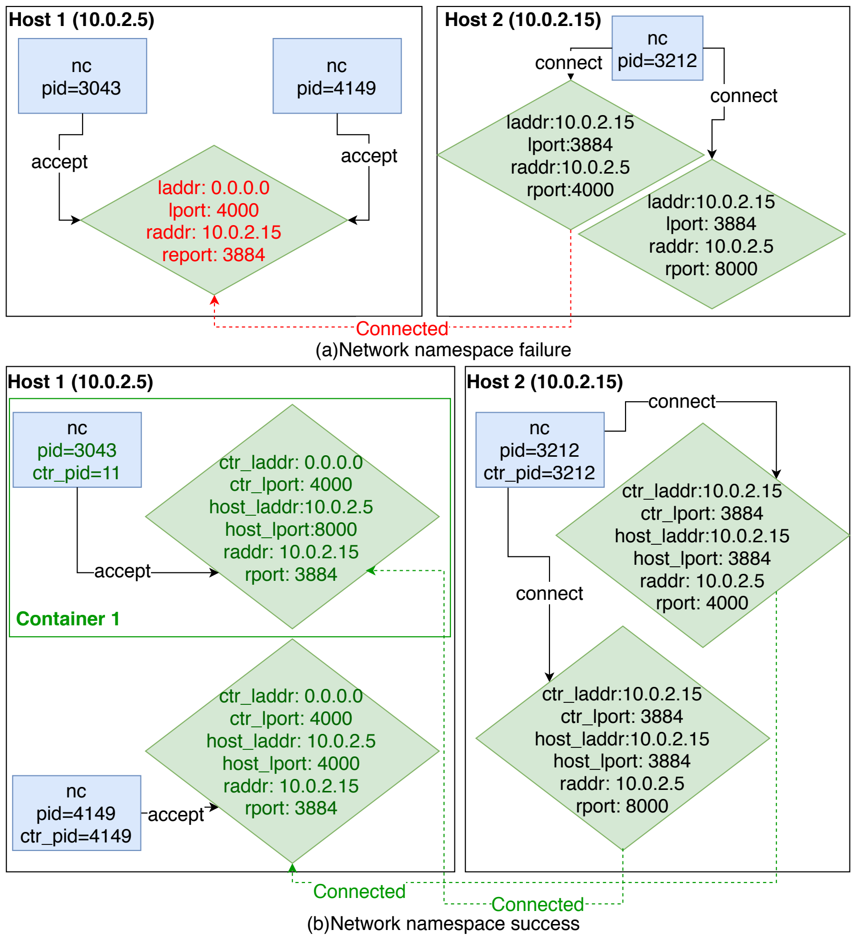


Figure 3.6. Network Namespace: Failure and Success

3.4.1.1. PID Namespace. Figure 3.2 shows a problematic audit record. It is created by a `runC` container runtime process inside a container and trying to finish the initialization. Syscall value 56 means that it is a `clone` system call, and its return value is the PID of the cloned child process. Here, we can see that exit value is 2, but the process 2 is usually a kernel-related process generated when the system is booted. It suggests process 2 cannot be the cloned child process of this `runC` runtime process, which is confirmed by our further investigation. So 2 cannot be the global PID for the cloned child process. It can only be a virtualized PID.

Figure 3.3 illustrates the subgraph exposing the fragmentation caused by PID namespace virtualization in the motivating example. The `bash` process 2976 expects that it created child process 10 which is actually process 3030.

3.4.1.2. Mount Namespace. Figure 3.4 shows a problematic audit record. A system call `openat` (inferred by syscall=257) is invoked by a process trying to read `/etc/passwd` in the container. As we can see, the CWD is `/` and the PATH is `/etc/passwd`. In fact, all files inside the container are stored under some directory specific to this container. This specific directory may vary due to different container engine choices. Taking Docker as an example, the specific directory is usually `/var/lib/docker/overlay2/container_hash/merged/` where `container_hash` is a hash string related to this container. So to get the global paths of the CWD `/` and the PATH `/etc/passwd`, the path of the specific directory needs to be added to them as the prefix.

Figure 3.5 illustrates the subgraph exposing the mount namespace virtualization problem described in the motivation example. Two `cat` processes (with PIDs 3030 and 4146), are attempting to read the `/etc/passwd` file, and the two files are confused with each other without mount-namespace awareness. CLARION's host-container mapping enables us to easily distinguish between them.

3.4.1.3. Network Namespace. Figure 3.6 illustrates the subgraph exposing the network namespace virtualization problem in the motivation example. Two `nc` processes (PID 3043 and 4149) are listening on socket (0.0.0.0/4000) within their respective containers, and one of them accepts a connection from (10.0.2.15/3884). Since the local IP addresses/ports are virtualized and remote IP addresses/ports are the same, the two sockets can be confused with each other without network-namespace awareness. Without establishing the host-container mapping of sockets inside the container, we are unable to attribute the connection to a socket inside the container, as illustrated in Figure 3.6.

3.4.1.4. Soundness Challenge on Other Audit Subsystems. We further investigated the impact of container virtualization on two alternative Linux audit subsystems, specifically Sysdig[**62**] and LTTng[**37**], to assess whether soundness challenge impacts other systems besides Linux Audit. We summarize our findings in Table 3.3. We find that Sysdig suffers from the same soundness challenges confronted by Linux Audit. LTTng provides host-container ID mappings using more low-level events[3] but the soundness challenge in mount namespace still persists. Our investigation shows that soundness challenge is not specific to Linux Audit.

3.4.1.5. Soundness Challenge on Rootless Containers. Rootless containers refer to the containers that can be created, run, and managed by unprivileged users. They differ from traditional containers in which they have a new unprivileged user namespace. In this user namespace, all UIDs and GIDs are mapped from the global user namespace, including a pseudo-root user. This core difference causes further effects in filesystem and networking in the rootless container. For filesystem, many Linux distributions do not allow mounting overlay filesystems in user namespaces. This limitation makes rootless containers inefficient. For networking, virtual Ethernet (veth) devices[4] cannot be created as only real root users have the privileges to do so.

[3]For example, the `sched_process_fork` event.

[4]Veth devices are a special type of Linux interface used in virtual networking. They are always created in pairs and usually serve as local Ethernet tunnel between namespaces in container networks.

As summarized in Table 3.2, the user namespace does not affect the soundness of provenance analysis. Further, although rootless containers have slightly different implementations for filesystem and networking (mentioned above), to support unprivileged root users, they do not affect provenance. Thus, we claim that rootless containers share the same soundness challenges faced by traditional containers.

3.4.1.6. Soundness Challenge on Other OS Platforms. We also investigated two alternative resource isolation techniques, specifically FreeBSD Jails and Solaris Zones, to see whether soundness challenge can also occur in other platforms. We summarize our findings in Table 3.4. Our key finding is that if semantics inconsistency exists in the low-level audit events related to virtualized system resources (e.g., PIDs, file paths, network addresses/ports), the resource-isolation technology will suffer from the soundness challenge. We assume this finding also extends to other OS platforms like Windows and MacOS. Semantic inconsistencies are at the core of the soundness challenge so the key to make CLARION feasible on those platforms is to systematically address such inconsistencies.

3.4.2. Clarity: Essential Container Semantics

We describe the challenges involved in automating the comprehension of essential container semantics. This is a feature that is unique to our provenance tracking system, and we believe it can greatly simplify the understanding and analysis of dataflow provenance in container-based microservice environments.

An important aspect of forensic analysis is accurately knowing what subgraphs correspond to which container so that we can effectively track the origins of a container microservice attack as well as assess the forensic impact of such attacks. For example, was the effect of the attack limited to the specific container or was it used as a stepping stone to other container targets? To effectively answer such questions, we need to demystify the boundary of containers in the provenance graph.

Initialization of containers is another frequent activity that explodes provenance graphs and may be abstracted to simplify analysis. Thus, if we can accurately identify subgraphs corresponding to initialization of each containers, we can produce simplified provenance graphs, effectively reducing the effort for forensic analysts by automatically annotating abnormal cross-container activity. Specifically, we investigate the container initialization regulation of several representative container engines, including Docker, rkt and LXC, and summarize the patterns observed in each of them.

3.5. System Design and Implementation

In this section, we provide a detailed description of the CLARION prototype design and the implementation that extends the SPADE provenance tracking system with additional container-specific extensions. Our design goal is to propose a solution for addressing *soundness* and *clarity* challenges by only using trusted information from the kernel while limiting extra instrumentation.

We claim that our solution is complete in handling all aliasing introduced by namespaces. First, we cover all system calls that can be used to manipulate namespaces generally, i.e., `clone`, `unshare` and `setns`. We investigate their semantics and provide associated provenance data models with consideration to different argument combinations as shown in Section 3.5.2. Second, we analyze all existing namespaces and understand what information will be aliased in the low-level audit and cause problems to provenance tracking as shown in Section 3.4.1. Our solution is designed to address all introduced problems below in Section 3.5.1.

3.5.1. Mapping Virtualized Namespaces

We summarize our virtualized namespace-mapping strategies in Table 3.5. For the soundness challenge, we establish a *host-container mapping* view on provenance graph artifacts that are impacted by most Linux namespaces because we believe this will provide the most clear view for users to understand the provenance. However, for the IPC namespace, the host view of an IPC object does not actually exist. Hence, we adopt a *namespace-labeling* approach.

3.5.1.1. PID Namespace. We considered multiple options to tackle the PID host-container mapping problem including: (*i*) directly using PPID (parent PID) to connect processes; (*ii*) using timestamps to map cloned child processes to its parent; (*iii*) using `/proc/PID/status` for mapping information; and (*iv*) using kernel module injection to get the PID mapping from kernel data structures.

We ultimately eliminated other options and chose to implement a kernel module for several reasons. We found that directly using PPID was infeasible because it sometimes points to the parent of the the process creating it. For the timestamp option, the granularity provided by audit record cannot guarantee that the order of process creation matches the order corresponding system call events. We also decided against using `/proc/PID/status` information as the `/proc` filesystem does not support asynchronous callbacks and the overhead of polling is prohibitive.

We implement our PID namespace host-container mapping solution as a kernel module that intercepts process-manipulation-related system calls, e.g., `clone`, `fork`, and `vfork`. Once those system calls are invoked by a process, we do not directly use the return value to determine the PID of its child process because it can be virtualized. Instead, we input this return value to a kernel helper function `pid_nr()` in /include/linux/pid.h to generate the global PID. Ultimately, we use the global PID to generate the sound provenance graph. However, we still capture both the global PID and virtualized PID for every process vertex such that a complete view can be provided.

3.5.1.2. Mount Namespace. To obtain the host-container mapping for file paths virtualized by containers, we leverage an empirically derived design principle about the mount namespace, that is consistent across state-of-the-art container engines, to develop an instrumentation-free solution.

This empirical design principle is that the newly created mount namespace needs the `init` process, i.e., the process with virtual PID 1, to provide a new filesystem view different from that in the parent mount namespace. It is achieved by using root directory change system calls, i.e., `pivot_root` and `chroot`, where new root directories are provided in their arguments. Specifically, state-of-the-art container engines make the `init` process move CWD to the root directory of a new container by using `chdir(container_root_path)` and then invoke a `pivot_root('.','.')` or a `chroot('.')` to wrap up the root directory change.

Therefore, if we monitor those root directory change system calls, we can use the CWD record associated with the `chdir` to find the host path of the container root directory, and then we attach this host path to every virtualized path as a prefix to establish the host-container mapping on file paths. Given the annotation in Table 3.6, the algorithm is described as four steps.

Step 1. Handle `chdir`. (input: PID 'p1', CWD 'cwd1'; operation: put((p1,cwd1), LastCWD)). We do this to record the last working directory for every process. With this information we can know what is the last CWD of the first process inside a new container, which will be the prefix for every virtualized path.

Step 2. Handle `pivot_root` or `chroot`. (input: PID 'p1'; operation: put((p1, get(p1, LastCWD)), Prefix)). When a root directory changing system call occurs, we label the corresponding process with the last CWD as the prefix.

Step 3. Handle virtualized PATH records, CWD records and arguments related to file operation system calls with path prefix. (input: PID 'p1', syscall 's1', operation: if 's1' is 'open','read','write' etc. Use get(p1, Prefix) to add a new annotation 'nsroot' representing the host prefix in the corresponding artifacts). This helps propagate the prefix from processes to file artifacts.

Step 4. Handle (`clone`, `fork`, `vfork`). (input: Parent PID 'p1', Child PID 'p2'; operation: put((p2, get(p1, Prefix)), Prefix)). The prefix (root directory) information will be propagated through process creation as kernel does.

We consider our mount namespace mapping solution to be robust because it relies on a standardized implementation technique for filesystem isolation and empirically validate its adoption across representative container engines including Docker, rkt and LXC.

For other cases where directories are shared between host and container than chroot-like cases, we claim that our solution still works well. Taking bind mount as an example, the key components in the bind mount provenance graph will be one process vertex which executes a `mount` system call along with two file artifacts representing the bound directories and two file artifacts are connected by an edge representing that `mount` system call. In this case, only the file path of the file artifact inside the container will be affected and our solution can still provide the host view of this file.

3.5.1.3. Network Namespace. For accurate provenance tracking of container network activity, CLARION needs to establish the host-container mapping for virtualized local IP addresses and ports. To this end, we design a Netfilter-based solution for tracking the host-container IP/port mapping and use the network namespace ID as a distinguisher. Netfilter is a Linux-kernel framework that provides hooks to monitor every ingress and egress packet, including packets from or to containers, on the host network stack [**44**]. The host network stack will do a source NAT for container egress packets and a destination NAT for container ingress packets before correctly forwarding those packets. Therefore, by monitoring the IP/port NAT about container ingress/egress packets on the host network stack, we can build the host-container mapping of local IP addresses and ports for sockets inside containers. We annotate each network socket artifact with the corresponding network namespace identifier, so sockets from different containers can be reliably distinguished.

The CLARION prototype implementation for the network namespace consists of two parts: network namespace identification and netfilter-based address mapping. For network namespace identification, we modify SPADE's kernel module to intercept network-related system calls and put the network namespace identifier of the calling process on the generated network socket. For netfilter-based mapping, we register kernel modules at the beginning and the end of netfilter hooks corresponding to NAT. Specifically, POST_ROUTING and LOCAL_INPUT are used for source NAT, while PRE_ROUTING and LOCAL_OUTPUT are used for destination NAT. The former two hooks provide the mapping for egress connections from container and the latter two provide the mapping for ingress connections.

Whenever a new mapping is added, we will search for the network device having the virtualized local IP address in the new mapping, by iterating through network namespaces using the function `ip_dev_find(struct net *net, __be32 addr)`. Through this, we find the container related to this virtualized local IP address and put the mapped global local IP address/port on the socket artifact that has the virtualized local IP address/port in the new mapping. As a special case, a socket may listen on `0.0.0.0` (IN_ADDR_ANY), i.e., it can accept connection on any local IP address. Hence, when we match socket artifacts with the virtualized local IP address/port in the container, we always treat `0.0.0.0` as a matched local IP and only check the local port.

3.5.1.4. IPC Namespace. The issue in the IPC namespace is that two different IPC objects from different IPC namespaces may have the same ID/name. Unlike other namespaces, the host-container mapping strategy for disambiguation does not extend to IPC object artifacts, because there is no corresponding host IPC object for virtualized IPC objects. Our design involves adding an IPC namespace ID to every IPC object artifact so that IPC objects from different containers can be uniquely distinguished.

The implementation of the IPC namespace solution was effected by adding IPC namespace IDs to IPC objects affected by namespace virtualization. Those objects consist of the POSIX message queue and all System V IPC objects, i.e., message queue, semaphore,

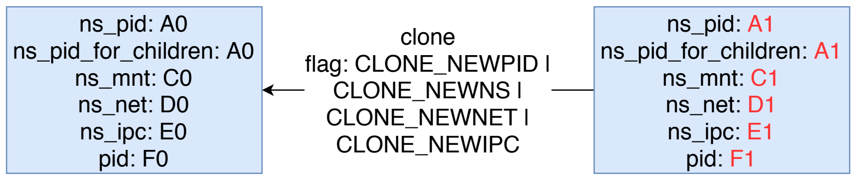


Figure 3.7. Handling the `clone` system call: a process vertex representing the child will be created with the new namespace label.

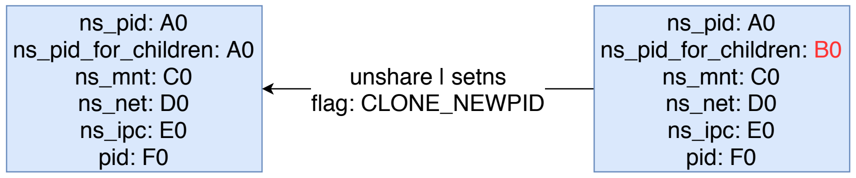


Figure 3.8. Handling the `unshare` and `setns` system calls on NEWPID: a process vertex representing the calling process itself will be created with the new assigned pid_for_children label.

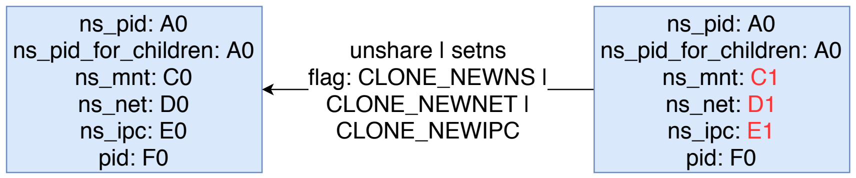
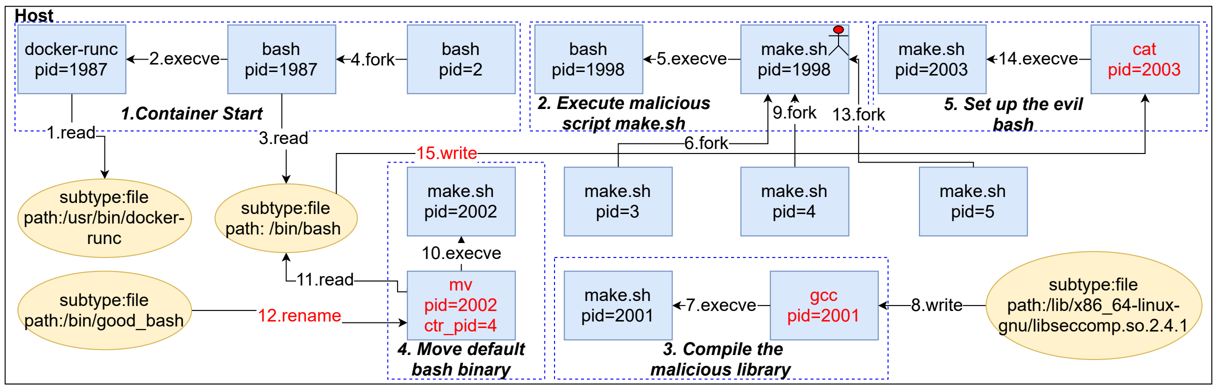


Figure 3.9. Handling `unshare` and `setns` system calls with other flags: a process vertex representing the calling process itself will be created with the new assigned namespace label.

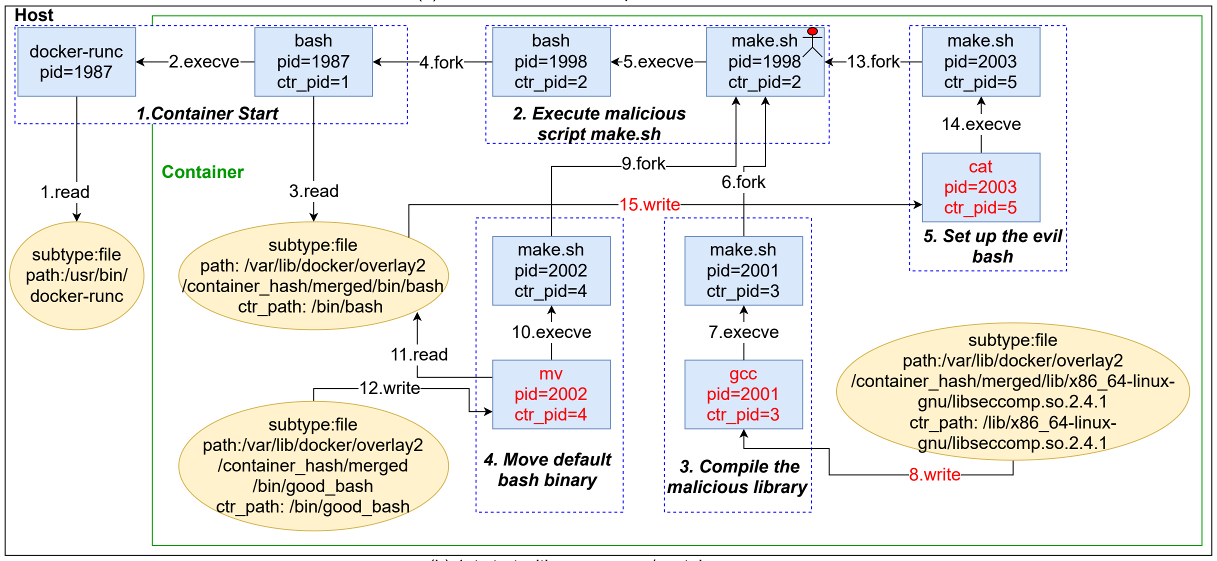
and shared memory. We assign and propagate IPC namespace ids by carefully interpreting process management system calls, e.g., `clone`, and IPC object management system calls, e.g., `msgget` and `msgsnd`.

3.5.2. Essential Container Semantic Patterns

To address the clarity challenge, we propose two essential container semantics which can significantly improve the quality of provenance graph. In addition, we design the semantic patterns for summarizing them during provenance tracking.



(a) 1st start without namespace/container awareness


Host
Container
docker-runc pid=1987
2.execve
bash pid=1987 ctr_pid=1
4.fork
1.Container Start
1.read
3.read
subtype:file path:/usr/bin/docker-runc
subtype:file path: /var/lib/docker/overlay2 /container_hash/merged/bin/bash ctr_path: /bin/bash
11.read
12.write
subtype:file path:/var/lib/docker/overlay2 /container_hash/merged /bin/good_bash ctr_path: /bin/good_bash
bash pid=1998 ctr_pid=2
5.execve
make.sh pid=1998 ctr_pid=2
2. Execute malicious script make.sh
9.fork
6.fork
15.write
make.sh pid=2002 ctr_pid=4
10.execve
mv pid=2002 ctr_pid=4
4. Move default bash binary
make.sh pid=2001 ctr_pid=3
7.execve
gcc pid=2001 ctr_pid=3
3. Compile the malicious library
8.write
13.fork
make.sh pid=2003 ctr_pid=5
14.execve
cat pid=2003 ctr_pid=5
5. Set up the evil bash
subtype:file path:/var/lib/docker/overlay2 /container_hash/merged/lib/x86_64-linux-gnu/libseccomp.so.2.4.1 ctr_path: /lib/x86_64-linux-gnu/libseccomp.so.2.4.1

(b) 1st start with namespace/container awareness

Figure 3.10. CVE 2019-5736: Provenance graph for 1st start without (top) and with (bottom) namespace/container awareness

3.5.2.1. Boundary of Containers. We begin by first providing a practical definition for a container at runtime. A container at runtime is a set of processes that share the same PID namespace. Usually processes inside a container can share multiple namespaces but, most critically, they at least have to share the PID namespace. In fact, while container runtimes often provide support for sharing other namespaces, e.g., mount, IPC, and network, between containers, none of them allow for sharing the PID namespace.

Next, we define the relationship between an artifact, e.g., file and network, and a container. An artifact relates to a container if and only if it can be accessed by a process inside that container. Here, "accessed" may refer to any type of read-write operation. An artifact may relate to several containers and thus may be used to infer the relationship between specific containers. An important challenge is labeling each process with the correct namespace identifier. We address this by carefully designing a new provenance data model for system calls related to namespace operations. There are three essential system calls for tracking the boundary of containers, i.e., `clone, unshare` and `setns`. `Clone` and `unshare` system calls are used for creating new namespaces; thus, they signal the process of creating a container boundary. `Setns` is used for aggregating two namespace together or making another process join a namespace.

We designed five different namespace labels (corresponding to PID, mount, network, IPC, and pid_for_children) and handle them when three essential namespace-related system calls (i.e., `clone`, `unshare`, and `setns`) occur, as shown in Figure 3.7, 3.8 and 3.9. All figures are illustrated in the OPM provenance data model format. The red areas highlight the changes between before and after. The implementation follows the Linux Kernel semantics for each system call and each namespace. The special case here is that if CLONE_NEWPID flag is specified for `unshare` or `setns` process, this only affects the child process generated by the calling process but does not affect the calling process itself. By adding namespace labels and handling namespace-related system calls, CLARION is able to capture the namespace information for every single process and leverage the PID namespace label to certify the boundary of each container.

3.5.2.2. Initialization of Containers. By analyzing several state-of-the-art container engines, we find that specific common pattern exist across containers that may be leveraged to identify the initialization of containers. In a nutshell, this pattern can be summarized as follows: start with an `unshare/clone` with new namespace flag specified, and end with an `execve` so that a new application can be launched inside the container. Slight differences exist across different container engines as described in Section 3.5. Identifying these patterns facilitates abstracting subgraphs in the provenance graph that corresponds to container spawning and initialization activity.

Here, we explain the container initialization patterns for Docker and rkt. For Docker, the initialization pattern is as follows:

- After receiving gRPC connection from `dockerd`, `containerd` will start a `containerd -shim`, which is responsible for starting a new container.
- This `containerd-shim` process will invoke several `runC` processes for initialization.
- One of those `runC` processes will invoke the `unshare` system call and this marks the beginning of the actual container initialization.
- The `runC` process calling `unshare` will clone several child processes to finish several initialization tasks including setting up `/proc`, `/rootfs`, and the network stack.
- Finally, it will clone a child process and make it execute the default container application, e.g., bash and apache.

Unlike centralized container engines like Docker, rkt does not have a daemon process that is responsible for starting a container. It has a more complex three-stage initialization pattern that begins once `rkt` is started with specified parameters to create a rkt container.

- **Stage 0:** It will use several instances of the `systemd` process to set up different namespaces including PID, Mount, Network, IPC and UTS.
- **Stage 1:** It will generate process inside the container with namespace restriction set up and call `chroot` to create a filesystem jail for this container.
- **Stage 2:** Finally, it will run the default application on this process.

We implement those patterns as a SPADE filter, and it automatically finds the starting point of those initialization patterns and attempts to do a backward traversal so the subgraph corresponding to initialization will be marked.

3.6. System Evaluation

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

- **Q1. Usefulness:** How *effective* is CLARION in dealing with real-world container microservice attacks?
- **Q2. Generality:** Are namespace disambiguation strategies implemented by CLARION for performing provenance tracking *generally* applicable across different container engines?
- **Q3. Performance:** Does CLARION provide an *efficient* provenance monitoring solution for real-world microservice deployments?

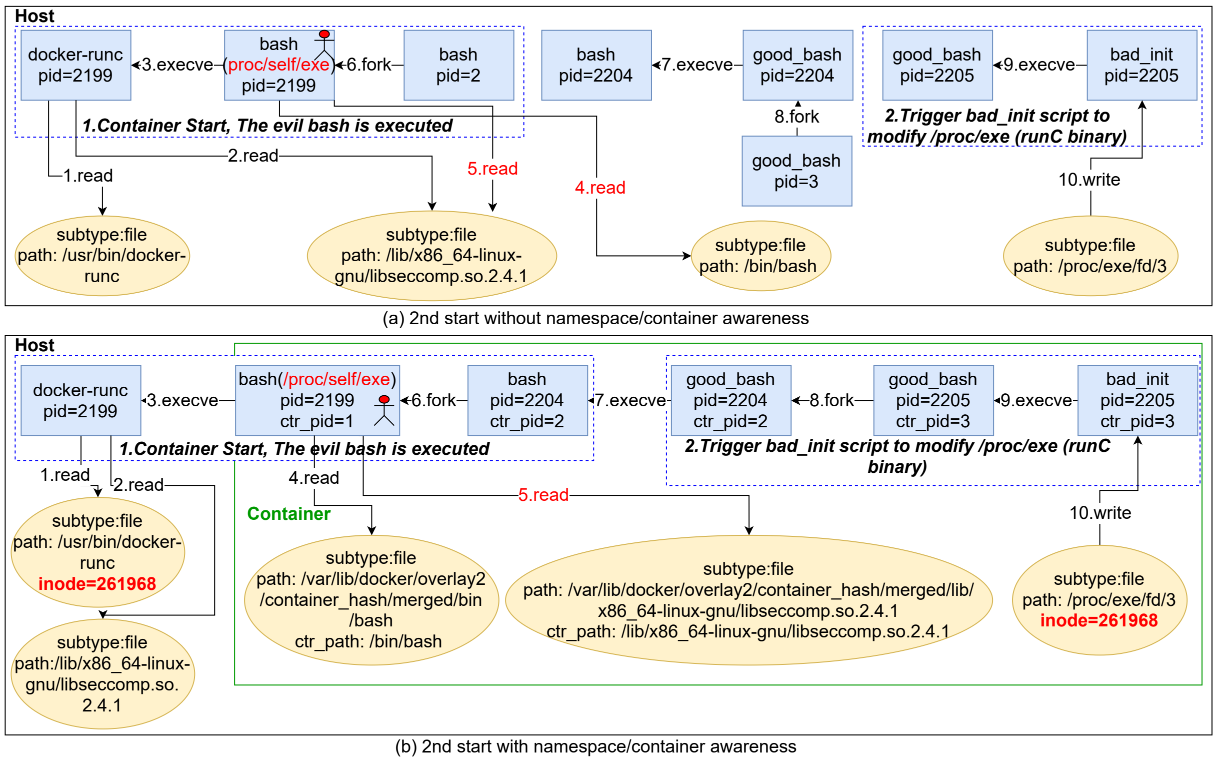

Host
docker-runc pid=2199
3.execve
bash (proc/self/exe) pid=2199
6.fork
bash pid=2
bash pid=2204
7.execve
good_bash pid=2204
good_bash pid=2205
9.execve
bad_init pid=2205
1.Container Start, The evil bash is executed
8.fork
good_bash pid=3
2.Trigger bad_init script to modify /proc/exe (runC binary)
2.read
1.read
5.read
4.read
10.write
subtype:file path: /usr/bin/docker-runc
subtype:file path: /lib/x86_64-linux-gnu/libseccomp.so.2.4.1
subtype:file path: /bin/bash
subtype:file path: /proc/exe/fd/3
(a) 2nd start without namespace/container awareness
Host
docker-runc pid=2199
3.execve
bash(/proc/self/exe) pid=2199 ctr_pid=1
6.fork
bash pid=2204 ctr_pid=2
7.execve
good_bash pid=2204 ctr_pid=2
8.fork
good_bash pid=2205 ctr_pid=3
9.execve
bad_init pid=2205 ctr_pid=3
1.Container Start, The evil bash is executed
2.Trigger bad_init script to modify /proc/exe (runC binary)
1.read
2.read
4.read
5.read
10.write
Container
subtype:file path: /usr/bin/docker-runc inode=261968
subtype:file path:/lib/x86_64-linux-gnu/libseccomp.so.2.4.1
subtype:file path: /var/lib/docker/overlay2/container_hash/merged/bin/bash ctr_path: /bin/bash
subtype:file path: /var/lib/docker/overlay2/container_hash/merged/lib/x86_64-linux-gnu/libseccomp.so.2.4.1 ctr_path: /lib/x86_64-linux-gnu/libseccomp.so.2.4.1
subtype:file path: /proc/exe/fd/3 inode=261968
(b) 2nd start with namespace/container awareness

Figure 3.11. CVE 2019-5736: Provenance graph for 2nd start without (top) and with (bottom) namespace/container awareness

3.6.1. Effectiveness Evaluation

To illustrate the effectiveness of CLARION for container-based provenance and forensic analysis, we evaluate against exploits of three recent CVEs affecting Docker. Specifically, we generate the provenance graphs with and without namespace and container awareness to show the capability of our solution. Then, we compare between the original provenance graphs generated by SPADE and CLARION.

The CVEs that we selected include CVE 2019-5736 (`runC`), CVE 2019-14271 (`docker-tar`) and CVE 2018-15664 (`docker-cp`). The first two exploits are particularly detrimental because they can lead to privilege escalation in the host machine. The third is a

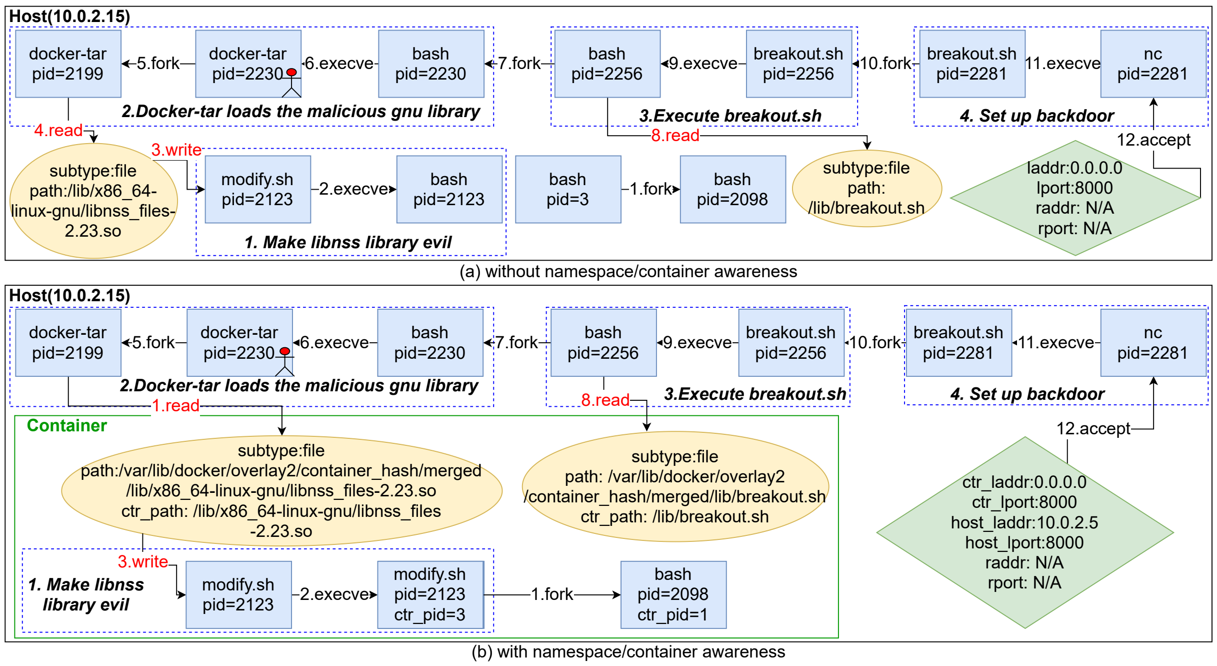

Host(10.0.2.15)
docker-tar pid=2199
5.fork
docker-tar pid=2230
6.execve
bash pid=2230
7.fork
bash pid=2256
9.execve
breakout.sh pid=2256
10.fork
breakout.sh pid=2281
11.execve
nc pid=2281
2.Docker-tar loads the malicious gnu library
3.Execute breakout.sh
4. Set up backdoor
4.read
3.write
8.read
12.accept
subtype:file path:/lib/x86_64-linux-gnu/libnss_files-2.23.so
modify.sh pid=2123
2.execve
bash pid=2123
1. Make libnss library evil
bash pid=3
1.fork
bash pid=2098
subtype:file path: /lib/breakout.sh
laddr:0.0.0.0 lport:8000 raddr: N/A rport: N/A
(a) without namespace/container awareness
Host(10.0.2.15)
docker-tar pid=2199
5.fork
docker-tar pid=2230
6.execve
bash pid=2230
7.fork
bash pid=2256
9.execve
breakout.sh pid=2256
10.fork
breakout.sh pid=2281
11.execve
nc pid=2281
2.Docker-tar loads the malicious gnu library
3.Execute breakout.sh
4. Set up backdoor
1.read
8.read
12.accept
Container
subtype:file path:/var/lib/docker/overlay2/container_hash/merged /lib/x86_64-linux-gnu/libnss_files-2.23.so ctr_path: /lib/x86_64-linux-gnu/libnss_files -2.23.so
subtype:file path: /var/lib/docker/overlay2 /container_hash/merged/lib/breakout.sh ctr_path: /lib/breakout.sh
ctr_laddr:0.0.0.0 ctr_lport:8000 host_laddr:10.0.2.5 host_lport:8000 raddr: N/A rport: N/A
3.write
1. Make libnss library evil
modify.sh pid=2123
2.execve
modify.sh pid=2123 ctr_pid=3
1.fork
bash pid=2098 ctr_pid=1
(b) with namespace/container awareness

Figure 3.12. CVE 2019-14271: Provenance graph for the `docker-tar` exploit without (top) and with (bottom) namespace/container awareness

race-condition (time-dependent) which requires multiple tries and some serendipity for successful exploitation.

3.6.1.1. CVE 2019-5736: runC Exploit. `runC` through 1.0-rc6 (as used in Docker before 18.09.2 and other platforms like LXC), allows attackers to overwrite the host `runC` binary (and consequently obtain host root access) by leveraging the ability to execute a command as root within one of these containers: (1) a new container with an attacker-controlled image, or (2) an existing container, to which the attacker previously had write access, that can be attached with docker exec. This occurs because of file-descriptor mishandling, related to `/proc/self/exe` [**14**]. Overwriting `runC` can lead to a privilege escalation attack by replacing `runC` binary with a backdoor program. The following four steps are used to demonstrate a successful exploitation using this vulnerability:

(1) Create a container where we have gcc installed.

(2) Create three files in this container with the `docker cp` command. Those files are (1) a script (`bad_init.sh`) that overwrites the executable (`/proc/self/exe`) of the process running this script; (2) a C program file (`bad_libseccomp.c`) that invokes `bad_init.sh`; and (3) a shell script (`make.sh`) for compiling `bad_libseccomp.c` and setting up the bait for `runC`.

(3) Start this container and execute `make.sh` that accomplishes two goals: (1) replaces the `seccomp` module with `bad_libseccomp.c`. Here `seccomp` module is a regular library which will be automatically loaded when an Ubuntu container starts; (2) replaces the default start-up process, i.e., the bash shell in Ubuntu containers, with `/proc/self/exe`.

(4) If and when this container is restarted, `runC` on the host loads the malicious `seccomp` module leading to execution of the malicious script (`bad_init.sh`), which overwrites the parent process, i.e., `runC` will be overwritten with the contents of `bad_init.sh`.

We illustrate the provenance graphs associated with this exploit in Figures 3.10, 3.11. This exploit consists of two starts of a malicious container. Figure 3.10 corresponds to the first start and Figure 3.11 corresponds to the second start.

We see that in the graphs without namespace awareness, the provenance graph fractures completely. Specifically, subgraphs corresponding to essential exploit steps are fractured, making it challenging for analysts to do backward and forward tracking given the attack points on `make.sh` and `bash`. Furthermore, ambiguity exists everywhere in

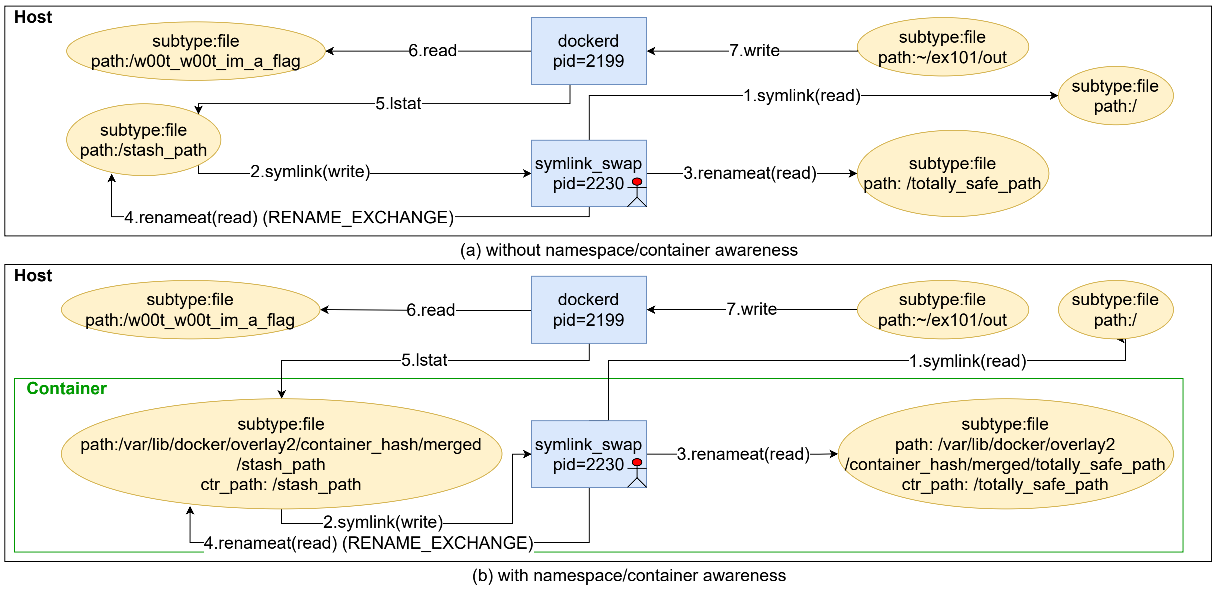

Host
subtype:file path:/w00t_w00t_im_a_flag
6.read
dockerd pid=2199
7.write
subtype:file path:~/ex101/out
5.lstat
1.symlink(read)
subtype:file path:/
subtype:file path:/stash_path
2.symlink(write)
symlink_swap pid=2230
3.renameat(read)
subtype:file path: /totally_safe_path
4.renameat(read) (RENAME_EXCHANGE)
(a) without namespace/container awareness
Host
subtype:file path:/w00t_w00t_im_a_flag
6.read
dockerd pid=2199
7.write
subtype:file path:~/ex101/out
subtype:file path:/
5.lstat
1.symlink(read)
Container
subtype:file path:/var/lib/docker/overlay2/container_hash/merged /stash_path ctr_path: /stash_path
symlink_swap pid=2230
3.renameat(read)
subtype:file path: /var/lib/docker/overlay2 /container_hash/merged/totally_safe_path ctr_path: /totally_safe_path
2.symlink(write)
4.renameat(read) (RENAME_EXCHANGE)
(b) with namespace/container awareness

Figure 3.13. CVE2018-15664: `docker cp` race condition exploit without and with namespace/container awareness. Steps 1 and 2 are attempts to establish the symlink between the stash path inside the container and the root path on the host. Steps 3 and 4 represent the renaming exchange between the symlinked stach path and the path of the file to be copied. Steps 5-7 show that `dockerd` didn't resolve the correct path and ultimately copies the incorrect file.

those namespace-unaware graphs. Among many points exposing ambiguity, the ambiguity between two `/lib/x86_64-linux-gnu/libseccmp.so.2.4.1` file artifacts in the second start is significant. If we cannot distinguish between them, we will not be able to understand that a malicious GNU library inside the container is linked with the `runC` instance (`/proc/self/exe`).

CLARION can successfully restore the connection between essential exploit steps in the namespace-aware provenance graphs and also resolve the associated ambiguity issues, making it very clear that a malicious container library is linked by the `runC` instance (which is anomalous).

3.6.1.2. CVE 2019-14271: docker-tar Exploit. Docker 19.03.x (prior to 19.03.1) linked against the GNU C Library (`glibc`) is vulnerable to code injection attacks, that may occur when the nsswitch facility dynamically loads a library inside the container using `chroot` [5] [**13**]. This code injection can affect the library files on the host and may be used to trigger privilege-escalation attacks.

We exploit this privilege-escalation vulnerability using `docker-tar`, a helper process spawned by the Docker engine that obviates the need to manually resolve symlinks in the container filesystem. First, `docker-tar` uses the `chroot` command to change its root to the container's root. This is done so that all symlinks are resolved relative to the container's root, preventing any ambiguities. After running `chroot`, `docker-tar` attempts to link with several standard `glibc` libraries, which induces the vulnerability. When `docker-tar` attempts to link with these libraries, it will instead link the library files inside the container. However, a malicious image could inject code inside those library files, such that the malicious code executes as part of the `docker-tar` process since they are linked by `docker-tar`. Specifically, two steps are required to demonstrate exploitation of this vulnerability:

(1) We modify `libnss_files.so` in the container image by a malicious script `modify.sh`, an example library linked with `docker-tar`, using a code injection attack such that it includes additional code to execute a malicious script, called `breakout.sh`, that sets up a backdoor on the host using `netcat`.

[5]The assumption here is that libraries within the containers are untrusted from the perspective of the host.

(2) When we then run the `docker-tar` command from a container using this image, the `docker cp` command is executed within the container that copies the library file to the host, leading to an open backdoor on the host.

Provenance graphs for comparison are shown in Figure 3.12. Similar to the first exploit, the namespace-unaware provenance graph is fractured. We see that bash process 2098 forked a child process but does nothing due to PID namespace fracturing. In addition, this graph implies that the `libnss` library on the host was compromised, which is incorrect. In contrast, CLARION eliminates graph fracturing and provides a sound and clear understanding of how this attack is initiated from inside the container. Specifically, (1) CLARION marks the boundary of containers so we know the starting malicious script `modify.sh` is run inside the container; and (2) CLARION provides the mapped path for the library file so we know the compromised `libnss_files.so` is inside the container.

3.6.1.3. CVE 2018-15664: Symlink TOCTOU Exploit. In Docker (versions inclusively before 18.06.1-ce-rc2), API endpoints behind the `docker cp` command are vulnerable to a symlink-exchange attack with Directory Traversal. This gives attackers arbitrary read-write access to the host filesystem with root privileges, because `daemon/archive.go` does not do archive operations on a frozen filesystem (or from within a `chroot`) [**12**].

When `docker cp` attempts to use a symlink from the container directory, it must find the absolute pathname file or directory in the context of the container filesystem. Failing to do so causes the symlink to be resolved in the host's filesystem, which allows symlinks created inside the container to affect files outside the container. When a user executes `docker cp` on a container filesystem, the Docker daemon process first executes a `FollowSymlinkInScope()` function, which returns the non-symlink path to the file/directory. Only after finding the actual path for both source and destination filenames does `docker cp` start copying files. One problem that arises from this process is that there is no guarantee that the filesystem won't change between running `FollowSymlinkInScope()` and copying the files. Here, a possible attack utilizing a Time-of-Check to Time-of-Use (TOCTOU) race condition is to have a process inside the container apply a symlink right after Docker verifies the symlink path, and right before docker begins writing the file. When docker uses the filename it resolved earlier, it will traverse the symlink to a file on the host machine.

Through this exploit, a container process could potentially overwrite any file in the container when the host tries to copy a file into that container. This includes crucial system files such as password and user records. In our example, we use the four steps shown below in the order when attackers win the race condition to reproduce the exploit:

(1) Host tries to copy a file `w00t_w00t_im_a_flag` from the container's filesystem to the host system by running `docker cp`.

(2) Docker engine resolves both source and destination filenames, traversing any necessary symlinks.

(3) Malicious process inside the container creates another directory (`stash_path`), applies a symlink to `/`, and performs a rename exchange of the original directory containing `w00t_w00t_im_a_flag`, i.e., `totally_safe_path`, with `stash_path`.

(4) Docker engine, uses the filename resolved at Step 2, and performs a read of the container filename, and writes the data to the host filesystem.

Once the malicious process wins the race condition (Step 3), the symlink will be resolved in the host's filesystem and `docker cp` ends up copying the `w00t_w00t_im_a_flag` in the host, rather than the one inside the container. For this exploit, the provenance generated by CLARION graph shown in Figure 3.13 does not show significant difference from the namespace-unaware graph, because there is only one malicious process which will be run from at container start. Yet, without namespace awareness, the analyst will not be able to know that the key malicious process, i.e., `symlink_swap`, is running inside a container.

3.6.2. Cross-container Evaluation

To demonstrate that our solution is generic to several popular container engines together with deeper insights about provenance graph statistics, we select LXC (a classical container engine), rkt (a container engine with the second highest market share), Mesos and Docker for evaluation.

3.6.2.1. Initialization Graphs. We show the provenance graphs for the initialization of a `hello-world` container within each container engine in Figures 3.14, 3.15, and 3.16.

We find the initialization provenance graphs for the three different container engines to be clear and intuitive. They show that even when varying initialization routines are employed by different container engines, (e.g., rkt doesn't start the container before it finishes changing root path, while the other two use the first process inside the container), our initialization patterns always detect them accurately. Moreover, CLARION successfully summarizes the container boundary for all three container engines.

3.6.2.2. Quantitative Provenance Graph Results. We measured the impact of CLARION on provenance graph statistics to quantitatively assess the implications of namespace awareness with various container engines. We selected five popular Docker images that

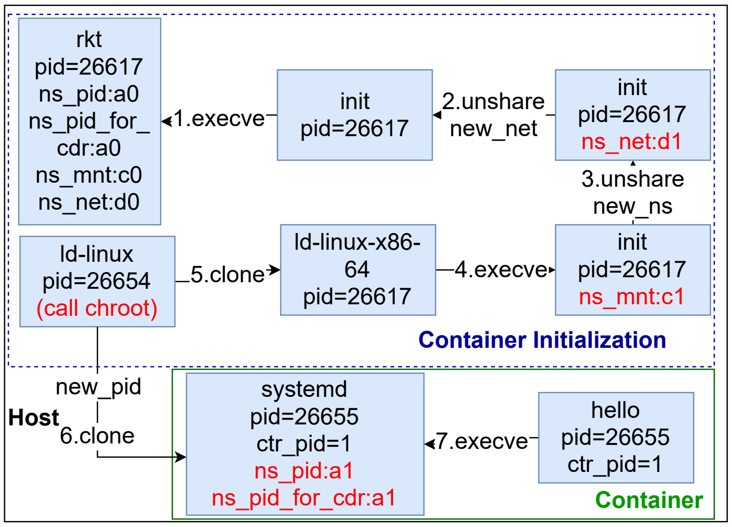


Figure 3.14. Initialization of a hello-world rkt container

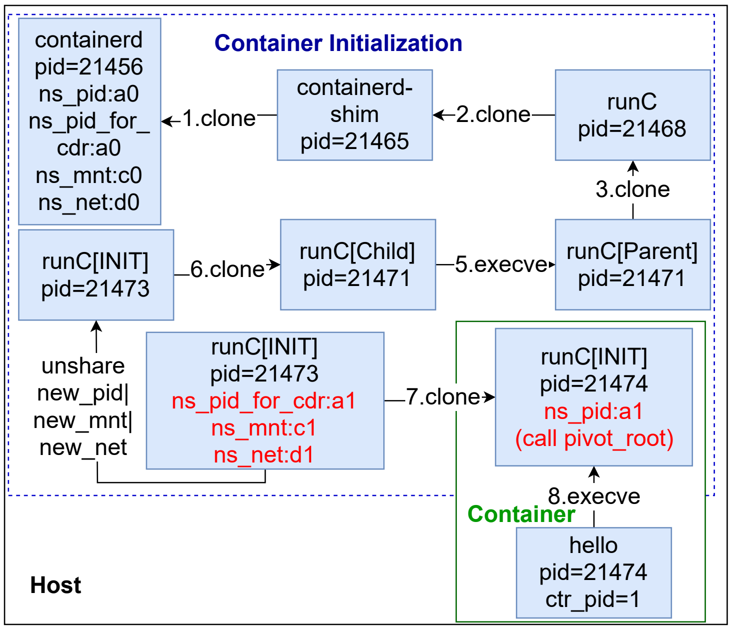


Figure 3.15. Initialization of a hello-world Docker container

cover typical use cases in microservices including the base OS (ubuntu), a popular database (redis), a continuous integration server (jenkins) and a web server (nginx). We ran those images on three popular container engines: Docker, rkt and Mesos. The results are

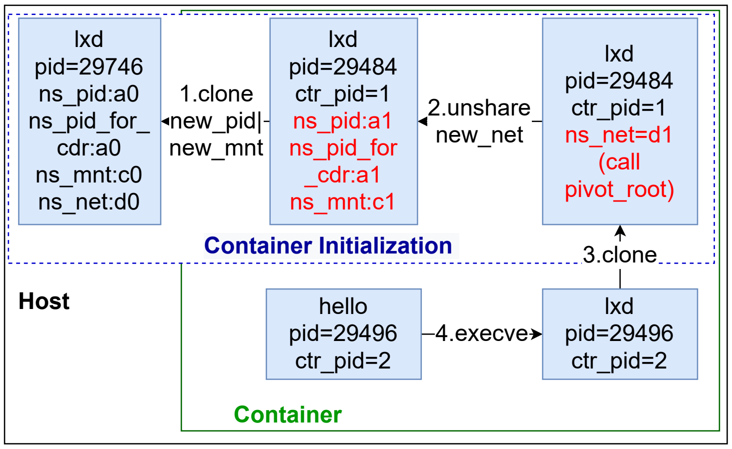


Figure 3.16. Initialization of a hello-world LXC container

reported in Tables 3.7, 3.8, and 3.9, respectively. For each image, we collected the behavior from container initialization to stable operation. In addition, we used two advanced configurations for nginx to highlight the effect of namespace awareness. *MT-4* indicates that we ran the nginx server with worker_process=4, while *MC-4* means we ran four nginx containers concurrently.

We see that in most cases, the total count of vertices and edges are not significantly impacted by the addition of namespace awareness. This is because it is possible for namespace unawareness to add or reduce vertices/edges, depending on the workload. For example, process cloning leads to more spurious vertices while false dependencies due to shared filenames in the mount namespace results in fewer vertices. Generally speaking, fewer vertices and more edges will be a better result because the provenance graph suffers from less fracturing. Hence, we count the (lost/extra) error vertices/edges in two common cases, i.e., process creation and file access, causing fragmentation and ambiguities. Though this may not cover all cases causing error vertices/edges, we believe it provides a useful lower bound to illustrate the severity of the soundness issue. Finally, in the case of components, we can observe significant differences when namespace awareness is turned on. Specifically, in nginx(*MC-4*) for rkt, we can see the components of SPADE are doubled in comparison to CLARION, meaning the corresponding provenance graph fractures significantly. This is because the four-container setting has more workload inside the containers and so the subgraph inside the container is much larger. Since the namespace-unaware system will fail to infer correct provenance inside containers, the whole graph becomes more fractured as well. These results underscore how, especially in microservice scenarios, namespace-unawareness can lead to significant errors due to both fragmentation and ambiguities.

3.6.3. Efficiency Evaluation

Our efficiency evaluation consists of two parts: runtime overhead evaluation and storage overhead evaluation. We deployed a microservice benchmark and conducted a performance comparison between SPADE and CLARION.

3.6.3.1. Experiment Setup. The server machine we used has a configuration of Xeon(R) E5-4669 CPU and 256 GB memory. The microservice benchmark we selected a very popular microservice demo, **Online Boutique**[**42**], provided by Google. It contains 10 representative microservices and a web-based e-commerce app in which users can browse items, add them to the cart, and purchase them (i.e., a typical use-case for modern microservices).

3.6.3.2. Runtime Overhead. To compute the runtime overhead, we started every microservice independently 100 times and recorded the cumulative time for those 100 microservice containers to be initialized. First, we performed this process for each microservice without any audit subsystem enabled to get a baseline. Next, we repeated this evaluation with Linux Audit, SPADE, CLARION, CamFlow, and Linux Audit with SE-Linux labeling [6].

We summarize the detailed results in Table 3.10. The incremental overhead is calculated by comparing CLARION's overhead with that of SPADE. The "overall overheads" are based on comparison against the performance of the Base system. We find that the additional runtime overhead on SPADE imposed by CLARION is under 5% which we consider to be acceptable.

The overall overhead of CLARION consists of SPADE overhead and CLARION's (PID namespace, Netfilter) kernel module overhead. By comparing values in the Base column with CLARION's overhead columns, we see that the major overhead originates from Linux Audit as opposed to extra modules introduced by CLARION.

3.6.3.3. Storage Overhead. We compare the size of raw logs collected by SPADE and CLARION in the aforementioned microservice environment with all 10 microservices. We collected logs for 24 hours and the results are shown in Table 3.11. We see that the additional storage overhead for CLARION is modest (under 5%) and much lower than CamFlow.

[6]Our objective is to obtain an estimate for Winnower's computational overhead. Unfortunately, because we do not have access to the Winnower system, we use Linux Audit with namespace-aware audit rules and SE-Linux-enabled Docker to obtain the results shown under SEL-Audit. We believe SEL-Audit results can serve as a lower-bound estimate of Winnower's computational overhead as Winnower uses Linux Audit and relies on SE-Linux labels. This does not measure the cost associated with Winnower's graph reduction or anomaly detection functionality.

3.7. Related Work

Container Security. With the growing popularity of container-based virtualization, numerous security issues have been identified in container orchestration systems [**43, 35, 19, 23**]. The reasons for these security issues may be attributed to a diverse set of flaws in design assumptions. For instance, to simplify support for file-system features like “bind mount”, container engines, such as Docker do not enable the user namespace by default because this leads to file access privilege problems. But disabling the user namespace also implies that the root user inside the container also becomes the root user outside the container. In several aforementioned security issues, attackers simply leverage this general vulnerability to achieve privilege escalation on the host OS. Given the prevalence of such security issues, developing defensive technology that supports security analysis in container environments is crucial. This paper describes a first step toward a robust forensics analysis framework for containerized application deployments.

Container Vulnerability Analysis. Many existing efforts [**106**] have focused on the problem of container system vulnerability analysis. One line of work leverages traditional static analysis techniques to perform compliance checking on container images, such as those built with Docker. However, they do not protect the integrity of container instances at runtime [**115, 98**]. Thus, contemporary container vulnerability analysis tools are limited in their ability to conduct long-term forensic analysis. Our study complements current container vulnerability analytics by providing a dynamic analysis view that leverages semantics-aware comprehension of attacks targeting running containers.

Provenance Tracking and Causality Analysis. Provenance tracking and causality analysis have played a vital role in system forensics [**97, 105, 100, 94**]. These tools build provenance/causal graphs by connecting system objects like processes, files, and sockets by using low-level events, such as system calls. When an attack entry point is identified, forward and backward tracking along graphs can then be performed to find the attack-related subgraphs. These allow analysts to get a clear understanding of the attack origin and its impact on the system. Several prior efforts have proposed mechanisms that seek to improve the quality of generated provenance/causal graphs [**97, 105, 94**] in different ways. While some of these attempt to mitigate the dependency explosion problem and eliminate unrelated data[**116**], others focus on real-time and scalable graph generation [**100**]. As described in Section 3.3, systems such as Winnower[**80**] and CamFlow[**110**] also have limitations. CamFlow has namespace awareness but not container awareness (i.e., it only extracts namespace identifiers, but does nothing to deal with container semantics or container boundaries.) In contrast, Winnower is container-aware but not namespace-aware. Although it uses SELinux label information to assign docker container IDs for process, file, and socket objects, those labels are not sufficient to fully disambiguate the effect of important syscalls like `clone`, `fork`. However, since both Winnower and run on SPADE, the two systems are complementary and could potentially be integrated. Our work is also more general and agnostic to specific container-management frameworks.

Alternative OS-level Virtualization Techniques. Multiple OS-level virtualization techniques exist on other operating system platforms. Among all those techniques, Solaris zones [**30**] and FreeBSD jails [**114**] show considerable similarity to Linux namespaces because both of them seek to provide isolation of system resources virtualized by Linux namespaces, e.g., process identifiers, filesystem and network stack, while sharing the same underlying kernel. Although conceptually similar, provenance effects from these techniques depend on multiple factors including virtualized resources, OS platforms, audit frameworks, etc. We provide a summary of our investigation into BSD Jail and Solaris Zones in Section 3.4.1.

Table 3.2. Namespace Virtualization: What / How Provenance?

Namespace	**What events will be affected?**	**How each namespace impacts provenance tracking?**
PID	Audit records related to process manipulation system calls (e.g., `clone`, `fork`) will be affected. In those records, the argument fields and return value field with PID semantics are virtualized but the PID fields themselves are virtualized. This leads to a semantic inconsistency.	The aforementioned inconsistency leads to fragmentation in the provenance graph when process creation happens, so the provenance tracking system fails to produce sound provenance information.
Mount	Audit records related to file operation system calls, e.g., `open`, `close` and `read`, will be affected. Just like the PID namespace, argument fields with file path semantics will be virtualized. In addition, the file path in CWD and PATH records will also be virtualized.	Two different files, accessed within two different containers, may have the same name which leads to ambiguity. Thus the provenance tracking system fails to produce sound provenance information.
Network	Audit records containing local IP addresses and local ports of a socket will be affected. Examples include the `bind` system call, which is the only system call directly providing local IP and local ports of a socket in its arguments, and other system calls like the `listen` system call providing socket file descriptors where local IP addresses and ports can be indirectly extracted.	Two sockets in two containers can have the same local IP address and local port leading to ambiguity. Furthermore, sockets inside the container are connected to a host port through port-mapping rules. Without explicit understanding of this mapping information, the provenance system fails to connect the incoming connection to the correct sockets, leading to fragmentation.
IPC	Audit records related to system calls handling SYSV IPC objects, i.e., message queue, semaphore and shared memory segmentation, and the POSIX message queue will be affected, e.g., `msgget`, `mq_open`, `shm_open`. The effect is that argument fields with the semantics of the ID/name of a SYSV IPC object or a POSIX message queue are virtualized.	Two IPC objects of the same type can have the same ID/name, and this will lead to ambiguity in the provenance graph.
User	The only affected data elements are UIDs and GIDs. They do not lead to fragmentation in the provenance graph. As for ambiguity, Linux Audit records can report the host view of UID and GID in the corresponding fields of every audit record so that ambiguity will also be resolved.	Since there is no impact, user namespace auditing is unchanged. Furthermore, most container engines do not use the user namespace in their default container initialization because it breaks access permission to critical libraries on the host and storage features like bind mount may be automatically disabled if the user namespace is enabled in the container.
Time, UTS and Cgroup	These namespaces do not affect dataflow in practice and thus do not directly impact provenance.	N/A

Table 3.3. Provenance Soundness on Sysdig and LTTng

Namespace	**Sysdig**	**LTTng**
PID	Soundness challenge persists because the return values and the arguments providing PID semantics will be virtualized in the audit records corresponding to process manipulation system calls, e.g., `clone`, `fork`.	Soundness challenge persists if only system call events are used in provenance tracking system because the return values and the arguments providing PID semantics will be virtualized. However, LTTng can provide the host-container PID mapping which eliminates the PID namespace soundness challenge.
Mount	Soundness challenge persists because the data fields providing file path semantics, e.g., `name` and `filename`, will be virtualized in the audit records corresponding to file operation system calls, e.g., `open`, `close`, and `read`.	Soundness challenge persists because the data fields providing file path semantics, e.g., `filename`, will be virtualized.
Network	Soundness challenge persists. The data fields having local IP addresses/ports will be virtualized. Examples include the argument (`addr`) of a `bind` system call and the translation of the argument (`fd`) being the socket file descriptor of a `listen` system call.	Local IP addresses and ports are still affected. However, since LTTng does not explicitly transform the `addr` argument in the `bind` system call to a socket address, the soundness challenge in network namespace is less severe.
IPC	Soundness challenge persists. Names/IDs of a SYSV IPC object or a POSIX message queue will be virtuailzed.	Soundness challenge persists. Names/IDs of a SYSV IPC object or a POSIX message queue virtualized by IPC namespace will be virtuailzed.
User	The return values and arguments of UID-manipulation system calls will be virtualized but soundness is not affected.	Soundness is not affected. Furthermore, clarity can be achieved since the UID/GID host-container mapping is provided.
Time, UTS and Cgroup	These do not affect dataflow in practice and thus do not directly impact provenance.	These do not affect dataflow in practice and thus do not directly impact provenance.

Table 3.4. Provenance Soundness in BSD Jails and Solaris Zones

Resource	BSD Jail	Solaris Zone
Process	BSD Jails use JID (Jail ID) to mark the processes inside a jail. Thus no virtualized PID is used and no soundness challenge will be introduced.	Zone ID is used to isolate the processes. Thus no virtualized PID is used and no soundness challenge will be introduced.
File-system	Ambiguity exists because filesystem isolation is also achieved by chroot-like operation and file path will be virtualized while the root directory path is specified by `jail` system call.	Ambiguity exists because a Zone needs a new root directory to be specified.
Network	This depends on what network isolation method is applied. If bind-filtering is applied, sockets are actually created under host network stack so that no soundness challenge would occur. Otherwise, if *epair* of VNET is used for network isolation, each jail would have a completely separate network stack just like what happens in Linux network namespace. Then both fragmentation and ambiguity can exist.	Both fragmentation and ambiguity can exist. When the default exclusive-IP setting is applied, Data-link acts just like veth pairs in Ubuntu and epair in BSD to provide the isolated network stack where sockets are virtualized.
IPC	Ambiguity exists. POSIX IPC objects are naturally isolated and System V IPC objects can be isolated with specific parameters so two IPC objects can have the same ID/name.	Ambiguity exists. System V IPC objects are naturally isolated and two System V IPC objects can have the same ID/name.
User	The same provenance effect as that in Table 3.2 will occur for jails.	The same provenance effect as that in Table 3.2 will occur for zones.
Time, UTS and Cgroup	These do not affect dataflow in practice and thus do not directly impact provenance.	These do not affect dataflow in practice and thus do not directly impact provenance.

Table 3.5. Namespace Provenance Mapping Strategies

Namespace	Affected Provenance Data	Strategy
PID	Process IDs	Host-container mapping
Mount	File paths	Host-container mapping
Network	Local IP addresses and ports	Host-container mapping
IPC	IPC Obejct IDs and names	Namespace labeling

Table 3.6. Operator Annotation

Annotation	Explanation
put((key,value), X)	put a pair (key,value) in a mapping X
get(key, X)	get the value from a mapping Y given the key

Table 3.7. Provenance Graph Statistics Comparison (Docker)

Service	**Error Vertices** (lost/extra)	**Error Edges**	**Vertices SPADE/ CLARION**	**Edges SPADE/ CLARION**	**Components SPADE/ CLARION**
ubuntu	58 (8/50)	900	4236 / 4152	19056 / 19066	22 / 22
redis	78 (18/60)	1612	4759 / 4677	22856 / 22871	23 / 22
jenkins	55 (2/53)	133	4673 / 4581	21024 / 21026	28 / 25
node	72 (9/63)	919	4473 / 4387	19371 / 19376	24 / 21
nginx	72 (18/54)	1558	4737 / 4637	20780 / 20841	26 / 21
nginx MT-4	73 (19/54)	1662	7467 / 7345	40711 / 40781	32 / 26
nginx MC-4	376 (135/241)	7492	23875 / 23233	119128 / 119372	49 / 31

Table 3.8. Provenance Graph Statistics Comparison (rkt)

Service	**Error Vertices** (lost/extra)	**Error Edges**	**Vertices SPADE/ CLARION**	**Edges SPADE/ CLARION**	**Components SPADE/ CLARION**
ubuntu	80 (59/21)	10076	19047 / 19031	88022 / 88114	28 / 27
redis	171 (145/26)	12540	19348 / 19330	90471 / 90573	26 / 26
jenkins	99 (84/15)	10749	19441 / 19420	90798 / 90893	28 / 28
node	138 (103/35)	13334	19600 / 19575	90029 / 90125	27 / 25
nginx	85 (69/16)	10671	19666 / 19617	90885 / 91063	34 / 28
nginx MT-4	101 (70/31)	15272	23761 / 23721	106599 / 106754	40 / 33
nginx MC-4	828 (726/102)	65022	92962 / 93158	425550 / 426194	66 / 36

Table 3.9. Provenance Graph Statistics Comparison (Mesos)

Service	**Error Vertices** (lost/extra)	**Error Edges**	**Vertices SPADE/ CLARION**	**Edges SPADE/ CLARION**	**Components SPADE/ CLARION**
ubuntu	10 (5/5)	241	28019 / 27932	76555 / 76561	18 / 17
redis	30 (18/12)	3149	19667 / 19574	59504 / 59507	17 / 17
jenkins	267 (210/57)	25453	34664 / 34560	141381 / 141387	26 / 24
node	21 (9/12)	1106	4960 / 4864	15492 / 15495	16 / 15
nginx	23 (20/3)	2389	5159 / 5067	17580 / 17582	20 / 17
nginx MT-4	23 (20/3)	2418	5185 / 5093	22545 / 22547	17 / 16
nginx MC-4	1402 (1383/19)	30304	19606 / 18817	66972 / 66982	30 / 22

Table 3.10. Runtime Overhead Comparison of Container Provenance Systems

Service	**Base** (secs)	**Linux Audit** (secs)	**SPADE** (secs)	**CLARION** (secs)	**Incremental Overhead (CLARION)**	**Overall Overhead (Audit + SPADE + CLARION)**	**Overall Overhead (CamFlow)**	**Overall Overhead (SEL-Audit)**
frontend	1503 s	1550 s	1558 s	1578 s	1.3%	3.7%	4.8%	32.4%
productcatalog service	668 s	679 s	681 s	691 s	1.5%	3.4%	9.1%	25.0%
currency service	1104 s	1139 s	1153 s	1169 s	1.4%	5.9%	12.9%	8.5%
payment service	1082 s	1123 s	1126 s	1143 s	1.5%	5.6%	11.5%	9.7%
shipping service	434 s	446 s	449 s	451 s	0.4%	3.9%	22.5%	25.8%
email service	929 s	960 s	1028 s	1068 s	3.9%	15.0%	1.2%	17.6%
checkout service	682 s	719 s	714 s	734 s	2.8%	7.6%	3.2%	13.9%
recommendation service	8726 s	9418 s	9337 s	9729 s	4.2%	11.5%	9.5%	19.5%
adservice	4438 s	4454 s	4518 s	4571 s	1.2%	3.0%	5.3%	8.5%
loadgenerator	200 s	208 s	212 s	215 s	1.4%	7.5%	20.4%	29.4%

Table 3.11. Storage Overhead Comparison

SEL-Audit	**CamFlow**	**SPADE**	**CLARION**	**Incremental Overhead**
168.79 GB	312.56 GB	174.68 GB	181.75 GB	4.05%

CHAPTER 4

Conclusion

In this dissertation, I elaborate a research problem in applying graph-based low-level event analysis on endpoint detection and response system: how can we *effectively* and *efficiently* leverage graph-based low-level system event analysis to perform intrusion response for *different platform/environment/tasks*? Then I investigate this problem in two projects, i.e., RATScope with Windows/operating system/detection setup and CLARION with Linux/container/forensics setup.

The brief conclusion is shown as follows.

- With the Windows/operating system/detection setup, RATScope uncovers the problem of missing arguments in Windows low-level events, i.e., ETW events, which must be addressed so that effectiveness can be achieved. Given a detection task, a meticulous graph pattern matching algorithm is also needed for providing efficiency.
- With the Linux/container/forensics setup, CLARION exposes the influence introduced by container virtualization in Linux low-level events, i.e., Linux Audit events, which usually cannot be realized by people. Given a forensic task, efforts on selecting system calls and system implementation optimization must be paid to reach efficiency.

Some further insights can be extracted here.

- For platforms, different systems usually have unique technical challenges related to its native low-level auditing system, e.g., missing argument problem in ETW. Specific technical solutions must be provided to achieve effectiveness. When multiple native low-level auditing systems exist, we should select the one with the best performance to get efficiency.
- For environments, different environments can introduce neglected problems when environments are theoretically similar, e.g., OSes and containers. Those problems could lead to severe effectiveness problems if not handled properly.
- For tasks, detection systems usually emphasize more on runtime efficiency since response delay is an important criterion for detection systems. In contrast, forensic systems pay more attention on storage efficiency because they store massive data on disks.

Finally, my publications are listed as follows.

- *Runqing Yang, Xutong Chen, Haitao Xu, Yueqiang Cheng, Chunlin Xiong, Linqi Ruan, Mohammad Kavousi, Zhenyuan Li, Liheng Xu, and Yan Chen, "RATScope: Recording and Reconstructing Missing RAT Attacks for Forensic Analysis with Semantics on Windows", in the IEEE Transactions on Dependable and Secure Computing, Oct. 2020, doi: 10.1109/TDSC.2020.3032570.
- Chunlin Xiong, Tiantian Zhu, Weihao Dong, Linqi Ruan, Runqing Yang, Yueqiang Cheng, Yan Chen, Shuai Cheng, and Xutong Chen, "CONAN: A Practical Real-time APT Detection System with High Accuracy and Efficiency", in the IEEE Transactions on Dependable and Secure Computing, Feb. 2020, doi: 10.1109/TDSC.2020.2971484.

- Xutong Chen, Hassaan Irshad, Yan Chen, Ashish Gehani, and Vinod Yegneswaran, "CLARION: Sound and Clear Provenance Tracking for Microservice Deployments", in the Proc. of USENIX Security 2021.

References

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

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

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

[4] Apache Mesos. http://mesos.apache.org/.

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

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

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

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

[9] Captain Hook:Pirating AVS to Bypass Exploit Mitigations. https://goo.gl/zVyuAL. accessed on 2020-2-10.

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

[11] CoreOS rkt. https://coreos.com/rkt/.

[12] CVE-2018-15664. https://nvd.nist.gov/vuln/detail/CVE-2018-15664.

[13] CVE-2019-14271. https://nvd.nist.gov/vuln/detail/CVE-2019-14271.

[14] CVE-2019-5736. https://nvd.nist.gov/vuln/detail/CVE-2019-5736.

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

[16] Desktop Operating System Market Share Worldwide. http://bit.ly/341OxHC. accessed on 2020-2-10.

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

[18] Docker. https://www.docker.com/.

[19] Docker Engine Large Integer Denial of Service Vulnerability. https://tools.cisco.com/security/center/resources/security-alerts-announcement.

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

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

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

[23] Hack Allows Escape of Play-with-Docker Containers. https://threatpost.com/hack-allows-escape-of-play-with-docker-containers/140831/.

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

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

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

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

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

[29] IDC data leakage report. https://blog.rapid7.com/2016/03/31/idc-says-70-of-successful-breaches-originate-on-the-endpoint/.

[30] Introduction to Solaris Zones. https://docs.oracle.com/cd/E19044-01/sol.containers/817-1592/zones.intro-1/index.html.

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

[32] Kernel Patch Protection — Wikipedia, The Free Encyclopedia. `https://goo.gl/S4idr7`. accessed on 2020-2-10.

[33] Kernel Self-Protection. `https://www.kernel.org/doc/html/latest/security/self-protection.html`.

[34] KHOBE – 8.0 earthquake for Windows desktop security software. `https://goo.gl/5UhzpQ`. accessed on 2020-2-10.

[35] Kinsing Malware Attacks Targeting Container Environments. `https://blog.aquasec.com/threat-alert-kinsing-malware-container-vulnerability`.

[36] Linux Namespace. `https://www.man7.org/linux/man-pages/man7/namespaces.7.html`.

[37] LTTng. `https://lttng.org/`.

[38] LXC: Linux Container. `https://linuxcontainers.org/lxd/docs/master/`.

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

[40] Metta. `http://bit.ly/2rJDVxb`. accessed on 2020-2-10.

[41] Microservice Architecture. `https://microservices.io/patterns/microservices.html`.

[42] Microservice Demo: Online Boutique. https://github.com/GoogleCloudPlatform/microservices-demo.

[43] Misconfiged containers lead to malicious cryptomining. https://www.darkreading.com/attacks-breaches/misconfigured-containers-again-targeted-by-/cryptominer-malware/.

[44] Netfilter Architecture. https://www.netfilter.org/.

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

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

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

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

[49] PANDA. http://bit.ly/2OhOyiv. accessed on 2020-2-10.

[50] Plague in (security) software drivers. https://goo.gl/kmycvb. accessed on 2020-2-10.

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

[52] Quantum of Surveillance: Familiar Actors and Possible False Flags in Syrian Malware Campaigns. https://bit.ly/35PPwey. accessed on 2020-2-10.

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

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

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

[56] S. Grubb. Redhat linux audit. https://people.redhat.com/sgrubb/audit/.

[57] Serverless Computing. https://aws.amazon.com/serverless/?nc1=h_ls.

[58] Shocking New Reveals From Sony Hack. https://goo.gl/18RnK9. accessed on 2020-2-10.

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

[60] Sony Pictures hack. https://goo.gl/t6oJcp. accessed on 2020-2-10.

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

[62] Sysdig. https://github.com/draios.

[63] Taintgrind: a Valgrind taint analysis tool. `http://bit.ly/2Kl0b5q`. accessed on 2020-2-10.

[64] Target's Data Breach: The Commercialization of APT. `https://goo.gl/cDYXCG`. accessed on 2020-2-10.

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

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

[67] VOLATILITY. `http://bit.ly/2rJFtat`. accessed on 2020-2-10.

[68] Windows-10-Mitigation-Improvement. `https://ubm.io/2IIVwtn`. accessed on 2020-2-10.

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

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

[71] 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.

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

[73] Adam Bates, Dave Jing Tian, Kevin RB Butler, and Thomas Moyer. Trustworthy whole-system provenance for the linux kernel. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 319–334, 2015.

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

[75] 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*, 2017.

[76] 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. In *USENIX Security*, 2018.

[77] 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.

[78] Ashish Gehani and Dawood Tariq. Spade: support for provenance auditing in distributed environments. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 101–120. Springer, 2012.

[79] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal De Lara. The taser intrusion recovery system. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 163–176, 2005.

[80] Wajih Ul Hassan, Lemay Aguse, Nuraini Aguse, Adam Bates, and Thomas Moyer. Towards scalable cluster auditing through grammatical inference over provenance graphs. In *Network and Distributed Systems Security Symposium*, 2018.

[81] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. NODOZE: Combatting Threat Alert Fatigue with Automated Provenance Triage. In *NDSS*, 2019.

[82] 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.

[83] 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.

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

[85] Taesoo Kim, Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, et al. Intrusion recovery using selective re-execution. In *OSDI*, pages 89–104, 2010.

[86] Samuel T King and Peter M Chen. Backtracking intrusions. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 223–236. ACM, 2003.

[87] Samuel T King and Peter M Chen. Backtracking intrusions. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 223–236, 2003.

[88] Samuel T King, Zhuoqing Morley Mao, Dominic G Lucchetti, and Peter M Chen. Enriching Intrusion Alerts Through Multi-Host Causality. In *NDSS*, 2005.

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

[90] 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 Security*, 2009.

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

[92] Srinivas Krishnan, Kevin Z Snow, and Fabian Monrose. Trail of bytes: efficient support for forensic analysis. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 50–60, 2010.

[93] 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.

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

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

[96] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High Accuracy Attack Provenance via Binary-based Execution Partition. In *NDSS*, 2013.

[97] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High Accuracy Attack Provenance via Binary-based Execution Partition. In *NDSS*, 2013.

[98] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. A measurement study on linux container security: Attacks and countermeasures. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 418–429, 2018.

[99] 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.

[100] 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.

[101] John Lyle and Andrew Martin. Trusted computing and provenance: better together. Usenix, 2010.

[102] 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.

[103] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. MPI: Multiple Perspective Attack Investigation with Semantic Aware Execution Partitioning. In *USENIX Security*, 2017.

[104] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting. In *NDSS*, 2016.

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

[106] Antony Martin, Simone Raponi, Théo Combe, and Roberto Di Pietro. Docker ecosystem–vulnerability analysis. *Computer Communications*, 122:30–43, 2018.

[107] 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.

[108] Sadegh M Milajerdi, Rigel Gjomemo, Birhanu Eshete, R Sekar, and VN Venkatakrishnan. HOLMES: real-time APT detection through correlation of suspicious information flows. In *S&P*, 2019.

[109] James Morris, Stephen Smalley, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *USENIX Security Symposium*, pages 17–31. ACM Berkeley, CA, 2002.

[110] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. Practical whole-system provenance capture. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 405–418, 2017.

[111] Kexin Pei, Saltaformaggio Gu, Zhongshu, and et al. HERCULE: attack story reconstruction via community discovery on correlated log graph. In *ACSAC*, 2016.

[112] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time. In *USENIX Security*, 2019.

[113] Mohammad Rezaeirad, Brown Farinholt, Hitesh Dharmdasani, Paul Pearce, Kirill Levchenko, and Damon McCoy. Schrödinger's RAT: Profiling the Stakeholders in the Remote Access Trojan Ecosystem. In *USENIX Security*, 2018.

[114] Matteo Riondato. Jails. `https://www.freebsd.org/doc/handbook/jails.html`.

[115] Byungchul Tak, Canturk Isci, Sastry Duri, Nilton Bila, Shripad Nadgowda, and James Doran. Understanding security implications of using containers in the cloud. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 313–319, 2017.

[116] Yutao Tang, Ding Li, Zhichun Li, Mu Zhang, Kangkook Jee, Xusheng Xiao, Zhenyu Wu, Junghwan Rhee, Fengyuan Xu, and Qun Li. Nodemerge: template based efficient data reduction for big-data causality analysis. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1324–1337, 2018.